
Language Reference
Menu Commands
Database Drivers
Template Language
CLARION
FOR WINDOWS
Version 1.5
TopSpeed
CORPORATION
Late Breaking News
Common Questions
Guide to Examples
Index to All Manuals

Clarion for Windows 1.5

TopSpeed Corporation
150 East Sample Road
Pompano Beach, Florida 33064
(305) 785-4555

TopSpeed Software Limited
Clare House, Thompsons Close
Harpenden, Herts AL5 4ES
United Kingdom
+44 (0)158 276 3200

Version 1.500
0995

The following employees participated in the creation of Clarion for Windows:

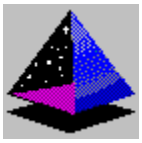
(In alphabetical order)

NANCY AUGUSTE
BRUCE BARRINGTON
LINDA BART
GEORGE BARWOOD
DAVID BAYLISS
SUSANNE BOSTIC
PAM BRECHLIN
JOHN BROADWATER
DOUG BROWN
TRENT BUCK
RICHARD CHAPMAN
ILKA CIOCANEL
JIM DeFABIA
CARON DETTMANN
SCOTT FERRETT
MARK FRITZINGER
BOB FOREMAN
VINCE GEORGE
GAVIN HALLIDAY
ROY HAWKES
SUSAN HELMS
JOHN HERRON, Jr.
DEBBY HERMAN
NIGEL HICKS
MARCIA HOLMES
KHRIS HOVEN
JOHN IACOVELLI
ANDY IRELAND
STEVEN JAMES
GEORGE LANITIS
LINDA LONIGRO
BARBARA KLEPEISZ
ANNA KRAUSS
BARRY LYNCH
BILLMcCOMBS
CARLOS MARRERO
CHARLENE MILLER
TOM MOSELEY
GILDAR PASSOS
OLE POULSEN
HELEN POUSTIAN
ROY RAFALCO
PATRICK RYAN
LISA SIMMONS
FRANCIS SINYANGWE
H. JOHN SCOTT
RICHARD TAYLOR
CHRISTINE TIMMIS

JEFF TRUDEL
RANDY WOOD
BOB ZAUNERE

Special thanks to the thousands of Beta testers who helped make this product possible.

Additional thanks to the shareholders and Team TopSpeed.

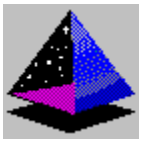


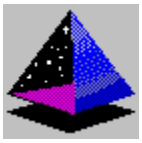
Team TopSpeed

Clarion for Windows Team TopSpeed Members:
Dave Harms (Team Leader)
Ross Santos
Bob Butler
Andy Stapleton
Larry Teames
Steve Bottomly
Randy Goodhew
Todd Seidel
Nigel Moss
Lee White

Clarion for DOS Team TopSpeed Members:
Nik Johnson (Team Leader)
Tom Stevens
Andy Stapleton
George Hale
Sam Bellamy
Nick Van Eldyk
Bruce Wells
Gregory Bailey

FidoNet/Internet Team TopSpeed Members:
Mike Gould
Colin Wynn
Ray Creighton





Language Reference

Language Reference

Language Reference

Language Reference

Language Reference

Language Reference

Language Reference

Language Reference

Language Reference

PRINT (print a report structure)

- Graphics Commands
- Data Files
- File Views
- Memory Queues
- Miscellaneous Procedures and Functions
- DDE Library Reference
- Keycodes
- Properties
- Events

Language Reference

Language Reference

Language Reference

Language Reference

Language Reference

Language Reference

Language Reference

Language Reference

Language Reference

Introduction

Clarion for Windows is an integrated environment for writing data processing applications and management information systems for microcomputers using the Windows operating environment. Clarion ´s programming language is the foundation of this environment. In this reference, the language is concisely documented in a modular fashion. Although this is not a text book, you should consult this reference when you want to know the precise syntax required to implement any declaration, statement, or function.

As far as possible, real-world example code is provided for each item.

Language Reference Organization

Introduction provides an introduction to the Clarion Language Reference. It provides a brief overview of the contents of each chapter, and a guide to help the reader understand the documentation conventions used throughout the book.

Program Source Code Format provides the general layout of a Clarion Windows program. Punctuation, special characters, reserved words, and a detailed description of the "building blocks" required to create modular, structured Clarion source code are documented here.

Declaring Variables describes the data types and attributes used to declare variables in a Clarion program. In addition, formatting masks, called "picture tokens," are defined and illustrated.

Expressions and Assignments defines the syntax required to combine variables, functions, and constants into numeric, string, or logical expressions. It also defines how the value of an expression is assigned to variables.

Control Statements describes compound executable statements that control program flow and operation.

Window Structures describes the APPLICATION and WINDOW data structures and all their components and attributes.

Window Commmands describes the executable statements and functions that are specific to APPLICATION and WINDOW structures.

Reports describes the REPORT data structure and all its components and attributes. The executable statements and functions that are specific to using a REPORT structure are also covered here.

Graphics Commands describes executable statements and functions that draw graphical figures in APPLICATION, WINDOW, and REPORT structures.

Data Files describes the FILE structure. This chapter covers the declarations, statements, and functions which access data files. The statements and functions required for multi-user and transaction processing systems are also documented here.

File Views describes the VIEW structure. This chapter covers the declarations, statements, and functions which access data files through the VIEW structure.

Memory Queues describes the QUEUE data structure, which is used to rapidly process information in random access memory. Along with all its components and attributes, the executable statements and functions that are specific to using a memory QUEUE are also covered here.

Miscellaneous Procedures and Functions documents the statements and functions that do not specifically apply to the subjects covered in chapters 1 through 12.

DDE Library Reference

Clarion Keycodes

Reference Item Format

Each Clarion programming language element listed in this reference is displayed in UPPER CASE letters. Components of the language are documented with a syntax diagram, a detailed description, and source code examples.

Items are documented in logical groupings, dependent upon their hierarchical relationships. Therefore, the "table of contents" for this part of the help is not listed in alphabetical order. You can find alphabetical listings in the syntax and attributes topics, or by pressing the **Search** button.

The documentation format used in this book is illustrated in the syntax diagram .

KEYWORD (short description of intended use)

[label] **KEYWORD(** | *parameter1* | [*parameter2*] **)** [**ATTRIBUTE1( )**] [**ATTRIBUTE2( )**]
| *alternate* |
| *parameter* |
| *list* |

KEYWORD A brief statement of what the KEYWORD does.

parameter1 A complete description of parameter1, along with how it relates to parameter2 and the KEYWORD.

parameter2 A complete description of parameter2, along with how it relates to parameter1 and the KEYWORD. Because it is enclosed in brackets, [ ], it is optional, and may be omitted.

alternate parameter list
A complete description of alternates to parameter1, along with how they relate to parameter2 and the KEYWORD.

ATTRIBUTE1 A sentence describing the relation of ATTRIBUTE1 to the KEYWORD.

ATTRIBUTE2 A sentence describing the relation of ATTRIBUTE2 to the KEYWORD.

A concise description of what the **KEYWORD** does. In many cases the KEYWORD will be an attribute of a keyword that was described in the preceding text. Sometimes a KEYWORD has no parameters and/or attributes.

Events Generated: If the KEYWORD generates events, they are listed here.

Return Data Type: The data type returned if KEYWORD is a function.

Errors Posted: If KEYWORD posts errors which may be trapped by the ERROR and ERRORCODE functions, they are listed here.

Example:

```
FieldOne = FieldTwo + FieldThree          !This is a source code example
FieldThree = KEYWORD(FieldOne,FieldTwo)     !Comments follow the "!" character
```

See Also:

Related Topic

Related Topic

Conventions and Symbols

Symbols are used in the syntax diagrams as follows:

Symbol	Meaning
[ ]	Brackets enclose an optional (not required) attribute or parameter.
( )	Parentheses enclose a parameter list.
| |	Vertical lines enclose parameters, where one, but only one, of the parameters is allowed.

Coding example conventions used throughout this manual:

CLARION KEYWORDS	All caps
DataNames	Mixed case with caps used for readability
Comments	Predominantly lower case

The purpose of these conventions is to make the code examples readable and clear.

Related Topic

CLICKING on a hotspot takes you to the related topic

Related Attribute

CLICKING on a hotspot takes you to the related topic

Program Source Code Format

Statement Format

Clarion is a "statement oriented" language. A statement oriented language makes use of the fact that its source code is contained in ASCII text files so every line of code is a separate record in the file. Therefore, the Carriage Return/Line Feed record delimiter can be used to eliminate punctuation.

In general, the Clarion statement format is:

```
label  STATEMENT[(parameters)] [,ATTRIBUTE[(parameters)]] ...
```

Attributes specify the properties of the item and are only used on data declarations. Executable statements take the form of a standard procedure call, except assignment statements (A = B) and control structures (such as IF, CASE, and LOOP).

A statement´s label must begin in column one (1) of the source code. A statement without a label must not start in column one. A statement is terminated by the end of the line. A statement too long to fit on one line can be continued by a vertical bar (|). The semi-colon is an optional statement separator that allows you to place more than one statement on a line.

Being a statement oriented language eliminates from Clarion much of the punctuation required in other languages to identify labels and separate statements. Blocks of statements are initiated by a single compound statement, and are terminated by an END statement (or period).

Declaration and Statement Labels

The language statements in a source module can be divided into two general categories: data declarations and executable statements, or simply "data" and "code."

During program execution, data declarations reserve memory storage areas that are manipulated by executable statements. A label is required for the data to be referenced in executable code. All variables, data structures, PROCEDUREs, FUNCTIONs, and ROUTINEs are referenced by labels.

A label defines a specific location in a PROGRAM. Any code statement may be identified and referenced by a label. This allows it to be used as the target of a GOTO statement. Each label on an executable statement adds ten bytes to the executable code size, even if not referenced.

The label on a PROCEDURE or FUNCTION statement is the procedure or function´s name. Using the label of a PROCEDURE in an executable statement executes the procedure. The label of a FUNCTION is used in expressions, or parameter lists of other functions, to assign the value returned by the function.

The rules for valid Clarion labels are:

A label MUST begin in column one (1) of the source code.

A label may contain letters (upper or lower case), numerals 0 through 9, the underscore character (_), and colon (:).

The first character must be a letter or the underscore character.

Labels are not case sensitive (i.e. CurRent and CURRENT are the same).

A label may not be a reserved word.

Structure Termination

Compound data structures are created when data declarations are nested within other data declarations. There are many compound data structures within the Clarion language: APPLICATION, WINDOW, REPORT, FILE, RECORD, GROUP, VIEW, etc. These compound data structures must be terminated by a period (.) or the keyword END. IF, CASE, EXECUTE, LOOP, BEGIN, and ACCEPT are all executable control structures which must also be terminated with a period or END statement.

Field Qualification

Variables declared as members of complex data structures (GROUP, QUEUE, FILE, RECORD, etc.) may have duplicate labels, as long as the duplicates are not contained within the same structure. To explicitly reference fields with duplicate labels in separate structures, you may use the PRE attribute on the structures just as it is documented (Prefix:FieldLabel) to provide unique names for each field. However, the PRE attribute is not required for this purpose and may be omitted.

Any field of any complex structure can be explicitly referenced by prepending the label of the structure containing the field to the field label, separated by a colon (StructureName:FieldLabel). You must use this Field Qualification syntax to reference any field in a complex structure that does not have a PRE attribute.

If the field is within nested complex data structures, you must prepend each successive level's structure label to the field label to explicitly reference the field (if the nested structure has a label). This means that, in the case of a FILE structure (without a PRE attribute) in which the RECORD structure has a label, the individual fields in the file must be referenced as FileLabel:RecordLabel:FieldLabel. If the FILE's RECORD structure does not have a label, the individual fields are referenced as FileLabel:FieldLabel.

Example:

```
MasterFile  FILE,DRIVER('TopSpeed')
Record      RECORD
AcctNumber      LONG        !Referenced as Masterfile:Record:AcctNumber
            . .

Detail      FILE,DRIVER('TopSpeed')
              RECORD
AcctNumber      LONG        !Referenced as Detail:AcctNumber
            . .

Memory      GROUP,PRE(Mem)
Message       STRING(30)     !May be referenced as Mem:Message or Memory:Message
            END

SaveQueue   QUEUE
Field1      LONG          !Referenced as SaveQueue:Field1
Field2      STRING        !Referenced as SaveQueue:Field2
            END

OuterGroup  GROUP
Field1      LONG          !Referenced as OuterGroup:Field1
Field2      STRING        !Referenced as OuterGroup:Field2
InnerGroup    GROUP
Field1        LONG        !Referenced as OuterGroup:InnerGroup:Field1
Field2        STRING      !Referenced as OuterGroup:InnerGroup:Field2
              END
            END
```

See Also: PRE

Reserved Words

The following keywords are reserved and may not be used as labels:

```
ACCEPT
AND
BEGIN
BREAK
BY
CASE
COMPILE
CYCLE
DO
EJECT
ELSE
ELSIF
EMBED
END
ENDEMBED
EXECUTE
EXIT
FUNCTION
GOTO
IF
INCLUDE
LOOP
MEMBER
NOT
OF
OMIT
OR
OROF
PROCEDURE
PROGRAM
RETURN
ROUTINE
SECTION
THEN
TIMES
TO
UNTIL
WHILE
XOR
```

The following keywords may be used as labels of data structures or executable statements. They may not be used as labels of PROCEDURE or FUNCTION statements:

APPLICATION
CODE
DETAIL
FILE
FOOTER
FORM
GROUP
HEADER
ITEM
MAP
MENU
MENUBAR
MODULE
OPTION
QUEUE
RECORD
REPORT
SUBTITLE
TITLE
TOOLBAR
VIEW
WINDOW

Special Characters

Initiators:

!	Exclamation point begins a source code comment.
?	Question mark begins a field equate label.
@	"At" sign begins a picture token.
*	Asterisk begins a parameter passed by address in a MAP prototype.

Terminators:

;	Semi-colon is an executable statement separator.
CR/LF	Carriage-return/Line-feed is an executable statement separator.
.	Period terminates a structure.
|	Vertical bar is the source code line continuation character.
#	Pound sign declares an implicit LONG variable.
$	Dollar sign declares an implicit REAL variable.
"	Double quote declares an implicit STRING variable.

Delimiters:

( )	Parentheses enclose a parameter list.
[ ]	Brackets enclose an array subscript list.
´ ´	Single quotes enclose a string constant.
{ }	Curly braces enclose a repeat count in a string constant, or a control field property parameter to a field equate label in an assignment statement.
< >	Angle brackets enclose an ASCII code in a string constant, or indicate a parameter in a MAP prototype which may be omitted.
:	Colon separates the start and stop positions of a string "slice."

Connecters:

.	Period is a decimal point used in numeric constants.
,	Comma connects parameters in a parameter list.
:	Colon connects a prefix to a label or character groups within a label.
$	Dollar sign connects the window to a field equate label in a control property assignment statement.
_	Underscore connects character groups within a label.

Operators:

+	Plus sign indicates addition.
-	Minus sign indicates subtraction.
*	Asterisk indicates multiplication.
/	Slash indicates division.
%	Percent sign indicates modulus division.
^	Carat indicates exponentiation.
<	Left angle bracket indicates less than.
>	Right angle bracket indicates greater than.
=	Equal sign indicates assignment or equivalence.
~	Tilde indicates the logical "NOT" operator.
&	Ampersand indicates concatenation.

Program Format

PROGRAM (declare a program)

PROGRAM
[**MAP**
prototypes
[**MODULE( )**
prototypes
END]
END]
global data
CODE
statements
[**RETURN**]
procedures or functions

PROGRAM The first declaration in a Clarion program source module. Required.

MAP Global procedure and function declarations.

MODULE Declare member source modules.

prototypes PROCEDURE and/or FUNCTION declarations.

global data Declare Global data which may be referenced by all procedures and functions.

CODE Begin executable statements.

statements Executable program instructions.

RETURN Terminate program execution. Return to operating system **control.**

procedures or functions
Source code for the procedures and functions in the PROGRAM module.

The **PROGRAM** statement is required to be the first declaration in a Clarion program source module. It may only be preceded by source code comments or a TITLE or SUBTITLE compiler directive. The PROGRAM source file name is used as the object (.OBJ) and executable (.EXE) file name, when compiled. The PROGRAM statement may have a label, but the label is ignored by the compiler.

A PROGRAM with PROCEDUREs and/or FUNCTIONs must have a MAP structure. The MAP declares the PROCEDURE and/or FUNCTION prototypes. Any PROCEDURE or FUNCTION contained in a separate source file must be declared in a MODULE structure within the MAP.

Data declared in the PROGRAM module, between the keywords PROGRAM and CODE, is Global data that may be accessed by any PROCEDURE or FUNCTION in the PROGRAM. Its memory allocation is Static.

Example:

```
        PROGRAM                     !Sample program declaration
Fahrenheit   REAL(0)                      !Global data declarations
Centigrade   REAL(0)
Window WINDOW(´Temperature Conversion´),CENTER,SYSTEM
        STRING(´Enter Fahrenheit Temperature: ´),AT(34,50,101,10)
        ENTRY(@N-04),AT(138,49,60,12),USE(Fahrenheit)
        STRING(´Centigrade Temperature:´),AT(34,71,80,10),LEFT
        ENTRY(@N-04),AT(138,70,60,12),USE(Centigrade),SKIP
        BUTTON(´Another´),AT(34,92,32,16),USE(?Another)
        BUTTON(´Exit´),AT(138,92,32,16),USE(?Exit)
      END
  CODE                                 !Begin executable code section
  OPEN(Window)
```

```
ACCEPT
  CASE ACCEPTED()
  OF ?Fahrenheit
    Centigrade = (Fahrenheit - 32) / 1.8
    DISPLAY(?Centigrade)
  OF ?Another
    Fahrenheit = 0
    Centigrade = 0
     DISPLAY
    SELECT(?Fahrenheit)
  OF ?Exit
    BREAK
  END
END
CLOSE(Window)
RETURN
```

See Also:

MAP

MODULE

PROCEDURE

FUNCTION

Data Declarations and Memory Allocation

MEMBER (identify member source file)

```
        MEMBER(program)
        [MAP
            prototypes
         END ]
[label] local data
        procedures or functions
```

MEMBER	The first statement in a source module that is not a PROGRAM source file. Required.
program	A string constant containing the filename (without extension) of a PROGRAM source file. This parameter is required.
MAP	Local procedure and function declarations. Any procedures or functions declared here may be referenced only by the procedures or functions in the MEMBER module.
prototypes	PROCEDURE and/or FUNCTION declarations.
local data	Declare Local Static data which may be referenced only by the procedures and functions whose source code is in the MEMBER module.

procedures or functions
Source code for the procedures and functions in the MEMBER module.

MEMBER is the first statement required to be in a source module that is not a PROGRAM source file. It may only be preceded by source code comments or a TITLE or SUBTITLE compiler directive. It is required at the beginning of any source file that contains PROCEDUREs or FUNCTIONs that are used by a PROGRAM. The MEMBER statement identifies the *program* to which the source MODULE belongs.

A MEMBER module may have a local MAP structure. Procedures and functions declared in this MAP are visible only to the other procedures and functions in the MEMBER module. The source code for the procedures and functions declared in this MEMBER MAP may be contained in the MEMBER source file, or another file.

If the source code for the PROCEDURE or FUNCTION declared in a MEMBER MAP is contained in a separate file, the PROCEDURE or FUNCTION´s *prototype* must be declared in a MODULE structure within the MEMBER MAP. That separate source file MEMBER MODULE must also contain its own MAP which declares the same *prototype* for that PROCEDURE or FUNCTION. Any PROCEDURE or FUNCTION not declared in the Global (PROGRAM) MAP must be declared in a local MAP in the MEMBER MODULE which contains its source code.

Data declared in the MEMBER module, after the keyword MEMBER and before the first PROCEDURE or FUNCTION statement, is Member Local data that may only be accessed by PROCEDUREs or FUNCTIONs within the module (unless passed as a parameter). Its memory allocation is Static.

Example:

```
!Source1 module contains:
   MEMBER(´OrderSys´)      !Module belongs to the OrderSys program
   MAP                    !Declare local procedures
     Func1(STRING),STRING    !Func1 is known only in both module
     MODULE(´Source2.clw´)
       HistOrd2            !HistOrd2 is known only in both modules
     END
   END

LocalData STRING(10)         !Declare data local to MEMBER module
```

```
HistOrd   PROCEDURE          !Declare order history procedure
HistData  STRING(10)         !Declare data local to PROCEDURE
  CODE
  LocalData = Func1(HistData)

Func1 FUNCTION(RecField)     !Declare local function
  CODE
    !Executable code statements


!Source2 module contains:
   MEMBER('OrderSys')      !Module belongs to the OrderSys program
   MAP                    !Declare local procedures
     HistOrd2             !HistOrd2 is known only in both modules
     MODULE('Source1.clw')
       Func1(STRING),STRING  !Func1 is known only in both modules
     END
   END

LocalData STRING(10)         !Declare data local to MEMBER module

HistOrd2 PROCEDURE          !Declare second order history procedure
  CODE
  LocalData = Func1(LocalData)
```

See Also:

MODULE

PROCEDURE

FUNCTION

Data Declarations and Memory Allocation

MAP (declare PROCEDURE and/or FUNCTION prototypes)

```
MAP
  prototypes
  [MODULE( )
    prototypes
   END ]
END
```

MAP Contains the *prototypes* which declare the functions, procedures and external source modules used in a PROGRAM or MEMBER module.

prototypes Declare a PROCEDURE or FUNCTION.

MODULE Declare a member source module.

A **MAP** structure contains the *prototypes* which declare the functions, procedures and external source modules used in a PROGRAM or MEMBER module. A MAP declared in the PROGRAM source module declares PROCEDUREs or FUNCTIONs that are available throughout the program. A MAP in a MEMBER module declares PROCEDUREs or FUNCTIONs that are available in that MEMBER module only.

The BUILTINS.CLW file is automatically included in your PROGRAM MAP structure by the compiler. This file contains prototypes of procedures and functions in the Clarion internal library that are available as part of the Clarion language. This file is provided because the compiler does not have the prototypes of these procedures and functions built into it.

Example:

```
!One file contains:
  PROGRAM             !Sample program in sample.cla
  MAP                !Begin map declaration
    LoadIt           ! LoadIt procedure
  END                !End of map

!A separate file contains:
 MEMBER('Sample')     !Declare MEMBER module
 MAP                !Begin local map declaration
  ComputeIt           ! compute it procedure
 END                !End of map
```

See Also:

PROGRAM

MEMBER

MODULE

FUNCTION and PROCEDURE Prototypes

MODULE (specify MEMBER source file)

MODULE(*sourcefile***)**
procedure prototype
function prototype
END

MODULE Names a MEMBER module or external library file.

sourcefile A string constant. If the sourcefile contains Clarion language source code, this specifies the filename (without extension) of the source file which contains the PROCEDUREs and/or FUNCTIONs. If the sourcefile is an external library, this string may contain any unique identifier.

procedure prototype The prototype of a PROCEDURE contained in the sourcefile.

function prototype The prototype of a FUNCTION contained in the source***file.***

A **MODULE** structure names a MEMBER module or external library file. It contains the *prototypes* for the PROCEDUREs and FUNCTIONs contained in the *sourcefile*. A MODULE structure can only be declared within a MAP structure.

Example:

```
!The "sample.cla" file contains:
  PROGRAM                 !Sample program in sample.cla
  MAP                    !Begin map declaration
   MODULE(´Loadit´)         ! source module loadit.cla
     LoadIt                !   load it procedure
    END                    ! end module
   MODULE(´Compute´)       ! source module compute.cla
     ComputeIt             !   compute it procedure
    END                    ! end module
  END                      !End map

!The "loadit.cla" file contains:
  MEMBER(´Sample´)        !Declare MEMBER module
  MAP                     !Begin local map declaration
   MODULE(´Process´)      ! source module process.cla
     ProcessIt            !   process it procedure
    END                    ! end module
  END                      !End map
```

See Also:

MEMBER

MAP

FUNCTION and PROCEDURE Prototypes

PROCEDURE (declare a procedure)

label **PROCEDURE** [(*parameter list*)]
local data
CODE
statements
[**RETURN**]

PROCEDURE Begins a section of source code that can be executed from within a PROGRAM.

label Names the PROCEDURE.

parameter list An optional list of variables which pass values to the PROCEDURE. This list defines the name of each parameter as used within the PROCEDURE´s source code. Each parameter is separated by a comma. The data type of each parameter is specified in the procedure´s prototype in the MAP structure.

local data Declare Local data which may be referenced only by this procedure.

CODE Begin executable statements.

statements Executable program instructions.

RETURN Terminate procedure execution. Return to the point from which the procedure was called.

PROCEDURE begins a section of source code that can be executed from within a PROGRAM. It is called by naming the PROCEDURE *label* (with its *parameter list*, if any) as an executable statement in the code section of a PROGRAM, PROCEDURE, or FUNCTION.

A PROCEDURE terminates and returns to its caller when a RETURN statement is executed. An implicit RETURN occurs at the end of the executable code. The end of executable code for the PROCEDURE is defined as the end of the source file, or the first encounter of a FUNCTION, ROUTINE, or another PROCEDURE.

Data declared within a PROCEDURE, between the keywords PROCEDURE and CODE, is Procedure Local data that can only be accessed by that PROCEDURE (unless passed as a parameter to another PROCEDURE or FUNCTION). This data is allocated memory on the stack upon entering the procedure, and de-allocated when it terminates.

A PROCEDURE must be declared in the MAP of a PROGRAM or MEMBER module. If declared in the PROGRAM MAP, it is available to any other procedure or function in the program. If declared in a MEMBER MAP, it is only available to other procedures or functions in that MEMBER module.

Example:

```
  PROGRAM              !Example program code
  MAP
   OpenFile(FILE)       !Procedure prototype with parameter
   ShoTime           !Procedure prototype without parameter
  END
  CODE
  OpenFile(FileOne)     !Call procedure to open file
  ShoTime              !Call ShoTime procedure
   !More executable statements

OpenFile PROCEDURE(AnyFile)    !Open any file
  CODE                        !Begin code section
  OPEN(AnyFile)                !Open the file
  IF ERRORCODE() = 2           !If file not found
    CREATE(AnyFile)              ! create it
```

```
  END
  RETURN                      !Return to caller

ShoTime  PROCEDURE              !Show time
Time   LONG                   !Local variable
Window WINDOW,CENTER
         STRING(@T3),USE(Time),AT(34,70)
         BUTTON(´Exit´),AT(138,92,32,16),USE(?Exit)
       END
  CODE                        !Begin executable code section
  Time = CLOCK()               !Get time from system
  OPEN(Window)
  ACCEPT
    CASE ACCEPTED()
    OF ?Exit
      BREAK
    END
  END
  RETURN                      !Return to caller
```

See Also:

FUNCTION and PROCEDURE Prototypes

Data Declarations and Memory Allocation

FUNCTION (declare a function)

label **FUNCTION** [(*parameter list*)]
local data
CODE
statements
RETURN(*value*)

FUNCTION	Begins a section of source code that can be executed from within a PROGRAM.
***l**abel*	Names the FUNCTION.
parameter list	An optional list of variables which pass values to the FUNCTION. This list defines the name of each parameter as used within the FUNCTION´s source code. Each parameter is separated by a comma. The data type of each parameter is specified in the procedure´s prototype in the MAP structure.
local data	Declare Local data which may be referenced only by this function.
CODE	Begin executable statements.
statements	Executable program instructions.
RETURN	Terminate function execution and return the value to the expression in which the function was used.
value	A numeric or string constant or variable which specifies the result of the function call.

FUNCTION begins a section of source code that can be executed by naming the FUNCTION label with its *parameter list* (empty parentheses are required if no parameters are passed). FUNCTION execution is terminated by a RETURN statement in its CODE section (required).

A function can be used as an expression component, or a parameter of a PROCEDURE or another FUNCTION. A FUNCTION may also be called in the same manner as a PROCEDURE, if the program logic does not require the RETURN *value*. In this case, the compiler will generate a warning which may be safely ignored.

Data declared within a FUNCTION, between the keywords FUNCTION and CODE, is Procedure Local data that can only be accessed by that FUNCTION (unless passed as a parameter to another PROCEDURE or FUNCTION). This data is allocated memory on the stack upon entering the function, and de-allocated when it terminates.

A FUNCTION must be declared in the MAP of a PROGRAM or MEMBER module. If declared in the PROGRAM MAP, it is available to any other procedure or function in the program. If declared in a MEMBER MAP, it is only available to other procedures or functions in the MEMBER module.

Example:

```
  PROGRAM
  MAP
    FullName(STRING,STRING,STRING),STRING   !Function prototype with parameters
    DayString,STRING                    !Function prototype without parameters
  END
TodayString STRING(9)
  CODE
  TodayString = DayString()               !Function call without parameters
                                    ! the () is required for a function
    !Global executable statements

  START(NewThread)                     !Clarion START function called as a
```

```
                                    ! procedure -- generates compiler warning
                                    ! but executes correctly

FullName FUNCTION(Last,First,Init)       !Full name function
  CODE                              !Begin executable code section
  IF Init = ''                       !If no middle initial
    RETURN(CLIP(First) & ' ' & Last)     ! return full name
  ELSE                              !Otherwise
    RETURN(CLIP(First) & ' ' & Init & '. ' & Last)
                                    ! return full name
  END

DayString FUNCTION                   !Day string function
  CODE                               !Begin executable code section
  Day# = (TODAY() % 7) + 1            !Find day of week from system date
  EXECUTE Day#                       !Execute, return day string
    RETURN('Sunday')
    RETURN('Monday')
    RETURN('Tuesday')
    RETURN('Wednesday')
    RETURN('Thursday')
    RETURN('Friday')
    RETURN('Saturday')
  END
```

See Also:

FUNCTION and PROCEDURE Prototypes

CODE (begin executable statements)

CODE

The **CODE** statement separates the data declaration section from the executable statement section within a PROGRAM, PROCEDURE, or FUNCTION. The first statement executed in a PROGRAM, PROCEDURE or FUNCTION is the statement following CODE.

Example:

```
OrdList  PROCEDURE                !Declare a procedure
!Data declarations go here
  CODE                          !This is the beginning of the "code" section
  !Executable statements go here
```

See Also:

PROGRAM

PROCEDURE

FUNCTION

ROUTINE (declare local subroutine)

label **ROUTINE**

ROUTINE Declares the beginning of a local subroutine of executable statements.

label The name of the ROUTINE.

ROUTINE declares the beginning of a local subroutine of executable statements. It is local to the PROCEDURE or FUNCTION in which it is written and must be at the end of the CODE section of the PROCEDURE or FUNCTION to which it belongs. All variables visible to the PROCEDURE or FUNCTION are available in the ROUTINE. This includes all Procedure Local, Module Local, and Global data.

A ROUTINE is called by the DO statement followed by the label of the ROUTINE. Program control following execution of a ROUTINE is returned to the statement following the calling DO statement. A ROUTINE is terminated by the end of the source module, or by another ROUTINE, PROCEDURE, or FUNCTION. The EXIT statement can also be used to terminate execution of a ROUTINE´s code (similar to RETURN in a PROCEDURE).

A ROUTINE is internally implemented by the compiler as a local procedure. Therefore, there are some efficiency issues that are not immediately obvious:

- DO and EXIT statements are very efficient.
- Accessing the PROCEDURE´s local data is less efficent than accessing module data.
- Implicit variables used only within the ROUTINE are less efficient than using local variables.
- Each RETURN statement within a ROUTINE incurs a 40-byte overhead.

Example:

```
SomeProc PROCEDURE
  CODE
  !Code statements
  DO Tally                !Call the routine
  !More code statements
Tally  ROUTINE               !Begin routine, end procedure
  IF CountVar < 55          !If less than 55
    CountVar += 1             ! increment counter
  ELSE                      ! otherwise
    CountVar = 0             ! reset the counter
    EXIT                    ! and exit the routine
  END                       !End if
```

See Also:

EXIT

DO (call a ROUTINE)

END (terminate a structure)

END

END terminates a data declaration structure or a compound executable statement. It is functionally equivalent to a period (.).

Example:

```
Customer FILE,DRIVER(´Clarion´)   !Declare a file
          RECORD                 ! begin record declaration
Name        STRING(20)
Number      LONG
          END                    ! end record declaration
         END                     !End file declaration

  CODE

  IF Number <> SavNumber           !Begin if structure
    DO GetNumber
  END                             !End if structure

  CASE Action                      !Begin case structure
  OF 1
    DO AddRec
  OF 2
    DO ChgRec
  OF 3
    DO DelRec
  END                             !End case structure
```

Statement Execution Sequence

In the CODE section of a Clarion program, statements are normally executed line-by-line, in the sequence in which they appear in the source module. Control statements, procedure calls, and function calls are used to modify this execution sequence.

PROCEDURE calls modify the execution sequence by branching to the called procedure and executing the code contained in it. Control returns to the executable statement following the procedure call when a RETURN statement is executed in the called procedure, or there are no more statements in the called procedure to execute.

FUNCTION calls modify the execution sequence by branching to the called function and executing the code contained in it. Control returns to the executable statement containing the function call when a RETURN statement is executed in the called function, returning the value of the function.

Control structures--IF , CASE, LOOP, and EXECUTE--change the execution sequence by evaluating expressions. When the expression is evaluated, the control structure conditionally executes statements contained within the structure.

Branching also occurs with the GOTO, DO, CYCLE, BREAK, EXIT, RETURN, and RESTART statements. These statements immediately and unconditionally alter the normal execution sequence.

The START function begins a new execution thread, unconditionally branching to that thread. However, the user may choose to activate another thread by clicking the mouse on the other thread´s active window.

Example:

```
  PROGRAM

  MAP
   ComputeTime(*GROUP)      !Passing a group parameter
   MatchMaster             !Passing no parameters
  END

ParmGroup GROUP             !Declare a group
FieldOne  STRING(10)
FieldTwo  LONG
         END
  CODE                    !Begin executable code
  FieldTwo = CLOCK()       !Executes 1st
  ComputeTime(ParmGroup)    !Executes 2nd, passes control to procedure
  MatchMaster              !Executes after procedure executes fully
```

PROCEDURE and FUNCTION Calls

procname[(*parameters*)]
return = *funcname*[(*parameters*)]

---

procname	The name of the PROCEDURE as declared in the procedure´s prototype in the MAP. If this is not the label of a PROCEDURE statement, compiler errors are issued.
parameters	An optional parameter list passed to the PROCEDURE or FUNCTION. A parameter list may be one or more variable labels or expressions. The *parameters* are separated by commas and are declared in the prototype in the MAP.
return	The label of a variable to receive the value returned by the FUNCTION.
funcname	The name of the FUNCTION as declared in the procedure´s prototype in the MAP. If this is not the label of a FUNCTION statement, compiler errors are issued.

A PROCEDURE is called by its label (including any parameter list) as a statement in the CODE section of a PROGRAM, PROCEDURE, or FUNCTION. The parameter list must match the parameter list declared in the procedure´s prototype in the MAP. Procedures cannot be called in expressions.

A FUNCTION is called by its label (including any parameter list) as a component of an expression or parameter list passed to another PROCEDURE or FUNCTION. The parameter list must match the parameter list declared in the function´s prototype in the MAP. A FUNCTION may also be called by its label (including any parameter list), in the same manner as a PROCEDURE, if its return value is not needed. This will generate a compiler warning that can be safely ignored.

Example:

```
  PROGRAM
  MAP
   ComputeTime(*GROUP)        !Passing a group parameter
   MatchMaster(),BYTE         !FUNCTION passing no parameters
  END
ParmGroup GROUP               !Declare a group
FieldOne  STRING(10)
FieldTwo  LONG
        END
  CODE
  FieldTwo = CLOCK()         !Built-in function called as expression
  ComputeTime(ParmGroup)      !Call the compute time procedure
  MatchMaster()              !Call the function as a procedure
```

See Also:

FUNCTION and PROCEDURE Prototypes

Procedure Prototyping

FUNCTION and PROCEDURE Prototypes

name[(*parameter list*)] [,*return type*] [,*calling convention*] [, **RAW**] [, **NAME( )**] [, **TYPE**] [, **DLL**] [, **PROC**][, **PRIVATE**]

---

name	The label of a PROCEDURE or FUNCTION statement.
parameter list	The data types of the parameters. Each parameter's data type may be followed by a label used to document the parameter (only). Each parameter may also include an assignment of the default value (a constant) to pass if the parameter is omitted.
return type	The data type the FUNCTION will RETURN**.**
calling convention	Specify the C or PASCAL stack-based parameter calling convention.
RAW	Specifies that STRING or GROUP parameters pass only the memory address (without passing the length of the passed string).
NAME	Specify an alternate, "external" name for the PROCEDURE or FUNCTION.
TYPE	Specify the prototype is a type definition for procedures passed as parameters.
DLL	Specify the PROCEDURE or FUNCTION is in a .DLL.
PROC	Specify the FUNCTION may be called as a PROCEDURE without generating a compiler warning.
PRIVATE	Specify the PROCEDURE or FUNCTION may be called only from another PROCEDURE or FUNCTION within the same MODULE.

All PROCEDUREs and FUNCTIONs in a PROGRAM must be declared as a prototype in a MAP. A prototype is defined as the *name* of the PROCEDURE or FUNCTION, an optional *parameter list*, and the data *return type* (if prototyping a FUNCTION). You may specify the parameter *calling convention*, if you are linking in objects that require stack-based parameter passing (such as objects that were not compiled with a Clarion TopSpeed compiler).

The optional *parameter list* is a list of the data types that are passed to the PROCEDURE or FUNCTION. Each passed parameter in the *parameter list* is delimited by commas, and the entire *parameter list* is enclosed in the parentheses following the *name*.

In the *parameter list*, each parameter's data type may be followed by a valid Clarion label which is completely ignored by the compiler (used only to document the purpose of the parameter). Each passed parameter's definition may also include the assignment of a constant value to the data type (or the documentary label, if present) that defines the default value to pass if the parameter is omitted.

Any parameter that may be omitted when the PROCEDURE or FUNCTION is called must be included in the prototype's *parameter list* and enclosed in angle brackets (< >) unless a default value is defined for the parameter. The OMITTED function allows you to test for unpassed paramters at runtime (except those parameters which have a default value defined).

You can optionally specify the C (right to left) or PASCAL (left to right and compatible with Win95 in 32-bit) stack-based parameter *calling convention* for your PROCEDURE or FUNCTION. This provides compatibility with third-party libraries written in other languages (if they were not compiled with a TopSpeed compiler). If you do not specify a *calling convention*, the default is the internal, register-based parameter passing convention used by all the TopSpeed compilers.

The RAW attribute allows you to pass just the memory address of a *?, STRING, or GROUP parameter (whether passed by value or by reference) to a non-Clarion language procedure or function. Normally,

STRING or GROUP parameters pass both the address and the length of the string. The RAW attribute eliminates the length portion. This is provided for compatibility with external library functions which expect only the address of the string.

The NAME attribute provides the linker an external name for the PROCEDURE or FUNCTION. This is also provided for compatibility with libraries written in other languages. For example: in some C language compilers, with the C calling convention specified, the compiler adds a leading underscore to the function name. The NAME attribute allows the linker to resolve the name of the function correctly.

The TYPE attribute indicates the prototype does not reference a specific PROCEDURE or FUNCTION. Instead, it defines a prototype *name* used in other prototypes to indicate the type of procedure passed to another PROCEDURE or FUNCTION as a parameter.

The DLL attribute specifies that the PROCEDURE or FUNCTION for prototype on which it is placed is in a .DLL. The DLL attribute is required for 32-bit applications because .DLLs are relocatable in a 32-bit flat address space, which requires one extra dereference by the compiler to address the procedure.

The PRIVATE attribute specifies that only another PROCEDURE or FUNCTION that is in the same MODULE may call it. This would most commonly be used on a prototype in a module's MAP structure, but may also be used in the global MAP.

When the *name* of a prototype is used in the *parameter list* of another prototype, it indicates the procedure being prototyped will receive the label of a PROCEDURE or FUNCTION that receives the same *parameter list* (and has the same *return type*, if it is a FUNCTION). A prototype with the TYPE attribute may not also have the NAME attribute.

Example:

```
MAP
  MODULE('Test')            !'test.clw' contains these procedure and functions
    MyProc1(LONG)           !LONG value-parameter
    MyProc2(<*LONG>)        !Omittable LONG variable-parameter
    MyProc3(LONG=23)        !Passes 23 if omitted
    MyProc4(LONG Count, REAL Sum)   !LONG passing a Count and REAL passing a Sum
    MyProc5(LONG Count=1, REAL Sum=0)!Count defaults to 1 and Sum to 0
    MyFunc1(*SREAL),REAL,C   !SREAL variable-parameter, REAL return, C call conv
    MyFunc2(FILE),STRING     !FILE entity-parameter, returning a STRING
    ProcType(FILE),TYPE      !Procedure-parameter type definition
    MyFunc3(ProcType),STRING
                            !ProcType procedure-parameter, returning a STRING,
                            ! must be passed a procedure that takes a FILE
                            ! as a parameter
    MyFunc4(FILE),STRING,PROC     !May be called as a procedure without warnings
    MyProc6(FILE),PRIVATE    !May only be called by other procs in TEST.CLW
  END
  MODULE('Party3.Obj')       !A third-party library
    Func46(*CSTRING),REAL,C,RAW
                            !Pass CSTRING address-only to C function
    Func47(*CSTRING),*CSTRING,C,RAW
                            !Returns pointer to a CSTRING
    Func48(REAL),REAL,PASCAL
                            !PASCAL calling convention
    Func49(SREAL),REAL,C,NAME('_func49')
                            !C convention and external function name
  END
  MODULE('STDFuncs.DLL')       !A standard functions .DLL
    Func50(SREAL),REAL,PASCAL,DLL
  END
END
```

See Also:

MAP

MEMBER

MODULE

NAME

PROCEDURE

FUNCTION

RETURN

Parameter Passing

DLL (set procedure defined externally in .DLL)

DLL([*flag*] **)**

DLL	Declares a PROCEDURE or FUNCTION defined externally in a .DLL.
flag	A numeric constant, equate, or Project system define which specifies the attribute as active or not. If the *flag* is zero, the attribute is not active, just as if it were not present. If the *flag* is any value other than zero, the attribute is active.

The **DLL** attribute specifies that the PROCEDURE or FUNCTION on whose prototype it is placed is defined in a .DLL. The DLL attribute is required for 32-bit applications because .DLLs are relocatable in a 32-bit flat address space, which requires one extra dereference by the compiler to address the procedure.

Example:

```
MAP
  MODULE('STDFuncs.DLL')       !A standard functions .DLL
    Func50(SREAL),REAL,PASCAL,DLL
  END
END
```

FUNCTION Return Types

A FUNCTION must RETURN a value. The data type to be returned is listed, separated by a comma, after the optional parameter list. Valid RETURN types are:

```
BYTE      SHORT     USHORT     LONG      ULONG     SREAL       REAL    DATE
TIME      STRING    CSTRING    *BYTE     *SHORT    *USHORT     *LONG
*ULONG    *SREAL    *REAL      *DATE     *TIME     *CSTRING
Untyped value-parameter return value (?)
```

An untyped value-parameter return value (?) indicates the data type of the value returned by the FUNCTION is not known. This functions in exactly the same manner as an untyped value-parameter. When the value is returned from the FUNCTION, standard Clarion Data Conversion Rules apply, no matter what data type is returned.

Functions which return pointers (the address of some data) should be prototyped with an asterisk prepended to the return data type. This is provided just for compatibility with external library functions (written in another language) which return only the address of data. The compiler automatically handles the returned pointer at runtime. Functions prototyped this way act just like a variable defined in the program--when the function is used in Clarion code, the data referenced by the returned pointer is automatically used. This data can be assigned to other variables, passed as parameters to procedures or functions, or the ADDRESS function may return the address of the data.

As an example of this, assume that the XYZ() function returns *CSTRING (pointer to a CSTRING), CStringVar is a CSTRING variable, and LongVar is a LONG variable. The simple Clarion assignment statement, CStringVar = XYZ(), places the data referenced by the XYZ() function´s returned pointer, in the CStringVar variable. The assignment, LongVar = ADDRESS(XYZ()), places the memory address of that data in the LongVar variable.

Example:

```
MAP
  MODULE(´Party3.Obj´)      !A third-party library
   Func46(*CSTRING),REAL,C,RAW
                            !Pass CSTRING address-only to C function
   Func47(*CSTRING),*CSTRING,C,RAW
                            !Returns pointer to a CSTRING
   Func48(REAL),REAL,PASCAL
                            !PASCAL calling convention
   Func49(SREAL),REAL,C,NAME(´_func49´)
                            !C convention and external function name
  END
END
```

See Also:

MAP

MEMBER

MODULE

NAME

FUNCTION

RETURN

RAW (pass address only)

RAW

The **RAW** attribute of a PROCEDURE or FUNCTION prototype specifies that STRING or GROUP parameters pass the memory address only. This allows you to pass just the memory address of a *?, STRING, or GROUP parameter, whether passed by value or by reference, to a non-Clarion language procedure or function. Normally, STRING or GROUP parameters pass the address and the length of the string. The RAW attribute eliminates the length portion. This is provided for compatibility with external library functions which expect only the address of the string.

Example:

```
MAP
  MODULE('Party3.Obj')      !A third-party library
   Func46(*CSTRING),REAL,C,RAW
                           !Pass CSTRING address-only to C function
  END
END
```

See Also:

FUNCTION and PROCEDURE Prototypes

Passing Parameters

NAME (set prototype's external name)

NAME(*constant* **)**

NAME Specifies an "external" name for the linker.

constant A string constant.

The **NAME** attribute specifies an "external" name for the linker. The NAME attribute may be placed on a FUNCTION or PROCEDURE Prototype. The *constant* supplies the external name used by the linker to identify the procedure or function from an external library.

Example:

```
PROGRAM
MAP
 MODULE('External.Obj')
  AddCount(LONG),LONG,C,NAME('_AddCount')   !C function named '_AddCount'
. .
```

See Also:

FUNCTION and PROCEDURE Prototypes

TYPE (specify procedure or function type defintion)

TYPE

The **TYPE** attribute specifies a prototype that does not reference an actual PROCEDURE or FUNCTION. Instead, it defines a prototype *name* to use in other prototypes to indicate the type of procedure passed to another PROCEDURE or FUNCTION as a parameter.

When the *name* of the TYPEd prototype is used in the *parameter list* of another prototype, the procedure being prototyped will receive, as a passed parameter, the label of a PROCEDURE or FUNCTION that has the same type of *parameter list* (and has the same *return type*, if it is a FUNCTION).

Example:

```
MAP
   ProcType(FILE),TYPE      !Procedure-parameter type definition
   MyFunc3(ProcType),STRING
                        !ProcType procedure-parameter, returning a STRING,
                  ! must be passed the label of a procedure that takes
                  ! a FILE as a required parameter
END
```

See Also:

FUNCTION and PROCEDURE Prototypes

PROC (set function called as procedure without warnings)

PROC

The **PROC** attribute specifies that the FUNCTION on whose prototype it is placed may be called as a PROCEDURE without generating compiler warnings. This allows you to use a FUNCTION as a PROCEDURE in those instances in which you do not need the return value from the FUNCTION.

You can now call a function as a procedure without generating a compiler warning if you prototype it with the PROC attribute. The built in functions which are commonly called as procedures have been prototyped this way.

This means statements which were written such as:

```
IF MESSAGE('Error Message Text').
```

can now be written:

```
MESSAGE('Error Message Text')
```

without a compiler warning.

The following built in functions are prototyped with the PROC attribute:

CALL()

COMMAND()

SEND()

MESSAGE()

START()

FILEDIALOG()

FONTDIALOG()

COLORDIALOG()

PRINTERDIALOG()

Example:

```
MAP
  MODULE('STDFuncs.DLL')       !A standard functions .DLL
    Func50(SREAL),REAL,PASCAL,PROC
  END
END
```

PRIVATE (set procedure private to a single module)

PRIVATE

The **PRIVATE** attribute specifies that the PROCEDURE or FUNCTION on whose prototype it is placed may be called only from a PROCEDURE or FUNCTION within the same source MODULE. This encapsulates it from other modules.

Example:

```
MAP
  MODULE('STDFuncs.DLL')       !A standard functions .DLL
    Func49(SREAL),REAL,PASCAL,PROC
    Proc50(SREAL),PRIVATE   !Callable only from Func49
  END
END
```

Parameter Passing

Parameter Types

There are four types of parameters that may be passed to a PROCEDURE or FUNCTION: **value-parameters**, **variable-parameters**, **entity-parameters**, and **procedure-parameters**.

Value-parameters are "passed by value." A copy of the variable passed in the parameter list of the "calling" PROCEDURE or FUNCTION is used in the "called" PROCEDURE or FUNCTION. The "called" PROCEDURE or FUNCTION cannot change the value of the variable passed to it by the "caller." Value-parameters are listed by data type in the PROCEDURE or FUNCTION prototype in the MAP. Valid value-parameters are:

BYTE SHORT USHORT LONG ULONG SREAL REAL BFLOAT4
BFLOAT8 DATE TIME STRING

Variable-parameters are "passed by address." A variable passed by address has only one memory address. Changing the value of the variable in the "called" PROCEDURE or FUNCTION also changes its value in the "caller." Variable-parameters are listed by data type with a leading asterisk (*) in the PROCEDURE or FUNCTION prototype in the MAP. Valid variable-parameters are:

*BYTE *SHORT *USHORT *LONG *ULONG *SREAL *REAL
*BFLOAT4 *BFLOAT8 *DECIMAL *PDECIMAL *DATE *TIME
*STRING *PSTRING *CSTRING *GROUP

Entity-parameters pass the name of a data structure to the "called" PROCEDURE or FUNCTION. Passing the entity allows the "called" PROCEDURE or FUNCTION to use those Clarion commands that require the label of the structure as a parameter. Entity-parameters are listed by entity type in the PROCEDURE or FUNCTION prototype in the MAP. Entity-parameters are always "passed by address." Valid entity-parameters are:

FILE VIEW KEY INDEX QUEUE APPLICATION WINDOW
REPORT

Procedure-parameters pass the name of another PROCEDURE or FUNCTION to the "called" PROCEDURE or FUNCTION. Procedure-parameters are listed by the name of a preceding prototype of the same type in the PROCEDURE or FUNCTION prototype in the MAP (which may or may not have the TYPE attribute). When called in executable code, the "called" PROCEDURE or FUNCTION must be passed the name of a PROCEDURE or FUNCTION whose prototype is exactly the same as the procedure named in the "called" procedure´s prototype.

Example:

```
MAP
 MODULE(´Test´)            !´test.clw´ contains these procedure and functions
   MyProc1(LONG)           !LONG value-parameter
   MyProc2(<*LONG>)        !Omittable LONG variable-parameter
   MyFunc1(*SREAL),REAL,C   !SREAL variable-parameter, REAL return, C call conv
   MyFunc2(FILE),STRING     !FILE entity-parameter, returning a STRING
   ProcType(FILE),TYPE      !Procedure-parameter type definition
   MyFunc3(ProcType),STRING
                          !ProcType procedure-parameter, returning a STRING,
                          ! must be passed a procedure that takes a FILE
                          ! as a parameter
 END
 MODULE(´Party3.Obj´)      !A third-party library
  Func46(*CSTRING),REAL,C,RAW
                          !Pass CSTRING address-only to C function
  Func47(*CSTRING),*CSTRING,C,RAW
```

```
                        !Returns pointer to a CSTRING
   Func48(REAL),REAL,PASCAL
                        !PASCAL calling convention
   Func49(SREAL),REAL,C,NAME(´_func49´)
                        !C convention and external function name
  END
END
```

See Also:

MAP

MEMBER

MODULE

NAME

PROCEDURE

FUNCTION

RETURN

Passing Parameters of Unspecified Data Type

The desire to write general purpose functions which perform some operation on a passed parameter, where the exact data type of the parameter may vary from one call to the next, is fairly common. Therefore, the function´s prototype must indicate that the data type of the parameter is unknown at compile time. The Clarion language allows for this with **untyped value-parameters** and **untyped variable-parameters**. These are polymorphic parameters; they may become any other data type depending upon the data type passed to the procedure or function.

Untyped value-parameters are represented in the PROCEDURE or FUNCTION prototype with a question mark (?). When the procedure executes, the parameter is dynamically typed and acts as a data object of the base type (LONG, STRING, or REAL) of the passed variable, or the base type of whatever it was last assigned. This means that the "assumed" data type of the parameter can change within the PROCEDURE or FUNCTION, allowing it to be treated as any data type.

An untyped value-parameter is "passed by value" to the PROCEDURE or FUNCTION and its assumed data type is handled by Clarion´s automatic Data Conversion Rules. Any changes made to the passed parameter within the PROCEDURE or FUNCTION do not affect the variable which was passed in.

Data types which may be passed as untyped value-parameters:

BYTE SHORT USHORT LONG ULONG SREAL REAL BFLOAT4
BFLOAT8 DECIMAL PDECIMAL DATE TIME STRING PSTRING
CSTRING GROUP (treated as a STRING) Untyped value-parameter (?)
Untyped Variable-parameter (*?)

Untyped variable-parameters are represented in the PROCEDURE or FUNCTION prototype with an asterisk and a question mark (*?). Inside the procedure, the parameter acts as a data object of the type of the variable passed in at runtime. This means the data type of the parameter is fixed during the execution of the PROCEDURE or FUNCTION.

An untyped variable-parameter is "passed by address" to the PROCEDURE or FUNCTION. Therefore, any changes made to the passed parameter within the PROCEDURE or FUNCTION are made directly to the variable which was passed in. This allows you to write polymorphic functions.

Within a PROCEDURE or FUNCTION which receives an untyped variable-parameter, it is not safe to make any assumptions about the data type coming in. The danger of making assumptions is the possiblity of assigning an out-of-range value which the variable´s actual data type cannot handle. If this happens, the result may be disastrously different from that expected.

Data types which may be passed as untyped variable-parameters:

BYTE SHORT USHORT LONG ULONG SREAL REAL BFLOAT4
BFLOAT8 DECIMAL PDECIMAL DATE TIME STRING PSTRING
CSTRING Untyped variable-parameter (*?)

The RAW attribute can be specified if the untyped variable-parameter (*?) is being passed to external library functions written in other languages than Clarion. This has the same effect as passing a C or C++ void * parameter.

Arrays may not be passed as either kind of untyped parameter.

Example:

```
PROGRAM
MAP
```

```
   Proc1(?)               !Untyped value-parameter
   Proc2(*?)              !Untyped variable-parameter
   Proc3(*?)              !Untyped variable-parameter (set to crash)
   Max(?,?),?             !Function returning Untyped value-parameter
  END
GlobalVar1 BYTE(3)        !BYTE initialized to 3
GlobalVar2 DECIMAL(8,2,3)
GlobalVar3 DECIMAL(8,1,3)
MaxInteger LONG
MaxString STRING(255)
MaxFloat REAL
  CODE
  Proc1(GlobalVar1)         !Pass in a BYTE, value is 3
  Proc2(GlobalVar2)         !Pass it a DECIMAL(8,2), value is 3.00 - it prints 3.33
  Proc2(GlobalVar3)         !Pass it a DECIMAL(8,1), value is 3.0 - it prints 3.3
  Proc3(GlobalVar1)         !Pass it a BYTE and watch it crash
  MaxInteger = Max(1,5)     !Max function returns the 5
  MaxString = Max('Z','A')   !Max function returns the 'Z'
  MaxFloat = Max(1.3,1.25) !Max function returns the 1.3
Proc1 PROCEDURE(ValueParm)
  CODE                             ! ValueParm starts at 3 and is a LONG
  ValueParm = ValueParm & ValueParm    !Now Contains '33' and is a STRING
  ValueParm = ValueParm / 10          !Now Contains 3.3 and is a REAL
Proc2 PROCEDURE(VariableParm)
  CODE
  VariableParm = 10 / 3               !Assign 3.33333333... to passed variable
Proc3 PROCEDURE(VariableParm)
  CODE
  LOOP
    IF VariableParm > 250 THEN BREAK.  !If passed a BYTE, BREAK will never happen
    VariableParm += 10
  END
Max FUNCTION(Val1,Val1)              !Find the larger of two passed values
  CODE
  IF Val1 > Val2                    !Check first value against second
    RETURN(Val1)                    ! return first, if largest
  ELSE                             !otherwise
    RETURN(Val2)                    ! return the second
  END
```

See Also:

FUNCTION and PROCEDURE Prototypes

Passing Parameters

Data Conversion Rules

Passing GROUPs and QUEUEs as Parameters

Passing a GROUP or a QUEUE to a PROCEDURE or FUNCTION which has been prototyped with GROUP or QUEUE types in its *parameter list* does not allow you to reference the component fields within the structure in the receiving PROCEDURE or FUNCTION. However, you can place the label of a GROUP or QUEUE in the prototype's *parameter list* to pass it by address and allow references to the component fields.

The GROUP or QUEUE named in the *parameter list* does not need the TYPE attribute, and does not have to be declared before the MAP structure, but it must be declared before the PROCEDURE or FUNCTION that will receive the parameter is called. This is the only case in the Clarion language that allows such a "forward reference."

The PROCEDURE or FUNCTION statement for the prototype may declare the local name of the passed group with a prefix to prevent name clashes, owever this is unnecessary as long as you use the Field Qualification syntax to reference members of the passed group. The passed group can be a "superset" of the named paremeter, as long as the first fields in the "superset" group are the same as the named group.

Example:

```
          PROGRAM
          MAP
            MyProc1(PassGroup,NameQue)
                 !Receives a GROUP defined the same as PassGroup and a QUEUE
                 ! defined the same as NameQue
          END
PassGroup   GROUP,PRE(Pas),TYPE   !Type definition: GROUP with 2 STRING(20) fields
F1          STRING(20)
F2          STRING(20)
          END
NameGroup   GROUP,PRE(Nme)          !Name group
First         STRING(20)            ! first name
Last        STRING(20)            ! last name
Company       STRING(30)
          END
NameQue     QUEUE,PRE(Que)          !Name Queue
First         STRING(20)            ! first name
Last        STRING(20)            ! last name
          END
  CODE
  MyProc1(NameGroup,NameQue)        !Pass NameGroup and NameQue as parameters

MyProc1     PROCEDURE(LG:PassedGroup,PassedQue)
  CODE
  PassedQue:First = LG:F1           !Assign value in Nme:First to Que:First
  PassedQue:Last  = LG:F2           !Assign value in Nme:Last to Que:Last
  ADD(PassedQue)                  !Add an entry into NameQue
```

See Also:

FUNCTION and PROCEDURE Prototypes

GROUP

QUEUEField Qualification

Passing Arrays as Parameters

An array may be passed to a PROCEDURE or FUNCTION. The prototype in the MAP structure must declare the array´s data type as a variable-parameter ("passed by address") with an empty subscript list. If the array is more than one dimension, commas must be used as position holders to indicate the number of dimensions in the array.

The calling statement should pass the entire array to the PROCEDURE or FUNCTION, not just one element.

Example:

```
  PROGRAM
  MAP
    MainProc
   AddCount(*LONG[,],*LONG[,])       !Passing two two-dimensional long arrays
  END
  CODE
  MainProc                        !Call first procedure

MainProc PROCEDURE
TotalCount  LONG,DIM(10,10)
CurrentCnt  LONG,DIM(10,10)
  CODE
  AddCount(TotalCount,CurrentCnt)    !Call the procedure passing the arrays

AddCount PROCEDURE(Tot,Cur)        !Procedure expects two arrays
  CODE
  LOOP I# = 1 TO MAXIMUM(Tot,1)        !Loop through first subscript
    LOOP J# = 1 TO MAXIMUM(Tot,2)     !Loop through second subscript
      Tot[I#,J#] += Cur[I#,J#]       ! increment TotalCount from CurrentCnt
    END
  END
  CLEAR(Cur)                        !Clear CurrentCnt array
  RETURN
```

See Also:

DIM

FUNCTION and PROCEDURE Prototypes

MAXIMUM

Program Structure Compiler Directives

Compiler Directives are statements that tell the compiler to take some action at compile time. These statements are not included in the executable program object code which the compiler generates. Therefore, there is no run-time overhead associated with their use.

BEGIN (define code structure)

```
BEGIN
  statements
END
```

BEGIN Declares a single code statement structure.

statements Executable program instructions.

The **BEGIN** compiler directive tells the compiler to treat the *statements* as a single structure. The BEGIN structure must be terminated by a period or the END statement.

BEGIN is used in an EXECUTE control structure to allow several lines of code to be treated as one.

Example:

```
EXECUTE Value
  Proc1          !Execute if Value = 1
  BEGIN          !Execute if Value = 2
    Value += 1
    Proc2
  END
  Proc3          !Execute if Value = 3
END
```

See Also:

EXECUTE

COMPILE (specify source to be compiled)

COMPILE(*terminator* [,*expression*]**)**

COMPILE	Specifies a block of source code lines to be included in the compilation.
terminator	A string constant that marks the last line of a block of source code.
expression	An expression allowing conditional execution of the COMPILE. The expression is: EQUATE = integer.

The **COMPILE** directive specifies a block of source code lines to be included in the compilation. The included block begins with the COMPILE directive and ends with the line that contains the same string constant as the *terminator*. The entire terminating line is included in the COMPILE block.

The optional *expression* parameter permits conditional COMPILE. The form of the *expression* is fixed. It is the label of an EQUATE statement, or a Conditional Switch set in the Project System, followed by an equal sign (=), followed by an integer constant. The code between COMPILE and the *terminator* is compiled only if the *expression* is true. Although the *expression* is not required, COMPILE without an *expression* parameter is not necessary because all source code is compiled unless explicitly omitted. COMPILE and OMIT are opposites and may not be nested within each other, or themselves.

Example:

```
Demo EQUATE(1)                  !Specify the Demo EQUATE value
  CODE
  COMPILE('EndDemoChk',Demo = 1)    !COMPILE only if Demo equate is turned on
  DO DemoCheck                  !Check for demo limits passed
EndDemoChk                      !End of conditional COMPILE code
```

See Also:

OMIT

EQUATE

EJECT (start new listing page)

EJECT([*module subtitle*]**)**

EJECT Starts a new page in a Clarion listing.

module subtitle A string constant containing the subtitle to be printed. On the next page of the listing, the *module subtitle* is printed in the first column of the third line.

The **EJECT** directive starts a new page and an optional new *module subtitle* in a Clarion listing. If the *module subtitle* parameter is omitted, the subtitle set by a previous SUBTITLE or EJECT directive will be used on the next page.

Example:

```
EJECT(´File Declarations´)    !Start new page, new subtitle
```

INCLUDE (compile code in another file)

INCLUDE(*filename* [,*section*]**)**

INCLUDE Specifies source code to be compiled which exists in a separate file which is not a MEMBER module.

filename A string constant that contains the DOS file specification for a source file. If the extension is omitted, .CLA is assumed.

section A string constant which is the *string* parameter of the SECTION directive marking the beginning of the source code to be included.

The **INCLUDE** directive specifies source code to be compiled which exists in a separate file which is not a MEMBER module. Starting on the line of the INCLUDE directive, the source file, or the specified *section* of that file, is compiled as if it appeared in sequence within the source module being compiled.

The compiler uses the Redirection File (CW.RED) to find the file, searching the path specified for that type of *filename* (usually by extension). This makes it unnecessary to provide a complete path in the *filename* to be included. A discussion of the Redirection file is in the *User´s Guide*.

Example:

```
GenLedger PROCEDURE            !Declare procedure
  INCLUDE(´filedefs.clw´)      !Include file definitions here
   CODE                       !Begin code section
  INCLUDE(´Setups´,´ChkErr´)   !Include error check from setups.clw
```

OMIT (specify source not to be compiled)

OMIT(*terminator* [,*expression*]**)**

OMIT	Specifies a block of source code lines to be omitted from the compilation.
terminator	A string constant that marks the last line of a block of source code.
expression	An expression allowing conditional execution of the OMIT. The expression must be: EQUATE = integer.

The **OMIT** directive specifies a block of source code lines to be omitted from the compilation. These lines may contain source code comments or a section of code that has been "stubbed out" for testing purposes. The omitted block begins with the OMIT directive and ends with the line that contains the same string constant as the *terminator*. The entire terminating line is included in the OMIT block.

The optional *expression* parameter permits conditional OMIT. The form of the *expression* is fixed. It is the label of an EQUATE statement, or a Conditional Switch set in the Project System, followed by an equal sign (=), followed by an integer constant. The OMIT directive executes only if the *expression* is true.

COMPILE and OMIT are opposites and may not be nested within each other, or themselves.

Example:

```
  OMIT(´**END**´)    !Unconditional OMIT
************************************************
*
* Main Program Loop
*
************************************************
**END**
Demo EQUATE(0)               !Specify the Demo EQUATE value
  CODE
  OMIT(´EndDemoChk´,Demo = 0)    !OMIT only if Demo is turned off
  DO DemoCheck               !Check for demo limits passed
EndDemoChk                   !End of omitted code
```

See Also:

COMPILE

EQUATE

SECTION (specify source code section)

SECTION(*string*)

SECTION Identifies the beginning of a block of executable source code or data declarations.

string A string constant which names the SECTION.

The **SECTION** compiler directive identifies the beginning of a block of executable source code or data declarations which may be INCLUDEd in source code in another file. The SECTION´s *string* parameter is used as an optional parameter of the INCLUDE directive to include a specific block of source code. A SECTION is terminated by the next SECTION or the end of the file.

Example:

```
 SECTION(´FirstSection´)       !Begin section

FieldOne STRING(20)
FieldTwo LONG

 SECTION(´SecondSection´)        !End previous section, begin new section

  IF Number <> SavNumber
    DO GetNumber
  END

 SECTION(´ThirdSection´)       !End previous section, begin new section

  CASE Action
  OF 1
    DO AddRec
  OF 2
    DO ChgRec
  OF 3
    DO DelRec
  END                         !Third section ends at end of file
```

See Also:

INCLUDE

SUBTITLE (print MODULE subtitle)

SUBTITLE(*module subtitle***)**

SUBTITLE Declares a listing subtitle printed in the first column of the third line of a Clarion listing.

module subtitle A string constant containing the subtitle to be printed.

A **SUBTITLE** is printed in the first column of the third line of a Clarion listing. The SUBTITLE directive does not print in the listing. The SUBTITLE directive must be placed at the beginning of a source module prior to the PROGRAM or MEMBER declarations. The subtitle remains the same on every page of the listing unless it is changed by an EJECT directive.

Example:

```
SUBTITLE(´Global Data Declarations´)
```

TITLE (print MODULE title)

TITLE(*module title***)**

TITLE Declares a listing title printed in the first column of the first line of a Clarion listing.

module title A string constant containing the title to be printed.

A **TITLE** is printed in the first column of the first line of a Clarion listing. The TITLE directive does not print in the listing. The TITLE directive must be placed at the beginning of a source module prior to the PROGRAM or MEMBER declarations. The title remains the same on every page of the listing.

Example:

```
TITLE(´ORDERSYS - Order Entry System Listing´)
```

Declaring Variables

Variable Declaration Statements

BYTE (one-byte unsigned integer)

label **BYTE(***initial value***)** [,**DIM( )**] [,**OVER( )**] [,**NAME( )**] [,**EXTERNAL**] [,**DLL**][,**STATIC**] [,**THREAD**] [,**AUTO**]

BYTE A one-byte unsigned integer.

```
Format: magnitude
       | ....... |
Bits:    7      0
Range: 0 to 255
```

initial value A numeric constant. If omitted, the initial value is zero.

DIM Dimension the variable as an array.

OVER Share a memory location with another variable.

NAME Specify an alternate, "external" name for the field.

EXTERNAL Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.

DLL Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.

STATIC Specify the variable´s memory is permanently allocated.

THREAD Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data**.**

AUTO Specify the variable has no *initial value*.

BYTE declares a one-byte unsigned integer.

Example:

```
Count1 BYTE                    !Declare one byte integer
Count2 BYTE,OVER(Count1)          !Declare OVER the one byte integer
Count3 BYTE,DIM(4)              !Declare it an array of 4 bytes
Count4 BYTE(5)                  !Declare with initial value
Count5 BYTE,EXTERNAL            !Declare as external
Count6 BYTE,NAME(´SixCount´)      !Declare with external name
ExampleFile FILE,DRIVER(´Clarion´) !Declare a file
Record     RECORD
Count5      BYTE,NAME(´Counter´)  !Declare with external name
          . .
```

SHORT (two-byte signed integer)

label **SHORT**([*initial value*]) [,**DIM( )**] [,**OVER( )**] [,**NAME( )**] [,**EXTERNAL**] [,**DLL**] [,**STATIC**] [,**THREAD**] [,**AUTO**]

SHORT A two-byte signed integer.

```
Format:    ±    magnitude
        | . | ............... |
Bits:    15   14            0
Range: -32,768 to 32,767
```

***i**nitial value* A numeric constant. If omitted, the initial value is zero.

DIM Dimension the variable as an array.

OVER Share a memory location with another variable.

NAME Specify an alternate, "external" name for the field.

EXTERNAL Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.

DLL Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.

STATIC Specify the variable´s memory is permanently allocated.

THREAD Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data.

AUTO Specify the variable has no *initial value*.

SHORT declares a two-byte signed integer, using the Intel 8086 word integer format. The high-order bit of this configuration is the sign bit (0 = positive, 1 = negative). Negative values are represented in standard two´s complement notation.

Example:

```
Count1 SHORT                      !Declare two-byte signed integer
Count2 SHORT,OVER(Count1)         !Declare OVER the two-byte signed integer
Count3 SHORT,DIM(4)               !Declare it an array of 4 shorts
Count4 SHORT(5)                 !Declare with initial value
Count5 SHORT,EXTERNAL             !Declare as external
Count6 SHORT,NAME(´SixCount´)       !Declare with external name
ExampleFile  FILE,DRIVER(´Clarion´)   !Declare a file
Record     RECORD
Count7       SHORT,NAME(´Counter´) !Declare with external name
         . .
```

USHORT (two-byte unsigned integer)

label **USHORT**([*initial value*]) [,**DIM( )**] [,**OVER( )**] [,**NAME( )**] [,**EXTERNAL**] [,**DLL**] [,**STATIC**] [,**THREAD**] [,**AUTO**]

USHORT A two-bye unsigned integer.

```
Format:      magnitude
        | ............... |
Bits:    15              0
Range:  0 to 65,535
```

initial value A numeric constant. If omitted, the initial value is zero.

DIM Dimension the variable as an array.

OVER Share a memory location with another variable.

NAME Specify an alternate, "external" name for the field.

EXTERNAL Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.

DLL Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.

STATIC Specify the variable´s memory is permanently allocated.

THREAD Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data.

AUTO Specify the variable has no *initial value*.

USHORT declares a two-byte unsigned integer in the Intel 8086 word format. There is no sign bit in this configuration.

Example:

```
Count1 USHORT                      !Declare two-byte unsigned integer
Count2 USHORT,OVER(Count1)           !Declare OVER the two-byte unsigned integer
Count3 USHORT,DIM(4)               !Declare it an array of 4 unsigned shorts
Count4 USHORT(5)                   !Declare with initial value
Count5 USHORT,EXTERNAL               !Declare as external
Count6 USHORT,NAME(´SixCount´)       !Declare with external name
ExampleFile FILE,DRIVER(´Btrieve´)     !Declare a file
Record      RECORD
Count7        USHORT,NAME(´Counter´)   !Declare with external name
          . .
```

LONG (four-byte signed integer)

label **LONG**([*initial value*]) [,**DIM( )**] [,**OVER( )**] [,**NAME( )**] [,**EXTERNAL**] [,**DLL**] [,**STATIC**] [,**THREAD**] [,**AUTO**]

LONG A four-byte unsigned integer.

```
Format:    ±          magnitude
         | . | ............................ |

Bits:    31   30                              0
Range: -2,147,483,648 to 2,147,483,647
```

initial value A numeric constant. If omitted, the initial value is zero.

DIM Dimension the variable as an array.

OVER Share a memory location with another variable.

NAME Specify an alternate, "external" name for the field.

EXTERNAL Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.

DLL Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.

STATIC Specify the variable´s memory is permanently allocated.

THREAD Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data**.**

AUTO Specify the variable has no *initial value*.

LONG declares a four-byte signed integer, using the Intel 8086 long integer format. The high-order bit is the sign bit (0 = positive, 1 = negative). Negative values are represented in standard two´s complement notation.

Example:

```
Count1 LONG                    !Declare four-byte signed integer
Count2 LONG,OVER(Count1)          !Declare OVER the four-byte signed integer
Count3 LONG,DIM(4)               !Declare it an array of 4 longs
Count4 LONG(5)                   !Declare with initial value
Count5 LONG,EXTERNAL             !Declare as external
Count6 LONG,NAME(´SixCount´)      !Declare with external name
ExampleFile FILE,DRIVER(´Clarion´) !Declare a file
Record      RECORD
Count7        LONG,NAME(´Counter´)  !Declare with external name
          . .
```

ULONG (four-byte unsigned integer)

label **ULONG**([*initial value*]) [,**DIM( )**] [,**OVER( )**] [,**NAME( )**] [,**EXTERNAL**] [,**DLL**] [,**STATIC**] [,**THREAD**] [,**AUTO**]

ULONG A four-byte unsigned integer.

```
Format:             magnitude
       | ............................. |

Bits:   31                           0
Range: 0 to 4,294,967,295
```

initial value A numeric constant. If omitted, the initial value is zero.

DIM Dimension the variable as an array.

OVER Share a memory location with another variable.

NAME Specify an alternate, "external" name for the field.

EXTERNAL Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.

DLL Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.

STATIC Specify the variable´s memory is permanently allocated.

THREAD Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data.

AUTO Specify the variable has no *initial value*.

ULONG declares a four-byte unsigned integer, using the Intel 8086 long integer format. There is no sign bit in this configuration.

Example:

```
Count1 ULONG                    !Declare four-byte unsigned integer
Count2 ULONG,OVER(Count1)       !Declare OVER four-byte unsigned integer
Count3 ULONG,DIM(4)             !Declare it an array of 4 unsigned longs
Count4 ULONG(5)                !Declare with initial value
Count5 ULONG,EXTERNAL           !Declare as external
Count6 ULONG,NAME(´SixCount´)     !Declare with external name
ExampleFile FILE,DRIVER(´Btrieve´)  !Declare a file
Record      RECORD
Count7        ULONG,NAME(´Counter´) !Declare with external name
          . .
```

SREAL (four-byte signed floating point)

label **SREAL**([*initial value*]) [,**DIM( )**] [,**OVER( )**] [,**NAME( )**] [,**EXTERNAL**] [,**DLL**] [,**STATIC**] [,**THREAD**] [,**AUTO**]

SREAL A four-byte floating point number.

```
Format:   ±   exponent     significand
       | . | ......... | ....................... |
Bits:   31  30        23                                    0
Range: 0, ± 1.175494e-38 .. ± 3.402823e+38 (6 significant digits)
```

initial value A numeric constant. If omitted, the initial value is zero.

DIM Dimension the variable as an array.

OVER Share a memory location with another variable.

NAME Specify an alternate, "external" name for the field.

EXTERNAL Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.

DLL Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.

STATIC Specify the variable´s memory is permanently allocated.

THREAD Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data.

AUTO Specify the variable has no *initial value*.

SREAL declares a four-byte floating point signed numeric variable, using the Intel 8087 short real (single precision) format.

Example:

```
Count1 SREAL                    !Declare four-byte signed floating point
Count2 SREAL,OVER(Count1)       !Declare OVER the four-byte
                               ! signed floating point
Count3 SREAL,DIM(4)             !Declare it an array of 4 floats
Count4 SREAL(5)                !Declare with initial value
Count5 SREAL,EXTERNAL           !Declare as external
Count6 SREAL,NAME(´SixCount´)     !Declare with external name
ExampleFile FILE,DRIVER(´Btrieve´) !Declare a file
Record     RECORD
Count7     SREAL,NAME(´Counter´) !Declare with external name
           . .
```

REAL (eight-byte signed floating point)

label **REAL**([*initial value*]) [,**DIM( )**] [,**OVER( )**] [,**NAME( )**] [,**EXTERNAL**] [,**DLL**] [,**STATIC**] [,**THREAD**] [,**AUTO**]

REAL An eight-byte floating point number.

```
Format:   ±    exponent              significand
        | . | ........... | ....................................... |
  Bits:   63  62          52
                  0

Range: 0, ± 2.225073858507201e-308 .. ± 1.79769313496231e+308
  (15 significant digits)
```

initial value A numeric constant. If omitted, the initial value is zero.

DIM Dimension the variable as an array.

OVER Share a memory location with another variable.

NAME Specify an alternate, "external" name for the field.

EXTERNAL Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.

DLL Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.

STATIC Specify the variable´s memory is permanently allocated.

THREAD Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data.

AUTO Specify the variable has no *initial value*.

REAL declares an eight-byte floating point signed numeric variable, using the Intel 8087 long real (double precision) format.

Example:

```
Count1 REAL                    !Declare eight-byte signed floating point
Count2 REAL,OVER(Count1)          !Declare OVER the eight-byte
                               !  signed floating point
Count3 REAL,DIM(4)               !Declare it an array of 4 reals
Count4 REAL(5)                   !Declare with initial value
Count5 REAL,EXTERNAL             !Declare as external
Count6 REAL,NAME(´SixCount´)      !Declare with external name
ExampleFile FILE,DRIVER(´Clarion´)  !Declare a file
Record      RECORD
Count5        REAL,NAME(´Counter´)    !Declare with external name
           . .
```

BFLOAT4 (four-byte signed floating point)

label **BFLOAT4**([*initial value*]) [,**DIM( )**] [,**OVER( )**] [,**NAME( )**] [,**EXTERNAL**] [,**DLL**] [,**STATIC**] [,**THREAD**] [,**AUTO**]

BFLOAT4 A four-byte floating point number.

```
Format:    exponent     ±            significand
       | .......... | . | ....................................... |
Bits:   31          23  22
            0
Range: 0, ± 5.87747e-39 .. ± 1.70141e+38
         (6 significant digits)
```

initial value A numeric constant. If omitted, the initial value is zero.

DIM Dimension the variable as an array.

OVER Share a memory location with another variable.

NAME Specify an alternate, "external" name for the field.

EXTERNAL Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.

DLL Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.

STATIC Specify the variable´s memory is permanently allocated.

THREAD Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data.

AUTO Specify the variable has no *initial value*.

BFLOAT4 declares a four-byte floating point signed numeric variable, using the Microsoft BASIC (single precision) format. This data type is normally used for compatibility with existing data since it is internally converted to a REAL before all arithmetic operations.

Example:

```
Count1 BFLOAT4                      !Declare four-byte signed floating point
Count2 BFLOAT4,OVER(Count1)           !Declare OVER the four-byte
                                   !  signed floating point
Count3 BFLOAT4,DIM(4)               !Declare array of 4 single-precision reals
Count4 BFLOAT4(5)                   !Declare with initial value
Count5 BFLOAT4,EXTERNAL             !Declare as external
Count6 BFLOAT4,NAME(´SixCount´)        !Declare with external name
ExampleFile FILE,DRIVER(´Btrieve´)     !Declare a file
Record      RECORD
Count5        BFLOAT4,NAME(´Counter´)    !Declare with external name
          . .
```

BFLOAT8 (eight-byte signed floating point)

label **BFLOAT8**([*initial value*]) [,**DIM( )**] [,**OVER( )**] [,**NAME( )**] [,**EXTERNAL**] [,**DLL**][,**STATIC**] [,**THREAD**] [,**AUTO**]

BFLOAT8 An eight-byte floating point number.

```
Format:   exponent    ±            significand
       | ........... | . | ..................................... |
Bits:   63          55  54
             0
Range: 0, ± 5.877471754e-39 .. ± 1.7014118346e+38
          (15 significant digits)
```

***i**nitial value* A numeric constant. If omitted, the initial value is zero.

DIM Dimension the variable as an array.

OVER Share a memory location with another variable.

NAME Specify an alternate, "external" name for the field.

EXTERNAL Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.

DLL Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.

STATIC Specify the variable´s memory is permanently allocated.

THREAD Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data.

AUTO Specify the variable has no *initial value*.

BFLOAT8 declares an eight-byte floating point signed numeric variable, using the Microsoft BASIC (double precision) format. This data type is normally used for compatibility with existing data since it is internally converted to a REAL before all arithmetic operations.

Example:

```
Count1 BFLOAT8                      !Declare eight-byte signed floating point
Count2 BFLOAT8,OVER(Count1)           !Declare OVER the eight-byte
                                   !  signed floating point
Count3 BFLOAT8,DIM(4)               !Declare it an array of 4 reals
Count4 BFLOAT8(5)                   !Declare with initial value
Count5 BFLOAT8,EXTERNAL             !Declare as external
Count6 BFLOAT8,NAME(´SixCount´)        !Declare with external name
ExampleFile FILE,DRIVER(´Btrieve´)     !Declare a file
Record      RECORD
Count5        BFLOAT8,NAME(´Counter´)    !Declare with external name
          . .
```

DECIMAL (signed packed decimal)

label **DECIMAL(***length* [,*places*] [,*initial value*]**)** [,**DIM( )**] [,**OVER( )**] [,**NAME( )**] [,**STATIC**] [,**THREAD**] [,**EXTERNAL**] [,**DLL**] [,**AUTO**]

DECIMAL A packed decimal floating point number.

```
Format:   ±                magnitude
       | . | ..........................................  |
Bits:  127  124
    0
Range: -9,999,999,999,999,999,999,999,999,999,999,999 to
       +9,999,999,999,999,999,999,999,999,999,999,999
```

length A required numeric constant containing the total number of decimal digits (integer and fractional portion combined) in the variable. The maximum *length* is 31.

places A numeric constant that fixes the number of decimal digits in the fractional portion (to the right of the decimal point) of the variable. It must be less than the *length* parameter. If omitted, the variable will be declared as an integer.

initial value A numeric constant. If omitted, the initial value is zero.

DIM Dimension the variable as an array.

OVER Share a memory location with another variable.

NAME Specify an alternate, "external" name for the field.

EXTERNAL Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.

DLL Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.

STATIC Specify the variable´s memory is permanently allocated.

THREAD Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data.

AUTO Specify the variable has no *initial value*.

DECIMAL declares a variable length packed decimal signed numeric variable. Each byte of a DECIMAL holds two decimal digits (4 bits per digit). The left-most byte holds the sign in its high-order nibble (0 = positive, anything else is negative) and one decimal digit. Therefore, DECIMAL variables always contain a fixed "odd" number of digits (DECIMAL(10) and DECIMAL(11) both use 6 bytes).

Example:

```
Count1 DECIMAL(5,0)                  !Declare three-byte signed packed decimal
Count2 DECIMAL(5),OVER(Count1)            !Declare OVER the three-byte
                                       !  signed packed decimal
Count3 DECIMAL(5,0),DIM(4)               !Declare it an array of 4 decimals
Count4 DECIMAL(5,0,5)                  !Declare with initial value
Count5 DECIMAL(5,0),EXTERNAL             !Declare as external
Count6 DECIMAL(5,0),NAME(´SixCount´)        !Declare with external name
ExampleFile FILE,DRIVER(´Clarion´)        !Declare a file
Record      RECORD
Count7        DECIMAL(5,0),NAME(´Counter´) !Declare with external name
          . .
```

PDECIMAL (signed packed decimal)

label **PDECIMAL(***length* [,*places*] [,*initial value*]**)** [,**DIM( )**] [,**OVER( )**] [,**NAME( )**] [,**EXTERNAL**] [,**DLL**] [,**STATIC**] [,**THREAD**] [,**AUTO**]

PDECIMAL A packed decimal floating point number.

```
Format:                  magnitude
        ±
       | ........................................... | . |
Bits:   127
    4    0
Range: -9,999,999,999,999,999,999,999,999,999,999,999 to
     +9,999,999,999,999,999,999,999,999,999,999,999
```

length A required numeric constant containing the total number of decimal digits (integer and fractional portion combined) in the variable. The maximum *length* is 31.

places A numeric constant that fixes the number of decimal digits in the fractional portion (to the right of the decimal point) of the variable. It must be less than the *length* parameter. If omitted, the variable will be declared as an integer.

initial value A numeric constant. If omitted, the initial value is zero.

DIM Dimension the variable as an array.

OVER Share a memory location with another variable.

NAME Specify an alternate, "external" name for the field.

EXTERNAL Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.

DLL Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.

STATIC Specify the variable´s memory is permanently allocated.

THREAD Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data.

AUTO Specify the variable has no *initial value*.

PDECIMAL declares a variable length packed decimal signed numeric variable in the Btrieve and IBM/EBCDIC type of format. Each byte of an PDECIMAL holds two decimal digits (4 bits per digit). The right-most byte holds the sign in its low-order nibble (0Fh or 0Ch = positive, 0Dh = negative) and one decimal digit. Therefore, PDECIMAL variables always contain a fixed "odd" number of digits (PDECIMAL(10) and PDECIMAL(11) both use 6 bytes).

Example:

```
Count1 PDECIMAL(5,0)                  !Declare three-byte signed packed decimal
Count2 PDECIMAL(5),OVER(Count1)            !Declare OVER the three-byte
                                         ! signed packed decimal
Count3 PDECIMAL(5,0),DIM(4)                !Declare it an array of 4 decimals
Count4 PDECIMAL(5,0,5)                     !Declare with initial value
Count5 PDECIMAL(5,0),EXTERNAL              !Declare as external
Count6 PDECIMAL(5,0),NAME(´SixCount´)       !Declare with external name
ExampleFile FILE,DRIVER(´Btrieve´)          !Declare a file
Record      RECORD
```

```
Count7        PDECIMAL(5,0),NAME(´Counter´) !Declare with external name
            . .
```

STRING (fixed-length string)

```
                |    length       |
label   STRING( |string constant  |) [,DIM( )][,OVER( )] [,NAME( )] [,EXTERNAL] [,DLL] [,STATIC]
                |    picture      |        [,THREAD] [,AUTO]
```

STRING A character string.

```
Format: A fixed number of bytes.
Size:   1 to 65,520 bytes in 16-bit, or 4MB in 32-bit.
```

length A numeric constant that defines the number of bytes in the STRING. String variables are not initialized unless given a *string constant*.

string constant The initial value of the STRING. The length of the STRING (in bytes) is set to the length of the *string constant*.

picture Used to format the values assigned to the STRING. The length is the number of bytes needed to contain the formatted STRING.

DIM Dimension the variable as an array.

OVER Share a memory location with another variable.

NAME Specify an alternate, "external" name for the field.

EXTERNAL Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.

DLL Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.

STATIC Specify the variable's memory is permanently allocated.

THREAD Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data.

AUTO Specify the variable has no *initial value*.

STRING declares a fixed-length character string. The memory assigned to the STRING is initialized to all blanks unless the AUTO attribute is present.

In addition to its explicit declaration, all STRING variables are also implicitly declared as STRING(1),DIM(*length of string*). This allows each character in the STRING to be addressed as an array element. If the STRING also has a DIM attribute, this implicit array declaration is the last (optional) dimension of the array (to the right of the explicit dimensions).

You may also directly address multiple characters within a STRING using the "string slicing" technique. This technique performs similar action to the SUB function, but is much more flexible and efficient. It is more flexible because a "string slice" may be used on both the destination and source sides of an assignment statement and the SUB function can only be used as the source. It is more efficient because it takes less memory than individual character assignments or the SUB function.

To take a "slice" of the STRING, the beginning and ending character numbers are separated by a colon (:) and placed in the implicit array dimension position within the square brackets ([]) of the STRING. The position numbers may be integer constants, variables, or expressions. If variables are used, there must be at least one blank space between the variable name and the colon separating the beginning and ending number (to prevent PREfix confusion).

Example:

```
Name       STRING(20)                    !Declare 20 byte name field
ArrayString  STRING(5),DIM(20)               !Declare array
Company     STRING('Clarion Software, Inc.')  !The software company - 22 bytes
Phone       STRING(@P(###)###-####P)       !Phone number field - 13 bytes
ExampleFile FILE,DRIVER('Clarion')         !Declare a file
Record      RECORD
NameField       STRING(20),NAME('Name')    !Declare with external name
          . .
  CODE
  NameField = 'Tammi'                     !Assign a value
  NameField[5] = 'y'                    ! change fifth letter
  NameField[5:6] = 'ie'                   ! and change a "slice"
                                       !  -- the fifth and sixth letters
  ArrayString[1] = 'First'                !Assign value to first element
  ArrayString[1,2] = 'u'                  !Change first element 2nd character
  ArrayString[1,2:3] = NameField[5:6]     !Assign slice to slice
```

CSTRING (fixed-length null terminated string)

```
                  |     length     |
label   CSTRING( |string constant |) [,DIM( )] [,OVER( )] [,NAME( )] [,EXTERNAL] [,DLL]
                  |     picture    |          [,STATIC] [,THREAD] [,AUTO]
```

CSTRING A character string.

```
Format: A fixed number of bytes.
Size:   2 to 65,520 bytes in 16-bit, or unlimited in 32-bit.
```

length A numeric constant that defines the number of bytes of storage the string will use. This must include a position for the terminating null character. String variables are not initialized unless given a *string constant*.

string constant A string constant containing the initial value of the string. The length of the string is set to the length of the *string constant* plus the terminating null character.

picture The picture token used to format the values assigned to the string. The length of the string is the number of bytes needed to contain the formatted string and the terminating null character.

DIM Dimension the variable as an array.

OVER Share a memory location with another variable.

NAME Specify an alternate, "external" name for the field.

EXTERNAL Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.

DLL Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.

STATIC Specify the variable's memory is permanently allocated.

THREAD Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data.

AUTO Specify the variable has no *initial value*.

CSTRING declares a character string terminated by a null character (ASCII zero). The memory assigned to the CSTRING is initialized to a zero length string unless the AUTO attribute is present.

CSTRING matches the string data type used in the "C" language and the "ZSTRING" data type of the Btrieve Record Manager. Storage and memory requirements are fixed-length, however the terminating null character is placed at the end of the data entered. CSTRING is internally converted to a STRING intermediate value for use during program execution. It should be used to achieve compatibility with outside files or procedures.

In addition to its explicit declaration, all CSTRINGs are implicitly declared as a CSTRING(1),DIM(*length of string*). This allows each character in the CSTRING to be addressed as an array element. If the CSTRING also has a DIM attribute, this implicit array declaration is the last (optional) dimension of the array (to the right of the explicit dimensions).

You may also directly address multiple characters within a CSTRING using the "string slicing" technique. This technique performs similar action to the SUB function, but is much more flexible and efficient. It is more flexible because a "string slice" may be used on both the destination and source sides of an assignment statement and the SUB function can only be used as the source. It is more efficient because

it takes less memory than individual character assignments or the SUB function.

To take a "slice" of the CSTRING, the beginning and ending character numbers are separated by a colon (:) and placed in the implicit array dimension position within the square brackets ([]) of the CSTRING. The position numbers may be integer constants, variables, or expressions. If variables are used, there must be at least one blank space between the variable name and the colon separating the beginning and ending number (to prevent PREfix confusion).

Since a CSTRING must be null-terminated, the programmer must be responsible for ensuring that an ASCII zero is placed at the end of the data if the field is only accessed through its array elements or as a "slice" (not as a whole entity).

Example:

```
Name       CSTRING(21)                !Declare 21 byte field - 20 bytes data
OtherName    CSTRING(21),OVER(Name)       !Declare field over name field
Contact      CSTRING(21),DIM(4)           !Array 21 byte fields - 80 bytes data
Company      CSTRING('Clarion Software, Inc.')  !23 byte string - 22 bytes data
Phone        CSTRING(@P(###)###-####P)     !  Declare 14 bytes - 13 bytes data
ExampleFile FILE,DRIVER('Btrieve')       !Declare a file
Record       RECORD
NameField       CSTRING(21),NAME('ZstringField') !Declare with external name
           . .
  CODE
  Name = 'Tammi'             !Assign a value
  Name[5] = 'y'              ! then change fifth letter
  Name[6] = 's'              ! then add a letter
  Name[7] = '<0>'              ! and handle null terminator
  Name[5:6] = 'ie'           ! and change a "slice"
                           !  -- the fifth and sixth letters
  Contact[1] = 'First'         !Assign value to first element
  Contact[1,2] = 'u'         !Change first element 2nd character
  Contact[1,2:3] = Name[5:6]  !Assign slice to slice
```

PSTRING (embedded length-byte string)

```
                |    length       |
label  PSTRING( |string constant  |) [,DIM( )] [,OVER( )] [,NAME( )] [,EXTERNAL] [,DLL] [,STATIC] [,THREAD]
                |    picture      |         [,AUTO]
```

PSTRING A character string.

```
Format:  A fixed number of bytes.
Size:    2 to 255 bytes.
```

length A numeric constant that defines the number of bytes in the string. This must include the first position length-byte.

string constant A string constant containing the initial value of the string. The length of the string is set to the length of the *string constant* plus the length-byte.

picture The picture token used to format the values assigned to the string. The length of the string is the number of bytes needed to contain the formatted string plus the first position length byte. String variables are not initialized unless given a *string constant*.

DIM Dimension the variable as an array.

OVER Share a memory location with another variable.

NAME Specify an alternate, "external" name for the field.

EXTERNAL Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.

DLL Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.

STATIC Specify the variable´s memory is permanently allocated.

THREAD Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data**.**

AUTO Specify the variable has no *initial value*.

PSTRING declares a character string with a leading length byte included in the number of bytes declared for the string. The memory assigned to the PSTRING is initialized to a zero length string unless the AUTO attribute is present.

PSTRING matches the string data type used by the Pascal language and the "LSTRING" data type of the Btrieve Record Manager. Storage and memory requirements are fixed-length, however, the leading length byte will contain the number of characters actually stored. PSTRING is internally converted to a STRING intermediate value for use during program execution. It should be used to achieve compatibility with outside files or procedures.

In addition to its explicit declaration, all PSTRINGs are implicitly declared as a PSTRING(1),DIM(*length of string*). This allows each character in the PSTRING to be addressed as an array element. If the PSTRING also has a DIM attribute, this implicit array declaration is the last (optional) dimension of the array (to the right of the explicit dimensions).

You may also directly address multiple characters within a PSTRING using the "string slicing" technique. This technique performs similar action to the SUB function, but is much more flexible and efficient. It is more flexible because a "string slice" may be used on both the destination and source sides of an assignment statement and the SUB function can only be used as the source. It is more efficient because

it takes less memory than individual character assignments or the SUB function.

To take a "slice" of the PSTRING, the beginning and ending character numbers are separated by a colon (:) and placed in the implicit array dimension position within the square brackets ([]) of the PSTRING. The position numbers may be integer constants, variables, or expressions. If variables are used, there must be at least one blank space between the variable name and the colon separating the beginning and ending number (to prevent PREfix confusion).

Since a PSTRING must have a leading length byte, the programmer must be responsible for ensuring that its value is always correct if the field is only accessed through its array elements or as a "slice" (not as a whole entity). The PSTRING´s length byte is addressed as element zero (0) of the array (the only case in Clarion where an array has a zero element). Therefore, the valid range of array indexes for a PSTRING(30) would be 0 to 29.

Example:

```
Name       PSTRING(21)                   !Declare 21 byte field - 20 bytes data
OtherName   PSTRING(21),OVER(Name)          !Declare field over name field
Contact     PSTRING(21),DIM(4)              !Array 21 byte fields - 80 bytes data
Company     PSTRING(´Clarion Software, Inc.´)  !23 byte string - 22 bytes data
Phone       PSTRING(@P(###)###-####P)        !Declare 14 bytes - 13 bytes data
ExampleFile FILE,DRIVER(´Btrieve´)          !Declare a file
Record      RECORD
NameField       PSTRING(21),NAME(´LstringField´) !Declare with external name
          . .
  CODE
  Name = ´Tammi´              !Assign a value
  Name[5] = ´y´               ! then change fifth letter
  Name[6] = ´s´               ! then add a letter
  Name[0] = ´<6>´               ! and handle length byte
  Name[5:6] = ´ie´            ! and change a "slice"
                             !  -- the fifth and sixth letters
  Contact[1] = ´First´          !Assign value to first element
  Contact[1,2] = ´u´          !Change first element 2nd character
  Contact[1,2:3] = Name[5:6]   !Assign slice to slice
```

DATE (four-byte date)

label **DATE** [,**DIM( )**] [,**OVER( )**] [,**NAME( )**] [,**EXTERNAL**] [,**DLL**] [,**STATIC**] [,**THREAD**] [,**AUTO**]

DATE A four-byte date.

```
Format:     year     mm     dd
          | ........ | .... | .... |
Bits:     31        15     7     0
Range:    year: 1 to 9999
         month: 1 to 12
           day: 1 to 31
```

DIM Dimension the variable as an array.

OVER Share a memory location with another variable.

NAME Specify an alternate, "external" name for the field.

EXTERNAL Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.

DLL Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.

STATIC Specify the variable´s memory is permanently allocated.

THREAD Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data.

AUTO Specify the variable has no *initial value*.

DATE declares a four-byte date variable. This format matches the "DATE" field type used by the Btrieve Record Manager. A DATE used in a numeric expression is converted to the number of days elapsed since December 28, 1800 (Clarion Standard Date - usually stored as a LONG). The valid Clarion Standard Date range is January 1, 1801 through December 31, 2099. Using an out-of-range date produces unpredictable results. DATE fields should be used to achieve compatibility with outside files or procedures.

Example:

```
DueDate     DATE                    !Declare a date field
OtherDate   DATE,OVER(DueDate)        !Declare field over date field
ContactDate DATE,DIM(4)                 !Array of 4 date fields
ExampleFile FILE,DRIVER(´Btrieve´)      !Declare a file
Record      RECORD
DateRecd      DATE,NAME(´DateField´)  !Declare with external name
          . .
```

See Also:

Standard Date

TIME (four-byte time)

label **TIME** [,**DIM( )**] [,**OVER( )**] [,**NAME( )**] [,**EXTERNAL**] [,**DLL**] [,**STATIC**] [,**THREAD**] [,**AUTO**]

TIME A four-byte time.

```
Format:     hh    mm    ss    hs
          | .... | .... | .... | .... |
Bits:     31    23    15     7     0
Range:      hours: 0 to 23
           minutes: 0 to 59
           seconds: 0 to 59
      seconds/100: 0 to 99
```

DIM Dimension the variable as an array.

OVER Share a memory location with another variable.

NAME Specify an alternate, "external" name for the field.

EXTERNAL Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.

DLL Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.

STATIC Specify the variable´s memory is permanently allocated.

THREAD Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data.

AUTO Specify the variable has no *initial value*.

TIME declares a four byte time variable. This format matches the "TIME" field type used by the Btrieve Record Manager. A TIME used in a numeric expression is converted to the number of hundredths of a second elapsed since midnight (Clarion Standard Time - usually stored as a LONG). TIME fields should be used to achieve compatibility with outside files or procedures.

Example:

```
ChkoutTime  TIME                      !Declare checkout time field
OtherTime   TIME,OVER(CheckoutTime)      !Declare field over time field
ContactTime TIME,DIM(4)                  !Array of 4 time fields
ExampleFile FILE,DRIVER(´Btrieve´)       !Declare a file
Record      RECORD
TimeRecd      TIME,NAME(´TimeField´)   !Declare with external name
          . .
```

See Also:

Standard Time

GROUP (compound data structure)

label **GROUP(** [*group*] **)** [,**PRE( )**] [,**DIM( )**] [,**OVER( )**] [,**NAME( )**] [,**EXTERNAL**] [,**DLL**] [,**STATIC**]
[,**THREAD**] [,**BINDABLE**] [, **TYPE**]
declarations
END

GROUP	A compound data structure.
group	The label of a previously declared GROUP, QUEUE, or RECORD structure from which it will inherit its structure. This may be a GROUP or QUEUE with the TYPE attribute.
PRE	Declare a label prefix for variables within the structure.
DIM	Dimension the variables into an array.
OVER	Share a memory location with another variable or structure.
NAME	Specify an alternate, "external" name for the field.
EXTERNAL	Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.
DLL	Specify the variable is defined in a .DLL. This is required in addition to the EXTERNAL attribute.
STATIC	Specify the variable's memory is permanently allocated.
THREAD	Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data.
BINDABLE	Specify all variables in the group may be used in dynamic expressions.
TYPE	Specify the GROUP is a type definition for GROUPs passed as parameters.
declarations	Multiple consecutive variable declarations.

A **GROUP** structure allows multiple variable declarations to be referenced by a single label. It may be used to dimension a set of variables, or to assign or compare sets of variables in a single statement. In large complicated programs, a GROUP structure is helpful for keeping sets of related data organized. A GROUP must be terminated by a period or the END statement.

The structure of a GROUP declared with the *group* parameter begins with the same structure as the named *group*; the GROUP inherits the fields of the named *group*. The GROUP may also contain its own *declarations* that follow the inherited fields. If the group parameter names a QUEUE or RECORD structure, only the fields are inherited and not the functionality implied by the QUEUE or RECORD.

When referenced in a statement or expression, a GROUP is treated as a STRING composed of all the variables within the structure. A GROUP structure may be nested within another data structure, such as a RECORD or another GROUP.

Because of their internal storage format, numeric variables (other than DECIMAL) declared in a group do not collate properly when treated as strings. For this reason, building a KEY on a GROUP that contains numeric variables may produce an unexpected collating sequence.

A GROUP with the BINDABLE attribute makes all the variables within the GROUP available for use in a dynamic expression. The contents of each variable's NAME attribute is the logical name used in the dynamic expression. If no NAME attribute is present, the label of the variable (including prefix) is used. Space is allocated in the .EXE for the names of all of the variables in the structure. This creates a larger program that uses more memory than it normally would. Therefore, the BINDABLE attribute should only

be used when a large proportion of the constituent fields are going to be used.

A GROUP with the TYPE attribute is not allocated any memory; it is only a type definition for GROUPs that are passed as parameters to PROCEDUREs or FUNCTIONs. This allows the receiving procedure to directly address component fields in the passed GROUP. The parameter declaration on the PROCEDURE or FUNCTION statement instantiates a local prefix for the passed GROUP as it names the passed GROUP for ther procedure. For example, PROCEDURE(LOC:PassedGroup) declares the procedure uses the LOC: prefix (along with the individual field names used in the type definition) to directly address component fields of the GROUP passed as the parameter.

Example:

```
          PROGRAM
PassGroup   GROUP,TYPE              !Type-definition for passed GROUP parameters
F1          STRING(20)             ! first field
F2          STRING(1)              ! middle field
F3          STRING(20)             ! last field
          END

   MAP
     MyProc1(PassGroup)           !Passes a GROUP defined the same as PassGroup
   END

NameGroup   GROUP,PRE(Nme)          !Name group
First         STRING(20)            ! first name
Middle      STRING(1)              ! middle initial
Last        STRING(20)             ! last name
          END                     !End group declaration

NameGroup2  GROUP(PassGroup),PRE(Nme2)  !Group that inherits PassGroup's fields
                                  ! resulting in Nme2:F1, Nme2:F2, and Nme2:F3
          END                     ! fields declared in this group

DateTimeGrp GROUP,PRE(Dtg),DIM(10)   !Date/time array
Date        LONG
Time        LONG
          END                     !End group declaration

FileNames   GROUP,BINDABLE          !Bindable group
FileName    STRING(8),NAME('FILE')  !Dynamic name: FILE
Dot         STRING('.')            !Dynamic name: Dot
Extension     STRING(3),NAME('EXT')   !Dynamic name: EXT
          END

  CODE
  MyProc1(NameGroup)              !Call proc passing NameGroup as parameter
  MyProc1(NameGroup2)               !Call proc passing NameGroup2 as parameter

MyProc1     PROCEDURE(LOC:PassedGroup)   !Proc to receive GROUP parameter
LocalVar  STRING(20)
  CODE
  LocalVar = LOC:F1                 !Assign value in Nme:First to LocalVar
                                  !  from passed parameter
```

LIKE (inherited data type)

new declaration **LIKE(***like declaration***)** [,**DIM( )**] [,**OVER( )**] [,**PRE( )**] [,**NAME( )**] [,**BINDABLE**]

LIKE Declares a variable whose data type is inherited from another variable.

new declaration The label of the new data element declaration.

like declaration The label of the data element declaration whose definition will be used.

DIM Dimension the variables into an array.

OVER Share a memory location with another variable or structure.

PRE Declare a label prefix for any variables within the structure. This is required when the like declaration is a GROUP.

NAME Specify an alternate, "external" name for the field.

EXTERNAL Specify the variable is defined, and its memory is allocated, in an external library. Not valid within FILE, QUEUE, or GROUP declarations.

STATIC Specify the variable´s memory is permanently allocated.

THREAD Specify memory for the variable is allocated once for each execution thread. Also implicitly adds the STATIC attribute on Procedure Local data.

BINDABLE Specify all variables in the group may be used in dynamic expressions.

LIKE tells the compiler to define the *new declaration* using the same definition as the *like declaration*, including all attributes. If the original *like declaration* changes, so does the *new declaration*.

The *new declaration* may use the DIM and OVER attributes. If the *like declaration* has a DIM attribute, the *new declaration* is already an array. If a further DIM attribute is added to the *new declaration*, the array is further dimensioned.

The PRE and NAME attributes may be used, if appropriate. If the *like declaration* already has these attributes, the *new declaration* will inherit them and compiler errors can occur. To correct this, specify a PRE or NAME attribute on the *new declaration* to override the inherited attribute.

Example:

```
Amount      REAL                !Define a field
QTDAmount     LIKE(Amount)          !Use same definition
YTDAmount     LIKE(QTDAmount)         !Use same definition again
MonthlyAmts   LIKE(Amount),DIM(12)    !Use same definition for array, 12 elements
AmtPrPerson   LIKE(MonthlyAmts),DIM(10)
                        !Use same definition for array of 120 elements (12,10)

Construct     GROUP,PRE(Con)          !Define a group
Field1      LIKE(Amount)          ! con:field1 - real
Field2      STRING(10)            ! con:field2 - string(10)
          END

NewGroup    LIKE(Construct),PRE(New)   !Define new group, with
                                ! new:field1 - real
                                ! new:field2 - string(10)

AmountFile  FILE,DRIVER(´Clarion´),PRE(Amt)
Record      RECORD
Amount        REAL              !Define a field
```

```
QTDAmount       LIKE(Amount)         !Use same definition
          . .
```

See Also:

DIM

OVER

PRE

NAME

Implicit Variables

Implicit variables are not declared in data declarations. They are created by the compiler when it first encounters them. Implicit variables are automatically initialized to blank or zero; they do not have to be explicitly assigned values before use. You may always assume that they contain blanks or zero before your program´s first assignment to them.

Any implicit variable used in the global data declaration area (between the keywords PROGRAM and CODE) is Global data, assigned static memory. Any implicit variable used between the keywords MEMBER and PROCEDURE (or FUNCTION) is Module data, assigned static memory. Any other implicit variable is Local data, assigned dynamic memory on the program´s stack.

Since the compiler dynamically creates implicit variables as they are encountered, there is a danger that problems may arise that can be difficult to trace. This is due to the lack of compile-time error and type checking on implicit variables. For example, if you spell incorrectly the name of a previously used implicit variable, the compiler will not tell you, but will simply create a new implicit variable with the new spelling. When your program checks the value in the original implicit variable, it will be incorrect. Therefore, implicit variables should be used with care and caution, and only within a limited scope (or not at all).

Implicit variables are generally used for: array subscripts, true/false switches, intermediate variables in complex calculations, loop control variables, etc. The Clarion language provides three types of implicit variables:

#	Pound sign names an implicit LONG variable, a label terminated by a # character.
$	Dollar sign names an implicit REAL variable, a label terminated by a $ character.
"	Double quote names an implicit 32 byte string, a label terminated by a " character.

Example:

```
LOOP Counter# = 1 TO 10                    !Implicit LONG loop counter
  ArrayField[Counter#] = Counter# * 2         ! to initialize an array
END

Address" = CLIP(City) & ´, ´ & State & ´  ´ & Zip     !Implicit STRING(32)
MESSAGE(Address")                          !Used to display a temporary value

Percent$ = ROUND((Quota / Sales),.1) * 100   !Implicit REAL
MESSAGE(FORMAT(Percent$,@P%<<<.##P))        !Used to display a temporary value
```

See Also:

Data Declarations and Memory Allocation

Reference Variables

A reference variable contains a reference to another data declaration (its "target"). You declare a reference variable by prepending an ampersand (&) to the data type of its target (&BYTE, &FILE, &LONG, &WINDOW, etc.). Depending upon the target´s data type, the reference variable may contain the target´s memory address, or a more complex internal data structure (describing the location and type of target data).

Valid reference variable declarations are: &BYTE, &SHORT, &USHORT, &LONG, &ULONG, &REAL, &SREAL, &BFLOAT8, &BFLOAT4, &DECIMAL, &PDECIMAL, &STRING, &CSTRING, &PSTRING, &GROUP, &QUEUE, &FILE, &VIEW, &WINDOW. Reference variables may not be declared within GROUP, FILE, QUEUE, or VIEW structures.

The &STRING, &CSTRING, &PSTRING, &DECIMAL, and &PDECIMAL reference variable declarations do not require length parameters, since all necessary information about the specific target data item is contained in the reference. This means a &STRING reference variable may contain a reference to any length STRING variable. A reference variable declared with &WINDOW can target either an APPLICATION, WINDOW, or REPORT structure.

The label of the reference variable is syntactically correct every place in executable code where its target is allowed. When used in a code statement, the reference variable is automatically "dereferenced" to supply the statement the value of its target (except for reference assignment statements). References cross thread boundaries, and so, may be used to reference data items in other execution threads.

The &= operator executes a reference assignment statement (destination &= source). This assigns the source´s reference to the destination reference variable.

Example:

```
App1  APPLICATION(´Hello´)
      END
App2  APPLICATION(´Buenos Dias´)
      END
AppRef &WINDOW                  !Reference to an APPLICATION, WINDOW, or REPORT
  CODE
  IF CTL:Language = ´English´     !If english language user
    AppRef &= App1               ! reference english application frame
  ELSE
    AppRef &= App2               ! else reference spanish application frame
  END
  OPEN(AppRef)                  !Open the referenced application frame window
```

See Also:

Reference Assignment Statements

THREAD

Attributes of Variables

PRE (set group label prefix)

PRE(*prefix***)**

PRE	Provides a label prefix for complex data structures.
prefix	Acceptable characters are alphabet letters, numerals 0 through 9, and the underscore character. A *prefix* must start with an alphabet character and must not be a reserved word. By convention, a *prefix* is 1-3 characters, although it can be longer.

The **PRE** attribute provides a label prefix for complex data structures. It is used to distinguish between identical variable names that occur in different structures. When referenced in executable statements, assignments, and parameter lists, a *prefix* is attached to a label by a colon (Pre:Label). PRE may be used with the following data structures discussed in this reference: GROUP, and LIKE.

Example:

```
G1     GROUP,PRE(Mem)   !Declare some memory variables
Message   STRING(30)      ! with the Mem prefix
Page     LONG
Line     LONG
Device   STRING(30)
       END

G2     LIKE(G1),PRE(Me2)    !Declare second GROUP LIKE the first
                          !Contains same variables with Me2 prefix
  CODE
  Mem:Message = ´Variable in original group´
  Me2:Message = ´Variable in LIKE group´
```

See Also:

Reserved Words

DIM (set array dimensions)

DIM(*dimension,...,dimension***)**

DIM Declares a variable as an array.

dimension A numeric constant which specifies the number of elements in this *dimension* of the array.

The **DIM** attribute declares a variable as an array. The variable is repeated the number of times specified by the *dimension* parameters. Multi-dimensional arrays may be thought of as nested. Each *dimension* in the array has a corresponding subscript. Therefore, referencing a variable in a three dimensional array requires three subscripts. There is no limit to the number of dimensions; however, the total size of an array must not exceed 65,520 bytes of data in 16-bit applications (there is no limit in 32-bit applications).

Subscripts identify which element of the array is being referenced. A subscript list contains a subscript for each *dimension* of the array. Each subscript is separated by a comma and the entire list is enclosed in brackets ([ ]). A subscript may be a numeric constant, expression, or function. The entire array may be referenced by the label of the array without a subscript list.

A GROUP structure is a special case. Each level of nesting adds subscripts to the GROUP and the variables within the GROUP. Data declared within the GROUP may be referenced exactly like the GROUP itself.

Example:

```
Scr    GROUP                !Characters on a text-mode screen
Row      GROUP,DIM(25)      !Twenty-five rows
Pos        GROUP,DIM(80)    !Two thousand positions
Attr         BYTE           !Attribute byte
Char         BYTE           !Character byte
       . . .                !Terminate the group structures
  ! In the group above:
  ! Scr is a 4,000 byte GROUP
  ! Row[1] is a 160 byte GROUP
  ! Pos[1,1] is a 2 byte GROUP
  ! Attr[1,1] is a BYTE
  ! Char[1,1] is a BYTE

Month   STRING(10),DIM(12)  !Dimension the month to 12
  CODE
  CLEAR(Month)              !Assign blanks to the entire array
  Month[1] = 'January'       !Load the months into the array
  Month[2] = 'February'
  Month[3] = 'March'
```

See Also:

MAXIMUM

EXTERNAL (set variable defined externally)

EXTERNAL([*member*] **)**

EXTERNAL Specifies the variable or FILE is defined in an external library.

member A string constant. Valid only on a FILE declaration. It contains the filename (without extension) of the MEMBER module containing the FILE definition without the EXTERNAL attribute. If the FILE is defined in a PROGRAM module, an empty *member* string ('') is required.

The **EXTERNAL** attribute specifies that the variable or FILE on which it is placed is defined in an external library. Therefore, a variable or FILE with the EXTERNAL attribute is declared and may be referenced in the Clarion code, but is not allocated memory. The memory for the variable or FILE is allocated by the external library. This allows the Clarion program access to variables or FILEs declared as public in external libraries.

The EXTERNAL attribute, without the *member* parameter, is valid only on variables declared outside FILE, QUEUE, or GROUP structures.

When using EXTERNAL(*member*) to declare a FILE shared by multiple libraries (.LIBs, or .DLLs and .EXE), only one library should define the FILE without the EXTERNAL attribute. All the other libraries (and the .EXE) should declare the FILE with the EXTERNAL attribute. This ensures that there is only one record buffer allocated for the FILE and all the libraries and the .EXE will reference the same memory when referring to data elements from that FILE.

The FILE declarations in all libraries (or .EXEs) that reference common files must be EXACTLY the same (with the appropriate addition of the EXTERNAL attribute). If they are not exactly the same, data corruption could occur. The actual consequence of incompatible FILE declarations is dependent upon the file driver for that file system. Any incompatibilities between libraries cannot be detected by the compiler or linker, therefore it is the programmer´s responsibility to ensure that consistency is maintained.

One suggested way of coding large systems using many .DLLs and/or .EXEs that share the same files would have one .DLL containing the actual FILE definition that only contains FILE and global variable definitions that are shared among all (or most) of the .DLLs and .EXEs. This makes one central library in which the actual file definitions are maintained. This one central .DLL is linked into all .EXEs that use those common files. All other .DLLs and/or .EXEs in the system would declare the common FILEs with the EXTERNAL attribute.

Example:

```
  PROGRAM
  MAP
   MODULE('LIB.LIB')
     AddCount                   !External library procedure
  . .

TotalCount  LONG,EXTERNAL       !A variable declared in an external library

Cust   FILE,PRE(Cus),EXTERNAL('')     !A File defined in a PROGRAM module
CustKey   KEY('Name')               ! whose .LIB is linked into this program
Record  RECORD
Name      STRING(20)
       . .

Contact   FILE,PRE(Con),EXTERNAL('LIB01')   !A File defined in a MEMBER module
ContactKey  KEY('Name')                ! whose .LIB is linked into this program
```

```
Record    RECORD
Name        STRING(20)
         . .

! The LIB.CLW file contains:
  PROGRAM
  MAP
   MODULE(´LIB01´)
     AddCount  !
  . .

TotalCount  LONG                      !The TotalCount variable definition

Cust     FILE,PRE(Cus)                !The Cust File definition where the
CustKey     KEY(´Cus:Name´)              ! record buffer is allocated
Record    RECORD
Name        STRING(20)
         . .

  CODE
  !Executable code ...

! The LIB01.CLW file contains:
 MEMBER(´LIB´)

Contact     FILE,PRE(Con)         !The Contact File definition where the
ContactKey    KEY(´Con:Name´)       ! record buffer is allocated
Record      RECORD
Name          STRING(20)
           . .

AddCount PROCEDURE
  CODE
  TotalCount += 1
```

See Also:

NAME

DLL (set variable defined externally in .DLL)

DLL([*flag*] **)**

DLL	Declares a variable defined externally in a .DLL.
flag	A numeric constant, equate, or Project system define which specifies the attribute as active or not. If the *flag* is zero, the attribute is not active, just as if it were not present. If the *flag* is any value other than zero, the attribute is active.

The **DLL** attribute specifies that the variable on which it is placed is defined in a .DLL. A variable with DLL attribute must also have the EXTERNAL attribute. The DLL attribute is required for 32-bit applications because .DLLs are relocatable in a 32-bit flat address space, which requires one extra dereference by the compiler to address the variable. The DLL attribute is valid only on variables declared outside FILE, QUEUE, or GROUP structures.

The variable declarations in all libraries (or .EXEs) that reference common variables must be EXACTLY the same (with the appropriate addition of the EXTERNAL and DLL attributes). If they are not exactly the same, data corruption could occur. Any incompatibilities between libraries cannot be detected by the compiler or linker, therefore it is the programmer's responsibility to ensure that consistency is maintained.

When using EXTERNAL and DLL to declare a variable shared by .DLLs and .EXE, only one .DLL should define the variable without the EXTERNAL and DLL attributes. All the other .DLLs (and the .EXE) should declare the variable with the EXTERNAL and DLL attributes. This ensures that there is only one memory allocation for the variable and all the .DLLs and the .EXE will reference the same memory when referring to that variable.

One suggested way of coding large systems using many .DLLs and/or .EXEs that share the same variables would have one .DLL containing the actual data definition that only contains FILE and global variable definitions that are shared among all (or most) of the .DLLs and .EXEs. This makes one central library in which the actual file definitions are maintained. This one central .DLL is linked into all .EXEs that use those common files. All other .DLLs and/or .EXEs in the system would declare the common variables with the EXTERNAL and DLL attributes.

Example:

```
TotalCount  LONG,EXTERNAL,DLL        !A variable declared in an external .DLL
```

See Also: EXTERNAL

NAME (set variable´s external name)

NAME([| *constant* |])
| *variable* |

NAME	Specifies an "external" name for the linker or file driver.
constant	A string constant.
variable	The label of a STRING variable declared in the global data declaration area or a MEMBER module´s data declaration area.

The **NAME** attribute specifies an "external" name for the linker or file driver. The NAME attribute is completely independent of the EXTERNAL attribute--there is no required connection between the two, although both attributes may be used on the same variable.

The NAME attribute may be placed on a FUNCTION or PROCEDURE Prototype, FILE, KEY, INDEX, MEMO, any field declared within a FILE, any field declared within a QUEUE structure, or any field not within a structure. The NAME attribute has different implications depending on where it is used.

NAME(*constant*) may be specified on a FUNCTION or PROCEDURE Prototype. The *constant* supplies the external name used by the linker to identify the procedure or function from an external library.

The NAME(*constant*) or NAME(*variable*) attribute on a FILE declaration specifies a DOS directory file specification. If the *constant* or *variable* does not contain a drive and path, the current drive and directory are assumed. If the extension is omitted, the directory entry assumes the file driver´s default value. Some file drivers require that KEYs, INDEXes, or MEMOs be in separate files. Therefore, a NAME may also be placed on a KEY, INDEX, or MEMO. A NAME attribute without a *constant* or *variable* defaults to the label of the declaration statement on which it is placed (including any specified prefix).

NAME(*constant*) may be used on any field declared within the RECORD structure. This provides the file driver with the name of a field as it may be used in that driver´s file system.

NAME(*constant*) may be used on any field declared within a QUEUE structure. This provides the capability of run time dynamic sorts.

NAME(*constant*) may be used on any variable declared outside of any structure. This provides the linker with an external name to identify a variable declared in an external library. If the variable also has the EXTERNAL attribute, it is declared, and its memory is allocated, as a public variable in the external library. Without the EXTERNAL attribute, it is declared, and its memory is allocated, in the Clarion program, and it is declared as an external variable in the external library.

Example:

```
  PROGRAM
  MAP
   MODULE(´External.Obj´)
    AddCount(LONG),LONG,C,NAME(´_AddCount´)   !C function named ´_AddCount´
  . .

Cust   FILE,PRE(Cus),NAME(CustName)          !Filename in CustName variable
CustKey   KEY(´Name´),NAME(´c:\data\cust.idx´)   !Declare key, cust.idx
Record  RECORD
Name      STRING(20)                        !Default NAME to ´Cus:Name´
       . .

SortQue  QUEUE,PRE(Que)
Field1   STRING(10),NAME(´FirstField´)     !QUEUE SORT NAME
```

```
Field2   LONG,NAME(´SecondField´)        !QUEUE SORT NAME
       END

CurrentCnt  LONG,EXTERNAL,NAME(´Cur´)       !Field declared public in
                                          ! external library as ´Cur´
TotalCnt  LONG,NAME(´Tot´)                  !Field declared external
                                          ! in external library as ´Tot´
```

See Also:

FUNCTION and PROCEDURE Prototypes

FILE

KEY

INDEX

QUEUE

EXTERNAL

OVER (set shared memory location)

OVER(*overvariable***)**

OVER Allows one memory address to be referenced two different ways.

overvariable The label of a variable that already occupies the memory to be shared.

The **OVER** attribute allows one memory address to be referenced two different ways. The variable declared with the OVER attribute must not be larger than the *overvariable* it is being declared OVER (it may be smaller, though).

You may declare a variable OVER an *overvariable* which is part of the parameter list passed into a PROCEDURE or FUNCTION.

A field within a GROUP structure cannot be declared OVER a *variable* outside that GROUP structure.

Example:

```
SomeProc PROCEDURE(PassedGroup)      !Proc receives a GROUP parameter

NewGroup GROUP,OVER(PassedGroup)    !Redeclare passed GROUP parameter
Field1     STRING(10)              !Compiler warning issued that
Field2     STRING(2)               ! NewGroup must not be larger
         END                     ! than PassedGroup

CustNote FILE,PRE(Csn)             !Declare CustNote file
Notes     MEMO(2000)               !The memo field
Record   RECORD
CustID     LONG
         . .

CsnMemoRow STRING(10),DIM(200),OVER(Csn:Notes)
                                 !Csn:Notes memo may be addressed
                                 ! as a whole or in 10-byte chunks
```

See Also:

DIM

STATIC (set local variable static)

STATIC

The **STATIC** attribute allows a variable declared within a PROCEDURE or FUNCTION to be allocated static memory instead of stack memory. This makes any value contained in the variable "persistent" from one instance of the procedure to the next.

Example:

```
SomeProc  PROCEDURE
AcctFile  STRING(64),STATIC       !STATIC needed for use as
                                ! Variable in NAME attribute

Transactions  FILE,DRIVER(´Clarion´),PRE(TRA),NAME(AcctFile)
AccountKey      KEY(TRA:Account),OPT,DUP
Record        RECORD
Account           SHORT             !Account code
Date              LONG            !Transaction Date
Amount            DECIMAL(13,2)    !Transaction Amount
            . .
```

See Also:

Data Declarations and Memory Allocation

THREAD (set thread-specific static variable)

THREAD

The **THREAD** attribute declares a static variable which is allocated memory separately for each execution thread in the program. This makes the value contained in the variable dependent upon which thread is executing. Whenever a new execution thread is begun, a new instance of the variable, specific to that thread, is created.

The variable must be allocated static memory so it should be declared as Local data with the STATIC attribute. It may also be declared as Global data or Module data.

This attribute creates runtime "overhead," particularly on Global or Module data. Therefore, it should be used only when absolutely necessary.

Example:

```
GlobalVar LONG,THREAD      !Each execution thread gets its own copy

SomeProc PROCEDURE
LocalVar LONG,THREAD    !Local threaded variable (automatically STATIC)
```

See Also:

START

Data Declarations and Memory Allocation

STATIC

BINDABLE (set dynamic expression string variables)

BINDABLE

The **BINDABLE** attribute declares a GROUP, QUEUE, FILE, or VIEW whose constituent variables are all available for use in a runtime expression string. The contents of each variable´s NAME attribute is the logical name used in the runtime expression string. If no NAME attribute is present, the label of the variable (including prefix) is used. Space is allocated in the .EXE for the names of all of the variables in the structure. This creates a larger program that uses more memory than it normally would. Therefore, the BINDABLE attribute should only be used when a large proportion of the constituent fields are going to be used.

Example:

```
FileNames GROUP,BINDABLE            !Bindable group
FileName  STRING(8),NAME(´FILE´)    !Dynamic name: FILE
Dot       STRING(´.´)             !Dynamic name: Dot
Extension   STRING(3),NAME(´EXT´)     !Dynamic name: EXT
        END      !
```

See Also:

BIND

UNBIND

EVALUATE

AUTO (uninitialized local variable)

AUTO

The **AUTO** attribute allows a variable, declared within a PROCEDURE or FUNCTION, to be allocated uninitialized stack memory. Without the AUTO attribute, a numeric variable is initialized to zero and a string variable is initialized to all blanks when its memory is assigned at run-time.

The AUTO attribute is used when you do not need to rely on an initial blank or zero value because you intend to assign some other value to the variable. This saves a small amount of run-time memory by eliminating the internal code necessary to perform the automatic initialization for the variable.

Example:

```
SomeProc  PROCEDURE
SaveCustID  LONG,AUTO      !Non-initialized local variable
```

TYPE (GROUP type definition)

TYPE

The **TYPE** attribute creates a "type definition" for a GROUP. The type definition can then be used in a LIKE statement to define other similar GROUPs. A GROUP with the TYPE attribute is not allocated any memory.

Example:

```
PassGroup   GROUP,TYPE              !Type-definition for passed GROUP parameters
F1          STRING(20)            ! first field
F2          STRING(1)             ! middle field
F3          STRING(20)            ! last field
          END

NameGroup   LIKE(PassGroup),PRE(Nme) !Name group
```

Data Declarations and Memory Allocation

Global, Local, Static, and Dynamic

Data declarations allocate memory to store the data values. Global, Local, Static, and Dynamic are terms that describe types of memory allocation.

The terms "Global" and "Local" refer to the "visibility" of data:

> "Global" means the data is visible and available to all procedures in the program.
>
> "Local" means the data has limited visibility. This may be limited to one procedure or function, or limited to a specific set of procedures and/or functions.

The terms "Static" and "Dynamic" refer to the persistence of the data´s memory allocation:

> "Static" means the data is allocated memory that is not released until the entire program is finished executing.
>
> "Dynamic" means the data is allocated memory on the program´s stack. Stack memory is released when the PROCEDURE or FUNCTION that allocated the stack memory returns to the place in the program from which it was called.

Data Declaration Sections

There are three areas where data can be declared in a Clarion program:

In the PROGRAM module, after the keyword PROGRAM and before the CODE statement. This is the **Global data** section.

In a MEMBER module, after the keyword MEMBER and before the first PROCEDURE or FUNCTION statement. This is the **Module data** section.

In a PROCEDURE or FUNCTION, after the keyword PROCEDURE (or FUNCTION) and before the CODE statement. This is the **Local data** section.

Global data is visible to executable statements and expressions in every PROCEDURE and FUNCTION in the PROGRAM. Global data is allocated in Static memory.

Module data is visible only to the set of PROCEDUREs and FUNCTIONs contained in the MEMBER module. Of course, it may be passed as a parameter to PROCEDUREs or FUNCTIONs in other MEMBER modules, if required. Module data is also allocated Static memory.

Local data is visible only within the PROCEDURE or FUNCTION in which it is declared. Of course, it may be passed as a parameter to any other PROCEDURE or FUNCTION. Local data is allocated Dynamic memory on the program´s stack. This can be overridden by using the STATIC attribute, making its value persistent between calls to the procedure.

Dynamic memory allocation for Local data allows a FUNCTION or PROCEDURE to be truly recursive, receiving a new copy of its local variables each time it is called.

See Also:

FUNCTION and PROCEDURE Prototypes

STATIC

Picture Tokens

Picture tokens provide a masking format for displaying and editing variables. Picture tokens may be used as parameters of STRING, ENTRY, or STRING OPTION declarations in SCREEN structures; as a parameter of STRING statements in a REPORT structure; as a parameter of some Clarion procedures and functions; or, the parameter of STRING, CSTRING and PSTRING variable declarations.

There are seven types of picture tokens:

Numeric and Currency Pictures

@N [*currency*] [*sign*] [*fill*] *size* [*grouping*] [*places*] [*sign*] [*currency*] [**B**]

@N — All numeric and currency pictures begin with @N.

currency — Either a dollar sign ($) or a string constant enclosed in tildes (~). When it precedes the *sign* indicator and there is no *fill* indicator, the *currency* symbol "floats" to the left of the high order digit. If there is a *fill* indicator, the *currency* symbol remains fixed in the left-most position. If the *currency* indicator follows the *size* and *grouping*, it appears at the end of the number displayed.

sign — Specifies the display format for negative numbers. If a hyphen precedes the *fill* and *size* indicators, negative numbers will display with a leading minus sign. If a hyphen follows the *size*, *grouping*, *places*, and *currency* indicators, negative numbers will display with a trailing minus sign. If parentheses are placed in both positions, negative numbers will be displayed enclosed in parentheses.

fill — Specifies leading zeros, spaces, or asterisks (*) in any leading zero positions, and suppresses *grouping*. If the *fill* indicator is omitted, leading zeros are suppressed.

0 (zero) Produces leading zeroes
_ (underscore) Produces leading spaces
***** (asterisk) Produces leading asterisks

size — The *size* is required to specify the total number of significant digits to display, including the number of digits in the *places* indicator and any formatting characters.

grouping — A *grouping* symbol, other than a comma (the default), can be placed to the right of the *size* indicator to specify a three digit group separator.

. (period) Produces periods
- (hyphen) Produces hyphens
_ (underscore) Produces spaces

places — Specifies the decimal separator symbol and the number of decimal digits. The number of decimal digits must be less than the size indicator. The decimal separator may be a period (.), grave accent (´ -- produces periods for *grouping* separators, unless overridden), or the letter "v" (used only for STRING field storage declarations--not for display).

. (period) Produces a period
´ (grave accent) Produces a comma
v Produces no decimal separator

B — Specifies that the format displays as blank whenever its value is zero.

The numeric and currency pictures format numeric values for screen display or in reports. If the value is greater than the maximum value the picture can display, a string of asterisks is displayed.

Example:

```
Numeric              Result   Format
@N94,550,000Nine digits, group with commas (default)
@N_9B  4550000Nine digits, no grouping, leading blanks if zero
@N09 004550000Nine digits, leading zero
@N*9 ***45,000Nine digits, asterisk fill, group with commas
@N9_ 4 550 000Nine digits, group with spaces
@N9. 4.550.000Nine digits, group with periods
```

```
Decimal              Result    Format
@N9.2  4,550.75Two decimal places, period decimal separator
@N_9.2B             4550.75    Two decimal places, period decimal separator, no
   grouping, blank if zero
@N_9´2 4550,75Two decimal places, comma decimal separator
@N9.´2 4.550,75Comma decimal separator, group with periods
@N9_´2 4 550,75Comma decimal separator, group with spaces,

Signed Result         Format
@N-9.2B           -2,347.25    Leading minus sign, blank if zero
@N9.2- 2,347.25-Trailing minus sign
@N(10.2)         (2,347.25)    Enclosed in parens when negative

Dollar Currency     Result     Format
@N$9.2B           $2,347.25    Leading dollar sign, blank if zero
@N$10.2-         $2,347.25-    Leading dollar sign, trailing minus when negative
@N$(11.2)       $(2,347.25)    Leading dollar sign, in parens when negative

Int´l Currency             Result     Format
@N12_´2~ F~             1 5430,50 F    France
@N~L. ~12´             L. 1.430.050    Italy
@N~£~12.2                 £1,240.50    United Kingdom
@N~kr~12´2               kr1.430,50    Norway
@N~DM~12´2               DM1.430,50    Germany
@N12_´2~ mk~            1 430,50 mk    Finland
@N12´2~ kr~             1.430,50 kr    Sweden

Storage-Only Pictures:
Variable1 STRING(@N_6v2)           !Declare as 6 bytes stored without decimal
  CODE
  Variable1 = 1234.56              !Assign value, stores ´123456´ in file
  MESSAGE(FORMAT(Variable1,@N_7.2))  !Display with decimal point: ´1234.56´
```

Scientific Notation Pictures

@Em.n[B]

@E	All scientific notation pictures begin with @E.
m	Determines the total number of characters in the format provided by the picture.
n	Indicates the number of digits that appear to the left of the decimal point.
B	Specifies that the format displays as blank when the value is zero.

The scientific notation picture formats very large or very small numbers. The format is a decimal number raised by a power of ten.

Example:

```
Picture                   Value          Result
@E9.0  1,967,865       .20e+007
@E12.1 1,967,865    1.9679e+006
@E12.1B                       0
@E12.1 -1,967,865  -1.9679e+006
@E12.1 .000000032   3.2000e-008
```

Date Pictures

@Dn[s][B]

@D	All date pictures begin with @D.
n	Determines the date picture format. Date picture formats range from 1 through 16.
s	A separation character. Slash (/) characters appear between the month, day, and year components of certain date picture formats. Following are alternate separation characters. . (period) Produces periods ´ (grave accent) Produces commas - (hyphen) Produces hyphens _ (underscore) Produces spaces
B	Specifies that the format displays as blank when the value is zero.

Dates may be stored in numeric variables (usually LONG), a DATE field (for Btrieve compatibility), or in a STRING declared with a date picture. A date stored in a numeric variable is called a "Clarion Standard Date." The stored value is the number of days since December 28, 1800. The date picture token converts the value into one of the 16 date formats.

Example:

```
Picture              Format          Result
@D1 mm/dd/yy        10/31/59
@D2 mm/dd/yyyy    10/31/1959
@D3 mmm dd, yyyy OCT 31,1959
@D4 mmmmmmmmm dd, yyyyOctober 31, 1959
@D5 dd/mm/yy        31/10/59
@D6 dd/mm/yyyy    31/10/1959
@D7 dd mmm yy      31 OCT 59
@D8 dd mmm yyyy  31 OCT 1959
@D9 yy/mm/dd        59/10/31
@D10 yyyy/mm/dd  1959/10/31
@D11 yymmdd           591031
@D12 yyyymmdd      19591031
@D13 mm/yy            10/59
@D14   mm/yyyy      10/1959
@D15   yy/mm          59/10
@D16 yyyy/mm        1959/10
@D17   Windows Control Panel setting for Short Date
@D18   Windows Control Panel setting for Long Date

   Alternate separators
@D1. mm.dd.yyPeriod separator
@D2- mm-dd-yyyyDash separator
@D5_ dd mm yyUnderscore produces space separator
@D6´ dd,mm,yyyyGrave accent produces comma separator
```

See Also:

Standard Date

Time Pictures

@Tn[s][B]

@T	All time pictures begin with @T.
n	Determines the time picture format. Time picture formats range from 1 through 6.
s	A separation character. By default, colon (:) characters appear between the hour, minute, and second components of certain time picture formats. The following s indicators provide an alternate separation character for these formats. . (period) Produces periods ´ (grave accent) Produces commas - (hyphen) Produces hyphens _ (underscore) Produces spaces
B	Specifies that the format displays as blank when the value is zero.

Times may be stored in a numeric variable (usually a LONG), a TIME field (for Btrieve compatibility), or in a STRING declared with a time picture. A time stored in a numeric variable is called a "Standard Time." The stored value is the number of hundredths of a second since midnight. The picture token converts the value to one of the six time formats.

Example:

```
Picture              Format          Result
@T1 hh:mm              17:30
@T2 hhmm                1730
@T3 hh:mmXM            5:30PM
@T4 hh:mm:ss         17:30:00
@T5 hhmmss             173000
@T6 hh:mm:ssXM     5:30:00PM
@T7  Windows Control Panel setting for Short Time
@T8  Windows Control Panel setting for Long Time


   Alternate separators
@T1. hh.mm Period separator
@T1- hh-mm   Dash separator
@T3_ hh mmXM Underscore produces space separator
@T4´ hh,mm,ssGrave accent produces comma separator
```

See Also:

Standard Time

Pattern Pictures

@P[<][#][x]P[B]

@P	All pattern pictures begin with the @P delimiter and end with the P delimiter. The case of the delimiters must be the same.
<	Specifies an integer position that is blank when zero.
#	Specifies an integer position.
x	Represents optional display characters. These characters appear in the final result string.
P	All pattern pictures must end with P. If a lower case @p delimiter is used, the ending P delimiter must also be lower case.
B	Specifies that the format displays as blank when the value is zero.

Pattern pictures contain optional integer positions and optional edit characters. Any character other than < or # is considered an edit character which will appear in the formatted picture string. The @P and P delimiters are case sensitive. Therefore, an upper case "P" can be included as an edit character if the delimiters are both lower case "p" and vice versa.

Pattern pictures do not recognize decimal points, in order to permit the period to be used as an edit character. Therefore, the value formatted by a pattern picture should be an integer. If a floating piont value is formatted by a pattern picture, only the integer portion of the number will appear in the result.

Example:

Picture	Value	Result
@P###-##-####P	215846377	215-84-6377
@P<#/##/##P	103159	10/31/59
@P(###)###-####P	3057854555	(305)785-4555
@P###/###-####P	7854555	000/785-4555
@p<#:##PMp	530	5:30PM
@P<#´ <#"P	506	5´ 6"
@P<#lb. <#oz.P	902	9lb. 2oz.
@P4##A-#P	112	411A-2
@PA##.C#P	312.45	A31.C2

Key-in Template Pictures

@K[@][#][<][x][\][?][^][_][|]K[B]

@K	All key-in template pictures begin with the @K delimiter and end with the K delimiter. The case of the delimiters must be the same.
@	Specifies only uppercase and lowercase alphabetic characters.
#	Specifies an integer 0 through 9.
<	Specifies an integer that is blank for high order zeros.
x	Represents optional constant display characters (any displayable character). These characters appear in the final result string.
\	Indicates the following character is a display character. This allows you to include any of the picture formatting characters (@,#,<,\,?,^,_,|) within the string as a display character.
?	Specifies any character may be placed in this position.
^	Specifies only uppercase alphabetic characters in this position.
_	Underscore specifies only lowercase alphabetic characters in this position.
|	Allows the operator to "stop here" if there are no more characters to input. Only the data entered and any display characters up to that point will be in the string result.
K	All key-in template pictures must end with K. If a lower case @k delimiter is used, the ending K delimiter must also be lower case.
B	Specifies that the format displays as blank when the value is zero.

Key-in pictures may contain integer positions (# <), alphabet character positions (@ ^ _), any character positions (?), and display characters. Any character other than a formatting indicator is considered a display character, which appears in the formatted picture string. The @K and K delimiters are case sensitive. Therefore, an upper case "K" may be included as a display character if the delimiters are both lower case "k" and vice versa.

Key-in pictures are used specifically with STRING, PSTRING, and CSTRING fields to allow custom field editing control and validation. Using a key-in picture containing any of the alphabet indicators (@ ^ _) on a numeric entry field produces unpredictable results.

Using the Insert typing mode for a key-in picture could produce unpredictable results. Therefore, key-in pictures always receive data entry in Overwrite mode, even if the INS attribute is present.

Example:

Picture	Value Entered	Result String
@K###-##-####K	215846377	215-84-6377
@K#####|-####K	33064	33064
@K#####|-####K	330643597	33064-3597
@K<# ^^^ ##K	10AUG59	10 AUG 59
@K(###)@@@-##\@##K	305abc4555	(305)abc-45@55
@K###/?##-####K	7854555	000/785-4555
@k<#:##^Mk	530P	5:30PM
@K<#´ <#"K	506	5´ 6"
@K4#_#A-#K	1g12	41g1A-2

String Pictures

@S*length*

@S	All string pictures begin with @S.
length	Determines the number of characters in the picture format.

A string picture describes an unformatted string of a specific *length*.

Example:

```
Name STRING(@S20)   !A 20 character string field
```

Compiler Directives

EQUATE (assign label)

```
                    | label    |
label    EQUATE(    | constant | )
                    | picture  |
```

EQUATE Assigns a label to another label or constant.

label The *label* of any statement preceding the EQUATE statement. This is used to declare an alternate statement label.

constant A numeric or string *constant*. This is used to declare a shorthand label for a constant value. It also makes a constant easy to locate and change.

picture A *picture* token. This is used to declare a shorthand label for a picture token. However, the screen and report formatter in the Clarion Editor will not recognize the equated label as a valid picture.

The **EQUATE** directive assigns a label to another label or constant. It does not use any run-time memory. The label of an EQUATE directive cannot be the same as its parameter.

Example:

```
Init     EQUATE(SetUpProg)              !Set alias label
Off      EQUATE(0)                     !Off means zero
On       EQUATE(1)                     !On means one
PI       EQUATE(3.1415927)              !The value of PI
EnterMsg EQUATE('Press Ctrl-Enter to SAVE')
SocSecPic EQUATE(@P###-##-####P)          !Soc-sec number picture
```

See Also:

Reserved Words

SIZE (memory size in bytes)

```
          | variable |
SIZE(     | constant | )
          | picture  |
```

SIZE Supplies the amount of memory used for storage.

variable The label of a previously declared variable.

constant A numeric or string constant.

picture A picture token.

SIZE directs the compiler to supply the amount of memory (in bytes) used to store the *variable*, *constant*, or *picture*.

Example:

```
SavRec  STRING(1),DIM(SIZE(Cus:Record)
                              !Dimension the string to size of record

StringVar STRING(SIZE(´Clarion Software, Inc.´))
                              !A string long enough for the constant

LOOP I# = 1 TO SIZE(ParseString)   !Loop for number of bytes in the string

PicLen = SIZE(@P(###)###-####P)    !Save size of the picture
```

Expressions and Assignments

Expressions

An expression is a mathematical, string, or logical formula that produces a value. An expression may be the source variable of an assignment statement, a parameter of a procedure or function, a subscript of an array (a dimensioned variable), or the condition of an IF, CASE, LOOP, or EXECUTE structure. Expressions may contain constant values, variables, and function calls connected by logical and/or arithmetic or string operators.

Expression Evaluation

Expressions are evaluated in the standard algebraic order of operations. The precedence of operations is controlled by operator type and placement of parentheses. Each operation produces an (internal) intermediate value used in subsequent operations. Parentheses may be used to group operations within expressions. Expressions are evaluated beginning with the inner-most set of parentheses and working through to the outer-most set.

Precedence levels for expression evaluation, from highest to lowest, are:

Level 1 **( )**	Parenthetical Grouping
Level 2 **-**	Unary Minus (Negative sign)
Level 3 **function call**	Gets the RETURN value
Level 4 **^**	Exponentiation
Level 5 *** / %**	Multiplication, Division, Modulus Division
Level 6 **+** -	Addition, Subtraction
Level 7 **&**	Concatenation

Expressions may produce numeric values, string values, or logical values (true/false evaluation). An expression may contain no operators at all; it may be a single variable, constant value, or function call.

Arithmetic Operators

An arithmetic operator combines two operands arithmetically to produce an intermediate value. The operators are:

+	Addition (A **+** B gives the sum of A and B)
-	Subtraction (A **-** B gives the difference of A and B)
*****	Multiplication (A ***** B multiples A by B)
/	Division (A **/** B gives divides A by B)
^	Exponentiation (A **^** B gives A raised to power of B)
%	Modulus Division (A **%** B gives the remainder of A divided by B)

Logical Operators

A logical operator compares two operands or expressions and produces a true or false condition. There are two types of logical operators: conditional and Boolean. Conditional operators compare two values or expressions. Boolean operators connect string, numeric, or logical expressions together to determine true-false logic. Operators may be combined to create complex operators.

Conditional Operators

=	Equal sign
<	Less than
>	Greater than

Boolean Operators

NOT	Boolean NOT
~	Tilde (Logical NOT)
AND	Boolean AND
OR	Boolean OR
XOR	Boolean XOR (eXclusive OR)

Combined operators

<>	Not equal
~=	Not equal
NOT =	Not equal
<=	Less than or equal to
=<	Less than or equal to
~>	Not greater than
NOT >	Not greater than
>=	Greater than or equal to
=>	Greater than or equal to
~<	Not less than
NOT <	Not less than

During logical evaluation, any non-zero value indicates a true condition, and a null (blank) string or zero value indicates a false condition.

Example:

```
Logical Expression              Result
A = B      True when A is equal to B
A < B      True when A is less than B
A > B      True when A is greater than B
A <> B, A ~= B, A NOT = B      True when A is not equal to B
A ~< B, A >= B, A NOT < B      True when A is not less than B
A ~> B, A <= B, A NOT > B      True when A is not greater than B
~ A,NOT A                      True when A is null or zero
A AND B                        True when A is true and B is true
A OR B     True when A is true,or B is true,or both true
A XOR B                        True when A is true or B is true, but not both
```

Numeric Constants

Numeric constants are fixed numeric values. They may occur in data declarations, in expressions, and as parameters of procedures, functions, or attributes. A numeric constant may be represented in decimal (base 10--the default), binary (base 2), octal (base 8), hexadecimal (base 16), or scientific notation formats. Formatting characters, such as dollar signs and commas, are not permitted in numeric constants.

Decimal (base ten) numeric constants may contain an optional leading minus sign (hyphen character), an integer, and an optional decimal with a fractional component. Binary (base two) numeric constants may contain an optional leading minus sign, the digits 0 and 1, and a terminating B or b character. Octal (base eight) numeric constants contain an optional leading minus sign, the digits 0 through 7, and a terminating O or o character. Hexadecimal (base sixteen) numeric constants contain an optional leading minus sign, the digits 0 through 9, alphabet characters A through F (representing the numbers 10 through 15) and a terminating H or h character. If the left-most character is a letter A through F, a leading zero must be used.

Example:

```
-924 !Decimal constants
76.346
1011b!Binary constants
-1000110B
3403o!Octal constants
-7041312O
-1FFBh!Hexadecimal constants
0CD1F74FH
```

Numeric Expressions

Numeric expressions may be used as parameters of procedures or functions, the condition of IF, CASE, LOOP, or EXECUTE structures, or as the source portion of an assignment statement where the destination is a numeric variable. A numeric expression may contain arithmetic operators and the concatenation operator, but they may not contain logical operators. When used in a numeric expression, string constants and variables are converted to numeric intermediate values. If the concatenation operator is used, the intermediate value is converted to numeric after the concatenation occurs.

Example:

```
Count + 1           !Add 1 to Count
(1 - N * N) / R     !N times N subtracted from 1 then divided by R
305 & 7854555       !Concatenate area code with phone number
```

See Also:

Data Conversion Rules

String Constants

A string constant is a set of characters enclosed in single quotes (apostrophes). The maximum length of a string constant is 255 characters. Characters that cannot be entered from the keyboard may be inserted into a string constant by enclosing their ASCII character codes in angle brackets (<>). ASCII character codes may be represented in decimal or hexadecimal numeric constant format.

In a string constant, a left angle bracket (<) initiates a scan for a right angle bracket. Therefore, to include a left angle bracket in a string constant requires two left angle brackets in succession. To include an apostrophe as part of the value inside a string constant requires two apostrophes in succession. Two apostrophes (´´), with no characters (or just spaces) between them, represents a null, or blank, string. Consecutive occurrences of the same character within a string constant may be represented by *repeat count* notation. The number of times the character is to be repeated is placed within curly braces ({ }) immediately following the character to repeat. To include a left curly brace ({) as part of the value inside a string constant requires two left curly braces ({{) in succession.

Example:

```
´string constant´     !A string constant
´It´´s a girl!´       !With embedded apostrophe
´<27,15>´             !Using decimal ASCII codes
´A << B´              !With embedded left angle, A < B
´*{20}´               !Twenty asterisks, repeat-count notation
´´                 !A null (blank) string
```

The Concatenation Operator

The concatenation operator (&) is used to append one string or variable to another. The length of the result string is the sum of the lengths of the two values being concatenated. Numeric data types may be concatenated with strings or other numeric variables or constants. In many cases, the CLIP function should be used to remove any trailing spaces from a string being concatenated to another string.

Example:

```
CLIP(FirstName) & ´ ´ Initial & ´. ´ & LastName     !Concatenate full name
´Clarion Software´ & ´, Inc.´                    !Concatenate two constants
```

See Also:

CLIP

Numeric Expressions

Data Conversion Rules

String Expressions

String expressions may be used as parameters of procedures, functions, and attributes, or as the source portion of an assignment statement when the destination is a string variable. String expressions may contain a single string or numeric variable, or a complex combination of sub-expressions, functions, and operations.

Example:

```
StringVar   STRING(30)
Name      STRING(10)
Weight      STRING(3)
Phone       LONG
  CODE
  StringVar= ´Address:´ & Cus:Address       !Concatenate a constant and variable

  StringVar = ´Phone:´ & ´ 305-´ & FORMAT(Phone,@P###-####P)
                                    !Concatenate constant valuess
                                    ! and FORMAT function´s return value

  StringVar = Weight & ´lbs.´             !Concatenate a constant and variable
```

Implicit String Arrays and String Slicing

In addition to their explicit declaration, all STRING, CSTRING and PSTRING variables have an implicit array declaration of one character strings, dimensioned by the length of the string. This is directly equivalent to declaring a second variable as:

```
StringVar STRING(10)
StringArray STRING(1),DIM(SIZE(StringVar)),OVER(StringVar)
```

This implicit array declaration allows each character in the string to be directly addressed as an array element, without the need of the second declaration.

If the string also has a DIM attribute, this implicit array declaration is the last (optional) dimension of the array (to the right of the explicit dimensions). The MAXIMUM function does not operate on the implicit dimension, you should use SIZE instead.

You may also directly address multiple characters within a string using the "string slicing" technique. This technique performs a similar function to the SUB function, but is much more flexible and efficient. It is more flexible because a "string slice" may be used as either the *destination* or *source* sides of an assignment statement, while the SUB function can only be used as the source. It is more efficient because it takes less memory than either individual character assignments or the SUB function.

To take a "slice" of the string, the beginning and ending character numbers are separated by a colon (:) and placed in the implicit array dimension position within the square brackets ([]) of the string. The position numbers may be integer constants, variables, or expressions. If variables are used, there must be at least one blank space between the variable name and the colon separating the beginning and ending number (to prevent PREfix confusion).

Example:

```
Name     STRING(15)
CONTACT   STRING(15),DIM(4)
  CODE
  Name = ´Tammi´            !Assign a value
  Name[5] = ´y´             ! then change fifth letter
  Name[6] = ´s´             ! then add a letter
  Name[0] = ´<6>´             ! and handle length byte
  Name[5:6] = ´ie´          ! and change a "slice"
                          !  -- the fifth and sixth letters
  Contact[1] = ´First´        !Assign value to first element
  Contact[1,2] = ´u´        !Change first element 2nd character
  Contact[1,2:3] = Name[5:6] !Assign slice to first element 2nd & 3rd characters
```

See Also:

STRING

CSTRING

PSTRING

Logical Expressions

Logical expressions evaluate true-false conditions in IF, LOOP UNTIL, and LOOP WHILE control structures. Control is determined by the final result (true or false) of the expression. Logical expressions are evaluated from left to right. The right operand of an AND, OR, or XOR logical expression will only be evaluated if it could affect the result. Parentheses should be used to eliminate ambiguous evaluation and to control evaluation precedence. The level or precedence for the logical operators is as follows:

Level 1	Conditional operators
Level 2	~, NOT
Level 3	AND
Level 4	OR, XOR

Example:

```
LOOP UNTIL KEYBOARD()        !True when user presses any key
 !some statements
END

IF A = B THEN RETURN.        !RETURN if A is equal to B

LOOP WHILE ~ Done#          !Loop while false (Done# = 0)
 !some statements
END

IF A >= B OR (C > B AND E = D) THEN RETURN.
                           !True if a >= b, also true if
                           ! both c > b and e = d.
                           !The second part of the expression
                           ! (after OR) is evaluated only if the
                           ! first part is not true.
```

Runtime Expression Strings

Clarion Database Developer for Windows has the ability to evaluate Clarion language expressions dynamically created at runtime, rather than at development time. This allows a Clarion program to contruct expressions "on the fly." This also makes it possible to allow an end-user to enter the expression to evaluate.

An expression is a mathematical or logical formula that produces a value; it is not a complete Clarion language statement. Expressions may only contain constant values, variables, or function calls connected by logical and/or arithmetic operators. An expression may be used as the source side of an assignment statement, a parameter of a procedure or function, a subscript of an array (a dimensioned variable), or the conditions of IF, CASE, LOOP, or EXECUTE structures.

Any program variable, and most of the internal Clarion functions, can be used as part of a runtime expression string. User-defined functions that fall within certain specific guidelines (described in the BIND statement documentation) may also be used in runtime expression strings.

All of the standard Clarion expression syntax is available for use in runtime expression strings. This includes parenthetical grouping and all the arithmetic, logical, and string operators. Dynamic expressions are evaluated just as any other Clarion expression and all the standard operator precedence level rules described in the Expression Evaluation section apply.

It takes three steps to use runtime expression strings:

The variables that are allowed to be used in the expressions must be explicitly declared with the BIND statement.

The expression must be built. This may involve concatenating user choices or allowing the user to directly type in their own expression.

The expression is passed to the EVALUATE function which returns the result. If the expression is not a valid Clarion expression, ERRORCODE is set.

Once the expression is evaluated, its result is used just as the result of any hard-coded expression would be. For example, a runtime expression string could provide a filter expression to eliminate certain records when viewing or printing a database (the FILTER expression of a VIEW structure is an implicit runtime expression string).

See Also:

BIND (declare runtime expression string variable)

UNBIND (free runtime expression string variable)

EVALUATE (return runtime expression string result)

BIND (declare runtime expression string variable)

BIND(| *name,variable* | **)**
| *name,function* |
| *group* |

BIND Identifies variables allowed to be used in dynamic expressions.

name A string constant containing the identifier used in the dynamic expression. This may be the same as the *variable* or *function* label.

variable The label of any variable (including fields in FILE, GROUP, or QUEUE structures) or passed parameter. If it is an array, it must have only one dimension.

function The label of a Clarion language FUNCTION that returns a STRING, REAL, or LONG value. If parameters are passed to the function, they must be STRING value-parameters (passed by value, not by address).

group The label of a GROUP, RECORD, or QUEUE structure declared with the BINDABLE attribute.

The **BIND** statement declares the logical name used to identify a variable or user-defined function in runtime expression strings. A variable or user-defined function must be identified with the BIND statement before it can be used in an expression string.

BIND(*name,variable*)
The specified *name* is used in the expression in place of the label of the *variable*.

BIND(*name,function*)
The specified *name* is used in the expression in place of the label of the *function*.

BIND(*group*) Declares all the variables within the GROUP, RECORD, or QUEUE (with the BINDABLE attribute) available for use in a dynamic expression. The contents of each variable´s NAME attribute is the logical name used in the dynamic expression. If no NAME attribute is present, the label of the variable (including prefix) is used.

A GROUP, RECORD, or QUEUE structure declared with the BINDABLE attribute has space allocated in the .EXE for the names of all of the data elements in the structure. This creates a larger program that uses more memory than it normally would. Also, the more variables that are bound at one time, the slower the EVALUATE function will work. Therefore, BIND(*group*) should only be used when a large proportion of the constituent fields are going to be used.

Example:

```
  PROGRAM
  MAP
    AllCapsFunc(STRING),STRING              !Clarion function
  END

Header   FILE,DRIVER(´Clarion´),PRE(Hea)    !header file layout
AcctKey       KEY(Hea:AcctNumber)
OrderKey    KEY(Hea:OrderNumber)
Record      RECORD
AcctNumber      LONG
OrderNumber     LONG
ShipToName      STRING(20)
ShipToAddr      STRING(20)
ShipToCity      STRING(20)
ShipToState     STRING(20)
```

```
ShipToZip       STRING(20)
          . .

Detail    FILE,DRIVER(´Clarion´),PRE(Dtl),BINDABLE    !Bindable RECORD structure
OrderKey    KEY(Dtl:OrderNumber)
Record      RECORD
OrderNumber     LONG
Item          LONG
Quantity      SHORT
          . .

  CODE
  BIND(´ShipName´,Hea:ShipToName)      !BIND a single variable
  BIND(Dtl:Record)                   !BIND a RECORD structure
  BIND(´SomeFunc´,AllCapsFunc)        !BIND a Clarion language function
  IF EVALUATE(´ShipName = SomeFunc(ShipName)´)
    MESSAGE(´Name is in ALL CAPS´)
  END

AllCapsFunc FUNCTION(PassedString)
  CODE
  RETURN(UPPER(PassedString))
```

See Also:

UNBIND

EVALUATE

UNBIND (free runtime expression string variable)

UNBIND([*name*] **)**

UNBIND Frees variables from use in runtime expression strings.

name A string constant that specifies the identifier used by the dynamic expression evaluator. If omitted, all bound variables are unbound.

The **UNBIND** statement frees logical names previously bound by the BIND statement. The more variables that are bound at one time, the slower the EVALUATE function works. Therefore, UNBIND should be used to free all variables and user-defined functions not currently available for use in runtime expression strings.

Example:

```
 PROGRAM
 MAP
   AllCapsFunc(STRING),STRING              !Clarion function
 END

Header    FILE,DRIVER(´Clarion´),PRE(Hea)   ! header file layout
AcctKey      KEY(Hea:AcctNumber)
OrderKey    KEY(Hea:OrderNumber)
Record      RECORD
AcctNumber     LONG
OrderNumber    LONG
ShipToName     STRING(20)
ShipToAddr     STRING(20)
ShipToCity     STRING(20)
ShipToState    STRING(20)
ShipToZip      STRING(20)
          . .

Detail    FILE,DRIVER(´Clarion´),PRE(Dtl),BINDABLE    !Bindable RECORD structure
OrderKey    KEY(Dtl:OrderNumber)
Record      RECORD
OrderNumber    LONG
Item          LONG
Quantity      SHORT
          . .

  CODE
  BIND(´ShipName´,Hea:ShipToName)
  BIND(Dtl:Record)
  BIND(´SomeFunc´,AllCapsFunc)
  UNBIND(´ShipName´)             !UNBIND the variable
  UNBIND(´SomeFunc´)             !UNBIND the Clarion language function
  UNBIND                         !UNBIND all bound variables

AllCapsFunc FUNCTION(PassedString)
  CODE
  RETURN(UPPER(PassedString))
```

See Also:

BIND

EVALUATE

EVALUATE (return runtime expression string result)

EVALUATE(*expression***)**

EVALUATE Evaluates runtime expression strings.

expression A string constant or variable containing the expression to evaluate.

The **EVALUATE** function returns the result of the *expression* as a STRING value. If the *expression* does not meet the rules of a valid Clarion expression, the result will be a null string, and the ERRORCODE function is set.

The more variables are bound at one time, the slower the EVALUATE function works. Therefore, BIND(*group*) should only be used when most of the *group´s* fields are needed, and UNBIND should be used to free all variables and user-defined functions not currently required for use in dynamic expressions.

Return Data Type: STRING

Errors Posted: 800 Illegal Expression
801 Variable Not Found

Example:

```
 PROGRAM
 MAP
   AllCapsFunc(STRING),STRING                          !Clarion function
 END
Header    FILE,DRIVER(´Clarion´),PRE(Hea),BINDABLE    !Declare header file layout
AcctKey      KEY(Hea:AcctNumber)
OrderKey    KEY(Hea:OrderNumber)
Record      RECORD
AcctNumber     LONG
OrderNumber    LONG
ShipToName     STRING(20)
ShipToAddr     STRING(20)
ShipToCity     STRING(20)
ShipToState    STRING(20)
ShipToZip      STRING(20)
          . .
StringVar     STRING(20)
  CODE
  BIND(´ShipName´,Hea:ShipToName)
  BIND(´SomeFunc´,AllCapsFunc)
  StringVar = ´SMITH´
  IF EVALUATE(´StringVar = SomeFunc(ShipName)´)
    DO SmithProcess
  END
AllCapsFunc FUNCTION(PassedString)
  CODE
  RETURN(UPPER(PassedString))
```

See Also:

BIND

UNBIND

Assignment Statements

Simple Assignment Statements

destination **=** *source*

---

destination The label of a variable or data structure property.

source A numeric or string constant, variable, function, expression, or data structure property.

The **=** sign assigns the value of *source* to the *destination*; it copies the value of the *source* expression into the *destination* variable. If *destination* and *source* are different data types, the value the *destination* receives from the *source* is dependent upon the Data Conversion Rules.

Example:

```
Name = ´JONES´                    !Variable = string constant
PI = 3.14159                      !Variable = numeric constant
Cosine = SQRT(1 - Sine * Sine)       !Variable = function return value
A = B + C + 3                     !Variable = numeric expression
Name = CLIP(FirstName) & ´ ´ Initial & ´. ´ & LastName
                                  !Variable = string expression
```

See Also:

Data Conversion Rules

Operating Assignment Statements

destination += *source*
destination -= *source*
destination *= *source*
destination /= *source*
destination ^= *source*
destination %= *source*

---

destination Must be the label of a variable.

source A constant, variable, function, or expression.

Operating assignment statements perform their operation on the *destination* and *source,* assigning the result to the *destination*. Operating assignment statements are more efficient than their equivalent operations.

Example:

```
Operating AssignmentFunctional Equivalent
A += 1              A = A + 1
A -= B              A = A - B
A *= -5             A = A * -5
A /= 100            A = A / 100
A ^= I + 1          A = A ^ (I + 1)
A %= 7              A = A % 7
```

Deep Assignment Statements

destination **:=:** *source*

---

destination	The label of a GROUP, RECORD, or QUEUE data structure, or an array.
source	The label of a GROUP, RECORD, or QUEUE data structure, or a numeric or string constant, variable, function, or expression.

The **:=:** sign executes a deep assignment statement which performs multiple individual component variable assignments from one data structure to another. The assignments are only performed between the variables within each structure that have exactly matching labels, ignoring all prefixes. The compiler looks within nested GROUP structures to find matching labels. Any variable in the *destination* which does not have a label exactly matching a variable in the *source*, is not changed.

Deep assignments are performed just as if each matching variable were individually assigned to its matching variable. This means that all normal data conversion rules apply to each matching variable assignment. For example, the label of a nested *source* GROUP may match a nested *destination* GROUP or simple variable. In this case, the nested *source* GROUP is assigned to the *destination* as a STRING, just as normal GROUP assignment is handled.

The name of a *source* array may match a *destination* array. In this case, each element of the *source* array is assigned to its corresponding element in the *destination* array. If the *source* array has more or fewer elements than the *destination* array, only the matching elements are assigned to the *destination*.

If the *destination* is an array variable that is not part of a GROUP, RECORD, or QUEUE, and the *source* is a constant, variable, or expression, then each element of the *destination* array is initialized to the value of the *source*. This is a much more efficient method of initializing an array to a specific value than using a LOOP structure and assigning each element in turn.

Example:

```
Group1 GROUP,PRE(G1)
S        SHORT
L        LONG
       END

Group2 GROUP,PRE(G2)
L        SHORT
S        REAL
T        LONG
       END

ArrayField  SHORT,DIM(1000)

  CODE
  Group2 :=: Group1     !Is equivalent to:
                    !  G2:S = G1:S
                    !  G2:L = G1:L
                    ! and performs all necessary data conversion

  ArrayField :=: 7    !Is equivalent to:
                    !   LOOP I# = 1 to 1000
                    !     ArrayField[I#] = 7
                    !   END
```

Reference Assignment Statements

destination &= *source*

---

destination	The label of a reference variable.
source	The label of another reference variable of the same type as the *destination*, or the label of a variable or data structure of the type referenced by the *destination*. This cannot be an expression, only a data label.

The **&=** sign executes a reference assignment statement which assigns to the *destination* reference variable the reference to the *source* variable. Depending upon the data type, the *destination* reference variable may receive the *source´s* memory address, or a more complex internal data structure (describing the location and type of *source* data).

The declarations of the *destination* reference variable and its *source* must match exactly; reference assignment does not perform automatic type conversion. For example, a reference assignment statement to a *destination* declared as &GROUP must have a *source* that is either another &GROUP reference variable, or the label of a GROUP structure.

Example:

```
Group1   GROUP,PRE(G1)
ShortVar  SHORT
LongVar1  LONG
LongVar2  LONG
         END

GroupRef &GROUP     !Reference a GROUP, only
LongRef   &LONG        !Reference a LONG, only

  CODE
  GroupRef &= Group1           !Assign GROUP reference
  IF SomeCondition            !Evaluate some condition
    LongRef &= G1:LongVar1     ! and reference an appropriate variable
  ELSE
    LongRef &= G1:LongVar2
  END
  LongRef += 1                !Increment either LongVar1 or LongVar2
                             ! depending upon which variable is referenced
```

See Also:

Reference Variables

CLEAR (clear a variable)

CLEAR(*label* [,*n*]**)**

CLEAR	Clears any value from a variable.
label	The label of a variable.
n	A numeric constant; 1 or -1. This parameter indicates a cleared value other than zero or blank. If *n* is 1, the variable is set to the highest possible value for that data type. For STRING, PSTRING and CSTRING, that is ASCII 255. If *n* is -1, the variable is set to the lowest possible value for that data type. For STRING, PSTRING and CSTRING, that is ASCII 0.

The **CLEAR** statement clears any value from the *label* variable. If *n* is omitted, numeric variables are cleared to zero, and string variables are cleared to spaces. If the *label* parameter is a GROUP, RECORD, or QUEUE structure name, all variables in the structure are cleared. If the variable has a DIM attribute, the entire array is cleared. A single element of an array cannot be CLEARed.

Example:

```
CLEAR(Count)         !Clear a variable
CLEAR(Cus:Record)      !Clear the record structure
CLEAR(Amount,1)        !Clear variable to highest possible value
CLEAR(Amount,-1)     !Clear variable to lowest possible value
```

Data Conversion Rules

The Clarion language provides automatic conversion between data types. However, some assignments can produce an unequal source and destination. Assigning an "out of range" value can produce unpredictable results.

Base Types

To facilitate automatic data type conversion, Clarion internally uses four Base Types to which all data items are automatically converted when any operation is performed on the data. These types are: STRING, LONG, DECIMAL, and REAL.These are all standard Clarion data types.

The STRING Base Type is used as the intermediate type for all string operations. The LONG, DECIMAL, and REAL Base Types are used in all arithmetic operations. Which numeric type is used, and when, is determined by the original data types of the operands and the type of operation being performed on them.

The "normal" Base Type for each data type is:

```
Base Type LONG:
    BYTE
    SHORT
    USHORT
    LONG
    DATE
    TIME
    Integer Constants

Base Type DECIMAL:
    ULONG
    DECIMAL
    PDECIMAL
    STRING(@Nx.y)
    Decimal Constants

Base Type REAL:
    SREAL
    REAL
    BFLOAT4
    BFLOAT8
    STRING(@Ex.y)
    Scientific Notation Constants
    Untyped (? and *?) Parameters

Base Type STRING:
    STRING
    CSTRING
    PSTRING
    String Constants
```

DATE and TIME data types are first converted to Clarion Standard Date and Clarion Standard Time intermediate values and have a LONG Base Type for all operations.

For the most part, Clarion´s internal use of these Base Types is transparent to the programmer and do not require any consideration when planning applications. However, for business programming with numeric data containing fractional portions (currency, for instance), using data types that have the DECIMAL Base Type has some significant advantages over REAL Base Types.

- DECIMAL supports 31 significant digits of accuracy for data storage while REAL only supports 15.
- DECIMAL automatically rounds to the precision specified by the data declaration, while REAL can create rounding problems due to the transalation of decimal (base 10) numbers to binary (base 2) for processing by the CPU´s Floating Point Unit (or Floating Point emulation software).

- On machines without a Floating Point Unit, DECIMAL is substantially faster than REAL.
- DECIMAL operations are closely linked with conventional (decimal) arithmetic.

BCD Operations and Functions

Clarion has a Binary Coded Decimal (BCD) library of operations and functions that execute in a manner similar to the manner in which decimal arithmetic is performed on paper. These operations use internal intermediate values with 31 digits accuracy on both sides of the decimal point.

The big advantage of the BCD operations is that it is very easy to "see" what is happening because they execute just as you would with pencil and paper. Simply imagine doing the computation long hand and throwing away numbers that go off the end of the page (rounding to the right).

Having 31 fixed decimal places either side of the decimal point there are numbers that cannot be represented in a BCD system which can be represented by a REAL. Therefore, understanding what is going on is useful.

Generally, the only cases where underflow will affect you is in division operations, usually when dividing by a multiple of 3. For example:

```
100000/3 = 33333.3333333333333333333333333333333
(100000/3)-INT(100000/3)*100000 =
33333.3333333333333333333333333300000
```

BCD computation times are very data sensitive; the time taken is proportional to how long the computation would take you by hand. Therefore, the longer the numbers involved, the longer the execution times. However, standard "tricks of the trade" (such as multiplying by a power of ten by shifting the decimal point) are spotted, making the BCD libraries fast in real world applications.

On a 486DX computer, BCD addition and subtraction is fractionally faster than casting to and from REALs. On chips without a math co-processor, the BCD libraries are MUCH faster than floating point operations.

The following operations may execute as BCD operations:

Addition (+), **Subtraction** (-), **Multiplication** (*)
: Performed as a BCD operation when neither operand has a REAL Base Type (both are LONG or DECIMAL) and one has the DECIMAL Base Type. Any digits appearing to the right of 1e31 disappear (wrap), and any to the left of 1e-30 are rounded up.

Division (/)
: Performed as a BCD operation when neither operand has a REAL Base Type (both are LONG or DECIMAL). Any digits appearing to the right of 1e31 disappear (wrap), and any to the left of 1e-30 are rounded up.

Exponentiation (^)
: Performed as a BCD operation when the first operand is a DECIMAL or LONG Base Type and the second operand is a LONG Base Type. Any digits appearing to the right of 1e31 disappear (wrap), and any to the left of 1e-30 are rounded.

ABS()
: Removes the sign from a DECIMAL variable or intermediate value and returns the DECIMAL value.

INT()
: Truncates a DECIMAL intermediate value and returns a DECIMAL value.

ROUND()
: If the second parameter is a LONG or DECIMAL Base Type, then rounding is performed as a BCD operation which returns a DECIMAL value. ROUND is very efficient as a BCD operation and should be used to compare REALs to DECIMALs at decimal width.

Type Conversion and Intermediate Results

Internally, a BCD intermediate result may have up to 31 digits of accuracy on both sides of the decimal point, so any two DECIMALs can be added with complete accuracy. Therefore, storage from BCD intermediate results to a data type can result in loss of precision. This is handled as follows :-

Decimal(x,y) = BCD
First the BCD value is rounded to y decimal places. If the result overflows x digits then leading digits are removed (this corresponds to "wrapping around" a decimal counter).

Integer = BCD Any digits to the right of the decimal point are ignored. The decimal is then converted to an integer with complete accuracy and then taken modulo 2^32.

String(@Nx.y) = BCD
The BCD value is rounded to y decimal places, the result is fitted into the pictured string. If overflow occurs, an invalid picture (####) results.

Real = BCD The most significant 15 digits are taken and the decimal point ´floated´ accordingly.

For those operations and functions that do not support DECIMAL types, the DECIMAL is converted to REAL first. In cases where more than 15 digits were available in the DECIMAL value, there is a loss of accuracy.

Note: Untyped parameters have an implicit REAL Base Type, therefore DECIMAL Base Type data passed as an Untyped Parameters will only have 15 digits of precision. DECIMAL Base Types can be passed as *DECIMAL parameters with no loss of precision.

When EVALUATEing a expression (or processing a VIEW FILTER) the REAL Base Type is used.

Simple Assignment Data Conversion

BYTE =

SHORT =

USHORT =

LONG =

DATE =

TIME =

ULONG =

REAL =

SREAL =

BFLOAT8 =

BFLOAT4 =

DECIMAL =

PDECIMAL =

STRING =

CSTRING =

PSTRING =

BYTE =

(SHORT, USHORT, LONG, or ULONG)

The destination receives the low-order 8 bits of the source.

(DECIMAL, PDECIMAL, REAL, SREAL, BFLOAT8, or BFLOAT4)

The source is first converted to a LONG, which truncates any decimal portion, then the destination receives the low-order 8 bits of the LONG.

(STRING, CSTRING, or PSTRING)

The source must be a numeric value with no formatting characters. The source is converted to a LONG, which truncates any decimal portion, then the destination receives the low-order 8 bits of the LONG.

SHORT =

BYTE The destination receives the value of the source.

(USHORT, LONG, or ULONG)
The destination receives the low-order 16 bits of the source.

(DECIMAL, PDECIMAL, REAL, SREAL, BFLOAT8, or BFLOAT4)
The source is first converted to a LONG, which truncates any decimal portion, then the destination receives the low-order 16 bits of the LONG.

(STRING, CSTRING, or PSTRING)
The source must be a numeric value with no formatting characters. The source is first converted to a LONG, which truncates any decimal portion, then the destination receives the low-order 16 bits of the LONG.

USHORT =

BYTE The destination receives the value of the source.

(SHORT, LONG, or ULONG)
The destination receives the low-order 16 bits of the source.

(DECIMAL, PDECIMAL, REAL, SREAL, BFLOAT8, or BFLOAT4)
The source is first converted to a LONG, which truncates any decimal portion, then the destination receives the low-order 16 bits of the LONG.

(STRING, CSTRING, or PSTRING)
The source must be a numeric value with no embedded formatting characters. The source is first converted to a LONG, which truncates any decimal portion, then the destination receives the low-order 16 bits of the LONG.

LONG =

(BYTE, SHORT, USHORT, or ULONG)

The destination receives the value and the sign of the source.

(DECIMAL, PDECIMAL, REAL, SREAL, BFLOAT8, or BFLOAT4)

The destination receives the value of the source, including the sign, up to 2_{31}. If the number is greater than 2_{31}, the destination receives the result of modulo 2_{31}. Any decimal portion is truncated.

(STRING, CSTRING, or PSTRING)

The source must be a numeric value with no embedded formatting characters. The source is first converted to a REAL, which is then converted to the LONG.

DATE =

(BYTE, SHORT, USHORT, or ULONG)

The destination receives the Btrieve format for the Clarion Standard Date for the value of the source.

(DECIMAL, PDECIMAL, REAL, SREAL, BFLOAT8, or BFLOAT4)

The source is first converted to a LONG as a Clarion Standard Date, which truncates any decimal portion, then the destination receives the Btrieve format for the Clarion Standard Date.

(STRING, CSTRING, or PSTRING)

The source must be a numeric value with no embedded formatting characters. The source is first converted to a LONG as a Clarion Standard Date, which truncates any decimal portion, then the destination receives the Btrieve format for the Clarion Standard Date.

TIME =

(BYTE, SHORT, USHORT, or ULONG)

The destination receives the Btrieve format for the Clarion Standard Time for the value of the source.

(DECIMAL, PDECIMAL, REAL, SREAL, BFLOAT8, or BFLOAT4)

The source is first converted to a LONG as a Clarion Standard Time, which truncates any decimal portion, then the destination receives the Btrieve format for the Clarion Standard Time.

(STRING, CSTRING, PSTRING)

The source must be a numeric value with no embedded formatting characters. The source is first converted to a LONG as a Clarion Standard Time, which truncates any decimal portion, then the destination receives the Btrieve format for the Clarion Standard Time.

ULONG =

(BYTE, SHORT, or USHORT)

The source is first converted to a LONG, then the destination receives the entire 32 bits of the LONG.

LONG The destination receives the entire 32 bits of the source.

(DECIMAL, PDECIMAL, REAL, SREAL, BFLOAT8, or BFLOAT4)

The source is first converted to a LONG, which truncates any decimal portion, then the destination receives the entire 32 bits of the LONG.

(STRING, CSTRING, or PSTRING)

The source must be a numeric value with no embedded formatting characters. The source is first converted to a LONG, which truncates any decimal portion, then the destination receives the entire 32 bits of the LONG.

REAL =

(BYTE, SHORT, USHORT, LONG, or ULONG)

The destination receives the full integer portion and the sign of the source.

(DECIMAL, PDECIMAL, SREAL, BFLOAT8, or BFLOAT4)

The destination receives the sign, integer portion, and the decimal portion of the source.

(STRING, CSTRING, PSTRING)

The source must be a numeric string value with no embedded formatting characters. The destination receives the sign, integer, and decimal portion of the number. Trailing spaces are ignored.

SREAL =

(BYTE, SHORT, USHORT, LONG, or ULONG)

The destination receives the sign and value of the source.

(DECIMAL, PDECIMAL, or REAL)

The destination receives the sign, integer, and fractional portion of the source.

(STRING, CSTRING, or PSTRING)

The source must be a numeric string value with no embedded formatting characters. The destination receives the sign, integer, and decimal portion of the number. Trailing spaces are ignored.

BFLOAT8 =

(BYTE, SHORT, USHORT, LONG, or ULONG)

The destination receives the sign and value of the source.

(DECIMAL, PDECIMAL, or REAL)

The destination receives the sign, integer, and fractional portion of the source.

(STRING, CSTRING, or PSTRING)

The source must be a numeric string value with no embedded formatting characters. The destination receives the sign, integer, and decimal portion of the number. Trailing spaces are ignored.

BFLOAT4 =

(BYTE, SHORT, USHORT, LONG, or ULONG)

The destination receives the sign and value of the source.

(DECIMAL, PDECIMAL, or REAL)

The destination receives the sign, integer, and fractional portion of the source.

(STRING, CSTRING, or PSTRING)

The source must be a numeric string value with no embedded formatting characters. The destination receives the sign, integer, and decimal portion of the number. Trailing spaces are ignored.

DECIMAL =

(BYTE, SHORT, USHORT, LONG, ULONG, or PDECIMAL)

The destination receives the sign and the value of the source, wrapping or rounding as appropriate.

(REAL, or SREAL)

The destination receives the sign, integer, and the high order part of the fraction from the source. The high order fractional portion is rounded in the destination.

(STRING, CSTRING, PSTRING)

The source must be a numeric string value with no embedded formatting characters. The destination receives the sign, integer, and decimal portion of the number. Trailing spaces are ignored.

PDECIMAL =

(BYTE, SHORT, USHORT, LONG, ULONG, or DECIMAL)

The destination receives the sign and the value of the source, wrapping or rounding as appropriate.

(REAL, SREAL, BFLOAT8, or BFLOAT4)

The destination receives the sign, integer, and the high order part of the fraction from the source. The high order fractional portion is rounded in the destination.

(STRING, CSTRING, or PSTRING)

The source must be a numeric string value with no embedded formatting characters. The destination receives the sign, integer, and decimal portion of the number. Trailing spaces are ignored.

STRING =

(BYTE, SHORT, USHORT, LONG, or ULONG)

The destination receives the sign and the unformatted number. The value is left justified in the destination.

(DECIMAL, PDECIMAL, REAL, SREAL, BFLOAT8, or BFLOAT4)

The destination receives the sign, integer, and fractional portion of the source (rounded into the string´s picture format). The value is left justified in the destination.

CSTRING =

(BYTE, SHORT, USHORT, LONG, or ULONG)

The destination receives the sign and the unformatted number. The value is left justified in the destination.

(DECIMAL, PDECIMAL, REAL, SREAL, BFLOAT8, or BFLOAT4)

The destination receives the sign, integer, and fractional portion of the source (rounded into the string´s picture format). The value is left justified in the destination.

PSTRING =

(BYTE, SHORT, USHORT, LONG, or ULONG)

The destination receives the sign and the unformatted number. The value is left justified in the destination.

(DECIMAL, PDECIMAL, REAL, SREAL, BFLOAT8, or BFLOAT4)

The destination receives the sign, integer, and fractional portion of the source (rounded into the string´s picture format). The value is left justified in the destination.

Program Control Statements

Control Structures

- CASE (conditional execution structure)
- EXECUTE (statement selection structure)
- IF (conditional execution structure)
- LOOP (iteration structure)

CASE (conditional execution structure)

CASE *condition*
OF *expression* [**TO** *expression*]
  statements
[**OROF** *expression* [**TO** *expression*]]
  statements
[**ELSE**]
  statements
END

CASE	Initiates a selective execution structure.
condition	A numeric or string variable or expression.
OF	The *statements* following an **OF** are executed when the *expression* following the OF option is equal to the *condition* of the CASE. There may be many OF options in a CASE structure.
expression	A numeric or string constant, variable, or expression.
TO	**TO** allows a range of values in an OF or OROF. The *statements* following the OF (or OROF) are executed if the value of the *condition* falls within the inclusive range specified by the *expressions*. The *expression* following OF (or OROF) must contain the lower limit of the range. The *expression* following TO must contain the upper limit of the range.
OROF	The *statements* following an **OROF** are executed when either the *expression* following the OROF or the OF option is equal to the *condition* of the CASE. There may be many OROF options associated with one OF option. An OROF may optionally be put on a separate line. An OROF does not terminate preceding *statements* groups, so control "falls into" the OROF *statements*.
ELSE	The *statements* following **ELSE** are executed when all preceding OF and OROF options have been evaluated as not equivalent. ELSE is not required; however, when used, it must be the last option in the CASE structure.
statements	Any valid Clarion executable source code.

A **CASE** structure selectively executes *statements* based on equivalence between the *condition* and *expression* or range of *expressions*. CASE structures may be nested within other executable structures and other executable structures may be nested within CASE structures.

Example:

```
CASE ACCEPTED()                    !Evaluate field edit routine
OF ?Name                         !If field is Name
  ERASE(?Address,?Zip)             ! erase Address through Zip
  GET(NameFile,NameKey)            ! get the record

  CASE Action                     !Evaluate Action
  OF 1                           ! adding record - does not exist
    IF NOT ERRORCODE()            ! should be a file error
      ErrMsg = ´ALREADY ON FILE´   !  otherwise display error message
      DISPLAY(?Address,?Zip)       !  display address through zipcode
      SELECT(?Name)               !  re-enter the name
    END
  OF 2 OROF 3                      ! change or delete - record exists
    DISPLAY(?Address,?Zip)         !  display address through zipcode
  END                            ! end case action
```

```
  CASE Name[1]                    !Get first letter of name
  OF ´A´ TO ´M´                   !Process first half of alphabet
  OROF ´a´ TO ´m´
    DO FirstHalf
  OF ´N TO ´Z´ OROF ´n´ TO ´z´    !Process second half of alphabet
    DO SecondHalf
  END                            !End case sub(name

OF ?Address                       !If field is address
  DO AddressVal                   ! call validation routine
END                              !End case accepted()
```

EXECUTE (statement selection structure)

```
EXECUTE expression
  statement 1
  statement 2
  [ BEGIN
        statements
  END ]
  statement n
END
```

EXECUTE	Initiates a single statement execution structure.
expression	A numeric expression or a variable that contains a numeric integer.
statement 1	A single statement that executes only when the *expression* is equal to 1.
statement 2	A single statement that executes only when the *expression* is equal to 2.
BEGIN	BEGIN marks the beginning of a structure containing a number of lines of code. The BEGIN structure will be treated as a single statement by the EXECUTE structure. The BEGIN structure is terminated by a period or the keyword END.
statement n	A single statement that executes only when the *expression* is equal to *n*.

An **EXECUTE** structure selects a single executable statement (or executable code structure) based on the value of the *expression*.

If the *expression* equals 1, the first statement (*statement 1*) executes. If *expression* equals 2, the second statement (*statement 2*) executes, and so on. If the value of the *expression* is zero, or greater than the total number of statements (or structures) within the EXECUTE structure, the EXECUTE is ignored.

EXECUTE structures may be nested within other executable structures and other executable structures (IF, CASE, LOOP, EXECUTE, and BEGIN) may be nested within an EXECUTE.

Example:

```
EXECUTE Transact        !Evaluate Transact
  ADD(Customer)         !Execute if Transact = 1
  PUT(Customer)         !Execute if Transact = 2
  DELETE(Customer)      !Execute if Transact = 3
END                   !End execute

EXECUTE CHOICE()        !Evaluate CHOICE() function
  OrderPart             !Execute if CHOICE() = 1
  BEGIN                 !Execute if CHOICE() = 2
    SavVendor" = Vendor
    UpdVendor
    IF Vendor <> SavVendor"
      Mem:Message = ´VENDOR NAME CHANGED´
  . .
  CASE VendorType         !Execute if CHOICE() = 3
  OF 1
    UpdPartNo1
  OF 2
    UpdPartNo2
  END
  RETURN              !Execute if CHOICE() = 4
END                   !End execute
```

See Also:

BEGIN

IF (conditional execution structure)

IF *logical expression* [**THEN**]
  statements
[**ELSIF** *logical expression* [**THEN**]
  statements]
[**ELSE**
  statements]
END

IF Initiates a conditional statement execution structure.

logical expression A numeric or string variable, expression, or function. A *logical expression* evaluates a condition. Control is determined by the result (true or false) of the expression. A zero (or blank) value evaluates as false, anything else is true.

THEN The *statements* following THEN are executed when the preceding *logical expression* is evaluated as true. If used, THEN must only be placed on the same line as the **IF** or **ELSIF**.

statements An executable statement, or a sequence of executable statements.

ELSIF The *logical expression* following an **ELSIF** is evaluated only when all preceding **IF** or **ELSIF** conditions were evaluated as false.

ELSE The *statements* following **ELSE** are executed when all preceding **IF** and **ELSIF** options were evaluated as false. ELSE is not required, however, when it is used, it must be the last option in the IF structure.

An **IF** structure controls program execution based on the outcome of one or more *logical expressions*. IF structures may have any number of ELSIF THEN statement groups. IF structures may be "nested" within other executable structures, and other executable structures may be nested within an IF structure.

Example:

```
IF Cus:TransCount = 1                          !If new customer
  AcctSetup                                   ! call account setup procedure
ELSIF Cus:TransCount > 10 AND Cus:TransCount < 100  !If regular customer
  DO RegularAcct                              ! process the account
ELSIF Cus:TransCount > 100                     !If special customer
  DO SpecialAcct                              ! process the account
ELSE                                         !Otherwise
  DO NewAcct                                   ! process the account
  IF Cus:Credit THEN CheckCredit ELSE CLEAR(Cus:CreditStat).
                                             ! verify credit status
END

IF ERRORCODE() THEN ErrHandler(Cus:AcctNumber,Trn:InvoiceNbr). !Handle errors
```

LOOP (iteration structure)

```
          | count TIMES                    |
LOOP [    | i = initial TO limit [ BY step ] | ]
          |   UNTIL logical expression      |
          |   WHILE logical expression      |
   statements
END
```

LOOP Initiates an iterative statement execution structure.

count A numeric constant, variable, or expression that determines the number of TIMES the *statements* in the LOOP are executed.

TIMES Executes *count* number of iterations of the *statements*.

i The label of a variable which is automatically incremented on each iteration of the LOOP.

= Assigns a new value to the increment (*i*) variable for each cycle of the LOOP.

initial A numeric constant, variable, or expression that specifies the initial value assigned to the increment variable (*i*) on the first pass through the LOOP structure.

TO A syntax conjunctive for the *limit* parameter.

limit When *i* is greater than *limit*, the LOOP structure control sequence terminates.

BY A syntax conjunctive for the *step* parameter.

step A numeric constant, variable, or expression. The *step* determines the quantity by which the *i* variable increments on each iteration of the LOOP. If the BY *step* parameter is omitted, *i* increments by 1.

UNTIL Evaluates the *logical expression* before each iteration of the LOOP. If the *logical expression* evaluates to true, the LOOP control sequence terminates.

WHILE Evaluates the *logical expression* before each iteration of the LOOP. If the *logical expression* evaluates to false, the LOOP control sequence terminates.

logical expression A numeric or string variable, expression, or function. A *logical expression* evaluates a condition. Control is determined by the result (true or false) of the expression. A zero (or blank) value evaluates as false, anything else is true.

statements An executable statement, or a sequence of executable statements.

A **LOOP** structure repetitively executes the *statements* within its structure. LOOP conditions are always evaluated at the top of the LOOP, before the LOOP is executed. LOOP structures may be nested within other executable code structures, and other executable code structures may be nested within a LOOP.

A LOOP with no parameters iterates continuously, unless a BREAK or RETURN statement is executed. BREAK discontinues the LOOP and continues program execution with the statement following the LOOP structure. All statements within a LOOP structure are executed unless a CYCLE statement is executed. CYCLE immediately sends program execution back to the top of the LOOP for the next iteration, without executing any statements following the CYCLE in the LOOP.

Example:

```
LOOP                                !Continuous loop
  Char = GetChar()                    ! get a character
  IF Char <> CarrReturn                 ! if it´s not a carriage return
    Field = CLIP(Field) & Char          !   append the character
  ELSE                              ! otherwise
```

```
   BREAK                          !  break out of the loop
. .                              !End if, end loop

IF ERRORCODE()                    !On error
  LOOP 3 TIMES                    ! loop three times
    BEEP                         !  sound the alarm
. .                              !End loop, end if

LOOP I# = 1 TO 365 BY 7             !Loop, increment I# by 7 each time
  GET(DailyTotal,I#)                ! read every 7th record
  DO WeeklyJob                      ! do the routine
END                              !End loop

SET(MasterFile)                     !Point to first record
LOOP UNTIL EOF(MasterFile)          !Process all the records
  NEXT(MasterFile)                  ! read a record
  ProcMaster                        ! call the procedure
END

LOOP WHILE KEYBOARD()               !Empty the keyboard buffer
  ASK                            ! without processing keystrokes
END
```

See Also:

BREAK

CYCLE

Control Statements

BREAK (immediately leave loop)

BREAK

The **BREAK** statement immediately terminates the LOOP or ACCEPT and transfers control to the first statement following the LOOP or ACCEPT loop structure. BREAK may only be used in a LOOP or ACCEPT loop structure.

Example:

```
LOOP                    !Loop
  ASK                   ! wait for a keystroke
  IF KEYCODE() = 256     ! if Esc key pressed
    BREAK                !  break out of the loop
  ELSE                  ! otherwise
    BEEP                !  sound the alarm
  END
END

ACCEPT                  !ACCEPT loop structure
  CASE ACCEPTED()
  OF ?Ok
    CallSomeProc
  OF ?Cancel
    BREAK                !  break out of the loop
  END
END
```

See Also:

LOOP

CYCLE

ACCEPT

CHAIN (execute another program)

CHAIN(*program***)**

CHAIN Terminates the current program and executes another.

program A string constant or variable containing the name of the program to execute. This may be any .EXE or .COM program.

CHAIN terminates the current program, closing all files and returning its memory to the operating system, and executes another *program*.

Example:

```
PROGRAM              !MainMenu program code
 CODE
 EXECUTE CHOICE()
   CHAIN(´Ledger´)       !Execute LEDGER.EXE
   CHAIN(´Payroll´)    !Execute PAYROLL.EXE
   RETURN             !Return to DOS
 END

  PROGRAM             !Ledger program code
 CODE
 EXECUTE CHOICE()
   CHAIN(´MainMenu´)     !Return to MainMenu program
   RETURN             !Return to DOS
 END

 PROGRAM               !Payroll program code
 CODE
 EXECUTE CHOICE()
   CHAIN(´MainMenu´)     !Return to MainMenu program
   RETURN             !Return to DOS
 END
```

CYCLE (go to top of loop)

CYCLE

The **CYCLE** statement passes control immediately back to the top of the LOOP or ACCEPT loop, where the LOOP condition is evaluated. CYCLE may only be used in a LOOP or ACCEPT loop structure.

In an ACCEPT loop, for certain events, CYCLE terminates an automatic action before it is performed (such as EVENT:Move).

Example:

```
SET(MasterFile)                 !Point to first record
LOOP                          !Process all the records
  NEXT(MasterFile)             ! read a record
  IF ERRORCODE() THEN BREAK.     !Get out of loop at end of file
  DO MatchMaster               ! check for a match
  IF NoMatch                   ! if match not found
    CYCLE                      !   jump to top of loop
  END
  DO TransVal                  ! validate the transaction
  PUT(MasterFile)               ! write the record
END
```

See Also:

LOOP

BREAK

ACCEPT

DO (call a ROUTINE)

DO *label*

DO	Executes a ROUTINE.
label	The label of a ROUTINE statement.

The **DO** statement is used to execute a ROUTINE local to a PROGRAM, PROCEDURE, or FUNCTION. When a ROUTINE completes execution, program control reverts to the statement following the DO statement. A ROUTINE may only be called within the CODE section containing the ROUTINE´s source code.

Example:

```
DO NextRecord    !Call the next record routine
DO CalcNetPay      !Call the calc net pay routine
```

EXIT (leave a ROUTINE)

EXIT

The **EXIT** statement immediately leaves a ROUTINE and returns program control to the statement following the DO statement that called it. An EXIT statement is not required. A ROUTINE with no EXIT statement terminates automatically when the entire sequence of statements in the ROUTINE is complete.

Example:

```
CalcNetPay  ROUTINE
  IF GrossPay = 0       !If no pay
    EXIT              ! exit the routine
  END
  NetPay = GrossPay - FedTax - Fica
  QtdNetPay += NetPay
  YtdNetPay += NetPay
```

GOTO (go to a label)

GOTO *label*

GOTO	Unconditionally transfers program control to another statement.
label	The label of another executable statement within the PROGRAM, PROCEDURE, FUNCTION, or ROUTINE.

The **GOTO** statement unconditionally transfers control from one statement to another. The target *label* of a GOTO must not be the label of a ROUTINE, PROCEDURE, or FUNCTION.

The scope of GOTO is limited to the currently executing ROUTINE, PROCEDURE, or FUNCTION; it may not target a *label* outside the ROUTINE, PROCEDURE, or FUNCTION in which it is used.

Example:

```
ComputeIt FUNCTION(Level)
  CODE
  IF Level = 0 THEN GOTO PassCompute.    !Skip rate calculation if no Level
  Rate = Level * MarkUp                 !Compute Rate
  RETURN(Rate)                         ! and return it
PassCompute RETURN(999999)              !Return bogus number
```

HALT (exit program)

HALT([*errorlevel*] [,*message*]**)**

HALT	Immediately terminates the program.
errorlevel	A positive integer constant or variable (range: 0 - 250) which is the exit code to pass to DOS, setting the DOS ERRORLEVEL. If omitted,the default is zero.
message	A string constant or variable which is typed on the screen after program termination.

The **HALT** statement immediately returns to the operating system, setting the *errorlevel* and optionally displaying a *message* after the program terminates. If a SHUTDOWN procedure is armed, it is executed before program termination.

If the program being HALTed was launched by a RUN or RUNSMALL statement within another Clarion program, the *errorlevel* exit code HALT sets may be determined by using the RUNCODE function in the launching program.

Example:

```
PasswordProc PROCEDURE
Password STRING(10)
Window WINDOW,CENTER
       ENTRY(@s10),AT(5,5),USE(Password),HIDE
      END
  CODE
  OPEN(Window)
  ACCEPT
    CASE ACCEPTED()
    OF ?Password
      IF Password <> ´Pay$MeMoRe´
        HALT(0,´Incorrect Password entered.´)
      END
    END
  END
```

See Also:

RUN

RUNCODE

IDLE (arm periodic procedure)

IDLE([*procedure*] [,*separation*]**)**

IDLE	Arms a *procedure* that periodically executes.
procedure	The label of a PROCEDURE. The *procedure* may not take any parameters.
separation	An integer that specifies the minimum wait time (in seconds) between calls to the *procedure*. A *separation* of 0 specifies continuous calls. If *separation* is omitted, the default value is 1 second.

An **IDLE** procedure is active while ASK or ACCEPT are waiting for user input. Only one IDLE procedure may be active at a time, and it executes on thread zero (0). Naming a new IDLE procedure overrides the previous one. An IDLE statement with no parameters disarms the IDLE process.

An IDLE *procedure* is usually prototyped in the PROGRAM´s MAP (not a MEMBER MAP). If prototyped in a MEMBER MAP, the IDLE statements which activate and de-activate it must be contained in a procedure or function within the same MEMBER module.

Example:

```
IDLE(ShoTime,10)      !Call shotime every 10 seconds
IDLE(CheckNet)        !Check network activity every 1 second
IDLE                 !Disarm idle procedure
```

See Also:

ASK

ACCEPT

PROCEDURE

MAP

RETURN (return to caller)

RETURN([*expression*] **)**

RETURN Terminates a PROGRAM, PROCEDURE, or FUNCTION.

expression The *expression* passes the return value of a FUNCTION back to the expression in which the FUNCTION was used. The *expression* is required for a FUNCTION and may not be used in a PROCEDURE or PROGRAM.

The **RETURN** statement terminates a PROGRAM, PROCEDURE, or FUNCTION, and passes control back to the caller. When RETURN is executed from the CODE section of a PROGRAM, the program is terminated, all files and windows are closed, and control is passed to the operating system.

RETURN is required in a FUNCTION and optional in a PROCEDURE or PROGRAM. If RETURN is not used in a PROCEDURE or PROGRAM, an implicit RETURN occurs at the end of the executable code. The end of executable code is defined as the end of the source file, or the beginning of another PROCEDURE, FUNCTION, or ROUTINE.

RETURN from a PROCEDURE or FUNCTION (whether explicit or implicit) automatically closes any local APPLICATION, WINDOW, REPORT, or VIEW structure opened in the PROCEDURE or FUNCTION. It does not automatically close any Global or Module Static APPLICATION, WINDOW, REPORT, or VIEW. It also closes and frees any local QUEUE structure declared without the STATIC attribute.

Example:

```
  IF Done# THEN RETURN.      !Quit when done

DayOfWeek FUNCTION(Date)     !Function to return the day of the week
  CODE
  EXECUTE (Date % 7) + 1     !Determine what day of week Date is
    RETURN(´Sunday´)        ! and RETURN the correct day string
    RETURN(´Monday´)
    RETURN(´Tuesday´)
    RETURN(´Wednesday´)
    RETURN(´Thursday´)
    RETURN(´Friday´)
    RETURN(´Saturday´)
  END
```

RUN (execute command)

RUN(*command* **)**

RUN Executes a *command* as if it were entered on the DOS command line.

command A string constant or variable containing the command to execute. This may include a full path and command line parameters.

The **RUN** statement executes a *command* to execute a DOS or Windows program. When the *command* executes, the new program is loaded as the ontop and active program. Execution control in the launching program returns immediately to the statement following RUN and the program continues executing as a background application. The user can return to the launching program by either terminating the launched program, or switching back to it through the Windows Task List.

If the *command* does not contain a path to the program, the following search sequence is followed:

1. The DOS current directory
2. The Windows directory
3. The Windows system directory
4. Each directory in the DOS PATH
5. Each directory mapped in a network

The successful execution of the *command* may be verified with the RUNCODE function, which returns the DOS exit code of the *command*. If unsuccessful, RUN posts the error to the ERROR and ERRORCODE functions.

Errors Posted: RUN may post any possible error (see Appendix B)

Example:

```
RUN(´notepad.exe readme.txt´)   !Run Notepad, automatically loading readme.txt file
RUN(ProgName )              !Run the command in the ProgName variable
```

See Also:

RUNCODE

STOP (suspend program execution)

STOP([*message*] **)**

STOP Suspends program execution and displays a message window.

message An optional string expression (up to 64K) which displays in the error window.

STOP suspends program execution and displays a message window. It offers the user the option of continuing the program or exiting. When exiting, it closes all files and frees the allocated memory.

Example:

```
PswdScreen  WINDOW
            STRING(´ Please Enter the Password ´),AT(5,5)
            ENTRY(@10),AT(20,5),USE(Password),HIDE    !Password storage field
          END
  CODE
  OPEN(PswdScreen)           !Open the password screen
  ACCEPT                    ! and get user input
    CASE ACCEPTED
    OF ?Password)
      IF Password <> ´PayMe$moRe´                !Correct password?
        STOP(´Incorrect Password Entered -- Access Denied´)
        HALT(0,´Incorrect password´)            !If not, throw them out
      END
    END
  END
```

Window Structures

Clarion Windows

Window Overview

In most Windows programs there are three types of screen windows used: application windows, document windows, and dialog boxes. An application window is the first window opened in a Windows program, and it usually contains the main menu as the entry point to the rest of the program. All other windows in the program are document windows or dialog boxes.

Along with these three screen window types, there are two user interface design conventions that are used in Windows programs: the Single Document Interface (SDI), and the Multiple Document Interface (MDI).

An SDI program usually only contains linear logic that allows the user to take only one execution path (thread) at a time; it does not open separate execution threads which the user may move between. This is the same type of program logic used in most DOS programs. An SDI program would not contain a Clarion APPLICATION structure as its application window. The Clarion WINDOW structure (without an MDI attribute) is used to define an SDI program´s application window, and the subsequent document windows or dialog boxes opened on top of it.

An MDI program allows the user to choose multiple execution paths (threads) and change from one to another at any time. This is a very common Windows program user interface. It is used by applications as a way of organizing and grouping windows which present several execution paths for the user to take.

A Clarion APPLICATION structure defines the MDI application window. The MDI application window acts as a parent for all the MDI child windows (document windows and dialog boxes), in that the child windows are clipped to its frame and automatically moved when the application frame is moved. They can also be concealed en masse by minimizing the parent. There may be only one APPLICATION open at any time in a Clarion Windows program.

Document windows and dialog boxes are very similar in that they are both defined as Clarion WINDOW structures. They differ in the conventional context in which they are commonly used and the conventions regarding appearance and attributes. In many cases, the difference is not distinguishable and does not matter. The generic term for both document windows and dialog boxes is "window" and that is the term used throughout this text.

Document windows usually display data. By convention they are movable and resizable. They usually have a title, a system menu, and maximize button. For example, in the Windows environment, the "Main" program group window that appears when you DOUBLE-CLICK on the "Main" icon in the Program Manager ´s desktop, is a document window.

Dialog boxes usually request information from the user or alert the user to some condition, usually prior to performing some action requested by the user. They may or may not be movable, and so, may or may not have a system menu and title. By convention, they are not resizable, although they can have a maximize button which gives the dialog two alternate sizes. A dialog box may be system modal (the user must respond before doing anything else in Windows), application modal (the user must respond before doing anything in the application), or modeless. For example, in the Clarion environment, the window that appears from the **F**ile menu´s **O**pen selection is an application modal dialog box that requests the name of the file to open.

Control Fields and Input Focus

The objects placed in an APPLICATION or WINDOW structure are "control fields." "Control" is a standard Windows term used to refer to any screen object--command buttons, text entry fields, radio buttons, list boxes, etc. In most DOS programs, the term "field" is usually used to refer to these objects. In this document, the terms "control" and "field" are generally interchangeable.

Controls appear only in MENUBARs, TOOLBARs, or WINDOW structures. Controls are available to the user to select and/or edit the data they contain only when it has "input focus." This occurs when the user uses the TAB key, the mouse, or an accelerator key combination to highlight the control.

A WINDOW also has "input focus" when it is the top WINDOW in the currently active execution thread. Since Clarion for Windows allows multi-threaded programs, the concept of which WINDOW currently has focus is important. Only the thread whose uppermost WINDOW has focus is active. The user may edit data in the WINDOW´s control fields only when it has focus.

Field Equate Labels

In WINDOW structures, every control field with a USE variable is assigned a field number by the compiler. By default, these field numbers begin with one (1) and are assigned to controls in the order they appear in the WINDOW structure code. The actual assigned numbers can be overridden by the second parameter of the USE attribute. The order of appearance in code determines the "natural" selection order of control fields for the ACCEPT structure (which may be altered with the SELECT statement). The order of appearance in code is independent of the control´s placement on the screen. Therefore, there is not necessarily any correlation between a control´s position on screen and the field number assigned by the compiler.

There are a number of statements that use these field numbers as parameters. It would be very tedious to "hard code" these numbers in order to use these statements. Therefore, Clarion provides a mechanism to address this problem: Field Equate Labels.

Field Equate Labels always begin with a question mark (?) followed by the name of the control´s USE variable. The leading question mark indicates to the compiler a Field Equate Label. They are very similar to normal EQUATE compiler directives. The compiler substitutes the field number for the Field Equate Label at compile time. This makes it unnecessary to know field numbers in advance.

Field Equate Labels for USE variables which are array elements always begin with a question mark (?) followed by the name of the USE variable followed by an underscore and a number (?ArrayField_1). Array elements from the same array are incrementally numbered beginning with one (1) for each element used in the same structure (?ArrayField_1, ?ArrayField_2, ...). Multi-dimensioned arrays are treated similarly (?ArrayField_1_1, ?ArrayField_1_2, ...).

Two or more controls with exactly the same USE variable in one WINDOW or APPLICATION structure would create the same Field Equate Label for all. Therefore, when the compiler encounters this condition, all Field Equate Labels for that USE variable are discarded. This makes it impossible to reference any of these controls in executable code, preventing confusion about which control you really want to reference. It also allows you to deliberately create this condition to display the contents of the variable in multiple controls using different display pictures. Some fields may have USE variables that can only be Field Equate Labels (a unique label with a leading question mark). This provides a way of referencing these fields in code statements.

In APPLICATION structures, every menu selection in the MENUBAR, and every control with a USE variable placed in the TOOLBAR, is assigned a number by the compiler. By default, these numbers begin with negative one (-1) and are decremented by one (1) in the order the menu selections and controls appear in the APPLICATION structure code.

Window Structure Statements

APPLICATION (declare an MDI frame window)

```
label   APPLICATION(´title´) [,AT( )] [,CENTER] [,SYSTEM] [,MAX] [,ICON( )] [,STATUS( )] [,HLP( )]
          [,CURSOR( )] [,TIMER( )] [,ALRT( )] [,ICONIZE] [,MAXIMIZE] [,MASK] [,FONT( )]
          [,MSG( )] [,IMM] [,AUTO] [,   | HSCROLL    | ] [, | DOUBLE   | ]
                                        | VSCROLL    |    | NOFRAME |
                                        | HVSCROLL   |    | RESIZE   |
    [ MENUBAR
        multiple menu and/or item declarations
      END ]
    [ TOOLBAR
        multiple control field declarations
      END ]
  END
```

APPLICATION Declares a Multiple Document Interface (MDI) frame.

label A valid Clarion label. A *label* is required on the APPLICATION statement.

title Specifies the title text for the application window.

AT Specifies the initial size and location of the application window. If omitted, default values are selected by the runtime library.

CENTER Specifies that the window´s initial position is centered in the screen by default. This attribute takes effect only if at least one parameter of the **AT** attribute is omitted.

SYSTEM Specifies the presence of a system menu.

MAX Specifies the presence of a maximize control.

ICON Specifies the presence of a minimize control, and names a file or standard icon identifier for the icon displayed when the window is minimized.

STATUS Specifies the presence of a status bar at the base of the application window.

HLP Specifies the "Help ID" associated with the APPLICATION window and provides the default for any child windows.

CURSOR Specifies a mouse cursor to be displayed when the mouse is positioned over the APPLICATION window. If omitted, the Windows default cursor is used.

TIMER Specifies periodic timed event generation.

ALRT Specifies "hot" keys active for the APPLICATION.

ICONIZE Specifies the APPLICATION is opened as an icon.

MAXIMIZE Specifies the APPLICATION is maximized when opened.

MASK Specifies pattern input editing mode of all ENTRY controls in the TOOLBAR.

FONT Specifies the default font for all controls in the toolbar.

MSG Specifies a string constant containing the default text to display in the status bar for all controls in the APPLICATION.

IMM Specifies the window generates events whenever it is moved or resized.

AUTO Specifies all toolbar controls´ USE variables re-display on screen each time through the ACCEPT loop.

HSCROLL Specifies that a horizontal scroll bar is automatically added to the application frame when any portion of a child window lies horizontally outside the visible area.

VSCROLL	Specifies that a vertical scroll bar is automatically added to the application frame when any portion of a child window lies vertically outside the visible area.
HVSCROLL	Specifies that both vertical and horizontal scroll bars are automatically added to the application frame when any portion of a child window lies outside the visible area.
DOUBLE	Specifies a double-width frame around the window. A window with this type of frame may not be resized.
NOFRAME	Specifies a window with no frame. A window with this type of frame may not be resized.
RESIZE	Specifies a thick frame around the window which does allow window resizing.
MENUBAR	Defines the menu structure (optional). The menu specified in an APPLICATION is the "Global menu."
TOOLBAR	Defines a toolbar structure (optional). The toolbar specified in an APPLICATION is the "Global toolbar."

APPLICATION declares a Multiple Document Interface (MDI) frame window. MDI is a part of the standard Windows interface, and is used by Windows applications to present several "views" in different windows. This is a way of organizing and grouping these. The MDI frame window (APPLICATION structure) acts as a "parent" for all the MDI "child" windows (WINDOW structures with the MDI attribute). These MDI "child" windows are clipped to the APPLICATION frame and automatically moved when the frame is moved, and can be totally concealed by minimizing the parent.

There may be only one APPLICATION window open at any time in a Clarion Windows program, and it must be opened before any MDI "child" windows may be opened. However, non-MDI windows may be opened before or after the APPLICATION is opened, and may be on the same execution thread as the APPLICATION.

An MDI "child" window must not be on the same execution thread as the APPLICATION. Therefore, any MDI "child" window called directly from the APPLICATION must be in a separate procedure so the START function can be used to begin a new execution thread. Once started, multiple MDI "child" windows may be called in the new thread.

A "conventional" APPLICATION window would have the ICON, MAX, STATUS, RESIZE, and SYSTEM attributes. This creates an application frame window with minimize and maximize buttons, a status bar, a resizable frame, and a system menu. It would also have a MENUBAR structure containing the global menu items, and may have a TOOLBAR with "shortcuts" to global menu items. These attributes create a standard Windows look and feel for the application frame.

An APPLICATION window may not contain controls except within its MENUBAR and TOOLBAR structures, and cannot be used for any output. For output, document windows or dialog boxes are required (defined using the WINDOW structure).

When the APPLICATION window is first opened, it remains hidden until the first DISPLAY statement or ACCEPT loop is encountered. This enables any changes to be made to the appearance before it is displayed. For example, the caption or size can be adjusted via runtime property assignment.

Example:

```
  !An MDI application frame window with system menu, minimize and maximize
  ! buttons, a status bar, scroll bars, and a resizable frame, containing the
  ! main menu and toolbar for the application:
MainWin APPLICATION(´My Application´),SYSTEM,MAX,ICON(´MyIcon.ICO´),STATUS |
   ,HVSCROLL,RESIZE
        MENUBAR
          MENU(´&File´),USE(?FileMenu)
```

```
          ITEM('&Open...'),USE(?OpenFile)
          ITEM('&Close'),USE(?CloseFile),DISABLE
          ITEM('E&xit'),USE(?MainExit)
        END
        MENU('&Edit'),USE(?EditMenu)
          ITEM('Cu&t'),USE(?CutText),KEY(CtrlX),STD(STD:Cut),DISABLE
          ITEM('&Copy'),USE(?CopyText),KEY(CtrlC),STD(STD:Copy),DISABLE
          ITEM('&Paste'),USE(?PasteText),KEY(CtrlV),STD(STD:Paste),DISABLE
        END
        MENU('&Window'),STD(STD:WindowList),LAST
          ITEM('&Tile'),STD(STD:TileWindow)
          ITEM('&Cascade'),STD(STD:CascadeWindow)
          ITEM('&Arrange Icons'),STD(STD:ArrangeIcons)
        END
        MENU('&Help'),USE(?HelpMenu)
          ITEM('&Contents'),USE(?HelpContents),STD(STD:HelpIndex)
          ITEM('&Search...'),USE(?HelpSearch),STD(STD:HelpSearch)
          ITEM('&How to Use Help'),USE(?HelpOnHelp),STD(STD:HelpOnHelp)
          ITEM('&About MyApp...'),USE(?HelpAbout)
        END
      END
      TOOLBAR
        BUTTON('E&xit'),USE(?MainExitButton)
        BUTTON('&Open'),USE(?OpenButton),ICON(ICON:Open)
      END
    END
CODE
OPEN(MainWin)           !Open APPLICATION
ACCEPT                 !Display APPLICATION and accept user input
  CASE ACCEPTED()         !Which control was chosen?
  OF ?OpenFile          !Open... menu selection
  OROF ?OpenButton      !Open button on toolbar
    START(OpenFileProc)   !Start new execution thread
  OF ?MainExit          !Exit menu selection
  OROF ?MainExitButton    !Exit button on toolbar
    BREAK               !Break ACCEPT loop
  OF ?HelpAbout         !About... menu selection
    HelpAboutProc       !Call application information procedure
  END
END
CLOSE(MainWin)          !Close APPLICATION
```

WINDOW (declare a dialog window)

```
label   WINDOW(´title´) [,AT( )] [,CENTER] [,SYSTEM] [,MAX] [,ICON( )] [,STATUS( )] [,HLP( )]
          [,CURSOR( )] [,MDI] [,MODAL] [,MASK] [,FONT( )] [,GRAY][,TIMER( )] [,ALRT( )]
          [,ICONIZE] [,MAXIMIZE] [,MSG( )] [,TOOLBOX][,PALETTE( )] [,DROPID( )] [,IMM]
          [,AUTO] [,  | HSCROLL    | ] [, | DOUBLE    | ]
                      | VSCROLL    |    | NOFRAME |
                      | HVSCROLL   |    | RESIZE    |
        [ MENUBAR
            menus and/or items
          END ]
        [ TOOLBAR
            controls
          END ]
          controls
        END
```

WINDOW	Declares a document window or dialog box.
label	A valid Clarion label. A *label* is required.
title	A string constant containing the window´s title text.
AT	Specifies the initial size and location of the window. If omitted, default values are selected by the runtime library.
CENTER	Specifies that the window´s initial position is centered on screen relative to its parent window, by default. This attribute takes effect only if at least one parameter of the AT attribute is omitted.
SYSTEM	Specifies the presence of a system menu.
MAX	Specifies the presence of a maximize control.
ICON	Specifies the presence of a minimize control, and names a file or standard icon identifier for the icon displayed when the window is minimized.
STATUS	Specifies the presence of a status bar for the window.
HLP	Specifies the "Help ID" associated with the window.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the window. This cursor is inherited by the WINDOW´s controls unless overridden.
MDI	Specifies that the window conforms to normal MDI child-window behavior.
MODAL	Specifies the window is "system modal" and must be closed before the user may do anything else.
MASK	Specifies pattern input editing mode of all ENTRY controls in this window.
FONT	Specifies the default font for all controls in this window.
GRAY	Specifies that the window has a gray background for use with 3-D look controls.
TIMER	Specifies periodic timed event generation.
ALRT	Specifies "hot" keys active when the window has focus.
ICONIZE	Specifies the window is opened as an icon.
MAXIMIZE	Specifies the window is maximized when opened.
MSG	Specifies a string constant containing the default text to display in the status bar for all

controls in the window.

TOOLBOX Specifies the window is "always on top" and its controls never retain focus.

PALETTE Specifies the number of hardware colors used for graphics in the window.

DROPID Specifies the window may serve as a drop target for drag-and-drop actions.

IMM Specifies the window generates events whenever it is moved or resized.

AUTO Specifies all toolbar controls´ USE variables re-display on screen each time through the ACCEPT loop.

HSCROLL Specifies that a horizontal scroll bar is automatically added to the window when any scrollable portion of the window lies horizontally outside the visible area.

VSCROLL Specifies that a vertical scroll bar is automatically added to the window when any scrollable portion of the window lies vertically outside the visible area.

HVSCROLL Specifies that both vertical and horizontal scroll bars are automatically added to the window when any scrollable portion of the window lies outside the visible area.

DOUBLE Specifies a double-width frame around the window. A window with this type of frame may not be resized.

NOFRAME Specifies a window with no frame. A window with this type of frame may not be resized.

RESIZE Specifies a thick frame around the window, which does allow window resizing.

MENUBAR Defines a menu structure (optional).

menus and/or items
MENU and/or ITEM declarations that define the menu selections.

TOOLBAR Defines a toolbar structure (optional).

controls Control field declarations that define tools available on the TOOLBAR, or the control fields in the WINDOW.

A **WINDOW** declares a document window or dialog box which may contain controls, and may be used to display output to the user. When the WINDOW is first opened, it remains hidden until the first DISPLAY statement or ACCEPT loop is encountered. This enables any changes to be made to the appearance before it is displayed. For example, the caption or size can be adjusted via runtime property assignment. Any previously opened WINDOW on the same execution thread is disabled.

A WINDOW automatically receives a single-width border frame unless one of the DOUBLE, NOFRAME, or RESIZE attributes are specified. Screen coordinates are measured in dialog units. A dialog unit is defined as one-quarter the average character width and one-eighth the average character height of the font specified in the WINDOW´s FONT attribute (or the system font, if no FONT attribute is specified on the WINDOW).

A WINDOW with the MODAL attribute is system modal; it takes exclusive control of the computer. This means that any other progam running in the background halts its execution until the MODAL WINDOW is closed. Therefore, the MODAL attribute should be used only when absolutely necessary. Also, the RESIZE attribute is ignored, and the WINDOW cannot be moved when the MODAL attribute is present.

A WINDOW without the MDI attribute, when opened in an MDI program, is application modal. This means that the user must respond before moving to any other window in the application. The user may, however, move to any other program running in Windows at the time. Non-MDI windows may be opened either before or after an APPLICATION is opened, and may be on the same execution thread as the APPLICATION.

A WINDOW with the MDI attribute is an MDI "child" window. MDI "child" windows are clipped to the APPLICATION frame and automatically moved when the frame is moved, and can be totally concealed by minimizing the parent APPLICATION. MDI "child" windows are modeless; the user may change to the top window of another execution thread, within the same application or any other application running in Windows, at any time. An MDI "child" window must not be on the same execution thread as the APPLICATION. Therefore, any MDI "child" window called directly from the APPLICATION must be in a separate procedure so the START function can be used to begin a new execution thread. Once started, multiple MDI "child" windows may be called in the new thread.

The MENUBAR specified in a WINDOW with the MDI attribute is automatically merged into the "Global menu" (from the APPLICATION) when the WINDOW receives focus unless either the WINDOW´s or APPLICATION´s MENUBAR has the NOMERGE attribute. A MENUBAR specified in a WINDOW without the MDI attribute is never merged into the "Global menu"--it always appears in the window itself.

The TOOLBAR specified in a WINDOW with the MDI attribute is automatically merged into the "Global toolbar" (from the APPLICATION) when the WINDOW receives focus, unless either the WINDOW´s or APPLICATION´s TOOLBAR has the NOMERGE attribute. The toolbar specified in a WINDOW without the MDI attribute is never merged into the "Global toolbar"--it always appears in the window itself.

A WINDOW with the TOOLBOX attribute is automatically "always on top" and its controls do not retain focus (just as if they all had the SKIP attribute). This creates a window whose controls all behave in the same manner as controls in the toolbar. Normally, a WINDOW with the TOOLBOX attribute would be executed in its own thread.

Example:

```
 !MDI child window with system menu, minimize and maximize buttons, status bar,
 ! scroll bars, a resizable frame, with menu and toolbar which are merged into the
 !application´s menubar and toolbar:
MDIChild WINDOW(´Child One´),MDI,SYSTEM,MAX,ICON(´Icon.ICO´),STATUS,HVSCROLL,RESIZE
          MENUBAR
            MENU(´File´),USE(?FileMenu)
              ITEM(´Close´),USE(?CloseFile)
            END
            MENU(´Edit´),USE(?EditMenu)
              ITEM(´Undo´),USE(?UndoText),KEY(CtrlZ),STD(STD:Undo)
              ITEM(´Cu&t´),USE(?CutText),KEY(CtrlX),STD(STD:Cut)
              ITEM(´Copy´),USE(?CopyText),KEY(CtrlC),STD(STD:Copy)
              ITEM(´Paste´),USE(?PasteText),KEY(CtrlV),STD(STD:Paste)
            END
          END
          TOOLBAR
            BUTTON(´Cut´),USE(?CutButton),ICON(ICON:Cut),STD(STD:Cut)
            BUTTON(´Copy´),USE(?CopyButton),ICON(ICON:Copy),STD(STD:Copy)
            BUTTON(´Paste´),USE(?PasteButton),ICON(ICON:Paste),STD(STD:Paste)
          END
          TEXT,HVSCROLL,USE(Pre:Field)
          BUTTON(´&OK´),USE(?Exit),DEFAULT
        END

 !Non-MDI, system menu, maximize button, status bar, non-resizable frame,
NonMDI WINDOW(´Dialog Window´),SYSTEM,MAX,STATUS
         TEXT,HVSCROLL,USE(Pre:Field),MSG(´Enter some text here´)
         BUTTON(´&OK´),USE(?Exit),DEFAULT
       END
 !System-modal window with non-resizable frame, with only a message and Ok button:
ModalWin WINDOW(´Modal Window´),MODAL
          IMAGE(ICON:Exclamation)
          STRING(´An ERROR has occurred´)
```

```
  BUTTON(´&OK´),USE(?Exit),DEFAULT
END
```

APPLICATION and WINDOW Attributes

ALRT (set window hot keys)

AT (set window position and size)

AUTO (set USE variable automatic re-display)

CENTER (set position and size)

CURSOR (set mouse cursor type)

DOUBLE, NOFRAME, RESIZE (set window border)

FONT (set window default font)

GRAY (set 3-D look background)

HLP (set windows on-line help identifier)

HSCROLL, VSCROLL, HVSCROLL (set window scroll bars)

ICON (set window icon)

ICONIZE (set window open as icon)

IMM (set immediate resize event notification)

MASK (set pattern editing data entry)

MAX (set maximize control)

MAXIMIZE (set window open maximized)

MDI (set MDI child window)

MODAL (set system modal window)

MSG (set window status bar message)

PALETTE (set number of hardware colors)

STATUS (set status bar)

SYSTEM (set system menu)

TOOLBOX (set toolbox window behavior)

TIMER (set periodic event)

ALRT (set window "hot" keys)

ALRT(*keycode***)**

ALRT Specifies a "hot" key active while the APPLICATION or WINDOW has focus.

keycode A numeric constant keycode or keycode equate.

The **ALRT** attribute specifies a "hot" key active while the APPLICATION or WINDOW has focus. When the user presses an ALRT "hot" key for the APPLICATION or WINDOW, two field-independent events, EVENT:PreAlertKey and EVENT:AlertKey, are generated. If the code executes a CYCLE statement when processing EVENT:PreAlertKey, you "shortstop" the EVENT:AlertKey, preventing library´s default action on the alerted keypress for the window.

You may have multiple ALRT attributes on one APPLICATION or WINDOW. The ALERT statement and the ALRT attribute of a window or control are completely separate. This means that clearing ALERT keys has no effect on any keys alerted by ALRT attributes.

Example:

```
Screen WINDOW,ALRT(F10Key),ALRT(F9Key)           !F10 and F9 alerted
         LIST,AT(109,48,50,50),USE(?List),FROM(Que),IMM
         BUTTON(´&Ok´),AT(111,108,,),USE(?Ok)
         BUTTON(´&Cancel´),AT(111,130,,),USE(?Cancel)
       END
  CODE
  OPEN(Screen)
  ACCEPT
    CASE EVENT()
    OF EVENT:PreAlertKey          !Pre-check alert events
      IF FOCUS() <> ?LIST           !Allow execution only on the list
        CYCLE                     !Terminate alert processing on other controls
      END
    OF EVENT:AlertKey               !Alert processing
      CASE KEYCODE()
      OF F9Key                   !Check for F9
        F9HotKeyProc              !Call hot key procedure
      OF F10Key                   !Check for F10
        F10HotKeyProc             !Call hot key procedure
      END
    END
  END
```

AT (set window position and size)

AT([*x*] [,*y*] [,*width*] [,*height*])

AT	Specifies the initial position and size of the window.
x	An integer constant or constant expression that specifies the initial horizontal position of the top left corner. If omitted, the runtime library provides a default value.
y	An integer constant or constant expression that specifies the initial vertical position of the top left corner. If omitted, the runtime library provides a default value.
width	An integer constant or constant expression that specifies the initial width. If omitted, the runtime library provides a default value.
height	An integer constant or constant expression that specifies the initial height. If omitted, the runtime library provides a default value.

The **AT** attribute defines the initial position and size of an APPLICATION or WINDOW. If any parameter is omitted, the runtime library provides a default value. The *x* and *y* parameters are relative to the top left hand corner of the video screen when the AT attribute is placed on an APPLICATION structure, or a WINDOW without the MDI attribute that is opened before any APPLICATION structure is opened by the program. They are relative to the top left hand corner of the APPLICATION when the AT attribute is placed on a WINDOW with the MDI attribute, or a WINDOW without the MDI attribute opened after an APPLICATION structure has been opened.

The *width* and *height* parameters specify the size of the "client area" or "workspace" of an APPLICATION. This is the area below the MENUBAR and above the status bar which defines the area in which the TOOLBAR is placed and MDI "child" windows are opened. On a WINDOW, they specify the size of the "workspace" which may contain control fields.

The values contained in the *x*, *y*, *width*, and *height* parameters are mreasured in to dialog units. Dialog units are defined as one-quarter the average character width by one-eighth the average character height. The size of a dialog unit is dependent upon the size of the default font for the window. This measurement is based on the font specified in the FONT attribute of the window, or the system default font specified by Windows.

Example:

```
WinOne WINDOW,AT(0,0,380,200),MDI  !top left corner, relative to app frame
       END

WinTwo WINDOW,AT(0,0,380,200)      !Top left corner, relative to video screen
       END
```

AUTO (set USE variable automatic re-display)

AUTO

The **AUTO** attribute specifies all window controls´ USE variables re-display on screen each time through the ACCEPT loop. This incurs some overhead, but ensures the data displayed is current, without requiring explicit DISPLAY statements.

Example:

```
WinOne WINDOW,AT(,,380,200),MDI,CENTER,AUTO  !All controls values always display
        !controls
      END
  CODE
  ACCEPT               !ACCEPT automatically re-dislays changed USE variables
  END
```

CENTER (set position and size)

CENTER

The **CENTER** attribute indicates that the window´s default width and height are centered. A WINDOW structure with the MDI attribute is centered on the APPLICATION. An APPLICATION structure is centered on the screen. A non-MDI WINDOW is centered on its parent (the window currently with focus when the non-MDI WINDOW is opened).

This attribute has no meaning unless at least one parameter of the AT attribute is omitted. This means that the CENTER attribute provides a default value for any omitted AT parameter.

Example:

```
 !Window centered relative to application frame:
WinOne WINDOW,AT(,,380,200),MDI,CENTER
       END

 !Window centered relative to its parent:
WinTwo WINDOW,AT(,,380,200),CENTER
       END
```

CURSOR (set mouse cursor type)

CURSOR(*file***)**

CURSOR Specifies a mouse cursor to display for the window.

file A string constant containing the name of a .CUR file, or an EQUATE naming a Windows-standard mouse cursor. The .CUR file is linked into the .EXE as a resource.

The **CURSOR** attribute specifies a mouse cursor to be displayed when the mouse is positioned over the window. This cursor is inherited by the controls in the window unless overridden.

The Windows standard mouse cursors contained in EQUATES.CLW are:

CURSOR:None	No mouse cursor
CURSOR:Arrow	The normal windows arrow cursor
CURSOR:IBeam	A capital "I" like a steel I-beam
CURSOR:Wait	An hourglass
CURSOR:Cross	A large plus sign
CURSOR:UpArrow	A vertical arrow
CURSOR:Size	A four-headed arrow
CURSOR:Icon	A box within a box
CURSOR:SizeNWSE	A double-headed arrow slanting left
CURSOR:SizeNESW	A double-headed arrow slanting right
CURSOR:SizeWE	A double-headed horizontal arrow
CURSOR:SizeNS	A double-headed vertical arrow
CURSOR:DragWE	A double-headed horizontal arrow

Example:

```
 !Window with Windows-standard large plus sign cursor
WinOne WINDOW,CURSOR(CURSOR:Cross)
       END

 !Window with custom cursor
WinTwo WINDOW,CURSOR(´CUSTOM.CUR´)
       END
```

DOUBLE, NOFRAME, RESIZE (set window border)

DOUBLE
NOFRAME
RESIZE

The **DOUBLE**, **NOFRAME**, and **RESIZE** attributes specify a WINDOW or APPLICATION border frame style other than the default single-width border. The **DOUBLE** attribute places a double-width border around the window and the **NOFRAME** attribute places no border on the window. A window with these frame types may not be resized.

The **RESIZE** attribute places a thick border frame around the window. This is the only type that allows the user to dynamically resize the window. RESIZE is ignored on any WINDOW with the MODAL attribute.

The RESIZE frame type is normally used on APPLICATION structures and WINDOW structures used as document windows, not dialog boxes. NOFRAME is usually used on "hidden" windows used only to activate an ACCEPT loop. DOUBLE is a common dialog box frame type.

Example:

```
 !A Window with a single-width border:
Win1 WINDOW
     END

 !A resizable Window:
Win2 WINDOW,RESIZE
     END

 !A Window with a double-width border:
Win3 WINDOW,DOUBLE
     END

 !A Window without a border:
Win4 WINDOW,NOFRAME
     END
```

FONT (set window default font)

FONT([*typeface*] [,*size*] [,*color*] [,*style*]**)**

FONT	Specifies the default display font for the window.
typeface	A string constant containing the name of the font. If omitted, the system font is used.
size	An integer constant containing the size (in points) of the font. If omitted, the system default font size is used.
color	A LONG integer constant containing the red, green, and blue values for the color of the font in the low-order three bytes, or an EQUATE for a standard Windows color value. If omitted, black is used.
style	An integer constant or constant expression or EQUATE specifying the strike weight and style of the font. If omitted, the weight is normal.

The **FONT** attribute on a WINDOW or APPLICATION structure specifies the default display font for all controls in the WINDOW or APPLICATION that do not have a FONT attribute. This is also the default font for newly created controls on the window, and is the font used by the SHOW and TYPE statements when writing to the window.

The *typeface* may name any font registered in the Windows system. The EQUATES.CLW file contains EQUATE values for standard *style* values. A *style* on the range zero (0) to one thousand (1000) specifies the strike weight of the font. You may add to that values that indicate italic, underline, or strikeout text. The following EQUATES are in EQUATES.CLW:

FONT:thin	EQUATE (100)
FONT:regular	EQUATE (400)
FONT:bold	EQUATE (700)
FONT:fixed	EQUATE (0800H)
FONT:italic	EQUATE (01000H)
FONT:underline	EQUATE (02000H)
FONT:strikeout	EQUATE (04000H)

Example:

```
 !A Window using 14 point Times New Roman
Win1 WINDOW,FONT(´Times New Roman´,14,00H)
     END

 !A Window using 14 point Times New Roman, Bold and Italic
Win2 WINDOW,FONT(´Times New Roman´,14,00H,FONT:italic+FONT:bold)
     END
```

GRAY (set 3-D look background)

GRAY

The **GRAY** attribute indicates that the WINDOW has a gray background, suitable for use with three-dimensional dialog controls. All controls on a WINDOW with the GRAY attribute are automatically given a three-dimensional appearance. Controls in a TOOLBAR are always automatically given a three-dimensional appearance, without the GRAY attribute.

This attribute is not valid on an APPLICATION structure.

The three-dimensional look may be disabled by SET3DLOOK.

Example:

```
 !A Window with 3-D controls
Win1 WINDOW,GRAY
     END
```

HLP (set window´s on-line help identifier)

HLP(*helpID***)**

HLP	Specifies the *helpID* for the APPLICATION, WINDOW, or control.
helpID	A string constant specifying the key used to access the Help system. This may be either a Help keyword or a "context string."

The **HLP** attribute specifies the *helpID* for the APPLICATION or WINDOW. Help, if available, is automatically displayed by Windows whenever the user presses F1.

If the user presses F1 to request help when the APPLICATION window is foremost and no menus are active, the APPLICATION´s *helpID* is used to locate the Help text. Otherwise, the library automatically uses the *helpID* of the active menu of uppermost control or window, searching up the hierarchy until an object with that *helpID* is found. The *helpID* of the APPLICATION is at the top of the hierarchy.

The *helpID* may contain a Help keyword or a "context string." A Help keyword is a keyword or phrase that is displayed in the Help Search dialog. When the user presses F1, if only one topic in the help file specifies this keyword, the help file is opened at that topic; if more than one topic specifies the keyword, the search dialog is opened for the user.

A "context string" is identified by a leading tilde (~) in the *helpID*, followed by a unique identifier (no spaces allowed) associated with exactly one help topic. When the user presses F1, the help file is opened at the specific topic associated with that "context string." If the tilde is missing, the *helpID* is assumed to be a help keyword.

Example:

```
 !A Window with a help context string:
Win1 WINDOW,HLP(´~Win1Help´)
     END

 !A Window with a help keyword:
Win2 WINDOW,HLP(´Window One Help´)
     END
```

HSCROLL, VSCROLL, HVSCROLL (set window scroll bars)

HSCROLL
VSCROLL
HVSCROLL

The **HSCROLL**, **VSCROLL**, and **HVSCROLL** parameters place scroll bars on an APPLICATION or WINDOW. HSCROLL adds a horizontal scroll bar to the bottom, VSCROLL adds a vertical scroll bar on the right side, and HVSCROLL adds both.

The vertical scroll bar allows a mouse to scroll up or down. The horizontal scroll bar allows a mouse to scroll left or right. The scroll bars appear whenever any scrollable portion of the APPLICATION or WINDOW lies outside the visible area on screen.

Example:

```
 !A Window with a horizontal scroll bar:
Win1 WINDOW,HSCROLL
     END

 !A Window with a vertical scroll bar:
Win2 WINDOW,VSCROLL
     END

 !A Window with both scroll bars:
Win2 WINDOW,HVSCROLL
     END
```

ICON (set window icon)

ICON([*file*] **)**

ICON	Specifies an icon to display for the APPLICATION or WINDOW.
file	A string constant containing the name of an .ICO file, or an EQUATE for the Windows-standard icon to display. The .ICO file is automatically linked into the .EXE as a resource.

The **ICON** attribute specifies an icon to display for the APPLICATION or WINDOW. On an APPLICATION or WINDOW, ICON also specifies the presence of a minimize control. The minimize control appears in the top right corner of the window as a downward pointing triangle (usually). When the user clicks the mouse on it, the window shrinks to an icon without halting its execution. When an APPLICATION or non-MDI WINDOW is minimized, the icon *file* is displayed in the operating system´s desktop; when a WINDOW with the MDI attribute is minimized, the icon *file* is displayed in the APPLICATION.

EQUATE statements for the Windows-standard icons are contained in the EQUATES.CLW file. The following list is a representative sample of these (see EQUATES.CLW for the complete list):

ICON:None	No icon
ICON:Application	
ICON:Question	?
ICON:Exclamation	!
ICON:Asterisk	*
ICON:VCRtop	>>|
ICON:VCRrewind	<<
ICON:VCRback	<
ICON:VCRplay	>
ICON:VCRfastforward	>>
ICON:VCRbottom	|<<
ICON:VCRlocate	?

Example:

```
 !A Window with a minimize button:
WinOne WINDOW,ICON(´MyIcon.ICO´)
       END

 !A Window with a minimize button:
WinTwo WINDOW,ICON(ICON:Application)
       END
```

ICONIZE (set window open as icon)

ICONIZE

The **ICONIZE** attribute specifies the APPLICATION or WINDOW is opened minimized as the icon specified by the ICON attribute. When an APPLICATION or non-MDI WINDOW is minimized, the icon *file* is displayed in the operating system´s desktop; when a WINDOW with the MDI attribute is minimized, the icon *file* is displayed in the APPLICATION.

Example:

```
 !A Window with a minimize button, opened as the icon:
Win2 WINDOW,ICON(´MyIcon.ICO´),ICONIZE
     END
```

IMM (set immediate resize event notification)

IMM

The **IMM** attribute on a WINDOW or APPLICATION specifies immediate event generation whenever the user moves or resizes the window. It generates one the following events before the action is executed:

EVENT:Move
EVENT:Size
EVENT:Restore
EVENT:Maximize
EVENT:Iconize

If the code that handles these events executes a CYCLE statement, the action is not performed. This allows you to prevent the user from moving or resizing the window. Once the action has been performed, one or more of the following events are generated:

EVENT:Moved
EVENT:Sized
EVENT:Restored
EVENT:Maximized
EVENT:Iconized

Multiple post-action events are generated because some of the actions have multiple results. For example, if the user CLICKS on the maximize button, EVENT:Maximize is generated. If there is no CYCLE statement executed as a result of this event, the action is performed, then EVENT:Maximized, EVENT:Moved, and EVENT:Sized are generated. This occurs because the window has been maximized, which also moves and resizes it at the same time.

Example:

```
Win2 WINDOW(´Some Window´),AT(58,11,174,166),MDI,DOUBLE,MAX,IMM
      LIST,AT(109,48,50,50),USE(?List),FROM(´Que´),IMM
      BUTTON(´&Ok´),AT(111,108,,),USE(?Ok)
      BUTTON(´&Cancel´),AT(111,130,,),USE(?Cancel)
    END
  CODE
  OPEN(Win2)
  ACCEPT
    CASE EVENT()
    OF EVENT:Move              !Prevent user from moving window
      CYCLE
    OF EVENT:Maximized         !When Maximized
      ?List{PROP:Height} = 100     ! resize the list
    OF EVENT:Restored           !When Restored
      ?List{PROP:Height} = 50      ! resize the list
    END
  END
```

MASK (set pattern editing data entry)

MASK

The **MASK** attribute specifies pattern input editing mode of all controls in this window. This means that, as the user types in data, each character is automatically validated against the control´s picture for proper input (numbers only in numeric pictures, etc.). This forces the user to enter data in the format specified by the control´s display picture.

If omitted, Windows free-input is allowed in the controls. Free-input means the user´s data is formatted to the control´s picture only after entry. This allows users to enter data as they choose and it is automatically formatted to the control´s picture after entry. If the user types in data in a format different from the control ´s picture, the libraries attempt to determine the format the user used, and convert the data to the control ´s display picture. For example, if the user types "January 1, 1995" into a control with a display picture of @D1, the runtime library formats the user´s input to "1/1/95." This action occurs only after the user completes data entry and moves to another control. If the runtime library cannot determine what format the user used, it will not update the USE variable. It then beeps and leaves the user on the same control with the data they entered, to allow them to try again.

Example:

```
 !A Window with pattern input editing enabled
Win2 WINDOW,MASK
    END
```

MAX (set maximize control)

MAX

The **MAX** attribute specifies a maximize control on the APPLICATION or WINDOW. The maximize control appears in the top right corner of the window as a box containing either an upward pointing triangle, or an upward pointing triangle above a downward pointing triangle (in Windows 3.1). When the user clicks the mouse on it, an APPLICATION or non-MDI WINDOW expands to occupy the full screen, an MDI WINDOW expands to occupy the entire APPLICATION. Once expanded, the maximize control appears as an upward pointing triangle above a downward pointing triangle. Click the mouse on it again, and the window returns to its previous size and the maximize control appears as an upward pointing triangle.

Example:

```
 !A Window with a maximize button:
Win2 WINDOW,MAX
    END
```

MAXIMIZE (set window open maximized)

MAXIMIZE

The **MAXIMIZE** attribute specifies the APPLICATION or WINDOW is opened maximized.When maximized, an APPLICATION or non-MDI WINDOW expands to occupy the full screen, and an MDI WINDOW expands to occupy the entire APPLICATION. Once expanded, the maximize control appears as an upward pointing triangle above a downward pointing triangle (in Windows 3.1).

Example:

```
 !A Window with a maximize button, opened maximized:
Win2 WINDOW,MAX,MAXIMIZE
    END
```

MDI (set MDI child window)

MDI

The **MDI** attribute specifies a WINDOW structure that acts as a "child" window to the APPLICATION. MDI "child" windows are clipped to the APPLICATION frame--they scroll only within the boundaries set by the display size of the APPLICATION. MDI "child" windows are automatically moved when the APPLICATION frame is moved, and can be totally concealed by minimizing the APPLICATION. A WINDOW with the MDI attribute cannot be opened unless there is a currently open APPLICATION.

MDI "child" windows are modeless; the user may change to the top window of another execution thread, within the same application or any other application running in Windows, at any time. An MDI "child" window must not be on the same execution thread as the APPLICATION. Therefore, any MDI "child" window called directly from the APPLICATION must be in a separate procedure so the START function can be used to begin a new execution thread. Once started, multiple MDI "child" windows may be called in the new thread.

A non-MDI WINDOW operates independently of any previously opened APPLICATION. It will, however, disable an APPLICATION if it or any of its MDI "child" windows are on the same execution thread. This makes a non-MDI window opened in an MDI program an "application modal" window which effectively disables the application while the user has the window open (unless it is opened in its own execution thread). It does not, however, prevent the user from changing to another application running under Windows.

Example:

```
 !An MDI child Window:
Win2 WINDOW,MDI
     END
```

MODAL (set system modal window)

MODAL

The **MODAL** attribute specifies the WINDOW is "system modal." This means that no other window (in the same or any other concurrent program) can receive focus while the MODAL window has focus--the MODAL window has exclusive control of the computer. MODAL windows are usually used for error messages, or messages which require immediate attention by the user, such as: "Please insert a disk in drive A:."

A WINDOW without the MODAL attribute, may be "application-modal" or "modeless." An application-modal window is a non-MDI window opened as the top window of an MDI execution thread. An application-modal window restricts the user from moving to another execution thread in the same application, but does not restrict them from changing to another Windows program.

A modeless window is an MDI "child" WINDOW (with the MDI attribute) without the MODAL attribute. From a modeless window, The top window on other execution threads may be selected by the mouse, keyboard, or menu commands. If so, the other window takes focus and becomes uppermost on the video display. Any window not on the top of its execution thread may not be selected to receive focus, even from a modeless window.

Example:

```
Win2 WINDOW,MODAL!A system-modal Window
     END
```

MSG (set window status bar message)

MSG(*text***)**

MSG	Specifies *text* to display in the status bar.
text	A string constant containing the message to display in the status bar.

The **MSG** attribute on an APPLICATION or WINDOW structure specifies the *text* to display in the first zone of the status bar when the control with focus has no MSG attribute of its own.

Example:

```
WinOne WINDOW,AT(0,0,160,400),MSG(´Enter Data´)       !Default MSG to use
         COMBO(@S8),AT(120,120,20,20),USE(?C1),FROM(Q1:F2),MSG(´Enter or Select´)
         TEXT,AT(20,0,40,40),USE(E2)                  !Default MSG used
         ENTRY(@S8),AT(100,200,20,20),USE(E2)         !Default MSG used
         CHECK(´&A´),AT(0,120,20,20),USE(?C7),MSG(´On or Off´)
         OPTION(´Option 1´),USE(OptVar),MSG(´Pick One or Two´)
           RADIO(´Radio 1´),AT(120,0,20,20),USE(?R1)
           RADIO(´Radio 2´),AT(140,0,20,20),USE(?R2)
         END
       END
```

PALETTE (set number of hardware colors)

PALETTE(*colors***)**

PALETTE Specifies the number of hardware colors displayed in the window.

colors An integer constant specifying the number of hardware colors displayed in the window.

The **PALETTE** attribute on a WINDOW specifies the number of hardware colors used in the window for graphics display. This enforces a particular set of colors for the graphics.

Example:

```
WinOne WINDOW,AT(0,0,160,400),PALETTE(256)          !Display 256-color
         IMAGE,AT(120,120,20,20),USE(ImageField)
       END
```

STATUS (set status bar)

STATUS([*widths*] **)**

STATUS	Specifies the presence of a status bar.
widths	A list of integer constants (separated by commas) specifying the size of each zone in the status bar. If omitted, the status bar has one zone the width of the window.

The **STATUS** attribute specifies the presence of a status bar at the base of the APPLICATION or WINDOW. The status bar of an MDI WINDOW is always displayed at the bottom of the APPLICATION. A WINDOW without the MDI attribute displays its status bar at the base of the WINDOW. If the STATUS attribute is not present on the APPLICATION or WINDOW, there is no status bar.

The status bar may be divided into multiple zones specified by the *widths* parameters. The size of each zone is specified in dialog units. A negative value indicates the zone is expandable, but has a minimum width indicated by the parameter´s absolute value. If no *widths* parameters are specified, a single expanding zone with no minimum width is created, which is equivalent to a STATUS(-1).

The first zone of the status bar is always used to display MSG attributes. The MSG attribute string is displayed in the status bar as long as its control field still has input focus. A control or menu item without a MSG attribute causes the status bar to revert to its former state (either blank or displaying the text previously displayed in the zone).

Text may be placed in, or retrieved from, any zone of the status bar using the runtime property assignment syntax. The text remains present until replaced. The status bar configuration can also be changed dynamically by using the runtime property assignment syntax.

Example:

```
 !An APPLICATION with a one-zone status bar:
MainWin APPLICATION,STATUS
       END

 !A WINDOW with a two-zone status bar:
Win1 WINDOW,STATUS(160,160)
     END
```

SYSTEM (set system menu)

SYSTEM

The **SYSTEM** attribute specifies the presence of a Windows system menu (also called the control menu) on the APPLICATION or WINDOW. This menu contains standard Windows menu selections, such as: Close, Minimize, Maximize (the window), and Switch To (another window). The actual selections available on a given window depend upon the attributes set for that window.

Example:

```
 !An APPLICATION with a system menu:
MainWin APPLICATION,SYSTEM
       END
 !A WINDOW with a system menu:
Win1 WINDOW,SYSTEM
     END
```

TOOLBOX (set toolbox window behavior)

TOOLBOX

The **TOOLBOX** attribute specifies a WINDOW that is "always on top." Neither the WINDOW nor its controls retain input focus. This creates control behavior as if all the controls in the WINDOW had the SKIP attribute. Normally, a WINDOW with the TOOLBOX attribute executes in its own execution thread to provide a set of tools to the window with input focus. The MSG attributes of the controls in the window appear in the status bar when the mouse cursor is positioned over the control.

Example:

```
 PROGRAM
MainWin APPLICATION(´My Application´)
        MENUBAR
          MENU(´File´),USE(?FileMenu)
            ITEM(´E&xit´),USE(?MainExit),LAST
          END
          MENU(´Edit´),USE(?EditMenu)
            ITEM(´Use Tools´),USE(?UseTools)
       . . .
Pre:Field      STRING(400)
UseToolsThread BYTE
ToolsThread    BYTE
  CODE
  OPEN(MainWin)
  ACCEPT
    CASE ACCEPTED()
    OF ?MainExit
      BREAK
    OF ?UseTools
      UseToolsThread = START(UseTools)
  . .

UseTools PROCEDURE                  !A procedure that uses a toolbox
MDIChild WINDOW(´Use Tools Window´),MDI
         TEXT,HVSCROLL,USE(Pre:Field)
         BUTTON(´&OK´),USE(?Exit),DEFAULT
       END
  CODE
  OPEN(MDIChild)                   !Open the window
  DISPLAY                          ! and display it
  ToolsThread = START(Tools)          !Pop up the toolbox
  ACCEPT
    CASE EVENT()                        !Check for user-defined events
    OF 401h                             ! posted by toolbox controls
      Pre:Field += ´ ´ & FORMAT(TODAY(),@D1) ! append date to end of field
    OF 402h
      Pre:Field += ´ ´ & FORMAT(CLOCK(),@T1) ! append time to end of field
    END
    CASE ACCEPTED()
    OF ?Exit
      POSTEVENT(400h,,ToolsThread)         !Signal to close tools window
      BREAK
  . .
  CLOSE(MDIChild)

Tools  PROCEDURE   !The toolbox procedure
Win1 WINDOW(´Tools´),TOOLBOX
      BUTTON(´Date´),USE(?Button1)
```

```
    BUTTON(´Time´),USE(?Button2)
   END
CODE
OPEN(Win1)
ACCEPT
  IF EVENT() = 400h THEN BREAK.             !Check for close window signal
  CASE ACCEPTED()
  OF ?Button1
    POSTEVENT(401h,,UseToolsThread)        !Post datestamp signal
  OF ?Button2
    POSTEVENT(402h,,UseToolsThread)    !Post timestamp signal
. .
CLOSE(Win1)
```

TIMER (set periodic event)

TIMER(*period***)**

TIMER	Specifies a periodic event.
period	An integer constant or constant expression specifying the interval between timed events, in hundredths of a second.

The **TIMER** attribute specifies generation of a periodic field-independent event whenever the time *period* passes. EQUATES.CLW contains EVENT:Timer which equates the timer-generated event. The FOCUS() function returns the number of the control that currently has focus at the time of the event.

Example:

```
RunClock PROCEDURE
ShowTime LONG

 !A WINDOW with a timed event occurring every second:
Win1 WINDOW,TIMER(100)
      STRING(@T4),USE(ShowTime)
    END
  CODE
  OPEN(Win1)
  ShowTime = CLOCK()
  ACCEPT
    CASE EVENT()
    OF EVENT:Timer
      ShowTime = CLOCK()
      DISPLAY
    END
  END
  CLOSE(Win1)
```

MENUBAR and TOOLBAR Structures

MENUBAR (declare a pulldown menu)

```
MENUBAR [, NOMERGE ]
   [ MENU( )
       [ ITEM( ) ]
       [ MENU( )
           [ ITEM( ) ]
         END ]
     END ]
   [ ITEM( ) ]
END
```

MENUBAR Declares the menu for an APPLICATION or WINDOW.

NOMERGE Specifies menu merging behavior.

MENU A menu item with an associated drop box containing other menu selections.

ITEM A menu item for selection.

The **MENUBAR** structure declares the pulldown menu selections displayed for an APPLICATION or WINDOW. MENUBAR must appear in the source code before any TOOLBAR or controls.

On an APPLICATION, the MENUBAR defines the Global menu selections for the program. These are active and available on all MDI "child" windows (unless the window´s own MENUBAR structure has the NOMERGE attribute). If the NOMERGE attribute is specified on the APPLICATION´s MENUBAR, then the menu is a local menu displayed only when no MDI child windows are open and there is no global menu.

On an MDI WINDOW, the MENUBAR defines menu selections that are automatically merged with the Global menu. Both the Global and the window´s menu selections are then active while the MDI "child" window has input focus. Once the window loses focus, its specific menu selections are removed from the Global menu. If the NOMERGE attribute is specified on an MDI WINDOW´s MENUBAR, the menu overwrites and replaces the Global menu.

On a non-MDI WINDOW, the MENUBAR is never merged with the Global menu. A MENUBAR on a non-MDI WINDOW always appears in the WINDOW, not on any APPLICATION which may have been previously opened.

Events generated by local menu items are sent to the WINDOW´s ACCEPT loop in the normal way. Events generated by global menu items are sent to the active event loop of the thread which opened the APPLICATION (in a normal multi-thread application this means the APPLICATION´s own ACCEPT loop).

Dynamic changes to menu items which reference the currently active window affect only the currently displayed menu, even if global items are changed. Changes made to the Global menu items when the APPLICATION is the current window, or which reference the global APPLICATION window affect the global portions of all menus, whether already open or not.

When a WINDOW´s MENUBAR is merged into an APPLICATION´s MENUBAR, the global menu selections appear first, followed by the local menu selections, unless the FIRST or LAST attributes are specified on individual menu selections.

Example:

```
 !An MDI application frame window with main menu for the application:
MainWin APPLICATION(´My Application´)
         MENUBAR
           MENU(´File´),USE(?FileMenu)
             ITEM(´Open...´),USE(?OpenFile)
```

```
            ITEM(´Close´),USE(?CloseFile),DISABLE
            ITEM(´E&xit´),USE(?MainExit),LAST
           END
           MENU(´Edit´),USE(?EditMenu)
            ITEM(´Cu&t´),USE(?CutText),KEY(CtrlX),STD(STD:Cut),DISABLE
            ITEM(´Copy´),USE(?CopyText),KEY(CtrlC),STD(STD:Copy),DISABLE
            ITEM(´Paste´),USE(?PasteText),KEY(CtrlV),STD(STD:Paste),DISABLE
           END
           MENU(´Help´),USE(?HelpMenu),LAST
            ITEM(´Contents´),USE(?HelpContents),STD(STD:HelpIndex)
            ITEM(´Search for Help On...´),USE(?HelpSearch),STD(STD:HelpSearch)
            ITEM(´How to Use Help´),USE(?HelpOnHelp),STD(STD:HelpOnHelp)
            ITEM(´About MyApp...´),USE(?HelpAbout)
           END
         END
       END

 !An MDI child window with menu for the window, merged into the
 ! application´s menubar:
MDIChild WINDOW(´Child One´),MDI
          MENUBAR
            MENU(´File´),USE(?FileMenu)       !Merges into File menu
              ITEM(´Close´),USE(?CloseFile)     !Supercedes main menu selection
              ITEM(´Pick...´),USE(?PickFile)      !Added to menu selections
            END
            MENU(´Edit´),USE(?EditMenu)       !Merges into Edit menu
              ITEM(´Undo´),USE(?UndoText),KEY(CtrlZ),STD(STD:Undo)  !Added to menu
                                !These items supercede main menu selections:
              ITEM(´Cu&t´),USE(?CutText),KEY(CtrlX),STD(STD:Cut)
              ITEM(´Copy´),USE(?CopyText),KEY(CtrlC),STD(STD:Copy)
              ITEM(´Paste´),USE(?PasteText),KEY(CtrlV),STD(STD:Paste)
            END
            MENU(´Window´),STD(STD:WindowList),LAST
              ITEM(´Tile´),STD(STD:TileWindow)
              ITEM(´Cascade´),STD(STD:CascadeWindow)
            END
          END
          TEXT,HVSCROLL,USE(Pre:Field)
          BUTTON(´&OK´),USE(?Exit),DEFAULT
        END
 !An MDI window with its own menu, overwriting the main menu:
MDIChild2 WINDOW(´Dialog Window´),MDI,SYSTEM,MAX,STATUS
          MENUBAR,NOMERGE
            MENU(´File´),USE(?FileMenu)
              ITEM(´Close´),USE(?CloseFile)
            END
            MENU(´Edit´),USE(?EditMenu)
              ITEM(´Undo´),USE(?UndoText),KEY(CtrlZ),STD(STD:Undo)
              ITEM(´Cu&t´),USE(?CutText),KEY(CtrlX),STD(STD:Cut)
              ITEM(´Copy´),USE(?CopyText),KEY(CtrlC),STD(STD:Copy)
              ITEM(´Paste´),USE(?PasteText),KEY(CtrlV),STD(STD:Paste)
            END
          END
         TEXT,HVSCROLL,USE(Pre:Field),MSG(´Enter some text here´)
          BUTTON(´&OK´),USE(?Exit),DEFAULT
         END

 !A non-MDI window with its own menu:
NonMDI WINDOW(´Dialog Window´),SYSTEM,MAX,STATUS
        MENUBAR
```

```
    MENU(´File´),USE(?FileMenu)
      ITEM(´Close´),USE(?CloseFile)
    END
    MENU(´Edit´),USE(?EditMenu)
      ITEM(´Undo´),USE(?UndoText),KEY(CtrlZ),STD(STD:Undo)
      ITEM(´Cu&t´),USE(?CutText),KEY(CtrlX),STD(STD:Cut)
      ITEM(´Copy´),USE(?CopyText),KEY(CtrlC),STD(STD:Copy)
      ITEM(´Paste´),USE(?PasteText),KEY(CtrlV),STD(STD:Paste)
    END
  END
  TEXT,HVSCROLL,USE(Pre:Field),MSG(´Enter some text here´)
  BUTTON(´&OK´),USE(?Exit),DEFAULT
END
```

TOOLBAR (declare a tool bar)

TOOLBAR [,**AT( )**] [,**CURSOR( )**] [,**FONT( )**] [,**NOMERGE**]
controls
END

TOOLBAR	Declares tools for an APPLICATION or WINDOW.
AT	Specifies the initial size of the toolbar. If omitted, default values are selected by the runtime library.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the TOOLBAR. If omitted, the WINDOW or APPLICATION structure´s CURSOR attribute is used, else the Windows default cursor is used.
FONT	Specifies the default display font for the controls in the TOOLBAR.
NOMERGE	Specifies tools merging behavior.
controls	Control field declarations that define the available tools.

The **TOOLBAR** structure declares the tools displayed for an APPLICATION or WINDOW. On an APPLICATION, the TOOLBAR defines the Global tools for the program. If the NOMERGE attribute is specified on the APPLICATION´s TOOLBAR, the tools are local and are displayed only when no MDI child windows are open; there are no global tools. Global tools are active and available on all MDI "child" windows unless an MDI "child" window´s TOOLBAR structure has the NOMERGE attribute. If so, the "child" window´s tools overwrite the Global tools.

On an MDI WINDOW, the TOOLBAR defines tools that are automatically merged with the Global toolbar. Both the Global and the window´s tools are then active while the MDI "child" window has input focus. Once the window loses focus, its specific tools are removed from the Global toolbar. If the NOMERGE attribute is specified on an MDI WINDOW´s TOOLBAR, the tools overwrite and replace the Global toolbar. On a non-MDI WINDOW, the TOOLBAR is never merged with the Global menu. A TOOLBAR on a non-MDI WINDOW always appears in the WINDOW, not on any APPLICATION which may have been previously opened.

Events generated by local tools are sent to the WINDOW´s ACCEPT loop in the normal way. Events generated by global tools are sent to the active event loop of the thread which opened the APPLICATION. In a normal multi-thread application, this means the APPLICATION´s own ACCEPT loop.

TOOLBAR controls generate events in the normal manner. However, they do not keep the focus, and cannot be operated from the keyboard unless accelerator keys are provided. As soon as user interaction with a TOOLBAR control is done, focus returns to the window and local control which previously had it.

Dynamic changes to tools which reference the currently active window affect only the currently displayed toolbar, even if global tools are changed. Changes made to the Global toolbar when the APPLICATION is the current window, or which reference the global APPLICATION´s window affect the global portions of all toolbars, whether already open or not.

When a WINDOW´s TOOLBAR is merged into an APPLICATION´s TOOLBAR, the global tools appear first, followed by the local tools. The toolbars are merged so that the fields in the WINDOW´s toolbar begin just right of the position specified by the value of the width parameter of the APPLICATION TOOLBAR´s AT attribute. The height of the displayed toolbar is the maximum height of the "tallest" tool, whether global or local. If any part of a control falls below the bottom, the height is increased accordingly.

Example:

```
!An MDI application frame window containing the
! main menu and toolbar for the application:
```

```
MainWin APPLICATION(´My Application´),SYSTEM,MAX,ICON(´MyIcon.ICO´),STATUS |
   ,HVSCROLL,RESIZE
         MENUBAR
           ITEM(´E&xit´),USE(?MainExit)
         END
         TOOLBAR
           BUTTON(´Exit´),USE(?MainExitButton)
         END
       END
 !An MDI child window with toolbar for the window, merged into the
 ! application´s toolbar:
MDIChildWINDOW(´Child One´),MDI
           TOOLBAR
             BUTTON(´Cut´),USE(?CutButton),STD(STD:Cut)
             BUTTON(´Copy´),USE(?CopyButton),STD(STD:Copy)
             BUTTON(´Paste´),USE(?PasteButton),STD(STD:Paste)
           END
           TEXT,HVSCROLL,USE(Pre:Field)
           BUTTON(´&OK´),USE(?Exit),DEFAULT
         END
 !An MDI window with its own toolbar, overwriting the main toolbar:
MDIChild2 WINDOW(´Dialog Window´),MDI,SYSTEM,MAX,STATUS
           TOOLBAR,NOMERGE
             BUTTON(´Cut´),USE(?CutButton),STD(STD:Cut)
             BUTTON(´Copy´),USE(?CopyButton),STD(STD:Copy)
             BUTTON(´Paste´),USE(?PasteButton),STD(STD:Paste)
           END
           TEXT,HVSCROLL,USE(Pre:Field),MSG(´Enter some text here´)
           BUTTON(´&OK´),USE(?Exit),DEFAULT
         END
 !A non-MDI window with its own toolbar:
NonMDI WINDOW(´Dialog Window´),SYSTEM,MAX,STATUS
         TOOLBAR
           BUTTON(´Cut´),USE(?CutButton),STD(STD:Cut)
           BUTTON(´Copy´),USE(?CopyButton),STD(STD:Copy)
           BUTTON(´Paste´),USE(?PasteButton),STD(STD:Paste)
         END
         TEXT,HVSCROLL,USE(Pre:Field),MSG(´Enter some text here´)
         BUTTON(´&OK´),USE(?Exit),DEFAULT
       END
```

MENUBAR and TOOLBAR Attributes

CURSOR (set toolbar mouse cursor type)

CURSOR(*file*)

CURSOR Specifies a mouse cursor to display for the TOOLBAR.

file A string constant containing the name of a .CUR file, or an EQUATE naming a Windows-standard mouse cursor. The .CUR file is linked into the .EXE as a resource.

The **CURSOR** attribute specifies a mouse cursor to be displayed when the mouse is positioned over the TOOLBAR. This cursor is inherited by the controls in the toolbar unless overridden.

EQUATE statements for the Windows-standard mouse cursors are contained in the EQUATES.CLW file. The following list is a representative sample of these (see EQUATES.CLW for the complete list):

CURSOR:None	No mouse cursor
CURSOR:Arrow	Normal windows arrow cursor
CURSOR:IBeam	Capital "I" like a steel I-beam
CURSOR:Wait	Hourglass
CURSOR:Cross	Large plus sign
CURSOR:UpArrow	Vertical arrow
CURSOR:Size	Four-headed arrow
CURSOR:Icon	Box within a box
CURSOR:SizeNWSE	Double-headed arrow slanting left
CURSOR:SizeNESW	Double-headed arrow slanting right
CURSOR:SizeWE	Double-headed horizontal arrow
CURSOR:SizeNS	Double-headed vertical arrow
CURSOR:DragWE	Double-headed horizontal arrow

Example:

```
 !Toolbar with large plus sign cursor
WinOne WINDOW
         TOOLBAR,CURSOR(´CURSOR:Cross´)
           BUTTON(´Cut´),USE(?CutButton),STD(STD:Cut)
           BUTTON(´Copy´),USE(?CopyButton),STD(STD:Copy)
           BUTTON(´Paste´),USE(?PasteButton),STD(STD:Paste)
         END
       END
```

FONT (set toolbar default font)

FONT([*typeface*] [,*size*] [,*color*] [,*style*]**)**

FONT	Specifies the default display font for the TOOLBAR.
typeface	A string constant containing the name of the font. If omitted, the system font is used.
size	An integer constant containing the size (in points) of the font. If omitted, the system default font size is used.
color	A LONG integer constant containing the red, green, and blue values for the color of the font in the low-order three bytes, or an EQUATE for a standard Windows color value. If omitted, black is used.
style	An integer constant or constant expression or EQUATE specifying the strike weight and style of the font. If omitted, the weight is normal.

The **FONT** attribute on a TOOLBAR structure specifies the default display font for all controls in the TOOLBAR that do not have a FONT attribute. The *typeface* may name any font registered in the Windows system. The EQUATES.CLW file contains EQUATE values for standard *style* values. A *style* on the range zero (0) to one thousand (1000) specifies the strike weight of the font. You may add to that values that indicate italic, underline, or strikeout text. The following EQUATES are in EQUATES.CLW:

```
FONT:thin          EQUATE(100)
FONT:regular           EQUATE(400)
FONT:bold          EQUATE(700)
FONT:fixed         EQUATE (0800H)
FONT:italic        EQUATE (01000H)
FONT:underline         EQUATE (02000H)
FONT:strikeout         EQUATE (04000H)
```

Example:

```
Win1 WINDOW  !A toolbar using 14 point Times New Roman
       TOOLBAR,FONT('Times New Roman',14,00H)
         BUTTON('Cut'),USE(?CutButton),STD(STD:Cut)
         BUTTON('Paste'),USE(?PasteButton),STD(STD:Paste)
       END
     END

Win2 WINDOW  !14 point Times New Roman, Bold and Italic
       TOOLBAR,FONT('Times New Roman',14,00H,FONT:italic+FONT:bold)
         BUTTON('Cut'),USE(?CutButton),STD(STD:Cut)
         BUTTON('Paste'),USE(?PasteButton),STD(STD:Paste)
       END
     END
```

NOMERGE (set merging behavior)

NOMERGE

The **NOMERGE** attribute indicates that the MENUBAR or TOOLBAR on a WINDOW should not be merged with the Global menu or toolbar.

The NOMERGE attribute on an APPLICATION´s MENUBAR indicates that the menu is local and to be displayed only when no MDI "child" windows are open and that there is no Global menu. The NOMERGE attribute on an APPLICATION´s TOOLBAR indicates that the tools are local and to be displayed only when no MDI "child" windows are open and that there are no Global tools.

Without the NOMERGE attribute, an MDI WINDOW´s menu and toolbar are automatically merged with the global menu and toolbar, and then displayed in the APPLICATION menu and toolbar. When NOMERGE is specified, the WINDOW´s menu and toolbar overwrite the Global menu and toolbar. The menu and toolbar displayed when the WINDOW has focus are only the WINDOW´s own menu and toolbar. However, they are still displayed on the APPLICATION.

A MENUBAR or TOOLBAR specified in a non-MDI WINDOW is never merged with the Global menu or toolbar--they appear in the WINDOW.

Example:

```
 !An MDI application frame window with local-only menu and toolbar:
MainWin  APPLICATION(´My Application´),SYSTEM,MAX,ICON(´MyIcon.ICO´),STATUS
         MENUBAR,NOMERGE
           ITEM(´E&xit´),USE(?MainExit)
         END
         TOOLBAR,NOMERGE
           BUTTON(´Exit´),USE(?MainExitButton)
         END
        END

 !MDI window with its own menu and toolbar, overwriting the application´s:
MDIChildWINDOW(´Dialog Window´),MDI,SYSTEM,MAX,STATUS
           MENUBAR,NOMERGE
             MENU(´Edit´),USE(?EditMenu)
               ITEM(´Cu&t´),USE(?CutText),KEY(CtrlX),STD(STD:Cut)
               ITEM(´Copy´),USE(?CopyText),KEY(CtrlC),STD(STD:Copy)
               ITEM(´Paste´),USE(?PasteText),KEY(CtrlV),STD(STD:Paste)
             END
           END
           TOOLBAR,NOMERGE
             BUTTON(´Cut´),USE(?CutButton),STD(STD:Cut)
             BUTTON(´Copy´),USE(?CopyButton),STD(STD:Copy)
             BUTTON(´Paste´),USE(?PasteButton),STD(STD:Paste)
           END
           TEXT,HVSCROLL,USE(Pre:Field),MSG(´Enter some text here´)
           BUTTON(´&OK´),USE(?Exit),DEFAULT
         END
```

MENUBAR Controls

- MENU (declare a menu box)
- ITEM (declare a menu item)

MENU (declare a menu box)

MENU(*text*) [,**USE**( )] [,**KEY**( )] [,**MSG**( )] [,**HLP**( )] [,**STD**( )] [,**RIGHT**] [,**DISABLE**]
[, | **FIRST** |]
| **LAST** |

MENU	Declares a menu box within a MENUBAR.
text	A string constant containing the display text for the menu selection.
USE	A field equate label to reference the menu selection in executable code.
KEY	Specifies an integer constant or keycode equate that immediately opens the menu.
MSG	Specifies a string constant containing the text to display in the status bar when the menu is pulled down.
HLP	Specifies a string constant containing the help system identifier for the menu.
STD	Specifies an integer constant or equate that identifies a "Windows standard behavior" for the menu.
RIGHT	Specifies the MENU appears at the far right of the action bar.
FIRST	Specifies the MENU appears at the left or top of the menu when merged.
LAST	Specifies the MENU appears at the right or bottom of the menu when merged.
DISABLE	Specifies the menu appears dimmed when the WINDOW or APPLICATION is first opened.

MENU declares a drop-down or cascading menu box structure within a MENUBAR structure. When the MENU is selected, the MENU and/or ITEM statements within the structure are displayed in a menu box. A MENU is not required to have any MENUs or ITEMs in it. A menu box usually appears (drops down) immediately below its *text* on the menu bar (or above, if there is no room below). When selected with ENTER or RIGHT ARROW, any subsequent menu drop-box appears (cascades) immediately to the right of the MENU *text* in the preceding menu box (or left, if there is no room to the right). LEFT ARROW backs up to the preceding menu. The KEY attribute designates a separate accelerator key for the field. This may be any valid Clarion keycode to immediately pull down the MENU.

The *text* string may contain an ampersand (&) which designates the following character as the accelerator "hot" key which is automatically underlined. If the MENU is on the menu bar, pressing the Alt key together with the accelerator key highlights and displays the MENU. If the MENU is within another MENU, pressing the accelerator key, alone, highlights and executes the MENU. If there is no ampersand in the *text*, the first non-blank character in the *text* string is the accelerator key for the MENU, but it will not be underlined. To include an ampersand as part of the *text*, place two ampersands together (&&) in the *text* string and only one will display.

Example:

```
 !An MDI application frame window with main menu for the application:
MainWin APPLICATION(´My Application´),SYSTEM,MAX,ICON(´MyIcon.ICO´),STATUS |
   ,HVSCROLL,RESIZE
        MENUBAR
          MENU(´File´),USE(?FileMenu),FIRST
            ITEM(´Open...´),USE(?OpenFile)
            ITEM(´Close´),USE(?CloseFile),DISABLE
            ITEM(´E&xit´),USE(?MainExit)
          END
          MENU(´Edit´),USE(?EditMenu),KEY(CtrlE),HLP(´EditMenuHelp´)
            ITEM(´Undo´),USE(?UndoText),KEY(CtrlZ),STD(STD:Undo),DISABLE
```

```
      ITEM(´Cu&t´),USE(?CutText),KEY(CtrlX),STD(STD:Cut),DISABLE
      ITEM(´Copy´),USE(?CopyText),KEY(CtrlC),STD(STD:Copy),DISABLE
      ITEM(´Paste´),USE(?PasteText),KEY(CtrlV),STD(STD:Paste),DISABLE
    END
    MENU(´Window´),STD(STD:WindowList),MSG(´Arrange or Select Window´),LAST
      ITEM(´Tile´),STD(STD:TileWindow)
      ITEM(´Cascade´),STD(STD:CascadeWindow)
      ITEM(´Arrange Icons´),STD(STD:ArrangeIcons)
    END
    MENU(´Help´),USE(?HelpMenu),LAST,RIGHT
      ITEM(´Contents´),USE(?HelpContents),STD(STD:HelpIndex)
      ITEM(´Search for Help On...´),USE(?HelpSearch),STD(STD:HelpSearch)
      ITEM(´How to Use Help´),USE(?HelpOnHelp),STD(STD:HelpOnHelp)
      ITEM(´About MyApp...´),USE(?HelpAbout)
    END
  END
END
```

ITEM (declare a menu item)

ITEM(*text***) [,USE( )] [,KEY( )] [,MSG( )] [,HLP( )] [,STD( )] [,CHECK] [,DISABLE]**
[, | FIRST |] [,SEPARATOR]
| LAST |

ITEM	Declares a menu choice within a MENUBAR or MENU structure.
text	A string constant containing the display text for the menu item.
USE	A field equate label to reference the menu item in executable code, or the variable used with CHECK.
KEY	Specifies an integer constant or keycode equate that immediately executes the menu item.
MSG	Specifies a string constant containing the text to display in the status bar when the menu item is highlighted.
HLP	Specifies a string constant containing the help system identifier for the menu item.
STD	Specifies an integer constant or equate that identifies a "Windows standard action" the menu item executes.
CHECK	Specifies an on/off ITEM.
DISABLE	Specifies the menu item appears dimmed when the WINDOW or APPLICATION is first opened.
FIRST	Specifies the ITEM appears at the top of the menu when menus are merged.
LAST	Specifies the ITEM appears at the bottom of the menu when menus are merged.
SEPARATOR	Specifies the ITEM displays a solid horizontal line across the menu box at run-time to delimit groups of menu selections. No other attributes may be specified with SEPARATOR.

ITEM declares a menu choice within a MENUBAR or MENU structure. The *text* string may contain an ampersand (&) which designates the following character as an accelerator "hot" key which is automatically underlined. If the ITEM is on the menu bar, pressing the Alt key together with the accelerator key highlights and executes the ITEM. If the ITEM is in a MENU, pressing the accelerator key, alone, when the menu is displayed, highlights and executes the ITEM. If there is no ampersand in the *text*, the first non-blank character in the *text* string is the accelerator key for the ITEM, which will not be underlined. To include an ampersand as part of the *text*, place two ampersands together (&&) in the *text* string and only one will display.

The KEY attribute designates a separate "hot" key for the field. This may be any valid Clarion keycode to immediately execute the ITEM´s action.

A cursor bar highlights individual ITEMs within the MENU structure. Each ITEM is usually associated with some code to be executed upon selection of that ITEM, unless the STD attribute is present. The STD atribute specifies a standard Windows action the menu item performs, such as Tile or Cascade the windows.

The SEPARATOR attribute creates an ITEM which serves only to delimit groups of menus selections so it should not have a *text* parameter, nor any other attributes. It creates a solid horizontal line across the menu box.

An ITEM that is not within a MENU structure is placed on the menu bar. This creates a menu bar selection which has no related drop-down menu. The normal convention to indicate this to the user is to terminate the *text* displayed for the item with an exclamation point (!). For example, the *text* for the ITEM

might contain ´Exit!´ to alert the user to the executable nature of the menu choice.

Example:

```
 !An MDI application frame window with main menu for the application:
MainWin APPLICATION(´My Application´),SYSTEM,MAX,ICON(´MyIcon.ICO´),STATUS |
   ,HVSCROLL,RESIZE
        MENUBAR
          ITEM(´E&xit!´),USE(?MainExit),FIRST
          MENU(´File´),USE(?FileMenu),FIRST
            ITEM(´Open...´),USE(?OpenFile) ,HLP(´OpenFileHelp´) ,FIRST
            ITEM(´Close´),USE(?CloseFile),HLP(´CloseFileHelp´),DISABLE
            ITEM(´Auto Increment´),USE(ToggleVar),CHECK
          END
          MENU(´Edit´),USE(?EditMenu),KEY(CtrlE),HLP(´EditMenuHelp´)
            ITEM(´Undo´),USE(?UndoText),KEY(CtrlZ),STD(STD:Undo),DISABLE
            ITEM,SEPARATOR
            ITEM(´Cu&t´),USE(?CutText),KEY(CtrlX),STD(STD:Cut),DISABLE
            ITEM(´Copy´),USE(?CopyText),KEY(CtrlC),STD(STD:Copy),DISABLE
            ITEM(´Paste´),USE(?PasteText),KEY(CtrlV),STD(STD:Paste),DISABLE
          END
          MENU(´Window´),STD(STD:WindowList),MSG(´Arrange or Select Window´),LAST
            ITEM(´Tile´),STD(STD:TileWindow)
            ITEM(´Cascade´),STD(STD:CascadeWindow)
            ITEM(´Arrange Icons´),STD(STD:ArrangeIcons)
            ITEM,SEPARATOR
          END
          MENU(´Help´),USE(?HelpMenu),LAST,RIGHT
            ITEM(´Contents´),USE(?HelpContents),STD(STD:HelpIndex)
            ITEM(´Search for Help On...´),USE(?HelpSearch),STD(STD:HelpSearch)
            ITEM(´How to Use Help´),USE(?HelpOnHelp),STD(STD:HelpOnHelp)
            ITEM(´About MyApp...´),USE(?HelpAbout),MSG(´Copyright Info´),LAST
          END
        END
      END
```

TOOLBAR and WINDOW Control Fields

- BOX (declare a window box control)
- BUTTON (declare a pushbutton control)
- CHECK (declare a window checkbox control)
- COMBO (declare an entry/list control)
- CUSTOM (declare a window .VBX custom control)
- ELLIPSE (declare a window ellipse control)
- ENTRY (declare a data entry control)
- GROUP (declare a group of window controls)
- IMAGE (declare a window graphic image control)
- LINE (declare a window line control)
- LIST (declare a window list control)
- OPTION (declare a group of window RADIO controls)
- PROMPT (declare a prompt control)
- RADIO (declare a window radio button control)
- REGION (declare a window region control)
- SPIN (declare a spinning list control)
- STRING (declare a window string control)
- TEXT (declare a multi-line data entry control)

BOX (declare a window box control)

BOX ,**AT( )** [,**USE( )**] [,**DISABLE**] [,**COLOR( )**] [,**FILL( )**] [,**ROUND**] [,**FULL**] [,**SCROLL**] [,**HIDE**]

BOX	Places a rectangular box on the window.
AT	Specifies the initial size and location of the control. If omitted, default values are selected by the runtime library.
USE	A field equate label to reference the control in executable code.
DISABLE	Specifies the control appears dimmed when the WINDOW (or APPLICATION) is first opened.
COLOR	Specifies the color for the border of the control. If omitted, the border is black.
FILL	Specifies the fill color for the control. If omitted, the box is not filled with color.
ROUND	Specifies the box corners are rounded. If omitted, the corners are square.
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter.
SCROLL	Specifies the control scrolls with the window.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened. UNHIDE must be used to display it.

The **BOX** control places a rectangular box on the WINDOW (or TOOLBAR) at the position and size specified by its AT attribute. This control cannot receive input focus and does not generate events.

Example:

```
MDIChild WINDOW(´Child One´),AT(0,0,320,200),MDI,MAX,HVSCROLL
         BOX,FILL(COLOR:MENU),FULL       !Filled, full screen, black border
         BOX,AT(0,0,20,20)              !Unfilled, black border
         BOX,AT(0,20,20,20),USE(?Box1),DISABLE
                                        !Unfilled, black border, dimmed
         BOX,AT(20,20,20,20),ROUND       !Unfilled, rounded, black border
         BOX,AT(40,40,20,20),FILL(COLOR:ACTIVEBORDER)
                                        !Filled, black border
         BOX,AT(60,60,20,20),COLOR(COLOR:ACTIVEBORDER)
                                        !Unfilled, active border color border
         BOX,AT(480,180,20,20),SCROLL      !Scrolls with screen
        END
```

BUTTON (declare a pushbutton control)

BUTTON(*text*) ,**AT( )** [,**CURSOR( )**] [,**USE( )**] [,**DISABLE**] [,**KEY( )**] [,**MSG( )**] [,**HLP( )**] [,**SKIP**] [,**STD( )**] [,**FONT( )**] [,**ICON( )**] [,**DEFAULT**] [,**IMM**][,**REQ**] [,**FULL**] [,**SCROLL**] [,**ALRT( )**] [,**HIDE**] [**DROPID( )**]

BUTTON	Places a pushbutton on the WINDOW or TOOLBAR.
text	A string constant containing the text to display on the button. This may contain an ampersand (&) to indicate the "hot" letter (accelerator key) for the button.
AT	Specifies the initial size and location of the control. If omitted, default values are selected by the runtime library.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control. If omitted, the WINDOW´s CURSOR attribute is used, else the Windows default cursor is used.
USE	A field equate label to reference the control in executable code.
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION is first opened.
KEY	Specifies an integer constant or keycode equate that immediately gives focus to and presses the button.
MSG	Specifies a string constant containing the text to display in the status bar when the control has focus.
HLP	Specifies a string constant containing the help system identifier for the control.
SKIP	Specifies the control does not receive input focus and may only be accessed with the mouse or accelerator key.
STD	Specifies an integer constant or equate that identifies a "Windows standard action" the control executes.
FONT	Specifies the display font for the control.
ICON	Specifies an .ICO file or standard icon to display on the button face.
DEFAULT	Specifies the BUTTON is automatically pressed when the user presses the ENTER key.
IMM	Specifies the control generates an event when the left mouse button is pressed, continuing as long as it is depressed. If omitted, an event is generated only when the left mouse button is pressed and released on the control.
REQ	Specifies that when the BUTTON is pressed, the runtime library automatically checks all ENTRY controls in the same WINDOW with the REQ attribute to ensure they contain data other than blanks or zeroes.
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter.
SCROLL	Specifies the control scrolls with the window.
ALRT	Specifies "hot" keys active for the control.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened. UNHIDE must be used to display it.
DROPID	Specifies the control may serve as a drop target for drag-and-drop actions.

The **BUTTON** control places a pushbutton on the WINDOW (or TOOLBAR) at the position and size specified by its AT attribute.

A BUTTON with the IMM attribute generates an event as soon as the left mouse button is pressed on the control and continues to do so until it is released. This allows the BUTTON control´s executable code to execute continuously until the mouse button is released. A BUTTON without the IMM attribute generates an event only when the left mouse button is pressed and released on the control.

A BUTTON with the REQ attribute is a "required control fields check" button. REQ attributes of ENTRY or TEXT control fields are not checked until a BUTTON with the REQ attribute is pressed or the INCOMPLETE function is called. Focus is given to the first required control which is blank or zero.

A BUTTON with an ICON attribute displays the icon on the button face in place of its *text* parameter. The *text* parameter then serves only for accelerator "hot" key definition.

Events Generated:

EVENT:Selected The control has received input focus.

EVENT:Accepted The control has been pressed by the user.

EVENT:PreAlertKey
The user pressed an ALRT attribute hot key.

EVENT:AlertKey The user pressed an ALRT attribute hot key.

EVENT:Drop A successful drag-and-drop to the control.

Example:

```
MDIChildWINDOW(´Child One´),AT(0,0,320,200),MDI,MAX,HVSCROLL
          BUTTON(´1´),AT(0,0,20,20),USE(?B1)
          BUTTON(´2´),AT(20,0,20,20),USE(?B2),KEY(F10Key)
          BUTTON(´3´),AT(40,0,20,20),USE(?B3),MSG(´Button 3´)
          BUTTON(´4´),AT(60,0,20,20),USE(?B4),HLP(´Button4Help´)
          BUTTON(´5´),AT(80,0,20,20),USE(?B5),STD(STD:Cut)
          BUTTON(´6´),AT(100,0,20,20),USE(?B6),FONT(´Arial´,12)
          BUTTON(´7´),AT(120,0,20,20),USE(?B7),ICON(ICON:Question)
          BUTTON(´8´),AT(140,0,20,20),USE(?B8),DEFAULT
          BUTTON(´9´),AT(160,0,20,20),USE(?B9),IMM
          BUTTON(´10´),AT(180,0,20,20),USE(?B10),CURSOR(CURSOR:Wait)
          BUTTON(´11´),AT(200,0,20,20),USE(?B11),REQ
          BUTTON(´12´),AT(220,0,20,20),USE(?B12),ALRT(F10Key)
          BUTTON(´13´),AT(240,0,20,20),USE(?B13),SCROLL
        END
  CODE
  OPEN(MDIChild)
  ACCEPT
    CASE ACCEPTED()
    OF ?B1
      !Perform some action
    END
  END
```

CHECK (declare a window checkbox control)

CHECK(*text***) ,AT( ) [,CURSOR( )] [,USE( )] [,DISABLE] [,KEY( )] [,MSG( )] [,HLP( )] [,SKIP]**
[,FONT( )] [,ICON( )] [,FULL] [,SCROLL] [,ALRT( )] [,HIDE] [DROPID( )]
[, | LEFT |]
| RIGHT |

CHECK	Places a check box on the WINDOW or TOOLBAR.
text	A string constant containing the text to display for the check box. This may contain an ampersand (&) to indicate the "hot" letter for the check box.
AT	Specifies the initial size and location of the control. If omitted, default values are selected by the runtime library.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control. If omitted, the WINDOW´s CURSOR attribute is used, else the Windows default cursor is used.
USE	The label of a numeric variable to receive the value of the check box, zero (0 = OFF) or one (1 = ON).
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION is first opened.
KEY	Specifies an integer constant or keycode equate that immediately gives focus to and toggles the box.
MSG	Specifies a string constant containing the text to display in the status bar when the control has focus.
HLP	Specifies a string constant containing the help system identifier for the control.
SKIP	Specifies the control does not receive input focus and may only be accessed with the mouse or accelerator key.
FONT	Specifies the display font for the control.
ICON	Specifies an .ICO file or standard icon to display on the button face of a "latching" pushbutton.
LEFT	Specifies that the text appears to the left of the check box.
RIGHT	Specifies that the text appears to the right of the check box (the default position).
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter.
SCROLL	Specifies the control scrolls with the window.
ALRT	Specifies "hot" keys active for the control.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened. UNHIDE must be used to display it.
DROPID	Specifies the control may serve as a drop target for drag-and-drop actions.

The **CHECK** control places a check box on the WINDOW (or TOOLBAR) at the position and size specified by its AT attribute.

A CHECK with an ICON attribute appears as a "latched" button with the icon displayed on the button face. When the button appears "up" the CHECK is off and the USE variable receives a zero (0); when it

appears "down" the CHECK is on and the USE variable receives a one (1).

Events Generated:

EVENT:Selected The control has received input focus.

EVENT:Accepted The control has been toggled by the user.

EVENT:PreAlertKey
The user pressed an ALRT attribute hot key.

EVENT:AlertKey The user pressed an ALRT attribute hot key.

EVENT:Drop A successful drag-and-drop to the control.

Example:

```
MDIChildWINDOW(´Child One´),AT(0,0,320,200),MDI,MAX,HVSCROLL
          CHECK(´1´),AT(0,0,20,20),USE(C1)
          CHECK(´2´),AT(0,20,20,20),USE(C2),KEY(F10Key)
          CHECK(´3´),AT(0,40,20,20),USE(C3),MSG(´Button 3´)
          CHECK(´4´),AT(0,60,20,20),USE(C4),HLP(´Check4Help´)
          CHECK(´5´),AT(20,80,20,20),USE(C5),LEFT
          CHECK(´6´),AT(0,100,20,20),USE(C6),FONT(´Arial´,12)
          CHECK(´7´),AT(0,120,20,20),USE(C7),ICON(ICON:Question)
          CHECK(´8´),AT(0,140,20,20),USE(C8),DEFAULT
          CHECK(´9´),AT(0,160,20,20),USE(C9),IMM
          CHECK(´10´),AT(0,180,20,20),USE(C10),CURSOR(CURSOR:Wait)
          CHECK(´11´),AT(0,200,20,20),USE(C11),ALRT(F10Key),DISABLE
        END
  CODE
  OPEN(MDIChild)
  ACCEPT
    CASE ACCEPTED()
    OF ?C1
      IF C1 = 1
        ENABLE(?C11)
      ELSE
        DISABLE(?C11)
      END
    END
  END
```

COMBO (declare an entry/list control)

```
COMBO(picture) ,FROM( ) ,AT( ) [,CURSOR( )] [,USE( )] [,DISABLE] [,KEY( )] [,MSG( )]
       [,HLP( )] [,SKIP] [,FONT( )] [,FORMAT( )] [,DROP] [,COLUMN] [,VCR] [,FULL]
       [,SCROLL] [,ALRT( )] [,HIDE] [,READONLY] [,REQ] [,NOBAR] [DROPID( )]
       [,  | MARK() |][,  | HSCROLL   |][,  | LEFT        |][,  | INS  |][,  | UPR |]
          | IMM      |   | VSCROLL   |   | RIGHT       |      | OVR  |     | CAP |
                         | HVSCROLL  |   | CENTER      |
                                         | DECIMAL     |
```

COMBO	Places a data entry field with an associated list of data items on the WINDOW or TOOLBAR.
picture	A display picture token that specifies the input format for the data entered into the control.
FROM	Specifies the origin of the data displayed in the list.
AT	Specifies the initial size and location of the control. If omitted, the runtime library chooses a value.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control. If omitted, the WINDOW´s CURSOR attribute is used, else the Windows default cursor is used.
USE	A field equate label to reference the control in executable code or the label of the variable that receives the value selected by the user.
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION is first opened.
KEY	Specifies an integer constant or keycode equate that immediately gives focus to the control.
MSG	Specifies a string constant containing the text to display in the status bar when the control has focus.
HLP	Specifies a string constant containing the help system identifier for the control.
SKIP	Specifies the control receives input focus to enter text only with the mouse or accelerator key and does not retain focus.
FONT	Specifies the display font for the control.
FORMAT	Specifies the display format of the data.
DROP	Specifies a drop-down list box and the number of elements the drop-down portion contains.
COLUMN	Specifies a field-by-field highlight bar on multi-column list boxes.
VCR	Specifies a VCR-type control to the left of the horizontal scroll bar (if present).
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter.
SCROLL	Specifies the control scrolls with the window.
ALRT	Specifies "hot" keys active for the control.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened. UNHIDE must be used to display it.

READONLY	Specifies the control does not allow data entry.
NOBAR	Specifies the highlight bar is displayed only when the LIST has focus.
DROPID	Specifies the control may serve as a drop target for drag-and-drop actions.
REQ	Specifies the control may not be left blank or zero.
MARK	Specifies multiple item selection mode.
IMM	Specifies generation of an event whenever the user presses any key.
HSCROLL	Specifies that a horizontal scroll bar is automatically added to the list box when any portion of the data item lies horizontally outside the visible area.
VSCROLL	Specifies that a vertical scroll bar is automatically added to the list box when any data items lie vertically outside the visible area.
HVSCROLL	Specifies that both vertical and horizontal scroll bars are automatically added to the list box when any portion of the data items lies outside the visible area.
LEFT	Specifies that the data is left justified within the list.
RIGHT	Specifies that the data is right justified within the list.
CENTER	Specifies that the data is centered within the list.
DECIMAL	Specifies that the data is aligned on the decimal point within the list.
INS / OVR	Specifies Insert or Overwrite entry mode (valid only on windows with the MASK attribute).
UPR / CAP	Specifies all upper case or proper name capitalization (First Letter Of Each Word Capitalized) entry.

The **COMBO** control places a data entry field with an associated list of data items on the WINDOW (or TOOLBAR) at the position and size specified by its AT attribute (a combination of an ENTRY and LIST control). The user may type in data or select an item from the list. The entered data is not automatically validated against the entries in the list. The data entry portion of the COMBO acts as an "incremental locator" to the list--as the user types each character, the highlight bar is positioned to the closest matching entry.

A COMBO with the DROP attribute displays only the currently selected data item on screen until the control has focus and the user presses the down arrow key, or CLICKS ON the the icon to the right of the displayed data item. When either of these occurs, the selection list appears ("drops down") to allow the user to select an item.

A COMBO with the IMM attribute generates an event every time the user moves the highlight bar to another selection, or pressed any key that causes the displayed entries to scroll. This allows an opportunity for the source code to re-fill the display QUEUE, or get the currently highlighted record to display other fields from the record. A COMBO with the VCR attribute has scroll control buttons like a **V**ideo **C**assette **R**ecorder to the left of the horizontal scroll bar (if there is one). These buttons allow the user to use the mouse to scroll through the list.

Events Generated:

EVENT:Selected	The control has received input focus.
EVENT:Accepted	The user has selected an entry.
EVENT:NewSelection	The current selection in the list has changed (highlight has moved up or down).

EVENT:PreAlertKey
The user pressed an ALRT attribute hot key.

EVENT:AlertKey The user pressed an ALRT attribute hot key.

EVENT:Drop A successful drag-and-drop to the control.

A COMBO with the IMM attribute also generates the following events:

EVENT:ScrollUp The highlight bar has attempted to move off the top of the LIST.

EVENT:ScrollDown
The highlight bar has attempted to move off the bottom of the LIST.

EVENT:PageUp The user pressed PgUp.

EVENT:PageDown
The user pressed PgDn.

EVENT:ScrollTop The user pressed Ctrl-PgUp.

EVENT:ScrollBottom
The user pressed Ctrl-PgDn.

EVENT:PreAlertKey
The user pressed a printable character or an ALRT attribute hot key.

EVENT:AlertKey The user pressed a printable character or an ALRT attribute hot key.

EVENT:Locate The user pressed the locator VCR button.

Example:

```
MDIChildWINDOW(´Child One´),AT(0,0,320,200),MDI,MAX,HVSCROLL
          COMBO(@S8),AT(0,0,20,20),USE(C1),FROM(Que)
          COMBO(@S8),AT(20,0,20,20),USE(C2),FROM(Que),KEY(F10Key)
          COMBO(@S8),AT(40,0,20,20),USE(C3),FROM(Que),MSG(´Button 3´)
          COMBO(@S8),AT(60,0,20,20),USE(C4),FROM(Que),HLP(´Check4Help´)
          COMBO(@S8),AT(80,0,20,20),USE(C5),FROM(Q) |
             ,FORMAT(´5C~List~15L~Box~´),COLUMN
          COMBO(@S8),AT(100,0,20,20),USE(C6),FROM(Que),FONT(´Arial´,12)
          COMBO(@S8),AT(120,0,20,20),USE(C7),FROM(Que),DROP(8)
          COMBO(@S8),AT(140,0,20,20),USE(C8),FROM(Que),HVSCROLL,VCR
          COMBO(@S8),AT(160,0,20,20),USE(C9),FROM(Que),IMM
          COMBO(@S8),AT(180,0,20,20),USE(C10),FROM(Que),CURSOR(CURSOR:Wait)
          COMBO(@S8),AT(200,0,20,20),USE(C11),FROM(Que),ALRT(F10Key)
          COMBO(@S8),AT(220,0,20,20),USE(C12),FROM(Que),LEFT
          COMBO(@S8),AT(240,0,20,20),USE(C13),FROM(Que),RIGHT
          COMBO(@S8),AT(260,0,20,20),USE(C14),FROM(Que),CENTER
          COMBO(@N8.2),AT(280,0,20,20),USE(C15),FROM(Que),DECIMAL
          END
  CODE
  OPEN(MDIChild)
  ACCEPT
    CASE ACCEPTED()
    OF ?C1
      LOOP X# = 1 to RECORDS(Que)     !Check for user´s entry in Que
        GET(Que,X#)
        IF C1 = Que THEN BREAK.      !Break loop if present
      END
      IF X# > RECORDS(Que)           !Check for BREAK
```

```
      Que = C1                    ! and add the entry
      ADD(Que)
    END
  END
END
```

See Also:

LIST

ENTRY

CUSTOM (declare a window .VBX custom control)

CUSTOM(*text***) ,AT( ) [,CLASS( )] [,CURSOR( )] [,USE( )] [,DISABLE] [,KEY( )] [,MSG( )]**
[,HLP( )] [,SKIP] [,FULL] [,SCROLL] [,ALRT( )] [,HIDE] [,FONT( )] [DROPID( )]
[,*property(value)***]**

CUSTOM	Places a Visual Basic .VBX control on the WINDOW or TOOLBAR.
text	A string constant containing the title for the control.
AT	Specifies the initial size and location of the control. If omitted, default values are selected by the control.
CLASS	Specifies the .VBX filename and type of control.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control. If omitted, the WINDOW´s CURSOR attribute is used, else the Windows default cursor is used.
USE	The label of a variable to receive the value of the control.
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION is first opened.
KEY	Specifies an integer constant or keycode equate that immediately gives focus to the control.
MSG	Specifies a string constant containing the text to display in the status bar when the control has focus.
HLP	Specifies a string constant containing the help system identifier for the control.
SKIP	Specifies the control does not receive input focus and may only be accessed with the mouse or accelerator key.
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter.
SCROLL	Specifies the control scrolls with the window.
ALRT	Specifies "hot" keys active for the control.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened. UNHIDE must be used to display it.
FONT	Specifies the display font for the control.
DROPID	Specifies the control may serve as a drop target for drag-and-drop actions.
property	A string constant containing the name of a custom property setting for the control.
value	A string constant containing the property value number or EQUATE for the *property*.

The **CUSTOM** control places a Visual Basic .VBX control on the WINDOW (or TOOLBAR) at the position and size specified by its AT attribute.

The *property* attribute allows you to specify any additional property settings the .VBX control may require. These are properties that need to be set for the .VBX control to properly function, and are not standard Clarion properties (such as AT, CURSOR, or USE). The custom control should only receive values for these properties that are defined for that control. Valid properties and values for those properties would be defined in the custom control´s documentation. You may have multiple *property* attributes on a single CUSTOM control.

Events Generated:

EVENT:VBXevent A VBX-specific event occurred. Interrogate the PROP:VBXEvent and PROP:VBXEventArg properties for the event.

EVENT:Selected The control has received input focus.

EVENT:Accepted The user has completed using the control.

EVENT:PreAlertKey
The user pressed an ALRT attribute hot key.

EVENT:AlertKey The user pressed an ALRT attribute hot key.

EVENT:Drop A successful drag-and-drop to the control.

Example:

```
MDIChild WINDOW(´Child One´),AT(0,0,320,200),MDI,MAX,HVSCROLL
         CUSTOM,AT(0,0,120,320),USE(C1), |
               CLASS(´graph.vbx´,´graph´),´graphstyle´(´2´)
       END
  CODE
  OPEN(MDIChild)
  ACCEPT
    CASE ACCEPTED()
    OF ?C1
      !Perform some action
    END
  END
```

ELLIPSE (declare a window ellipse control)

ELLIPSE ,**AT( )** [,**USE( )**] [,**DISABLE**] [,**COLOR( )**] [,**FILL( )**] [,**FULL**] [,**SCROLL**] [,**HIDE**]

ELLIPSE	Places a "circular" figure on the WINDOW or TOOLBAR.
AT	Specifies the initial size and location of the control. If omitted, default values are selected by the runtime library.
USE	A field equate label to reference the control in executable code.
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION is first opened.
COLOR	Specifies the color for the border of the ellipse. If omitted, the ellipse has a black border.
FILL	Specifies the fill color for the control. If omitted, the ellipse is not filled with color.
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter.
SCROLL	Specifies the control scrolls with the window.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened. UNHIDE must be used to display it.

The **ELLIPSE** control places a "circular" figure on the WINDOW (or TOOLBAR) at the position and size specified by its AT attribute. The ellipse is drawn inside a "bounding box" defined by the *x*, *y*, *width*, and *height* parameters of it sAT attribute. The *x* and *y* parameters specify the starting point, and the *width* and *height* parameters specify the horizontal and vertical size of the "bounding box." This control cannot receive input focus and does not generate events.

Example:

```
MDIChild WINDOW(´Child One´),AT(0,0,320,200),MDI,MAX,HVSCROLL
         ELLIPSE,FILL(COLOR:MENU),FULL       !Filled, full screen, black border
         ELLIPSE,AT(0,0,20,20)                    !Unfilled, black border
         ELLIPSE,AT(0,20,20,20),USE(?Box1),DISABLE   !Dimmed
         ELLIPSE,AT(20,20,20,20),ROUND       !Unfilled, rounded, black border
         ELLIPSE,AT(40,40,20,20),FILL(COLOR:ACTIVEBORDER)
                                               !Filled, black border
         ELLIPSE,AT(60,60,20,20),COLOR(COLOR:ACTIVEBORDeR)
                                    !Unfilled, active border color border
         ELLIPSE,AT(480,180,20,20),SCROLL  !Scrolls with screen
        END
```

ENTRY (declare a data entry control)

```
ENTRY(picture) ,AT() [,CURSOR()] [,USE()] [,DISABLE] [,KEY()] [,MSG()] [,HLP()] [,SKIP]
       [,FONT()] [,IMM] [,PASSWORD] [,REQ] [,FULL] [,SCROLL][,ALRT()] [,HIDE]
       [,READONLY] [DROPID( )] [,   | INS     | ] [, | CAP | ] [, | LEFT        | ]
                                    | OVR     |     | UPR |     | RIGHT       |
                                                                | CENTER      |
                                                                | DECIMAL     |
```

ENTRY	Places a data entry field on the WINDOW or TOOLBAR.
picture	A display picture token that specifies the input format for the data entered into the control.
AT	Specifies the initial size and location of the control. If omitted, default values are selected by the runtime library.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control. If omitted, the WINDOW´s CURSOR attribute is used, else the Windows default cursor is used.
USE	The label of the variable that receives the value entered into the control by the user.
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION is first opened.
KEY	Specifies an integer constant or keycode equate that immediately gives focus to the control.
MSG	Specifies a string constant containing the text to display in the status bar when the control has focus.
HLP	Specifies a string constant containing the help system identifier for the control.
SKIP	Specifies the control receives input focus to enter text only with the mouse or accelerator key and does not retain focus.
FONT	Specifies the display font for the control.
IMM	Specifies immediate event generation whenever the user presses any key.
PASSWORD	Specifies non-display of the data entered (password mode).
REQ	Specifies the control may not be left blank or zero.
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter.
SCROLL	Specifies the control scrolls with the window.
ALRT	Specifies "hot" keys active for the control.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened. UNHIDE must be used to display it.
READONLY	Specifies the control does not allow data entry.
DROPID	Specifies the control may serve as a drop target for drag-and-drop actions.
INS / OVR	Specifies Insert or Overwrite entry mode (valid only on windows with the MASK attribute).
UPR / CAP	Specifies all upper case or proper name capitalization (First Letter Of Each Word

Capitalized) entry.

LEFT Specifies that the data entered is left justified within the area specified by the AT attribute.

RIGHT Specifies that the data entered is right justified within the area specified by the AT attribute.

CENTER Specifies that the data entered is centered within the area specified by the AT attribute.

DECIMAL Specifies that the data entered is aligned on the decimal point within the area specified by the AT attribute.

The **ENTRY** control places a data entry field on the WINDOW (or TOOLBAR) at the position and size specified by its AT attribute. Data entered is formatted according to the *picture*, and the variable specified in the USE attribute receives the data entered when the user has completed data entry and moves on to another control. Data entry scrolls horizontally to allow the user to enter data to the full length of the variable. Therefore, the right and left arrow keys move within the data in the ENTRY control.

An ENTRY control with the PASSWORD attribute displays asterisks when the user enters data. This is useful for password-type variables. An ENTRY control with the SKIP attribute is used for seldom-used data entry. Display-only data should be declared with the READONLY attribute.

The MASK attribute on a WINDOW specifies pattern input editing mode of all controls in the window. This means that, as the user types in data, each character is automatically validated against the control´s picture for proper input (numbers only in numeric pictures, etc.). This forces the user to enter data in the format specified by the control´s display picture. If omitted, Windows free-input is allowed in the controls. This is Windows´ default data entry mode. Free-input means the user´s data is formatted to the control´s picture only after entry. This allows users to enter data as they choose and it is automatically formatted to the control´s picture after entry. If the user types in data in a format different from the control´s picture, the libraries attempt to determine the format the user used, and convert the data to the control´s display picture. For example, if the user types "January 1, 1995" into a control with a display picture of @D1, the runtime library formats the user´s input to "1/1/95." This action occurs only after the user completes data entry and moves to another control. If the runtime library cannot determine what format the user used, it will not update the USE variable. It then beeps and leaves the user on the same control with the data they entered, to allow them to try again.

Events Generated:

EVENT:Selected The control has received input focus.

EVENT:Accepted The user has completed data entry in the control.

EVENT:PreAlertKey
The user pressed an ALRT attribute hot key.

EVENT:AlertKey The user pressed an ALRT attribute hot key.

EVENT:Drop A successful drag-and-drop to the control.

An ENTRY with the IMM attribute also generates the following events:

EVENT:NewSelection
The user has pressed a key.

Example:

```
MDIChildWINDOW(´Child One´),AT(0,0,320,200),MDI,MAX,HVSCROLL
```

```
  ENTRY(@S8),AT(0,0,20,20),USE(E1)
  ENTRY(@S8),AT(20,0,20,20),USE(E2),KEY(F10Key)
  ENTRY(@S8),AT(40,0,20,20),USE(E3),MSG(´Button 3´)
  ENTRY(@S8),AT(60,0,20,20),USE(E4),HLP(´Entry4Help´)
  ENTRY(@S8),AT(80,0,20,20),USE(E5),DISABLE
  ENTRY(@S8),AT(100,0,20,20),USE(E6),FONT(´Arial´,12)
  ENTRY(@S8),AT(120,0,20,20),USE(E7),REQ,INS,CAP
  ENTRY(@S8),AT(140,0,20,20),USE(E8),SCROLL,OVR,UPR
  ENTRY(@S8),AT(160,0,20,20),USE(E9),IMM
  ENTRY(@S8),AT(180,0,20,20),USE(E10),CURSOR(CURSOR:Wait)
  ENTRY(@S8),AT(200,0,20,20),USE(E11),ALRT(F10Key)
  ENTRY(@S8),AT(220,0,20,20),USE(E12),LEFT
  ENTRY(@S8),AT(240,0,20,20),USE(E13),RIGHT
  ENTRY(@S8),AT(260,0,20,20),USE(E14),CENTER
  ENTRY(@N8.2),AT(280,0,20,20),USE(E15),DECIMAL
END
```

GROUP (declare a group of window controls)

```
GROUP(text) ,AT() [,CURSOR()] [,USE()] [,DISABLE] [,KEY()] [,MSG()]   [,HLP()] [,FONT()]
                [,BOXED] [,FULL] [,SCROLL] [,HIDE] [,ALRT( )] [,SKIP]
    controls
END
```

GROUP	Declares a group of controls that may be referenced as one entity.
text	A string constant containing the prompt for the group of controls. This may contain an ampersand (&) to indicate the "hot" letter for the prompt. The *text* is displayed on screen only if the BOXED attribute is also present.
AT	Specifies the initial size and location of the control. If omitted, default values are selected by the runtime library.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control (or any control within the GROUP). If omitted, the window´s CURSOR attribute is used, else the Windows default cursor is used.
USE	A field equate label to reference the control in executable code.
DISABLE	Specifies the GROUP control and the controls in the GROUP appear dimmed when the WINDOW or APPLICATION is first opened.
KEY	Specifies an integer constant or keycode equate that immediately gives focus to the first control in the GROUP.
MSG	Specifies a string constant containing the default text to display in the status bar when any control in the GROUP has focus.
HLP	Specifies a string constant containing the default help system identifier for any control in the GROUP.
FONT	Specifies the display font for the control and the default for all the controls in the GROUP.
BOXED	Specifies a single-track border around the group of controls with the text at the top of the border.
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter.
SCROLL	Specifies the GROUP control and the controls in the GROUP scroll with the window.
HIDE	Specifies the GROUP control and the controls in the GROUP do not appear when the WINDOW or APPLICATION is first opened. UNHIDE must be used to display them.
ALRT	Specifies "hot" keys active for the controls in the GROUP.
SKIP	Specifies the controls in the GROUP do not receive input focus and may only be accessed with the mouse or accelerator key.
controls	Control declarations that may be referenced as the GROUP.

The **GROUP** control declares a group of controls that may be referenced as one entity. GROUP allows the user to use the cursor keys instead of the TAB key to move between the *controls* in the GROUP, and provides default MSG and HLP attributes for all controls in the GROUP.

This control cannot receive input focus and does not generate events.

Example:

```
MDIChild WINDOW(´Child One´),AT(0,0,320,200),MDI,MAX,HVSCROLL
         GROUP(´Group 1´),USE(?G1),KEY(F10Key)
           ENTRY(@S8),AT(0,0,20,20),USE(?E1)
           ENTRY(@S8),AT(20,0,20,20),USE(?E2)
         END
         GROUP(´Group 2´),USE(?G2),MSG(´Group 2´)
           ENTRY(@S8),AT(40,0,20,20),USE(?E3)
           ENTRY(@S8),AT(60,0,20,20),USE(?E4)
         END
         GROUP(´Group 3´),USE(?G3),AT(80,0,20,20),BOXED
           ENTRY(@S8),AT(80,0,20,20),USE(?E5)
           ENTRY(@S8),AT(100,0,20,20),USE(?E6)
         END
         GROUP(´Group 4´),USE(?G4),FONT(´Arial´,12)
           ENTRY(@S8),AT(120,0,20,20),USE(?E7)
           ENTRY(@S8),AT(140,0,20,20),USE(?E8)
         END
         GROUP(´Group 5´),USE(?G5),CURSOR(CURSOR:Wait)
           ENTRY(@S8),AT(160,0,20,20),USE(?E9)
           ENTRY(@S8),AT(180,0,20,20),USE(?E10)
         END
         GROUP(´Group 6´),USE(?G6),SCROLL
           ENTRY(@S8),AT(200,0,20,20),USE(?E11)
           ENTRY(@S8),AT(220,0,20,20),USE(?E12)
         END
         GROUP(´Group 7´),USE(?G7),HLP(´Group7Help´)
           ENTRY(@S8),AT(240,0,20,20),USE(?E13)
           ENTRY(@S8),AT(260,0,20,20),USE(?E14)
         END
       END
```

IMAGE (declare a window graphic image control)

IMAGE(*file***) ,AT( ) [,USE( )] [,DISABLE] [,FULL] [,SCROLL] [,HIDE] [, | HSCROLL |]**
| VSCROLL |
| HVSCROLL |

IMAGE	Places a graphic image on the WINDOW or TOOLBAR.
file	A string constant containing the name of the file to display. The file is linked into the .EXE as a resource.
AT	Specifies the initial size and location of the control. If omitted, default values are selected by the runtime library.
USE	A field equate label to reference the control in executable code.
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION is first opened.
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter.
SCROLL	Specifies the control scrolls with the window.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened. UNHIDE must be used to display it.
HSCROLL	Specifies that a horizontal scroll bar is automatically added to the IMAGE control when the graphic image is wider than the area specified for display.
VSCROLL	Specifies that a vertical scroll bar is automatically added to the IMAGE control when the graphic image is taller than the area specified for display.
HVSCROLL	Specifies that both vertical and horizontal scroll bars are automatically added to the IMAGE control when the graphic image is larger than the display area.

The **IMAGE** control places a graphic image on the WINDOW (or TOOLBAR) at the position and size specified by its AT attribute. This may be a bitmap (.BMP), icon (.ICO), PaintBrush (.PCX), Graphic Interchange Format (.GIF), JPEG (.JPG), or Windows metafile (.WMF). This control cannot receive input focus and does not generate events.

Example:

```
MDIChild WINDOW(´Child One´),AT(0,0,320,200),MDI,MAX,HVSCROLL
         IMAGE(´PIC.BMP´),AT(0,0,20,20),USE(?I1)
         IMAGE(´PIC.WMF´),AT(40,0,20,20),USE(?I3),SCROLL
       END
```

See Also:

How to Assign an Image to Display at Runtime

LINE (declare a window line control)

LINE ,**AT( )** [,**USE( )**] [,**DISABLE**] [,**COLOR( )**] [,**FULL**] [,**SCROLL**] [,**HIDE**]

LINE	Places a straight line on the WINDOW or TOOLBAR.
AT	Specifies the initial size and location of the control. If omitted, default values are selected by the runtime library.
USE	A field equate label to reference the control in executable code.
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION is first opened.
COLOR	Specifies the color for the line. If omitted, the color is black.
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter.
SCROLL	Specifies the control scrolls with the window.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened. UNHIDE must be used to display it.

The **LINE** control places a straight line on the WINDOW (or TOOLBAR) at the position and size specified by its AT attribute. The *x* and *y* parameters of the AT attribute specify the starting point of the line. The *width* and *height* parameters of the AT attribute specify the horizontal and vertical distance to the end point of the line. If these are both positive numbers, the line slopes to the right and down from its starting point. If the *width* is negative, the line slopes left; if the *height* is negative, the line slopes left. If either the *width* or *height* is zero, the line is horizontal or vertical. This control cannot receive input focus and does not generate events.

Width	Height	Result
positive	positive	right and down from start point
negative	positive	left and down from start point
positive	negative	right and up from start point
negative	negative	left and up from start point
zero	positive	vertical, down from start point
zero	negative	vertical, up from start point
positive	zero	horizontal, right from start point
negative	zero	horizontal, left from start point

Example:

```
MDIChild WINDOW(´Child One´),AT(0,0,320,200),MDI,MAX,HVSCROLL
         LINE,AT(60,60,20,20),COLOR(COLOR:ACTIVEBORDER)                  !Border color
         LINE,AT(480,180,20,20),SCROLL                        !Scrolls with screen
       END
```

LIST (declare a window list control)

```
LIST ,FROM( ) ,AT( ) [,CURSOR( )] [,USE( )] [,DISABLE] [,KEY( )] [,MSG( )] [,HLP( )] [,SKIP]
     [,FONT( )] [,FORMAT( )] [,DROP] [,COLUMN] [,VCR] [,FULL] [,SCROLL] [,NOBAR]
     [,ALRT( )] [,HIDE] [,DRAGID( )] [,DROPID( )]
     [,  | MARK( )  | ] [,  | HSCROLL    | ] [,  | LEFT       | ]
        | IMM       |      | VSCROLL    |      | RIGHT      |
                           | HVSCROLL   |      | CENTER     |
                                                 | DECIMAL    |
```

LIST	Places a scrolling list of data items on the WINDOW or TOOLBAR.
FROM	Specifies the origin of the data displayed in the list.
AT	Specifies the initial size and location of the control. If omitted, the runtime library chooses a value.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control. If omitted, the WINDOW´s CURSOR attribute is used, else the Windows default cursor is used.
USE	A field equate label to reference the control in executable code, or the label of the variable that receives the value selected by the user.
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION is first opened.
KEY	Specifies an integer constant or keycode equate that immediately gives focus to the control.
MSG	Specifies a string constant containing the text to display in the status bar when the control has focus.
HLP	Specifies a string constant containing the help system identifier for the control.
SKIP	Specifies the control does not receive input focus and may only be accessed with the mouse or accelerator key.
FONT	Specifies the display font for the control.
FORMAT	Specifies the display format of the data.
DROP	Specifies a drop-down list box and the number of elements the drop-down portion contains.
COLUMN	Specifies a field-by-field highlight bar on multi-column list boxes.
VCR	Specifies a VCR-type control to the left of the horizontal scroll bar (if present).
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter.
SCROLL	Specifies the control scrolls with the window.
NOBAR	Specifies the highlight bar is displayed only when the LIST has focus.
ALRT	Specifies "hot" keys active for the control.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened. UNHIDE must be used to display it.
DRAGID	Specifies the control may serve as a drag host for drag-and-drop actions.

DROPID	Specifies the control may serve as a drop target for drag-and-drop actions.
MARK	Specifies multiple items selection mode.
IMM	Specifies generation of an event whenever the user presses any key.
HSCROLL	Specifies that a horizontal scroll bar is automatically added to the list box when any portion of the data item lies horizontally outside the visible area.
VSCROLL	Specifies that a vertical scroll bar is automatically added to the list box when any data items lie vertically outside the visible area.
HVSCROLL	Specifies that both vertical and horizontal scroll bars are automatically added to the list box when any portion of the data items lies outside the visible area.
LEFT	Specifies that the data is left justified within the LIST.
RIGHT	Specifies that the data is right justified within the LIST.
CENTER	Specifies that the data is centered within the LIST.
DECIMAL	Specifies that the data is aligned on the decimal point within the LIST.

The **LIST** control places a scrolling list of data items on the WINDOW (or TOOLBAR) at the position and size specified by its AT attribute. The data items displayed in the LIST come from a QUEUE or STRING specified by the FROM attribute. The CHOICE function returns the QUEUE entry number (the value returned by POINTER(queue)) of the selected item when the EVENT:Accepted event has been generated by the LIST. The data displayed in the LIST is automatically refreshed every time through the ACCEPT loop, whether the AUTO attribute is present or not.

A LIST with the DROP attribute displays only the currently selected data item on screen until the control has focus and the user presses the down arrow key, or CLICKS ON the the icon to the right of the displayed data item. When either of these occurs, the selection list appears ("drops down") to allow the user to select an item.

A LIST with the IMM attribute generates an event every time the user moves the highlight bar to another selection, or pressed any key that causes the displayed entries to scroll. This allows an opportunity for the source code to re-fill the display QUEUE, or get the currently highlighted record to display other fields from the record. If VSCROLL is also present, the vertical scroll bar is always displayed and when the end-user CLICKS on the scroll bar, events are generated but the list does not move (executable code should perfrom this action). You can interrogate the PROP:VscrollPos property to determine the scroll thumb´s position from 0 (top) to 255 (bottom).

A LIST with the VCR attribute has scroll control buttons like a **V**ideo **C**assette **R**ecorder to the left of the horizontal scroll bar (if there is one). These buttons allow the user to use the mouse to scroll through the list.

A LIST with the DRAGID attribute can serve as a drag-and-drop host, providing information to be moved or copied to another control. A LIST with the DROPID attribute can serve as a drag-and-drop target, receiving information from another control. These attributes work together to specify drag-and-drop "signatures" that define a valid target for the operation. The DRAGID() and DROPID() functions, along with the SETDROPID procedure, are used to perform the data exchange.

Events Generated:

EVENT:Selected	The control has received input focus.
EVENT:Accepted	The user has selected an entry from the control.
EVENT:NewSelection	

The current selection in the list has changed (highlight has moved up or down).

EVENT:PreAlertKey
The user pressed an ALRT attribute hot key.

EVENT:AlertKey The user pressed an ALRT attribute hot key.

A LIST with the IMM attribute also generates the following events:

EVENT:ScrollUp The highlight bar has attempted to move off the top of the LIST.

EVENT:ScrollDown
The highlight bar has attempted to move off the bottom of the LIST.

EVENT:PageUp The user pressed PgUp.

EVENT:PageDown
The user pressed PgDn.

EVENT:ScrollTop The user pressed Ctrl-PgUp.

EVENT:ScrollBottom
The user pressed Ctrl-PgDn.

EVENT:PreAlertKey
The user pressed a printable character or an ALRT attribute hot key.

EVENT:AlertKey The user pressed a printable character or an ALRT attribute hot key.

EVENT:Locate The user pressed the locator VCR button.

EVENT:ScrollDrag
The scroll bar "thumb" is being moved.

A LIST with the DRAGID attribute also generates the following events:

EVENT:Dragging The mouse cursor is over a potential drag target.

EVENT:Drag The mouse cursor has been released over a drag target.

A LIST with the DROPID attribute also generates the following events:

EVENT:Drop The mouse cursor has been released over a drag target.

Example:

```
MDIChild WINDOW(´Child One´),AT(0,0,320,200),MDI,MAX,HVSCROLL
         LIST,AT(0,0,20,20),USE(?L1),FROM(Que),IMM
         LIST,AT(20,0,20,20),USE(?L2),FROM(Que),KEY(F10Key)
         LIST,AT(40,0,20,20),USE(?L3),FROM(Que),MSG(´Button 3´)
         LIST,AT(60,0,20,20),USE(?L4),FROM(Que),HLP(´Check4Help´)
         LIST,AT(80,0,20,20),USE(?L5),FROM(Q),FORMAT(´5C~List~15L~Box~´),COLUMN
         LIST,AT(100,0,20,20),USE(?L6),FROM(Que),FONT(´Arial´,12)
         LIST,AT(120,0,20,20),USE(?L7),FROM(Que),DROP(6)
         LIST,AT(140,0,20,20),USE(?L8),FROM(Que),HVSCROLL,VCR
```

```
  LIST,AT(180,0,20,20),USE(?L10),FROM(Que),CURSOR(CURSOR:Wait)
  LIST,AT(200,0,20,20),USE(?L11),FROM(Que),ALRT(F10Key)
  LIST,AT(220,0,20,20),USE(?L12),FROM(Que),LEFT
  LIST,AT(240,0,20,20),USE(?L13),FROM(Que),RIGHT
  LIST,AT(260,0,20,20),USE(?L14),FROM(Que),CENTER
  LIST,AT(280,0,20,20),USE(?L15),FROM(Que),DECIMAL
END
```

See Also:

COMBO

DRAGID

DROPID

SETDROPID

OPTION (declare a group of window RADIO controls)

```
OPTION(text) ,AT( ) [,CURSOR( )] [,USE( )] [,DISABLE] [,KEY( )] [,MSG( )] [,HLP( )] [,BOXED]
             [,FULL] [,SCROLL] [,HIDE] [,FONT( )] [,ALRT( )] [,SKIP] [DROPID( )]
  radios
END
```

OPTION	Declares a group of RADIO controls.
text	A string constant containing the prompt for the group of controls. This may contain an ampersand (&) to indicate the "hot" letter for the prompt. The *text* is displayed on screen only if the BOXED attribute is also present.
AT	Specifies the initial size and location of the control. If omitted, default values are selected by the runtime library.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control. If omitted, the WINDOW´s CURSOR attribute is used, else the Windows default cursor is used.
USE	The label of a string variable to receive the value of the RADIO string (with any accelerator key ampersand stripped out) selected by the user.
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION is first opened.
KEY	Specifies an integer constant or keycode equate that immediately gives focus to the currently selected RADIO in the OPTION control.
MSG	Specifies a string constant containing the default text to display in the status bar when any control in the OPTION has focus.
HLP	Specifies a string constant containing the default help system identifier for any control in the OPTION.
BOXED	Specifies a single-track border around the RADIO controls with the text at the top of the border.
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter.
SCROLL	Specifies the control scrolls with the window.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened. UNHIDE must be used to display it.
FONT	Specifies the display font for the control and the default for all the controls in the OPTION.
ALRT	Specifies "hot" keys active for the controls in the OPTION.
SKIP	Specifies the controls in the OPTION do not receive input focus and may only be accessed with the mouse or accelerator key.
DROPID	Specifies the control may serve as a drop target for drag-and-drop actions.
radios	Multiple RADIO control declarations.

The **OPTION** control declares a group of RADIO controls that offer the user a list of choices. The multiple RADIO controls in the OPTION structure define the choices offered to the user.

Input focus changes between the OPTION´s RADIO controls are signalled only to the individual RADIO controls affected. This means the events generated when the user changes input focus within an OPTION structure are field-specific events for the affected RADIO controls, not the OPTION structure which contains them.

The variable named in the OPTION structure´s USE attribute receives the text of the RADIO control selected by the user. The CHOICE(*?Option*) function returns the number of the selected RADIO. No RADIO button selected is a valid option, which occurs only when the OPTION structure´s USE variable does not contain a value duplicated by one of its component RADIO controls. This condition only lasts until the user has selected one of the RADIOs.

Events Generated:

EVENT:Selected One of the OPTION´s RADIO controls has received input focus.

EVENT:Accepted One of the OPTION´s RADIO controls has been selected by the user.

EVENT:PreAlertKey
The user pressed an ALRT attribute hot key.

EVENT:AlertKey The user pressed an ALRT attribute hot key.

EVENT:Drop A successful drag-and-drop to the control.

Example:

```
MDIChild WINDOW(´Child One´),AT(0,0,320,200),MDI,MAX,HVSCROLL
         OPTION(´Option 1´),USE(OptVar1),KEY(F10Key),HLP(´Option1Help´)
           RADIO(´Radio 1´),AT(0,0,20,20),USE(?R1)
           RADIO(´Radio 2´),AT(20,0,20,20),USE(?R2)
          END
          OPTION(´Option 2´),USE(OptVar2),MSG(´Option 2´),SCROLL
            RADIO(´Radio 3´),AT(40,0,20,20),USE(?R3)
            RADIO(´Radio 4´),AT(60,0,20,20),USE(?R4)
          END
          OPTION(´Option 3´),USE(OptVar3),AT(80,0,20,20),BOXED
            RADIO(´Radio 5´),AT(80,0,20,20),USE(?R5)
            RADIO(´Radio 6´),AT(100,0,20,20),USE(?R6)
          END
         OPTION(´Option 4´),USE(OptVar4),FONT(´Arial´,12),CURSOR(CURSOR:Wait)
           RADIO(´Radio 7´),AT(120,0,20,20),USE(?R7)
           RADIO(´Radio 8´),AT(140,0,20,20),USE(?R8)
         END
       END
```

See Also:

RADIO

PROGRESS (declare a progress control)

PROGRESS, **AT( )** [,**CURSOR( )**] [,**USE( )**] [,**DISABLE**] [,**FULL**] [,**SCROLL**]
[,**HIDE**] [,**DROPID( )**] [,**RANGE( )**]

PROGRESS	Places a control that displays the current progress of a batch process in the WINDOW or TOOLBAR.
AT	Specifies the initial size and location of the control. If omitted, default values are selected by the runtime library.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control. If omitted, the WINDOWs CURSOR attribute is used, else the Windows default cursor is used.
USE	The label of the variable containing the value of the current progress, or a field equate label to reference the control in executable code.
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION is first opened.
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter.
SCROLL	Specifies the control scrolls with the window.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened. UNHIDE must be used to display it.
DROPID	Specifies the control may serve as a drop target for drag-and-drop actions.
RANGE	Specifies the range of values the progress bar displays. If omitted, the default range is zero (0) to one hundred (100).

The **PROGRESS** control declares a control that displays a progress bar. This usually displays the current percentage of completion of a batch process.

If a variable is named as the USE attribute, the progress bar is automatically updated whenever the value in that variable changes. If the USE attribute is a field equate label, you must directly update the display by assigning a value (within the range defined by the RANGE attribute) to the controls PROP:progress property (an undeclared property equate -- see *Undeclared Properties*).

This control cannot receive input focus and does not generate events.

Example:

```
BackgroundProcess   PROCEDURE        !Background processing batch process

ProgressVariable  LONG

Win  WINDOW(Batch Processing...),AT(,,400,400),TIMER(1),MDI,CENTER
       PROGRESS,AT(100,100,200,20),USE(ProgressVariable),RANGE(0,200)
       PROGRESS,AT(100,140,200,20),USE(?ProgressBar),RANGE(0,200)
       BUTTON(Cancel),AT(190,300,20,20),STD(STD:Close)
     END

  CODE
  OPEN(Win)
  OPEN(File)
  ?ProgressVariable{PROP:rangehigh} = RECORDS(File)
```

```
?ProgressBar{PROP:rangehigh} = RECORDS(File)
SET(File)                        !Set up a batch process
ACCEPT
  CASE EVENT()
  OF EVENT:CloseWindow
    BREAK
  OF EVENT:Timer                      !Process records when timer allows it
    ProgressVariable += 3               !Auto-updates 1st progress bar
    LOOP 3 TIMES
      NEXT(File)
      IF ERRORCODE() THEN BREAK.
      ?ProgressBar{PROP:progress} = ?ProgressBar{PROP:progress} + 1
                                    !Manually update 2nd progress bar
      !Perform some batch processing code
. . .
CLOSE(File)
```

PROMPT (declare a prompt control)

```
PROMPT(text) ,AT( ) [,CURSOR( )] [,USE( )] [,DISABLE] [,FONT( )] [,FULL] [,SCROLL]
        [,HIDE] [,   | LEFT     | ]
                     | RIGHT    |
                     | CENTER   |
```

PROMPT	Places a prompt for the next active control following it, in the WINDOW or TOOLBAR.
text	A string constant containing the text to display. This may contain an ampersand (&) to indicate the "hot" letter for the prompt.
AT	Specifies the initial size and location of the control. If omitted, default values are selected by the runtime library.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control. If omitted, the WINDOW´s CURSOR attribute is used, else the Windows default cursor is used.
USE	A field equate label to reference the control in executable code.
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION is first opened.
FONT	Specifies the font used to display the text.
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter.
SCROLL	Specifies the control scrolls with the window.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened. UNHIDE must be used to display it.
LEFT	Specifies that the prompt is left justified.
RIGHT	Specifies that the prompt is right justified.
CENTER	Specifies that the prompt is centered.

The **PROMPT** control places a prompt for the next active control following the PROMPT in the WINDOW or TOOLBAR structure. The prompt *text* is placed on the WINDOW (or TOOLBAR) at the position and size specified by its AT attribute.

The *text* may contain an ampersand (&) to indicate the letter immediately following the ampersand is the "hot" letter for the prompt. By default, the "hot" letter displays with an underscore below it to indicate its special purpose. This "hot" letter, when pressed in conjunction with the ALT key, changes input focus to the next control following the PROMPT in the WINDOW or TOOLBAR structure, which is capable of receiving focus.

Disabling or hiding the control directly following the PROMPT in the window structure does not autmatically disable or hide the PROMPT; it must also be explicitly disabled or hidden, otherwise the PROMPT will then refer to the next currently active control following the disabled control. This allows you to place one PROMPT control on the window that will apply to any of multiple controls (if only one will be active at a time). If the next active control is a BUTTON, it is pressed when the user presses the PROMPT´s "hot key."

To include an ampersand as part of the prompt *text*, place two ampersands together (&&) in the *text* string and only one will display.

This control cannot receive input focus and does not generate events.

Example:

```
MDIChild WINDOW(´Child One´),AT(0,0,320,200),MDI,MAX,HVSCROLL
          PROMPT(´Enter Data:´),AT(10,100,20,20),USE(?P1),CURSOR(CURSOR:Wait)
          ENTRY(@S8),AT(100,100,20,20),USE(E1)
          PROMPT(´Enter More Data:´),AT(10,200,20,20),USE(?P2),CURSOR(CURSOR:Wait)
          ENTRY(@S8),AT(100,200,20,20),USE(E2)
          ENTRY(@D1),AT(100,200,20,20),USE(E3)
        END
  CODE
  OPEN(MDIChild)
   IF SomeCondition
    HIDE(?E2)        !Prompt will refer to E3
  ELSE
    HIDE(?E3)        !Prompt will refer to E2
  END
```

RADIO (declare a window radio button control)

RADIO(*text***) ,AT( ) [,CURSOR( )] [,USE( )] [,DISABLE] [,KEY( )] [,MSG( )] [,HLP( )] [,SKIP]**
[,FONT( )] [,ICON( )] [,FULL] [,SCROLL] [,HIDE] [,ALRT( )] [DROPID( )][, | LEFT |]
| RIGHT |

RADIO	Places a radio button on the WINDOW or TOOLBAR.
text	A string constant containing the text to display for the radio button. This may contain an ampersand (&) to indicate the "hot" letter for the radio button.
AT	Specifies the initial size and location of the control. If omitted, default values are selected by the runtime library.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control. If omitted, the WINDOW´s CURSOR attribute is used, else the Windows default cursor is used.
USE	A field equate label to reference the control in executable code.
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION is first opened.
KEY	Specifies an integer constant or keycode equate that immediately selects the radio button.
MSG	Specifies a string constant containing the text to display in the status bar when the control has focus.
HLP	Specifies a string constant containing the help system identifier for the control.
SKIP	Specifies the control does not receive input focus and may only be accessed with the mouse or accelerator key.
FONT	Specifies the display font for the control.
ICON	Specifies an .ICO file or standard icon to display on the face of a "latching" button.
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter.
SCROLL	Specifies the control scrolls with the window.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened. UNHIDE must be used to display it.
LEFT	Specifies the text appears to the left of the radio button.
RIGHT	Specifies the text appears to the right of the radio button (this is the default position).
ALRT	Specifies "hot" keys active for the control.
DROPID	Specifies the control may serve as a drop target for drag-and-drop actions.

The **RADIO** control places a radio button on the WINDOW (or TOOLBAR) at the position and size specified by its AT attribute. A RADIO control may only be placed within an OPTION control. When selected by the user, the RADIO *text* (with any accelerator key ampersand stripped out) is placed in the OPTION´s USE variable. A RADIO with an ICON attribute appears as a "latched" pushbutton with the icon on the button face. When the icon appears "up" the RADIO is off; when it appears "down" the RADIO is on and the OPTION´s USE variable receives the value in the selected RADIO´s *text* parameter.

Events Generated:

EVENT:Selected The control has received input focus.

EVENT:Accepted The control has been selected by the user.

EVENT:PreAlertKey
The user pressed an ALRT attribute hot key.

EVENT:AlertKey The user pressed an ALRT attribute hot key.

EVENT:Drop A successful drag-and-drop to the control.

Example:

```
MDIChild WINDOW(´Child One´),AT(0,0,320,200),MDI,MAX,HVSCROLL
         OPTION(´Option 1´),USE(OptVar1)
           RADIO(´Radio 1´),AT(0,0,20,20),USE(?R1),KEY(F10Key)
           RADIO(´Radio 2´),AT(20,0,20,20),USE(?R2),MSG(´Radio 2´)
         END
         OPTION(´Option 2´),USE(OptVar2)
           RADIO(´Radio 3´),AT(40,0,20,20),USE(?R3),FONT(´Arial´,12)
           RADIO(´Radio 4´),AT(60,0,20,20),USE(?R4),CURSOR(CURSOR:Wait)
         END
         OPTION(´Option 3´),USE(OptVar3)
           RADIO(´Radio 5´),AT(80,0,20,20),USE(?R5),HLP(´Radio5Help´)
           RADIO(´Radio 6´),AT(100,0,20,20),USE(?R6)
         END
         OPTION(´Option 4´),USE(OptVar4)
           RADIO(´Radio 7´),AT(120,0,20,20),USE(?R7),ICON(´Radio1.ICO´)
           RADIO(´Radio 8´),AT(140,0,20,20),USE(?R8),ICON(´Radio2.ICO´)
         END
         OPTION(´Option 5´),USE(OptVar5)
           RADIO(´Radio 9´),AT(100,20,20,20),USE(?R9),LEFT
           RADIO(´Radio 10´),AT(120,20,20,20),USE(?R10),LEFT
         END
         OPTION(´Option 6´),USE(OptVar6),SCROLL
           RADIO(´Radio 11´),AT(200,0,20,20),USE(?R11),SCROLL
           RADIO(´Radio 12´),AT(220,0,20,20),USE(?R12),SCROLL
         END
       END
```

See Also:

OPTION

REGION (declare a window region control)

REGION ,**AT( )** [,**CURSOR( )**] [,**USE( )**] [,**DISABLE**] [,**FILL**] [,**COLOR( )**] [,**IMM**] [,**FULL**] [,**SCROLL**] [,**HIDE**] [,**DRAGID( )**] [,**DROPID( )**]

REGION	Defines an area in the WINDOW or TOOLBAR.
AT	Specifies the initial size and location of the control. If omitted, default values are selected by the runtime library.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control. If omitted, the WINDOW´s CURSOR attribute is used, else the Windows default cursor is used.
USE	A field equate label to reference the control in executable code.
DISABLE	Specifies the control is disabled when the WINDOW or APPLICATION is first opened.
FILL	Specifies the red, green, and blue component values that create the fill color for the control. If omitted, the region is not filled with color.
COLOR	Specifies the border color of the control. If omitted, there is no border.
IMM	Specifies control generates an event whenever the mouse is moved in the region.
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter.
SCROLL	Specifies the control scrolls with the window.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened. UNHIDE must be used to display it.
DRAGID	Specifies the control may serve as a drag host for drag-and-drop actions.
DROPID	Specifies the control may serve as a drop target for drag-and-drop actions.

The **REGION** control defines an area on screen at the position and size specified by its AT attribute. Generally, tracking the position of the mouse is the reason for defining a REGION. The MOUSEX and MOUSEY functions can be used to determine the exact position of the mouse when the event occurs. Use of the IMM attribute causes some excess code and speed overhead at runtime, so it should be used only when necessary. This control cannot receive input focus.

A REGION with the DRAGID attribute can serve as a drag-and-drop host, providing information to be moved or copied to another control. A REGION with the DROPID attribute can serve as a drag-and-drop target, receiving information from another control. These attributes work together to specify drag-and-drop "signatures" that define a valid target for the operation. The DRAGID() and DROPID() functions, along with the SETDROPID procedure, are used to perform the data exchange. Since a REGION can be defined over any other control, you can write drag-and-drop code between any two controls. Simply define REGION controls to handle the required drag-and drop functionality.

Events Generated:

EVENT:Accepted The mouse has been clicked by the user in the region.

A REGION with the IMM attribute also generates the following events:

EVENT:MouseIn The mouse has entered the region.

EVENT:MouseOut The mouse has left the region.

EVENT:MouseMove
The mouse has moved within the region.

A REGION with the DRAGID attribute also generates the following events:

EVENT:Dragging The mouse cursor is over a potential drag target.

EVENT:Drag The mouse cursor has been released over a drag target.

A REGION with the DROPID attribute also generates the following events:

EVENT:Drop The mouse cursor has been released over a drag target.

Example:

```
MDIChild WINDOW(´Child One´),AT(0,0,320,200),MDI,MAX,HVSCROLL
         REGION,AT(10,100,20,20),USE(?R1)
         REGION,AT(100,100,20,20),USE(?R2),CURSOR(CURSOR:Wait)
         REGION,AT(10,200,20,20),USE(?R3),IMM
         REGION,AT(100,200,20,20),USE(?R4),COLOR(COLOR:ACTIVEBORDER)
         REGION,AT(10,300,20,20),USE(?R4),FILL(COLOR:ACTIVEBORDER)
        END
```

SHEET (declare a group of TAB controls)

```
SHEET ,AT( ) [,CURSOR( )] [,USE( )] [,DISABLE] [,KEY( )] [,FULL] [,SCROLL] [,HIDE]
                [,FONT( )] [,DROPID( )] [,WIZARD] [,SPREAD]
  tabs
END
```

SHEET	Declares a group of TAB controls.
AT	Specifies the initial size and location of the control. If omitted, default values are selected by the runtime library.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control. If omitted, the WINDOW's CURSOR attribute is used, else the Windows default cursor is used.
USE	The label of a variable to receive the choice. If this is a string variable, it receives the value of the TAB string (with any accelerator key ampersand stripped out) currently selected by the user. If a numeric variable, it receives the number of the TAB currently selected by the user (the value returned by the CHOICE() function).
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION is first opened.
KEY	Specifies an integer constant or keycode equate that immediately gives focus to the currently selected TAB in the SHEET control.
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter.
SCROLL	Specifies the control scrolls with the window.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened. UNHIDE must be used to display it.
FONT	Specifies the display font for the control and the default for all the controls in the SHEET.
ALRT	Specifies "hot" keys active for controls in the SHEET.
DROPID	Specifies the control may serve as a drop target for drag-and-drop actions.
WIZARD	Specifies the SHEET's TAB controls do not appear. The user is moved from TAB to TAB under the program's control (usually with "Next" and "Previous" buttons).
SPREAD	Specifies the TABs are evenly spaced on one line.
tabs	Multiple TAB control declarations.

The **SHEET** control declares a group of TAB controls that offer the user multiple "pages" of controls for the window. The multiple TAB controls in the SHEET structure define the "pages" displayed to the user.

Input focus changes between the SHEET's TAB controls are signalled only to the individual TAB controls affected. This means the events generated when the user changes input focus within a SHEET structure are field-specific events for the affected TAB controls, not the SHEET structure which contains them.

A string variable as the SHEET structure's USE attribute receives the text of the TAB control selected by the user, and the CHOICE(*?Option*) function returns the number of the selected TAB control. If the SHEET structure's USE attribute is a numeric variable, it receives the number of the TAB control selected by the user (the same value returned by the CHOICE function).

Events Generated:

EVENT:Selected One of the SHEET's TAB controls has received input focus.

EVENT:Accepted One of the SHEET's TAB controls has been selected by the user.

EVENT:PreAlertKey
The user pressed an ALRT attribute hot key.

EVENT:AlertKey The user pressed an ALRT attribute hot key.

EVENT:Drop A successful drag-and-drop to the control.

Example:

```
MDIChild WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
          SHEET,AT(0,0,320,175),USE(SelectedTab)
            TAB('Tab One'),USE(?TabOne)
              OPTION('Option 1'),USE(OptVar1),KEY(F10Key),HLP('Option1Help')
                RADIO('Radio 1'),AT(20,0,20,20),USE(?R1)
                RADIO('Radio 2'),AT(40,0,20,20),USE(?R2)
              END
              OPTION('Option 2'),USE(OptVar2),MSG('Option 2')
                RADIO('Radio 3'),AT(60,0,20,20),USE(?R3)
                RADIO('Radio 4'),AT(80,0,20,20),USE(?R4)
              END
              PROMPT('Enter Data:'),AT(100,100,20,20),USE(?P1)
              ENTRY(@S8),AT(100,140,32,20),USE(E1)
              PROMPT('Enter More Data:'),AT(100,200,20,20),USE(?P2)
              ENTRY(@S8),AT(100,240,32,20),USE(E2)
            END
            TAB('Tab Two'),USE(?TabTwo)
              OPTION('Option 3'),USE(OptVar3)
                RADIO('Radio 1'),AT(20,0,20,20),USE(?R5)
                RADIO('Radio 2'),AT(40,0,20,20),USE(?R6)
              END
              OPTION('Option 4'),USE(OptVar4)
                RADIO('Radio 3'),AT(60,0,20,20),USE(?R7)
                RADIO('Radio 4'),AT(80,0,20,20),USE(?R8)
              END
              PROMPT('Enter Data:'),AT(100,100,20,20),USE(?P3)
              ENTRY(@S8),AT(100,140,32,20),USE(E3)
              PROMPT('Enter More Data:'),AT(100,200,20,20),USE(?P4)
              ENTRY(@S8),AT(100,240,32,20),USE(E4)
            END
          END
          BUTTON('Ok'),AT(100,180,20,20),USE(?Ok)
          BUTTON('Cancel'),AT(200,180,20,20),USE(?Cancel)
        END
```

See Also: TAB

SPIN (declare a spinning list control)

```
SPIN(picture) ,AT( ) [,CURSOR( )] [,USE( )] [,DISABLE] [,KEY( )] [,MSG( )]   [,HLP( )] [,SKIP]
        [,FONT( )] [,FULL] [,SCROLL] [,ALRT( )] [,HIDE][,READONLY] [,REQ] [,IMM]
        [DROPID( )] [,  | LEFT    | ] [, | INS   | ] , | RANGE()[,STEP]  | [,   | UPR    |]
                        | RIGHT   |     | OVR |     | FROM( )           |     | CAP    |
                        | CENTER  |
                        | DECIMAL |
```

SPIN	Places a "spinning" list of data items on the WINDOW or TOOLBAR.
picture	A display picture token that specifies the format for the data displayed in the control.
AT	Specifies the initial size and location of the control. If omitted, default values are selected by the runtime library.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control. If omitted, the WINDOW´s CURSOR attribute is used, else the Windows default cursor is used.
USE	A field equate label to reference the control in executable code or the label of the variable that receives the value selected by the user.
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION is first opened.
KEY	Specifies an integer constant or keycode equate that immediately gives focus to the control.
MSG	Specifies a string constant containing the text to display in the status bar when the control has focus.
HLP	Specifies a string constant containing the help system identifier for the control.
SKIP	Specifies the control receives input focus to enter text only with the mouse or accelerator key and does not retain focus.
FONT	Specifies the display font for the control.
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter.
SCROLL	Specifies the control scrolls with the window.
ALRT	Specifies "hot" keys active for the control.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened. UNHIDE must be used to display it.
READONLY	Specifies the control does not allow data entry.
REQ	Specifies the control may not be left blank or zero.
IMM	Specifies immediate event generation whenever the user presses any key.
DROPID	Specifies the control may serve as a drop target for drag-and-drop actions.
LEFT	Specifies that the data is left justified within the area specified by the AT attribute.
RIGHT	Specifies that the data is right justified within the area specified by the AT attribute.
CENTER	Specifies that the data is centered within the area specified by the AT attribute.

DECIMAL Specifies that the data is aligned on the decimal point within the area specified by the AT attribute.

INS / OVR Specifies Insert or Overwrite entry mode (valid only on windows with the MASK attribute).

RANGE Specifies the range of values the user may choose.

STEP Specifies the increment/decrement amount of the choices within the specified RANGE. If omitted, the STEP is 1.0.

FROM Specifies the origin of the choices displayed for the user.

UPR / CAP Specifies all upper case or proper name capitalization (First Letter Of Each Word Capitalized) entry.

The **SPIN** control places a "spinning" list of data items on the WINDOW (or TOOLBAR) at the position and size specified by its AT attribute. The "spinning" list displays only the current selection with a pair of buttons to the right to allow the user to "spin" through the available selections (similar to a slot machine wheel).

If the SPIN control offers the user regularly spaced numeric choices, the RANGE attribute specifies the valid range of values from which the user may choose. The STEP attribute then works in conjunction with RANGE to increment/decrement those values by the specified amount. If the choices are not regular, or are string values, the FROM attribute is used instead of RANGE and STEP. The FROM attribute provides the SPIN control its list of choices from a memory QUEUE or a string. Using the FROM attribute, you may provide the user any type of choices in the SPIN control.

The user may select an item from the list or type in the desired value, so this control also acts as an ENTRY control.

Events Generated:

EVENT:Selected The control has received input focus.

EVENT:Accepted The user has selected a value from the control.

EVENT:NewSelection
The user has changed the displayed value.

EVENT:PreAlertKey
The user pressed an ALRT attribute hot key.

EVENT:AlertKey The user pressed an ALRT attribute hot key.

EVENT:Drop A successful drag-and-drop to the control.

Example:

```
MDIChild WINDOW(´Child One´),AT(0,0,320,200),MDI,MAX,HVSCROLL
         SPIN(@S8),AT(0,0,20,20),USE(SpinVar1),FROM(Que)
         SPIN(@N3),AT(20,0,20,20),USE(SpinVar2),RANGE(1,999),KEY(F10Key)
         SPIN(@N3),AT(40,0,20,20),USE(SpinVar3),RANGE(5,995),STEP(5)
         SPIN(@S8),AT(60,0,20,20),USE(SpinVar4),FROM(Que),HLP(´Check4Help´)
         SPIN(@S8),AT(80,0,20,20),USE(SpinVar5),FROM(Que),MSG(´Button 3´)
         SPIN(@S8),AT(100,0,20,20),USE(SpinVar6),FROM(Que),FONT(´Arial´,12)
         SPIN(@S8),AT(120,0,20,20),USE(SpinVar7),FROM(Que),DROP
         SPIN(@S8),AT(140,0,20,20),USE(SpinVar8),FROM(Que),HVSCROLL,VCR
         SPIN(@S8),AT(160,0,20,20),USE(SpinVar9),FROM(Que),IMM
         SPIN(@S8),AT(180,0,20,20),USE(SpinVar10),FROM(Que),CURSOR(CURSOR:Wait)
         SPIN(@S8),AT(200,0,20,20),USE(SpinVar11),FROM(Que),ALRT(F10Key)
         SPIN(@S8),AT(220,0,20,20),USE(SpinVar12),FROM(Que),LEFT
```

```
  SPIN(@S8),AT(240,0,20,20),USE(SpinVar13),FROM(Que),RIGHT
  SPIN(@S8),AT(260,0,20,20),USE(SpinVar14),FROM(Que),CENTER
  SPIN(@N8.2),AT(280,0,20,20),USE(SpinVar15),FROM(Que),DECIMAL
END
```

STRING (declare a window string control)

```
STRING(text) ,AT( ) [,CURSOR( )] [,USE( )] [,DISABLE] [,FONT( )] [,FULL] [,SCROLL] [,HIDE]
        [,TRN] [, | LEFT    | ]
                  | RIGHT   |
                  | CENTER  |
                  | DECIMAL |
```

STRING	Places the *text* on the WINDOW or TOOLBAR.
text	A string constant containing the text to display, or a display picture token to format the variable specified in the USE attribute.
AT	Specifies the initial size and location of the control. If omitted, default values are selected by the runtime library.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control. If omitted, the WINDOW´s CURSOR attribute is used, else the Windows default cursor is used.
USE	A field equate label to reference the control in executable code, or a variable whose contents are displayed in the format of the picture token declared instead of string text.
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION is first opened.
FONT	Specifies the font used to display the text.
LEFT	Specifies that the text is left justified within the area specified by the AT attribute.
RIGHT	Specifies that the text is right justified within the area specified by the AT attribute.
CENTER	Specifies that the text is centered within the area specified by the AT attribute.
DECIMAL	Specifies that the text is aligned on the decimal point within the area specified by the AT attribute.
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter.
SCROLL	Specifies the control scrolls with the window.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened. UNHIDE must be used to display it.
TRN	Specifies the text or USE variable characters transparently display over the background.

The **STRING** control places the *text* on the WINDOW (or TOOLBAR) at the position and size specified by its AT attribute.

If the *text* parameter is a picture token instead of a string constant, the contents of the variable named in the USE attribute are formatted to that display picture, at the position and size specified by the AT attribute. This makes the STRING with a USE variable a "display-only" control for the variable. The data displayed in the STRING is automatically refreshed every time through the ACCEPT loop, whether the AUTO attribute is present or not.

There is a difference between ampersand (&) use in STRING and PROMPT controls. An ampersand in a STRING displays as part of the *text*, while an ampersand in a PROMPT defines the prompt´s "hot" letter.

A STRING with the TRN attribute displays characters transparently, without obliterating the background. This means only the pixels required to create each character are written to screen. This allows the

STRING to be placed directly on top of an IMAGE without destroying the background picture.

This control cannot receive input focus and does not generate events.

Example:

```
MDIChild WINDOW(´Child One´),AT(0,0,320,200),MDI,MAX,HVSCROLL
         STRING(´String Constant´),AT(10,0,20,20),USE(?S1)
         STRING(@S30),AT(10,20,20,20),USE(StringVar1)
         STRING(@S30),AT(10,20,20,20),USE(StringVar2),CURSOR(CURSOR:Wait)
         STRING(@S30),AT(10,20,20,20),USE(StringVar3),FONT(´Arial´,12)
        END
```

TAB (declare a page of a SHEET control)

```
TAB( text ) [,CURSOR( )] [,USE( )] [,KEY( )] [,MSG( )] [,HLP( )]
            [,REQ] [,SKIP] [DROPID( )] [,TIP( )]
  controls
END
```

TAB	Declares a group of controls that constitute one of the multiple pages of controls contained within a SHEET structure.
text	A string constant containing the text to display on the TAB.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control. If omitted, the WINDOWs CURSOR attribute is used, else the Windows default cursor is used.
USE	Specifies a field equate label to reference the control in executable code.
KEY	Specifies an integer constant or keycode equate that immediately gives focus to the control.
MSG	Specifies a string constant containing the default text to display in the status bar when any control in the TAB has focus.
HLP	Specifies a string constant containing the default help system identifier for any control in the TAB.
REQ	Specifies that when another TAB is selected, the runtime library automatically checks all ENTRY controls in the same TAB structure with the REQ attribute to ensure they contain data other than blanks or zeroes.
SKIP	Specifies the controls in the TAB do not receive input focus through the TAB key sequence and may only be accessed with the mouse or accelerator key.
DROPID	Specifies the control may serve as a drop target for drag-and-drop actions.
TIP	Specifies the text that displays as balloon help when the mouse cursor pauses over the control.
controls	Multiple control declarations. This should not contain any SHEET controls (nested SHEET structures are not supported).

The **TAB** structure declares a group of controls that constitute one of the multiple pages of controls contained within a SHEET structure. The multiple TAB controls in the SHEET structure define the pages displayed to the user. The SHEET structures USE attribute receives the *text* of the TAB control selected by the user.

Input focus changes between the SHEETs TAB controls are signalled only to the individual TAB controls affected. This means the events generated when the user changes input focus within a SHEET structure are field-specific events for the affected TAB controls, not the SHEET structure which contains them.

Events Generated:

EVENT:Selected	The TAB control has received input focus.
EVENT:Accepted	The TAB control has been selected by the user.
EVENT:Drop	A successful drag-and-drop to the control.

Example:

```
MDIChild WINDOW(Child One),AT(0,0,320,200),MDI,MAX,HVSCROLL
          SHEET,AT(0,0,320,175),USE(SelectedTab)
            TAB(Tab One),USE(?TabOne)
              OPTION(Option 1),USE(OptVar1),KEY(F10Key),HLP(Option1Help)
                RADIO(Radio 1),AT(20,0,20,20),USE(?R1)
                RADIO(Radio 2),AT(40,0,20,20),USE(?R2)
              END
              OPTION(Option 2),USE(OptVar2),MSG(Option 2)
                RADIO(Radio 3),AT(60,0,20,20),USE(?R3)
                RADIO(Radio 4),AT(80,0,20,20),USE(?R4)
              END
              PROMPT(Enter Data:),AT(100,100,20,20),USE(?P1)
              ENTRY(@S8),AT(100,140,32,20),USE(E1)
              PROMPT(Enter More Data:),AT(100,200,20,20),USE(?P2)
              ENTRY(@S8),AT(100,240,32,20),USE(E2)
            END
            TAB(Tab Two),USE(?TabTwo)
              OPTION(Option 3),USE(OptVar3)
                RADIO(Radio 1),AT(20,0,20,20),USE(?R5)
                RADIO(Radio 2),AT(40,0,20,20),USE(?R6)
              END
              OPTION(Option 4),USE(OptVar4)
                RADIO(Radio 3),AT(60,0,20,20),USE(?R7)
                RADIO(Radio 4),AT(80,0,20,20),USE(?R8)
              END
              PROMPT(Enter Data:),AT(100,100,20,20),USE(?P3)
              ENTRY(@S8),AT(100,140,32,20),USE(E3)
              PROMPT(Enter More Data:),AT(100,200,20,20),USE(?P4)
              ENTRY(@S8),AT(100,240,32,20),USE(E4)
            END
          END
          BUTTON(Ok),AT(100,180,20,20),USE(?Ok)
          BUTTON(Cancel),AT(200,180,20,20),USE(?Cancel)
        END
```

See Also: SHEET

TEXT (declare a multi-line data entry control)

```
TEXT ,AT( ) [,CURSOR( )] [,USE( )] [,DISABLE] [,KEY( )] [,MSG( )] [,HLP( )] [,SKIP] [,FONT( )]
     [,REQ] [,FULL] [,SCROLL] [,ALRT( )] [,HIDE] [,READONLY] [DROPID( )] [UPR]
     [,  | INS  | ] [, | HSCROLL   | ] [,   | LEFT       | ]
        | OVR |    | VSCROLL   |      | RIGHT      |
                   | HVSCROLL  |      | CENTER     |
```

TEXT	Places a multi-line data entry field on the WINDOW or TOOLBAR.
AT	Specifies the initial size and location of the control. If omitted, default values are selected by the runtime library.
CURSOR	Specifies a mouse cursor to display when the mouse is positioned over the control. If omitted, the WINDOW´s CURSOR attribute is used, else the Windows default cursor is used.
USE	The label of the variable that receives the value entered into the control by the user.
DISABLE	Specifies the control appears dimmed when the WINDOW or APPLICATION is first opened.
KEY	Specifies an integer constant or keycode equate that immediately gives focus to the control.
MSG	Specifies a string constant containing the text to display in the status bar when the control has focus.
HLP	Specifies a string constant containing the help system identifier for the control.
SKIP	Specifies the control receives input focus to enter text only with the mouse or accelerator key and does not retain focus.
FONT	Specifies the display font for the control.
REQ	Specifies the control may not be left blank or zero.
FULL	Specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter.
SCROLL	Specifies the control scrolls with the window.
ALRT	Specifies "hot" keys active for the control.
HIDE	Specifies the control does not appear when the WINDOW or APPLICATION is first opened. UNHIDE must be used to display it.
READONLY	Specifies the control does not allow data entry.
DROPID	Specifies the control may serve as a drop target for drag-and-drop actions.
INS / OVR	Specifies Insert or Overwrite entry mode (valid only on windows with the MASK attribute).
UPR	Specifies all upper case entry.
HSCROLL	Specifies that a horizontal scroll bar is automatically added to the text field when any portion of the data lies horizontally outside the visible area.
VSCROLL	Specifies that a vertical scroll bar is automatically added to the text field when any of the data lies vertically outside the visible area.
HVSCROLL	Specifies that both vertical and horizontal scroll bars are automatically added to the text

field when any portion of the data lies outside the visible area.

LEFT Specifies that the text is left justified within the area specified by the AT attribute.

RIGHT Specifies that the text is right justified within the area specified by the AT attribute.

CENTER Specifies that the text is centered within the area specified by the AT attribute.

The **TEXT** control places a multi-line data entry field on the WINDOW (or TOOLBAR) at the position and size specified by its AT attribute. The variable specified in the USE attribute receives the data entered when the user has completed data entry and moves on to another control.

Events Generated:

EVENT:Selected The control has received input focus.

EVENT:Accepted The user has completed data entry in the control.

EVENT:PreAlertKey
The user pressed an ALRT attribute hot key.

EVENT:AlertKey The user pressed an ALRT attribute hot key.

EVENT:Drop A successful drag-and-drop to the control.

Example:

```
MDIChild  WINDOW(´Child One´),AT(0,0,320,200),MDI,MAX,HVSCROLL
          TEXT,AT(0,0,40,40),USE(E1),ALRT(F10Key),CENTER
          TEXT,AT(20,0,40,40),USE(E2),KEY(F10Key),HLP(´Text4Help´)
          TEXT,AT(40,0,40,40),USE(E3),SCROLL,OVR,UPR
          TEXT,AT(60,0,40,40),USE(E4),CURSOR(CURSOR:Wait),RIGHT
          TEXT,AT(80,0,40,40),USE(E5),DISABLE,FONT(´Arial´,12)
          TEXT,AT(100,0,40,40),USE(E6),HVSCROLL,LEFT
          TEXT,AT(120,0,40,40),USE(E7),REQ,INS,CAP,MSG(´Text Field 7´)
        END
```

Control Field Attributes

ROUND (set round-cornered window BOX)

SCROLL (set scrolling control)

SEPARATOR (set separator line ITEM)

SKIP (set Tab key skip)

STD (set standard behavior)

STEP (set SPIN increment)

TRN (set transparent window string)

USE (set control variable or equate label)

VCR (set VCR control)

ALRT (set control "hot" keys)

ALRT(*keycode***)**

ALRT Specifies a "hot" key active while the control has focus.

keycode A numeric constant keycode or keycode equate.

The **ALRT** attribute specifies a "hot" key active while the control has focus. When the user presses an ALRT "hot" key for a control, two field-specific events, EVENT:PreAlertKey and EVENT:AlertKey, are generated. If the code executes a CYCLE statement when processing EVENT:PreAlertKey, you "shortstop" the EVENT:AlertKey, preventing library´s default action on the alerted keypress for the control.

You may have multiple ALRT attributes on one control. The ALERT statement and the ALRT attribute of a window or control are completely separate. This means that clearing ALERT keys has no effect on any keys alerted by ALRT attributes.

Example:

```
WinOne WINDOW,AT(0,0,160,400)
         ENTRY,AT(6,40),USE(SomeVar1),ALRT(F9Key)    !F9 alerted for control
         ENTRY,AT(60,40),USE(SomeVar2),ALRT(F10Key)  !F10 alerted for control
       END
  CODE
  OPEN(WinOne)
  ACCEPT
    CASE FIELD()
    OF ?SomeVar1
      CASE EVENT()
      OF EVENT:PreAlertKey       !Pre-check alert events
        IF NOT SomeVar1
          CYCLE                  !Terminate alert processing on other controls
        END
      OF EVENT:AlertKey            !Alert processing
        DO F9Routine
      END
    OF ?SomeVar2
      CASE EVENT()
      OF EVENT:AlertKey            !Alert processing
        DO F10Routine
      END
    END
  END
```

AT (set control position and size in window)

AT([*x*] [,*y*] [,*width*] [,*height*])

AT	Defines the position and size of a control.
x	An integer constant or constant expression that specifies the horizontal position of the top left corner. If omitted, the runtime library provides a default value (zero).
y	An integer constant or constant expression that specifies the vertical position of the top left corner. If omitted, the runtime library provides a default value (zero).
width	An integer constant or constant expression that specifies the width. If omitted, the runtime library provides a default value.
height	An integer constant or constant expression that specifies the height. If omitted, the runtime library provides a default value.

The **AT** attribute defines the position and size of a control. If any parameter is omitted, the runtime library provides a default value.

The values contained in the *x*, *y*, *width*, and *height* parameters are measured in dialog units. Dialog units are defined as one-quarter the average character width by one-eighth the average character height. The size of a dialog unit is dependent upon the size of the default font for the window. This measurement is based on the font specified in the FONT attribute of the window, or the system default font specified by Windows.

Example:

```
 !Measurement in dialog units
WinOne WINDOW,AT(0,0,160,400)
         ENTRY,AT(8,40,80,8) !Approx. 2 characters in, 5 down, 20 wide, 1 high
       END

 !Measurement in Tousandths of an Inch
WinOne WINDOW,AT(0,0,160,400)
         ENTRY,AT(1000,1000,2000,250) !1" in & down, 2" wide, 1/4" high
       END

 !Measurement in Millimeters
WinOne WINDOW,AT(0,0,160,400)
         ENTRY,AT(100,100,200,50) !1 cm in and down, 2 cm wide, 50 mm high
       END

 !Measurement in Points
WinOne WINDOW,AT(0,0,160,400)
         ENTRY,AT(72,72,144,18) !1" in & down, 2" wide, 1/4" high
       END
```

BOXED (set window controls group border)

BOXED

The **BOXED** attribute specifies a single-track border around a GROUP or OPTION structure. The *text* parameter of the GROUP or OPTION control appears in a gap at the top of the border box. If BOXED is omitted, the *text* parameter of the GROUP or OPTION control is not displayed on screen.

CAP, UPR (set display case)

CAP
UPR

The **CAP** and **UPR** attributes specify the automatic case of text entered into ENTRY or TEXT controls when the MASK attribute is on the window. UPR specifies all upper case.

The CAP attribute specifies "Proper Name Capitalization," where the first letter of each word is capitalized and all other letters are lower case. The user can override this default behavior by pressing the SHIFT key to allow an upper case letter in the middle of a name (allowing for names such as, "McDowell") or SHIFT while CAPS-LOCK is on, forcing a lower case first letter (allowing for names such as, "von Richtofen").

CHECK (set on/off ITEM)

CHECK

The **CHECK** attribute specifies an ITEM that may be either ON or OFF. When ON, a check appears to the left of the menu selection and the USE variable receives the value one (1). When OFF, the check to the left of the menu selection disappears and the USE variable receives the value zero (0).

CLASS (set .VBX custom control class)

CLASS(*file* [,*name*] **)**

CLASS	The specifies the filename and type of .VBX custom control.
file	A string constant containing the name of the .VBX file (including the .VBX extension) in which the custom control is implemented.
name	A string constant containing the name of the custom control type from the .VBX file. If omitted, the first control type defined in the .VBX file is used.

The **CLASS** attribute specifies the filename and type of .VBX custom control. The *name* parameter identifies the specific control to use in a .VBX that contains multiple controls.

Example:

```
WinOne WINDOW,AT(0,0,160,400)
         CUSTOM,AT(0,0,120,320),CLASS(´graph.vbx´,´graph´),´graphstyle´(´2´)
       END
```

COLOR (set control display color)

COLOR(*rgb***)**

COLOR	Specifies display color.
rgb	A LONG or ULONG integer constant, or constant EQUATE, containing the red, green, and blue components that create the color in the three low-order bytes (bytes 0, 1, and 2), or an EQUATE for a standard Windows color value.

The **COLOR** attribute specifies the display color of a BOX, LINE, ELLIPSE, or REGION control. On a BOX, ELLIPSE, or REGION, the color specified is the color used for the border.

EQUATEs for Windows´ standard colors are contained in the EQUATES.CLW file. Windows automatically finds the closest match to the specified *rgb* color value for the hardware on which the program is run.

Windows standard colors may be reconfigured by the user in the Windows Control Panel. Any control using a Windows standard color is automatically repainted with the new color when this occurs.

Example:

```
WinOne WINDOW,AT(0,0,160,400)
         BOX,AT(20,20,20,20),COLOR(COLOR:ACTIVEBORDER)
                                          !Windows´ active border color
         BOX,AT(100,100,20,20),COLOR(00FF0000h)  !Blue
         BOX,AT(140,140,20,20),COLOR(0000FF00h)  !Green
         BOX,AT(180,180,20,20),COLOR(000000FFh)  !Red
       END
```

COLUMN (set list box highlight bar)

COLUMN

The **COLUMN** attribute specifies a field-by-field highlight bar on a LIST or COMBO control with multiple display columns.

CURSOR (set control mouse cursor type)

CURSOR(*file***)**

CURSOR Specifies a mouse cursor to display for the control.

file A string constant containing the name of a .CUR file, or an EQUATE naming a Windows-standard mouse cursor. The .CUR file is linked into the .EXE as a resource.

The **CURSOR** attribute specifies a mouse cursor to be displayed when the mouse is positioned over the control.

EQUATE statements for the Windows-standard mouse cursors are contained in the EQUATES.CLW file. The following list is a representative sample of these (see EQUATES.CLW for the complete list):

CURSOR:None	No mouse cursor
CURSOR:Arrow	Normal windows arrow cursor
CURSOR:IBeam	Capital "I" like a steel I-beam
CURSOR:Wait	Hourglass
CURSOR:Cross	Large plus sign
CURSOR:UpArrow	Vertical arrow
CURSOR:Size	Four-headed arrow
CURSOR:Icon	Box within a box
CURSOR:SizeNWSE	Double-headed arrow slanting left
CURSOR:SizeNESW	Double-headed arrow slanting right
CURSOR:SizeWE	Double-headed horizontal arrow
CURSOR:SizeNS	Double-headed vertical arrow
CURSOR:DragWE	Double-headed horizontal arrow

Example:

```
WinOne WINDOW,AT(0,0,160,400)
         REGION,AT(20,20,20,20),CUSOR(CURSOR:IBeam)
         REGION,AT(100,100,20,20),CURSOR(´Custom.CUR´)
       END
```

DEFAULT (set enter key button)

DEFAULT

The **DEFAULT** attribute specifies a BUTTON that is automatically pressed when the user presses the ENTER key. Only one active BUTTON on a window should have this attribute.

DISABLE (set control dimmed at open)

DISABLE

The **DISABLE** attribute specifies a control that is disabled when the WINDOW or APPLICATION is opened. The disabled control may be activated with the ENABLE statement.

DROP (set list box behavior)

DROP(*count***)**

DROP Specifies the list appears only when the user presses an arrow cursor key or clicks on the drop icon.

count An integer constant that specifies the number of elements displayed.

The **DROP** attribute specifies that the selection list appears only when the user presses an arrow cursor key or clicks on the drop icon to the right of the currently selected value display. Once it drops into view, the list displays *count* number of elements. If the DROP attribute is omitted, the LIST or COMBO control always displays the number of data items specified by the *height* parameter of the control´s AT of the selection list.

The DROP attribute does not work on a WINDOW with the MODAL attribute and should not be used.

Example:

```
WinOne WINDOW,AT(0,0,160,400)
         LIST,AT(120,0,20,20),USE(?L7),FROM(Que1),DROP(6)
         COMBO(@S8),AT(120,120,20,20),USE(?C7),FROM(Que2),DROP(8)
       END
```

DRAGID (set drag-and-drop host signatures)

DRAGID(*signature* [, *signature*] **)**

DRAGID Specifies a LIST or REGION control that can serve as a drag-and-drop host.

signature A string constant containing an identifier used to indicate valid drop targets. Any *signature* that begins with a tilde (~) indicates that the information can also be dragged to an external (Clarion) program. A single DRAGID may contain up to 16 *signatures*.

The **DRAGID** attribute specifies a LIST or REGION control that can serve as a drag-and-drop host. DRAGID works in conjunction with the DROPID attribute. The DRAGID *signature* strings (up to 16) define validation keys to match against the *signature* parameters of the target control´s DROPID. This provides control over where successful drag-and-drop operations are allowed.

A drag-and-drop operation occurs when the user drags information from a control with the DRAGID attribute to a control with the DROPID attribute. For a successful drag-and-drop operation, both controls must have at least one identical *signature* string in their respective DRAGID and DROPID attributes.

Example:

```
WinOne WINDOW,AT(0,0,160,400)
        LIST,AT(120,0,20,20),USE(?List1),FROM(Que1),DRAGID(´FromList1´)
                                    !Allows drags, but not drops
        LIST,AT(120,120,20,20),USE(?List2),FROM(Que2),DROPID(´FromList1´)
                                    !Allows drops from List1, but no drags
       END
  CODE
  OPEN(WinOne)
  ACCEPT
    CASE EVENT()
    OF EVENT:Drag           !When a drag event is attempted
      IF DRAGID()           ! check for success
        SETDROPID(Que1)       ! and setup info to pass
      END
    OF EVENT:Drop           !When drop event is successful
      Que2 = DROPID()         ! get dropped info
      ADD(Que2)             ! and add it to the queue
    END
  END
```

See Also:

DROPID

DROPID (set drag-and-drop target signatures)

DROPID(*signature* [, *signature*] **)**

DROPID Specifies a control that can serve as a drag-and-drop target.

signature A string constant containing an identifier used to indicate valid drag hosts. A single DROPID may contain up to 16 *signatures*. Any *signature* that begins with a tilde (~) indicates that the information can also be dropped from an external (Clarion) program. A DROPID *signature* of ´~FILE´ indicates the target accepts a comma-delimited list of filenames dragged from the Windows File Manager.

The **DROPID** attribute specifies a control that can serve as a drag-and-drop target. DROPID works in conjunction with the DRAGID attribute. The DROPID *signature* strings (up to 16) define validation keys to match against the *signature* parameters of the host control´s DRAGID. This provides control over where successful drag-and-drop operations are allowed.

A drag-and-drop operation occurs when the user drags information from a control with the DRAGID attribute to a control with the DROPID attribute. For a successful drag-and-drop operation, both controls must have at least one identical *signature* string in their respective DRAGID and DROPID attributes.

Example:

```
WinOne WINDOW,AT(0,0,160,400)
        LIST,AT(120,0,20,20),USE(?List1),FROM(Que1),DRAGID(´FromList1´)
                                !Allows drags, but not drops
        LIST,AT(120,120,20,20),USE(?List2),FROM(Que2),DROPID(´FromList1´,´~FILE´)
                              !Allows drops from List1 or the Window File Manager,
                              ! but no drags
       END
  CODE
  OPEN(WinOne)
  ACCEPT
    CASE EVENT()
    OF EVENT:Drag             !When a drag event is attempted
      IF DRAGID()             ! check for success
        SETDROPID(Que1)         ! and setup info to pass
      END
    OF EVENT:Drop             !When drop event is successful
      Que2 = DROPID()           ! get dropped info
      ADD(Que2)               ! and add it to the queue
    END
  END
```

See Also:

DRAGID

FILL (set display fill color)

FILL(*rgb***)**

FILL	Specifies display fill color.
rgb	A LONG or ULONG integer constant, or constant EQUATE, containing the red, green, and blue components that create the color in the three low-order bytes (bytes 0, 1, and 2) or an EQUATE for a standard Windows color value.

The **FILL** attribute specifies the display fill color of a BOX, ELLIPSE, or REGION control. If omitted, the control is not filled with color.

Example:

```
WinOne WINDOW,AT(0,0,160,400)
         BOX,AT(20,20,20,20),FILL(COLOR:ACTIVEBORDER)
                                            !Windows´ active border color
         BOX,AT(100,100,20,20),FILL(00FF0000h)    !Blue
         BOX,AT(140,140,20,20),FILL(0000FF00h)    !Green
         BOX,AT(180,180,20,20),FILL(000000FFh)    !Red
       END
```

FIRST, LAST (set MENU or ITEM position)

FIRST
LAST

The **FIRST** and **LAST** attributes specify menu selection positioning within the global pulldown menu, when a WINDOW´s MENUBAR is merged into the global menu. The order of priorities is:

1. Global selections with FIRST attribute
2. Local selections with FIRST attribute
3. Global selections without FIRST or LAST attributes
4. Local selections without FIRST or LAST attributes
5. Global selections with LAST attribute
6. Local selections with LAST attribute

FONT (set control font)

FONT([*typeface*] [,*size*] [,*color*] [,*style*]**)**

FONT	Specifies the display font for a control.
typeface	A string constant containing the name of the font. If omitted, the default font is used.
size	An integer constant containing the size (in points) of the font. If omitted, the system default font size is used.
color	A LONG integer constant containing the red, green, and blue values for the color of the font in the low-order three bytes, or an EQUATE for a standard Windows color value. If omitted, the default font color is used.
style	An integer constant, constant expression, or EQUATE specifying the strike weight and style of the font. If omitted, the default font weight is used.

The **FONT** attribute specifies the display font for the control, overriding any FONT specified on the WINDOW.

The *typeface* may name any font registered in the Windows system. The EQUATES.CLW file contains EQUATE values for standard *style* values. A *style* on the range zero (0) to one thousand (1000) specifies the strike weight of the font. You may add to that values that indicate italic, underline, or strikeout text. The following EQUATES are in EQUATES.CLW:

FONT:thin	EQUATE (100)
FONT:regular	EQUATE (400)
FONT:bold	EQUATE (700)
FONT:fixed	EQUATE (0800H)
FONT:italic	EQUATE (01000h)
FONT:underline	EQUATE (02000h)
FONT:strikeout	EQUATE (04000h)

Example:

```
WinOne WINDOW,AT(0,0,160,400)
        LIST,AT(120,0,20,20),USE(?L7),FROM(Que1),FONT(´Arial´,14,0FFh)
                        !14 point Arial typeface, Red, normal
        LIST,AT(120,120,20,20),USE(?C7),FROM(Que2),FONT(´Arial´,14,0,700)
                        !14 point Arial typeface, Black, Bold
        LIST,AT(120,240,20,20),USE(?C7),FROM(Que2),FONT(´Arial´,14,0,700+01000h)
                        !14 point Arial typeface, Black, Bold Italic
      END
```

FORMAT (set LIST or COMBO layout)

FORMAT(*format string***)**

FORMAT Specifies the display format of the data in the LIST or COMBO control.

format string A string constant specifying the display format.

The **FORMAT** attribute specifies the display format of the data in the LIST or COMBO control. The *format string* contains the information for single or multi-column formatting of the data.

The *format string* contains "field-specifiers" which map to the fields of the QUEUE. Multiple "field-specifiers" may be grouped together as a "field-group" in square brackets (**[ ]**) to display as a single unit.

Only the fields in the QUEUE for which there are "field-specifiers" are included in the display. This means that, if there are two fields specified in the *format string* and three fields in the QUEUE, only the two specified in the *format string* are displayed in the LIST or COMBO control.

The following describes the components allowed in a *format string*:

"Field-specifier" format: *width justification* [**(***indent***)**] [*modifiers*]

width A required integer defining the width of the field. Specified in dialog units.

justification A single capital letter (**L** , **R** , **C** , or **D**) that specifies **L**eft, **R**ight, **C**enter, or **D**ecimal justification. One is required.

indent An optional integer, enclosed in parentheses, that specifies the indent from the justification. This may be negative. With left (**L**) justification, *indent* defines a left margin; with right (**R**) or decimal (**D**), it defines a right margin; and with center (**C**), it defines an offset from the center of the field (negative = left offset).

modifiers: Optional special characters (listed below) to modify the display format of the field or group. Multiple *modifiers* may be used on one field or group.

~header~ [justification [(indent)]] A header string enclosed in tildes, followed by optional justification and/or indent, displays the header at the top of the list. The header uses the same justification and indent as the field, if not specifically overidden.

@picture@ The *picture* formats the field for display. The trailing @ is required to define the end of the *picture*, so that display pictures like @N12~Kr~ can be used in the format string without creating ambiguity.

? A question mark defines the locator field for a COMBO list box with a selector field. For a drop-down multi-column list box, this is the value displayed in the current-selection box.

#number# The *number* enclosed in pound signs (#) indicates the QUEUE field to display. Following fields in the format string without an explicit #*number*# are taken in order from the fields following the #*number*# field. For example, #2# on the first field in the format string indicates starting with the second field in the QUEUE, skipping the first. If the number of fields specified in the format string are >= the number of fields in the QUEUE, the format "wraps around" to the start of the QUEUE.

_ An underscore underlines the field.

/ A slash causes the next field to appear on a new line (only used on a field within a group).

| A vertical bar places a vertical line to the right of the field.

M An M allows the field or group of fields to be dynamically re-sized at runtime. This allows the user to drag the right vertical bar (if present) or right edge of the data area.

F An F creates a fixed column in the list that stays on screen when the user horizontally pages through the fields (by the HSCROLL attribute). Fixed fields or groups must be at the start of the list. This is ignored if placed on a field within a group.

S(*integer*) An S followed by an *integer* in parentheses adds a scroll bar to the group. The *integer* defines the total number of dialog units to scroll. This allows large fields to be displayed in a small column width. This is ignored if placed on a field within a group.

"Field-group" format: **[** *multiple field-specifiers* **]** [**(***size***)**] [*modifiers*]

multiple field-specifiers
A list of field-specifiers contained in square brackets (**[ ]**) that cause them to be treated as a single display unit.

size An optional integer, enclosed in parentheses, that specifies the default width of the group. If omitted, the size is calculated from the enclosed fields.

modifiers The "field-group" *modifiers* act on the entire group of fields. These are the same *modifiers* listed above.

Example:

```
StrLists     PROCEDURE
ThisFormat     STRING(200),AUTO  !Current selected display format string
TQ           QUEUE,AUTO         !Display formats list box FROM queue
TT             STRING(200)
             END
TD           QUEUE,AUTO         !Data list box FROM queue
FName           STRING(20)
LName           STRING(20)
Init          STRING(4)
Wage          REAL
Address         STRING(40)
State           STRING(10)
             END
Win WINDOW(´List Boxes´),AT(0,0,366,181),SYSTEM,DOUBLE,MDI
     LIST,AT(13,6,346,12),USE(ThisFormat),FROM(TQ),FORMAT(´400L´),DROP(6)
     LIST,AT(0,34,366,146),FORMAT(´80L80L16L60L160L40L´),FROM(TD),USE(?Show) |
             ,HVSCROLL
   END
  CODE
  LOOP 20 TIMES
    RandomAlphaData(FName)
    RandomAlphaData(LName)
    RandomAlphaData(Init)
    Wage = RANDOM(1,5000)
    RandomAlphaData(Address)
    RandomAlphaData(State)
    ADD(TD)
  END
  TT = ´80L80L16L60L160L40L´       !Single-row data without headers
  ThisFormat = TT
  ADD(TQ)
  TT = ´80C~First Name~80C~Last Name~16L~Intls~60R~Wage~160C~Address~40C~State~|´
  ADD(TQ)                          !Single-row data with headers
   TT = ´80C~First Name~80C~Last Name~16L~Intls~60D(10)~Wage~160C~Address~40C~State~|´
```

```
  ADD(TQ)                          !With headers and aligned decimal
  TT = ´[80C80C16L]~Name~|M[60D(10)@N$12.2@]~Wage~|M[160C40C]~Address~|M´
  ADD(TQ)                          !Added vertical size bars between columns
  TT = ´[80L~ForeName~/80L~Surname~16R~Init~]|M[60D(10)]~Wage~|M[160C40C]~Address~|´
  ADD(TQ)                          !Vertical size bars and multi-line name column
  TT = ´[80L~ForeName~/80L~Surname~16R~Init~](60)F|M[60D(10)]´ & |
       ´~Wage~|M[160C40C]~Address~|´
  ADD(TQ)                          !Fixed Multi-line name column w/ Hscroll bar
  OPEN(Win)
  ACCEPT
    CASE ACCEPTED()
    OF ?ThisFormat                 !When user selects a format
      ?Show{PROP:format} = ThisFormat      ! change FORMAT attribute to new format
    END
  END

RandomAlphaData PROCEDURE(Field)       !MAP Prototype is: RandomAlphaData(*STRING)
  CODE
  y# = RANDOM(1,SIZE(Field))          !Random fill size
  LOOP x# = 1 to y#                   !Fill each character with
    Field[x#] = CHR(RANDOM(97,122))    ! a random lower case letter
  END
```

FROM (set window listbox data source)

FROM(*source*)

FROM	Specifies the source of the data elements displayed in a LIST, COMBO, or SPIN.
source	The label of a QUEUE, a field within a QUEUE, or a string constant containing the data items to display in the list.

The **FROM** attribute specifies the source of the data elements displayed in a LIST, COMBO, or SPIN.

For a SPIN control, the *source* would usually be a QUEUE field or string. If the *source* is a QUEUE with multiple fields, only the first field is displayed in the SPIN.

For LIST and COMBO controls, the data elements are formatted for display according to the information in the FORMAT attribute. If the label of a QUEUE is specified as the *source*, all fields in the QUEUE are displayed. If the label of one field in a QUEUE is specified as the *source*, only that field is displayed.

If a string constant is specified as the *source*, the individual data elements to display in the LIST must be delimited by a vertical bar (|) character. To include a vertical bar as part of one data element, place two adjacent vertical bars in the string (||), and only one will be displayed. To indicate that an element is empty, place at least one blank space between the two vertical bars delimiting the elements (| |).

Example:

```
Que1 QUEUE,PRE(Q1)
F1     LONG
F2     STRING(8)
     END

Win1 WINDOW,AT(0,0,160,400)
       LIST,AT(120,0,20,20),USE(?L1),FROM(Que1),FORMAT(´5C~List~15L~Box~´),COLUMN
       COMBO(@S8),AT(120,120,20,20),USE(?C1),FROM(Q1:F2)
       SPIN(@N8.2),AT(280,0,20,20),USE(SpinVar1),FROM(Q1:F1)
       SPIN(@S4),AT(280,0,20,20),USE(SpinVar2),FROM(´Mr.|Mrs.|Ms.|Dr.´)
     END
```

FULL (set full-screen)

FULL

The **FULL** attribute specifies the control expands to occupy the entire size of the WINDOW for any missing AT attribute width or height parameter.

FULL may not be specified for TOOLBAR controls.

HIDE (set control hidden at open)

HIDE

The **HIDE** attribute specifies the control does not appear when the WINDOW or APPLICATION is first opened. UNHIDE must be used to display it.

HLP (set control´s on-line help identifier)

HLP(*helpID***)**

HLP	Specifies the *helpID* for the control.
helpID	A string constant specifying the key used to access the Help system. This may be either a Help keyword or a "context string."

The **HLP** attribute specifies the *helpID* for the control. Help, if available, is automatically displayed by Windows whenever the user presses F1. If the user presses F1 to request help when the control has input focus, the library uses the control´s *helpID* to search the help file until an object with that *helpID* is found.

The *helpID* may contain a Help keyword or a "context string." A Help keyword is a keyword or phrase that is displayed in the Help Search dialog. When the user presses F1, if only one topic in the help file specifies this keyword, the help file is opened at that topic; if more than one topic specifies the keyword, the search dialog is opened for the user.

A "context string" is identified by a leading tilde (~) in the *helpID*, followed by a unique identifier (no spaces allowed) associated with exactly one help topic. When the user presses F1, the help file is opened at the specific topic associated with that "context string." If the tilde is missing, the *helpID* is assumed to be a help keyword.

Example:

```
Win1 WINDOW
       ENTRY(@s30),USE(SomeVariable),HLP(´~Entry1Help´)!A help context string
       ENTRY(@s30),USE(SomeVariable),HLP(´Control Two Help´)!A help keyword
     END
```

HSCROLL, VSCROLL, HVSCROLL (set control scroll bars)

HSCROLL
VSCROLL
HVSCROLL

The **HSCROLL**, **VSCROLL**, and **HVSCROLL** attributes place scroll bars on a COMBO, LIST, IMAGE, or TEXT control. HSCROLL adds a horizontal scroll bar to the bottom; VSCROLL adds a vertical scroll bar on the right side, and HVSCROLL adds both.

The vertical scroll bar allows a mouse to scroll the control´s display up or down. The horizontal scroll bar allows a mouse to scroll the control´s display left or right. The scroll bars appear whenever any scrollable portion of the control lies outside the visible area on screen.

When you place VSCROLL on a LIST with the IMM attribute, the vertical scroll bar is always present, even when the list is not full. When the user clicks on the scroll bar, events are generated, but the list contents do not move (executable code should perform this task). You can interrogate the PROP:VscrollPos property to determine the scroll thumb´s position in the range 0 (top) to 100 (bottom).

ICON (set control icon)

ICON([*file*] **)**

ICON Specifies an icon to display as the control.

file A string constant or EQUATE containing the name of an .ICO file or Windows standard icon to display. The .ICO file is automatically linked into the .EXE as a resource.

The **ICON** attribute specifies an icon to display as the control. The icon is displayed on the button face of the control. The ICON attribute may be specified on a BUTTON, RADIO, or CHECK control. For RADIO and CHECK controls, the ICON attribute creates "latched" pushbuttons, where the control button appears "down" when on and "up" when off.

EQUATE statements for the Windows-standard icons are contained in the EQUATES.CLW file. The following list is a representative sample of these (see EQUATES.CLW for the complete list):

```
ICON:None                   No icon
ICON:Application
ICON:Question               ?
ICON:Exclamation            !
ICON:Asterisk               *
ICON:VCRtop                 >>|
ICON:VCRrewind              <<
ICON:VCRback                <
ICON:VCRplay                >
ICON:VCRfastforward     >>
ICON:VCRbottom              |<<
ICON:VCRlocate              ?
```

Example:

```
MDIChildWINDOW(´Child One´),AT(0,0,320,200),MDI,MAX,HVSCROLL
          OPTION(´Option´),USE(OptVar)
            RADIO(´Radio 1´),AT(120,0,20,20),USE(?R1),ICON(´Radio1.ICO´)
            RADIO(´Radio 2´),AT(140,0,20,20),USE(?R2),ICON(´Radio2.ICO´)
          END
         CHECK(´&A´),AT(0,120,20,20),USE(?C7),ICON(ICON:Asterisk)
         BUTTON(´&1´),AT(120,0,20,20),USE(?B7),ICON(ICON:Question)
        END
```

IMM (set immediate event notification)

IMM

The **IMM** attribute specifies immediate generation of an event.

On a REGION control, the IMM attribute generates an event whenever the mouse enters, moves within, or leaves the area specified by the REGION´s AT attribute. The exact position of the mouse can be deteremined by the MOUSEX and MOUSEY functions.

On a BUTTON control, the IMM attribute indicates the BUTTON generates an event when the left mouse button is pressed down on the control, instead of on its release. The event is continuously generated as along as the user keeps the mouse button pressed.

The IMM attribute specifies immediate event generation each time the user presses any keystroke on a LIST or COMBO control, usually requiring the QUEUE to be re-filled. When the user presses a printable character, EVENT:AlertKey is generated. It does the same thing on an ENTRY or SPIN control.

INS, OVR (set typing mode)

INS
OVR

The **INS** and **OVR** attributes specify the typing mode for an ENTRY or TEXT control when the MASK attribute is present on the window. INS specifies insert mode while OVR specifies overwrite mode. These modes are only active on windows with the MASK attribute.

KEY (set control execution keycode)

KEY(*keycode***)**

KEY Specifies a "hot" key for the control

keycode A Clarion Keycode or keycode equate label.

The **KEY** attribute specifies a "hot" key to immediately give focus to the control or execute the control´s associated action.

The following controls receive focus:

COMBO
CUSTOM
ENTRY
GROUP
LIST
OPTION
PROMPT
SPIN
TEXT

The following controls both receive focus and immediately execute:

BUTTON
CHECK
CUSTOM
RADIO

Example:

```
WinOne WINDOW,AT(0,0,160,400)
         COMBO(@S8),AT(120,120,20,20),USE(?C1),FROM(Q1:F2),KEY(F1Key)
         LIST,AT(120,0,20,20),USE(?L1),FROM(Que1),KEY(F2Key)
         SPIN(@N8.2),AT(280,0,20,20),USE(SpinVar1),FROM(Q),KEY(F3Key)
         TEXT,AT(20,0,40,40),USE(E2),KEY(F4Key)
         PROMPT(´Enter &Data in E2:´),AT(10,200,20,20),USE(?P2),KEY(F5Key)
         ENTRY(@S8),AT(100,200,20,20),USE(E2),KEY(F6Key)
         BUTTON(´&1´),AT(120,0,20,20),USE(?B7),KEY(F7Key)
         CHECK(´&A´),AT(0,120,20,20),USE(?C7),KEY(F8Key)
         OPTION(´Option´),USE(OptVar),KEY(F9Key)
           RADIO(´Radio 1´),AT(120,0,20,20),USE(?R1),KEY(F10Key)
           RADIO(´Radio 2´),AT(140,0,20,20),USE(?R2),KEY(F11Key)
         END
       END
```

Note: Using the Property Assignment Syntax to reset this attribute for a control may be unsuccesful. (1119)

LEFT, RIGHT, CENTER, DECIMAL (set display justification)

LEFT([*indent*] **)**
RIGHT([*indent*] **)**
CENTER([*indent*] **)**
DECIMAL([*indent*] **)**

---

indent An integer constant specifying the amount of offset from the justification point. This is in dialog units.

The **LEFT**, **RIGHT**, **CENTER**, and **DECIMAL** attributes specify the justification of data displayed. LEFT specifies left justification, RIGHT specifies right justification, CENTER specifies centered text, and DECIMAL specifies numeric data aligned on the decimal point.

The *indent* parameter on the CENTER attribute specifies an offset from the center (negative = left offset). On the DECIMAL attribute, *indent* specifies the offset of the decimal point from the right.

The CHECK and RADIO controls allow LEFT or RIGHT only (without an *indent* parameter). The TEXT control allows only LEFT(*indent*), RIGHT(*indent*), or CENTER(*indent*).

The following controls allow LEFT(*indent*), RIGHT(*indent*), CENTER(*indent*), or DECIMAL(*indent*):

COMBO
ENTRY
LIST
SPIN
STRING

Example:

```
WinOne WINDOW,AT(0,0,160,400)
         COMBO(@S8),AT(120,120,20,20),USE(?C1),FROM(Q1:F2),RIGHT(4)
         LIST,AT(120,0,20,20),USE(?L1),FROM(Que1),CENTER
         SPIN(@N8.2),AT(280,0,20,20),USE(SpinVar1),FROM(Q),DECIMAL(8)
         TEXT,AT(20,0,40,40),USE(E2),LEFT(8)
         ENTRY(@S8),AT(100,200,20,20),USE(E2),LEFT(4)
         CHECK(´&A´),AT(0,120,20,20),USE(?C7),LEFT
         OPTION(´Option´),USE(OptVar)
           RADIO(´Radio 1´),AT(120,0,20,20),USE(?R1),LEFT
           RADIO(´Radio 2´),AT(140,0,20,20),USE(?R2),RIGHT
         END
       END
```

MARK (set multiple selection mode)

MARK(*flag***)**

MARK Enables multiple items selection.

flag The label of a QUEUE field.

The **MARK** attribute enables multiple items selection from a LIST or COMBO control. When an item in the LIST is selected, the appropriate *flag* field is set to true (1). Each marked entry is automatically highlighted in the LIST or COMBO. Changing the value of the *flag* field also changes the screen display for the related LIST or COMBO entry.

If the MARK attribute is specified on the LIST or COMBO, the IMM attribute may not be.

Example:

```
Que1    QUEUE,PRE(Q1)
MarkFlag  BYTE
F1        LONG
F2        STRING(8)
        END

WinOne WINDOW,AT(0,0,160,400)
         LIST,AT(120,0,20,20),USE(?L1),FROM(Q1:F1),MARK(Q1:MarkFlag)
         COMBO(@S8),AT(120,120,,),USE(?C1),FROM(Q1:F2),MARK(Q1:MarkFlag)
       END
```

MSG (set control status bar message)

MSG(*text***)**

MSG Specifies *text* to display in the status bar.

text A string constant containing the message to display in the status bar.

The **MSG** attribute specifies the *text* to display in the first zone of the status bar when the control has focus.

Example:

```
WinOne WINDOW,AT(0,0,160,400)
         COMBO(@S8),AT(120,120,20,20),USE(?C1),FROM(Q1:F2),MSG(´Enter or Select´)
         LIST,AT(120,0,20,20),USE(?L1),FROM(Que1),MSG(´Select One´)
         SPIN(@N8.2),AT(280,0,20,20),USE(SpinVar1),FROM(Q),MSG(´Choose One´)
         TEXT,AT(20,0,40,40),USE(E2),MSG(´Enter Text´)
         ENTRY(@S8),AT(100,200,20,20),USE(E2),MSG(´Enter Data´)
         CHECK(´&A´),AT(0,120,20,20),USE(?C7),MSG(´On or Off´)
         OPTION(´Option 1´),USE(OptVar),MSG(´Pick One or Two´)
           RADIO(´Radio 1´),AT(120,0,20,20),USE(?R1)
           RADIO(´Radio 2´),AT(140,0,20,20),USE(?R2)
         END
         OPTION(´Option´),USE(OptVar)
           RADIO(´Radio 1´),AT(120,40,20,20),USE(?R1),MSG(´Pick One´)
           RADIO(´Radio 2´),AT(140,40,20,20),USE(?R2),MSG(´Pick Two´)
         END
       END
```

NOBAR (set no highlight bar)

NOBAR

The **NOBAR** attribute specifies the currently selected element in the LIST is only highlighted when the LIST control has focus.

PASSWORD (set data non-display)

PASSWORD

The **PASSWORD** attribute specifies non-display of the data entered in the ENTRY control. When the user types in data, asterisks are displayed on screen for each character entered.

RANGE (set SPIN range limits)

RANGE(*lower,upper***)**

RANGE	Specifies the valid range of data values the user may select in a SPIN control.
lower	A numeric constant that specifies the lower inclusive limit of valid data.
upper	A numeric constant that specifies the upper inclusive limit of valid data.

The **RANGE** attribute specifies the valid range of data values the user may select in a SPIN control. This attribute works in conjunction with the STEP attribute to provide the user with choices in the SPIN control. When RANGE and STEP are used, the SPIN control´s FROM attribute is not.

Example:

```
WinOne WINDOW,AT(0,0,160,400)
         SPIN(@N4.2),AT(280,0,20,20),USE(SpinVar1),RANGE(.05,9.95),STEP(.05)
         SPIN(@n3),AT(280,0,20,20),USE(SpinVar2),RANGE(5,995),STEP(5)
       END
```

READONLY (set display-only)

READONLY

The **READONLY** attribute specifies a display-only COMBO, ENTRY, SPIN or TEXT control. The control may receive input focus with the mouse, but may not enter data. If the user attempts to change the displayed value, a beep warns the user that data entry is not allowed.

REQ (set required entry)

REQ

The **REQ** attribute specifies an ENTRY or TEXT control that may not be left blank or zero. The REQ attribute on an ENTRY or TEXT control is not checked until a BUTTON with the REQ attribute is pressed, or the INCOMPLETE() function is called.

When a BUTTON with the REQ attribute is pressed, or the INCOMPLETE() function is called, all ENTRY and TEXT controls with the REQ attribute are checked to ensure they contain data. The first control encountered in this check that does not contain data inmediately receives input focus.

RIGHT (set MENU position)

RIGHT

The **RIGHT** attribute specifies the MENU is placed at the right end of the action bar.

ROUND (set round-cornered window BOX)

ROUND

The **ROUND** attribute specifies a BOX control with rounded corners.

SCROLL (set scrolling control)

SCROLL

The **SCROLL** attribute specifies a control that moves with the window when the WINDOW scrolls. This allows "virtual" windows larger than the physical video display.

The presence of the SCROLL attribute means that the control stays fixed at a position in the window relative to the top left corner of the virtual window, whether that position is currently in view or not. This means that the control appears to move as the window scrolls.

If the SCROLL attribute is omitted, the control stays fixed at a position in the window relative to the top left corner of the currently visible portion of the window. This means that the control appears to stay in the same position on screen while the rest of the window scrolls. This is useful for controls which should stay visible to the user at all times (such as Ok or Cancel buttons).

Mixing controls with and without the SCROLL attribute on the same WINDOW can result in multiple controls appearing to occupy the same screen position. This occurs because the controls with SCROLL move and the controls without SCROLL do not. This condition is temporary and scrolling the window will correct the situation. The situation can be avoided entirely by careful placement of controls in the window. For example, you can place all controls without SCROLL at the bottom of the window then place all controls with SCROLL above them extending to the right and left. This would create a window that only scrolls horizontally.

SEPARATOR (set separator line ITEM)

SEPARATOR

The **SEPARATOR** attribute specifies an ITEM in a MENU that displays a horizontal line to group ITEMs within the MENU. No other attributes may be specified for the ITEM.

SKIP (set Tab key skip)

SKIP

The **SKIP** attribute specifies the control may only be accessed with the mouse or an accelerator key. Controls that allow data entry receive input focus only during data entry and the control does not retain focus. Controls that do not allow data entry do not receive or retain input focus. The effect of this is to create the same behavior as a control in a toolbar. When the mouse cursor is over a control with the SKIP attribute, the control´s MSG attribute is displayed in the status bar.

STD (set standard behavior)

STD(*behavior***)**

STD	Specifies standard Windows *behavior*.
behavior	An integer constant or EQUATE specifying the identifier of a standard windows behavior.

The **STD** attribute specifies the control activates some standard Windows action. This action is automatically executed by the runtime library and does not generate an event.

EQUATE statements for the standard Windows actions are contained in the EQUATES.CLW file. The following list is a representative sample of these (see EQUATES.CLW for the complete list):

STD:WindowList	List of open MDI windows
STD:TileWindow	Tile Windows
STD:CascadeWindow	Cascade Windows
STD:ArrangeIcons	Arrange Icons
STD:HelpIndex	Help Contents
STD:HelpSearch	Help Search dialog

Example:

```
MDIChildWINDOW(´Child One´),MDI,SYSTEM,MAX
          MENUBAR
            MENU(´Edit´),USE(?EditMenu)
              ITEM(´Undo´),USE(?UndoText),KEY(CtrlZ),STD(STD:Undo)
              ITEM(´Cu&t´),USE(?CutText),KEY(CtrlX),STD(STD:Cut)
              ITEM(´Copy´),USE(?CopyText),KEY(CtrlC),STD(STD:Copy)
              ITEM(´Paste´),USE(?PasteText),KEY(CtrlV),STD(STD:Paste)
            END
          END
          TOOLBAR
            BUTTON(´Cut´),USE(?OpenButton),ICON(ICON:Cut),STD(STD:Cut)
            BUTTON(´Copy´),USE(?OpenButton),ICON(ICON:Copy),STD(STD:Copy)
            BUTTON(´Paste´),USE(?OpenButton),ICON(ICON:Paste),STD(STD:Paste)
          END
        END
```

STEP (set SPIN increment)

STEP(*count***)**

STEP Specifies a SPIN control RANGE attribute´s increment/decrement value.

count A numeric constant specifying the amount to increment or decrement.

The **STEP** attribute specifies the amount by which a SPIN control´s value is incremented or decremented within its valid RANGE. The default STEP value is 1.0.

Example:

```
WinOne WINDOW,AT(0,0,160,400)
        SPIN(@N4.2),AT(280,0,20,20),USE(SpinVar1),RANGE(.05,9.95),STEP(.05)
        SPIN(@n3),AT(280,0,20,20),USE(SpinVar2),RANGE(5,995),STEP(5)
       END
```

TIP (set 'balloon help' text)

TIP(*string* **)**

TIP Specifies the text to display when the mouse cursor pauses over the control.

string A string constant that specifies the text to display.

The **TIP** attribute on a control specifies the text.to display in a balloon help box when the mouse cursor pauses over the control. Although there is no specific limit on the number of characters, the *string* should not be longer than can be displayed on the screen.

Although it is valid on any control that can gain focus for user input, this attribute is most commonly used on BUTTON controls with the ICON attribute that are placed on the TOOLBAR. This allows the user to quickly determine the controls purpose without accessing the on-line Help system.

Example:

```
WinOne WINDOW,AT(0,0,160,400)
         TOOLBAR
           BUTTON(E&xit),USE(?MainExitButton),ICON(ICON:hand),TIP(Exit Window)
           BUTTON(&Open),USE(?OpenButton),ICON(ICON:Open),TIP(Open a File)
         END
    COMBO(@S8),AT(120,120,20,20),USE(?C1),FROM(Q1:F2)
         ENTRY(@S8),AT(100,200,20,20),USE(E2)
       END
```

TRN (set transparent window string)

TRN

The **TRN** attribute on a STRING control specifies the characters display transparently, without obliterating the background over which the STRING is placed. Only the pixels required to create each character are written to the screen. This allows the STRING to be placed directly on top of an IMAGE without destroying the background picture.

Example:

```
WinOne WINDOW,AT(0,0,160,400)
         IMAGE(´PIC.BMP´),USE(?I1),FULL            !Full window image
         STRING(´String Constant´),AT(10,0,20,20),USE(?S1),TRN
                                                 !Transparent string on image
       END
```

USE (set control variable or equate label)

USE(| *label* | [,*number*] **)**
| *variable* |

USE	Specifies a variable or field equate label for the control.
label	A field equate label to reference the control in executable code.
variable	The variable to receive the value entered in the control.
number	An integer constant that specifies the number the compiler equates to the field equate label for the control.

The **USE** attribute specifies a variable or field equate label for the control. USE with a *label* parameter simply provides a mechanism for executable source code statements to reference the control. Some controls only allow a field equate *label* as the USE parameter, not a *variable*. These controls are: PROMPT, IMAGE, LINE, BOX, ELLIPSE, GROUP, RADIO, REGION, MENU, and BUTTON. USE with a *variable* parameter supplies the control with a variable to update by operator entry. This is applicable to an ITEM with the CHECK attribute, or an ENTRY, OPTION, SPIN, TEXT, LIST, COMBO, CHECK, or CUSTOM.

All controls in an APPLICATION or WINDOW are automatically assigned numbers by the compiler. For an APPLICATION´s MENUBAR controls, these numbers start at negative one (-1) and decrement by one (1) for each MENU and ITEM in the MENUBAR. On a WINDOW, these numbers start at one (1) and increment by one (1) for each control in the WINDOW.

The USE attribute´s *number* parameter allows you to specify the actual field number the compiler assigns to the control. This *number* also is used as the new starting point for subsequent field numbering for fields without a *number* parameter in their USE attribute. Subsequent controls without a *number* parameter in their USE attribute are incremented (or decremented) relative to the last *number* assigned.

Two or more controls with exactly the same USE variable in one WINDOW or APPLICATION structure would create the same Field Equate Label for all, therefore, when the compiler encounters this condition, all Field Equate Labels for that USE variable are discarded. This makes it impossible to reference any of these controls in executable code, preventing confusion about which control you really want to reference. It also allows you to deliberately create this condition to display the contents of the variable in multiple controls with different display pictures.

Example:

```
WinOne WINDOW,AT(0,0,160,400)
         COMBO(@S8),AT(120,120,20,20),USE(?C1),FROM(Q1:F2)
         ENTRY(@S8),AT(100,200,20,20),USE(E2)
       END
```

VALUE (set RADIO control OPTION USE variable assignment)

VALUE(*string* **)**

VALUE Specifies the value assigned to the OPTION structures USE variable when the RADIO control is selected by the user.

string A string constant that specifies the value to assign.

The **VALUE** attribute specifies the value that is automatically assigned to the OPTION structures USE variable when the RADIO control is selected by the user. This attribute overrides the RADIO controls *text* parameter.

All automatic type conversion rules apply to the *string* assigned to the OPTION structures USE variable. Therefore, if the *string* contains only numeric data and the USE variable is a numeric data type, it receives the numeric value of the *string*.

Example:

```
WinOne WINDOW,AT(0,0,160,400)
          OPTION(Option 1),USE(OptVar1),MSG(Pick One or Two)
          RADIO(Radio 1),AT(120,0,20,20),USE(?R1),VALUE(10) !OptVar1 receives 10
          RADIO(Radio 2),AT(140,0,20,20),USE(?R2),VALUE(20) !OptVar1 receives 20

        END
OPTION(Option 2),USE(OptVar2),MSG(Pick One or Two)
          RADIO(Radio 1),AT(120,0,20,20),USE(?R1),VALUE(10) !OptVar2 receives 10
          RADIO(Radio 2),AT(140,0,20,20),USE(?R2),VALUE(20) !OptVar2 receives 20

        END
             END
```

VCR (set VCR control)

VCR([*field*] **)**

VCR	Places Video Cassette Recorder (VCR) style buttons on a LIST or COMBO control.
field	A field equate label that specifies the ENTRY control to use as a locator for a LIST (not valid on a COMBO).

The **VCR** attribute places **V**ideo **C**assette **R**ecorder (VCR) style buttons on a LIST or COMBO control. The VCR style buttons affect the scrolling characteristics of the data displayed in the LIST or COMBO.

There are six buttons displayed as the VCR:

|<	Top of list
<<	Page Up
<	Entry Up
>	Entry Down
>>	Page Down
>|	Bottom of list

On a LIST control´s VCR(*field*), there also appears a button with a question mark (?) in the middle of the other buttons. This is the locator button that gives focus to the control specified by the *field* parameter. When the user enters data and then presses TAB on the locator *field*, the LIST scrolls to its closest matching entry.

Example:

```
MDIChildWINDOW(´Child One´),AT(0,0,320,200),MDI,MAX,HVSCROLL
          LIST,AT(140,0,20,20),USE(?L8),FROM(Que),HVSCROLL,VCR
          ENTRY(@S8),AT(100,200,20,20),USE(E2)
          LIST,AT(140,100,20,20),USE(?L8),FROM(Que),HVSCROLL,VCR(?E2)
        END
```

Window Commands

Event Processing

Event-driven programming

Windows programs are generally event-driven. This means the user causes an event by clicking the mouse on a screen control or pressing a key. Every user action in the program results in Windows sending a message to the program which owns the window telling it what the user has done. Once Windows has sent the message signaling an event to the program, the program has the opportunity to handle the event in the appropriate manner. This basically means the Windows programming paradigm is exactly opposite from the DOS programming paradigm--the operating system (Windows) tells the program what to do, instead of the program telling the operating system what to do.

Writing a Windows program in a programming language other than Clarion becomes very complex, because the program must be coded to explicitly handle every message from Windows. Common tasks, such as re-drawing graphics that have been overwritten by a window that was open and is now closed, must be explicitly coded in the program.

These common tasks could be handled automatically by writing generic procedures to accomplish the task and call them every time the need arises. Of course, in other programming languages, you would have to write these procedures yourself. In Clarion for Windows, they are already written and included as part of our runtime library. The Clarion language, therefore, has persistent graphics commands that do not require an explicit re-draw each time they are overwritten (unlike other languages).

In Clarion Windows programs, most of the messages from Windows are automatically handled internally by the ACCEPT event processor. These are the common events handled by the runtime library (screen re-draws, etc.). Only those events that actually may require program action are passed on by ACCEPT to your Clarion code. The net effect of this is to make your programming job easier by removing the low-level "drudgery" code from your program, allowing you to concentrate on the high-level aspects of programming, instead.

There are two types of events passed on to the program by ACCEPT: **Field-specific** and **Field-independent** events.

A **Field-specific** event occurs when the user presses a key that may require the program to perform a specific action related to that control.

A **Field-independent** event does not relate to any one control but requires some program action (for example, to close a window, quit the program, or change execution threads). Most of these events cause the system to become modal, since they require a response before the program may continue.

ACCEPT (the event processor)

```
ACCEPT
  statements
END
```

ACCEPT	The event handler.
statements	Executable code statements.

The **ACCEPT** loop is the event handler that processes events generated by Windows for the APPLICATION or WINDOW structures. An ACCEPT loop and a window are bound together, in that, when the window is opened, the next ACCEPT loop encountered will process all events for that window.

ACCEPT operates in the same manner as a LOOP--the BREAK and CYCLE statements can be used within it. The ACCEPT loop cycles for every event that requires program action. ACCEPT waits until the Clarion runtime library sends it an event that the program should process, then cycles through to execute its *statements*. During the time ACCEPT is waiting, the Clarion runtime library has control, automatically handling common events from Windows that do not need specific program action (such as screen re-draws).

The current contents of all STRING control USE variables (in the top window of each thread) automatically display on screen each time the ACCEPT loop cycles to the top. This eliminates the need to explicitly issue a DISPLAY statement to update the video display for display-only data. USE variable contents for any other control automatically display on screen for any event generated for that control, unless PROP:Auto is turned on to automatically display all USE variables each time through the ACCEPT loop.

Within the ACCEPT loop, the program determines what happened by using the following functions:

EVENT()	Returns a value indicating what happened. Symbolic constants for events are in the EQUATES.CLW file.
FIELD()	Returns the field number for the control to which the event refers, if the event is a field-specific event.
ACCEPTED()	Returns the field number for the control to which the event refers for the EVENT:Accepted event.
SELECTED()	Returns the field number for the control to which the event refers for the EVENT:Selected event.
FOCUS()	Returns the field number of the control that has input focus, no matter what event occurred.
MOUSEX()	Returns the x-coordinate of the mouse cursor.
MOUSEY()	Returns the y-coordinate of the mouse cursor.

Two events cause an implicit BREAK from the ACCEPT loop. These are the events that signal the close of a window (EVENT:CloseWindow) or close of a program (EVENT:CloseDown). The program´s code need not check for these events as they are handled automatically. However, the code may check for them and execute some specific action, such as displaying a "You sure?" window or handling some housekeeping details. A CYCLE statement at that point returns to the top of the ACCEPT loop without exiting the window or program.

Similarly, there are several other events whose action can also be terminated by a CYCLE statement: EVENT:PreAlertKey, EVENT:Move, EVENT:Size, EVENT:Restore, EVENT:Maximize, and EVENT:Iconize. A CYCLE statement in response to any of these events stops the normal action and prohibits generation

of the related EVENT:AlertKey, EVENT:Moved, EVENT:Sized, EVENT:Restored, EVENT:Maximized, or EVENT:Iconized.

Example:

```
CODE
OPEN(Window)
ACCEPT                          !Event handler
  CASE FIELD()
  OF 0                          !Handle Field-independent events
    CASE EVENT()
    OF EVENT:Move
       CYCLE                    !Do not allow user to move the window
    OF EVENT:Suspend
      CASE FOCUS()
      OF ?Field1
          !Save some stuff
     END
    OF EVENT:Resume
        !Restore the stuff
    END
  OF ?Field1                     !Handle events for Field1
    CASE EVENT()
    OF EVENT:Selected
        ! pre-edit code for field1
    OF EVENT:Accepted
        ! completion code for field1
    END
  OF ?Field2
    CASE EVENT()
    OF EVENT:Selected
        ! pre-edit code for field2
    OF EVENT:Accepted
        ! completion code for field2
    END
  END
```

See Also:

EVENT

FIELD

FOCUS

ACCEPTED

SELECTED

CYCLE

ALERT (set event generation key)

ALERT([*first-keycode*] [,*last-keycode*]**)**

ALERT	Specifies keys that generate an event.
first-keycode	A numeric keycode or keycode equate label. This may be the lower limit in a range of keycodes.
last-keycode	The upper limit keycode, or keycode equate label, in a range of keycodes.

ALERT specifies a key, or an inclusive range of keys, as event generation keys. Two field-independent events, EVENT:PreAlertKey and EVENT:AlertKey, are generated when the user presses the ALERTed key. If the code executes a CYCLE statement when processing EVENT:PreAlertKey, you "shortstop" the EVENT:AlertKey, preventing the library´s default action on the alerted keypress for the window.

The ALERT statement with no parameters clears all ALERT keys. Any key with a keycode may be used as the parameter of an ALERT statement. ALERT generates field-independent events, since it is not associated with any particular control. When EVENT:AlertKey is generated by an ALERT key, the USE variable of the control that currently has input focus is not automatically updated (use UPDATE if this is required).

The ALERT statement alerts its keys separately from the ALRT attribute of a window or control. This means that clearing all ALERT keys has no effect on any keys alerted by ALRT attributes.

Example:

```
Screen WINDOW,ALRT(F10Key),ALRT(F9Key) !F10 and F9 alerted
         LIST,AT(109,48,50,50),USE(?List),FROM(Que),IMM
         BUTTON(´&Ok´),AT(111,108,,),USE(?Ok)
         BUTTON(´&Cancel´),AT(111,130,,),USE(?Cancel)
       END
  CODE
  ALERT                          !Turn off all alerted keys
  ALERT(F1Key,F12Key)              !Alert all function keys
  ALERT(279)                     !Alert the Ctrl-Esc key
  OPEN(Screen)
  ACCEPT
    CASE EVENT()
    OF EVENT:PreAlertKey           !Pre-check alert events
      IF KEYCODE() = F4Key         !Dis-Allow F4 key
        CYCLE                    !Terminate alert processing
      END
    OF EVENT:AlertKey              !Alert processing
      CASE KEYCODE()
      OF 279                   !Check for Ctrl+Esc
        BREAK
      OF F9Key                 !Check for F9
        F9HotKeyProc             !Call hot key procedure
      OF F10Key                  !Check for F10
        F10HotKeyProc            !Call hot key procedure
      END
    END
  END
```

See Also:

UPDATE

EVENT (return event number)

EVENT( )

The **EVENT** function returns a number indicating what caused ACCEPT to alert the program that something has happened that it may need to handle. There are EQUATEs listed in EQUATES.CLW for all the events the program may need to handle.

There are two types of events generated by ACCEPT: field-specific and field-independent events. Field-specific events affect a single control, while field-independent events affect the window or program. The type of event can be determined by the values returned by the ACCEPTED, SELECTED, and FIELD functions. If you need to know which field has input focus on a field-independent event, use the FOCUS function.

For field-specific events:

The FIELD function returns the field number of the control on which the event occurred. The ACCEPTED function returns the field number if the event is EVENT:Accepted. The SELECTED function returns the field number if the event is EVENT:Selected.

For field-independent events:

The FIELD, ACCEPTED, and SELECTED functions all return zero (0).

Return Data Type: SHORT

Example:

```
ACCEPT
  CASE EVENT()
  OF EVENT:Selected
    CASE SELECTED()
    OF ?Control1
      !Pre-edit code here
    OF ?Control2
      !Pre-edit code here
    END
  OF EVENT:Accepted
    CASE ACCEPTED()
    OF ?Control1
      !Post-edit code here
    OF ?Control2
      !Post-edit code here
    END
  OF EVENT:Suspend
     !Save some stuff
  OF EVENT:Resume
     !Restore the stuff
  END
END
```

POST (post user-defined event)

POST(*event* [,*control*] [,*thread*] **)**

POST	Posts an event.
event	An integer constant, variable, expression, or EQUATE containing an event number. A value in the range 400h to 0FFFh is a User-defined event.
control	An integer constant, EQUATE, variable, or expression containing the field number of the control affected by the event. If omitted, the event is field-independent.
thread	An integer constant, EQUATE, variable, or expression containing the execution thread number whose ACCEPT loop is to process the event. If omitted, the event is posted to the current thread.

POST posts an event to the currently active ACCEPT loop of the specified *thread*. This may be User-defined events, or any other event. User-defined event numbers can be defined as any integer between 400h and 0FFFh. Any *event* posted with a *control* specified is a field-specific event, while those without are field-independent events.

Example:

```
Win1    WINDOW(´Tools´),AT(156,46,32,28),TOOLBOX
          BUTTON(´Date´),AT(0,0,,),USE(?Button1)
          BUTTON(´Time´),AT(0,14,,),USE(?Button2)
        END
  CODE
  OPEN(Win1)
  ACCEPT
    IF EVENT() = EVENT:User THEN BREAK.       !Detect user-defined event
    CASE ACCEPTED()
    OF ?Button1
      POST(EVENT:User,,UseToolsThread)
                                  !Post field-independent event to other thread
    OF ?Button2
      POST(EVENT:User)              !Post field-independent event to this thread
    END
  END
  CLOSE(Win1)
```

YIELD (allow event processing)

YIELD

YIELD temporarily gives control to Windows to allow other concurrently executing Windows applications to process events they need to handle (except those events that would post messages back to the program containing the YIELD statement ,or events that would change focus to the other application).

YIELD is used to ensure that long batch processing in a Clarion application does not completely "lock out" other applications from completing their tasks. This is known as "cooperative multi-tasking" and ensures that your Windows programs peacefully co-exist with any other Windows applications.

Within your Clarion application, YIELD only allows control to pass to EVENT:Timer events in other execution threads. This allows you to code a "background" procedure in its own execution thread using the TIMER attribute to perform some long batch processing without requiring the user to wait until the task is complete before continuing with other work in the application. This is an industry-standard Windows method of doing background processing within an application.

The example code below demonstrates both approaches to performing batch processing: making the user wait for the process to complete, and processing in the background. Only the WaitForProcess procedure requires the YIELD statement, because it takes full control of the program. Background processing using EVENT:Timer does not need a YIELD statement, since the ACCEPT loop automatically performs cooperative multi-tasking with other Windows applications.

Example:

```
StartProcess  PROCEDURE
Win  WINDOW(´Choose a Batch Process´),MDI
        BUTTON(´Full Control´),USE(?FullControl)
        BUTTON(´Background´),USE(?Background)
        BUTTON(´Close´),USE(?Close)
       END
  CODE
  OPEN(Win)
  ACCEPT
    CASE FIELD()
    OF ?FullControl
      DISABLE(FIRSTFIELD(),LASTFIELD())    !Disable all buttons
      WaitForProcess                     ! and call the batch process procedure
      ENABLE(FIRSTFIELD(),LASTFIELD())     !Enable buttons when batch is complete
    OF ?Background
       X# = START(BackgroundProcess)   !Start new execution thread for the process
    OF ?Close
      BREAK
    END
  END

WaitForProcess PROCEDURE           !Full control Batch process
  CODE
  SETCURSOR(CURSOR:Wait)           !Alert user to batch in progress
  SET(File)                       !Set up a batch process
  LOOP
    NEXT(File)
    IF ERRORCODE() THEN BREAK.
    !Perform some batch processing code
    YIELD                         !Yield to other applications and EVENT:Timer
  END
```

```
  SETCURSOR                         !Restore mmouse cursor

BackgroundProcess   PROCEDURE       !Background processing batch process
Win  WINDOW(´Batch Processing...´),TIMER(1),MDI
       BUTTON(´Cancel´),STD(STD:Close)
     END
  CODE
  OPEN(Win)
  SET(File)                         !Set up a batch process
  ACCEPT
    CASE EVENT()
    OF EVENT:CloseWindow
      BREAK
    OF EVENT:Timer                  !Process records whenever the timer allows it
      LOOP 3 TIMES
        NEXT(File)
        IF ERRORCODE() THEN BREAK.
        !Perform some batch processing code
  . . .
```

Multi-Threaded Applications

Multi-Threading and MDI

A multi-threaded application allows the user the ability to switch between multiple execution threads at runtime, as they choose. This makes the Windows Multiple Document Interface (MDI) approach to programming possible. A single Windows application may have a maximum of 64 execution threads concurrently available.

The first execution thread in any program is the main program code. This opens an APPLICATION structure as the MDI "parent" window, containing the main menu selections for the application.

The menu selections in the APPLICATION´s MENUBAR call the START function to begin each subsequent execution thread. The procedures called by START usually open an MDI "child" WINDOW, as a document window or dialog box. These windows allow the user to perform the tasks the application is designed to perform.

The last MDI "child" WINDOW opened (and not closed) in any execution thread is the "top" window in the thread and has input focus when that thread is executing. The user can switch between execution threads by using the mouse to CLICK on the top window of another execution thread. Thread switching can also be accomplished by selecting an open window from an MDI window list in the main menu, if the APPLICATION´s menu contains this standard Windows menu item.

Multi-Threading vs. Multi-Tasking

Multi-threading, as the term is used here, should not be confused with the ability to have the computer perform multiple tasks concurrently. Multiple execution threads do not necessarily imply multi-tasking, because only one thread normally executes at a time.

Windows allows cooperative, non-preemptive, multi-tasking between separately executing applications in any mode, and preemptive multi-tasking in 386 enhanced mode. Preemptive multi-tasking is based on "time slicing" between the applications and the amount of time each simultaneously executing application receives is governed by the end user´s Windows configuration. See your Windows documentation for an explanation of Windows´ multi-tasking settings.

A form of cooperative, non-preemptive, multi-threading (similar to inter-application multi-tasking) can be accomplished within a single Clarion application by using the TIMER attribute. This is not based on "time slicing" between execution threads. Instead, each execution thread gains control and does not relinquish it until it executes an ASK or ACCEPT statement.

When the top window of an execution thread has the TIMER attribute, a timer event (EVENT:Timer) is periodically generated to cycle its ACCEPT loop to process the event. This occurs even if the thread does not currently have input focus. Therefore, if you want to perform this type of multi-threading, you must ensure that any lengthy execution code includes YIELD statements that occasionally execute to allow the timer events in other threads to generate and execute.

START (return new execution thread)

START(*procedure* [,*stack*] **)**

START	Begins a new execution thread.
procedure	The label of the first PROCEDURE to call on the new execution thread. The *procedure* must have been prototyped not to receive any parameters.
stack	An integer constant or variable containing the size of the stack to allocate to the new execution thread. If omitted, the default stack is 10,000 bytes.

The **START** function begins a new execution thread, calling the *procedure* and returning the number assigned to the new thread. The returned thread number number is used by procedures and functions whose action may be performed on any execution thread, such as SETTARGET. The maximum number of simultaneously available execution threads in a single application is 64.

The first execution thread in any program is the main program code, which is always numbered one (1). Therefore, the lowest value START can return is two (2), when the first START function is executed in a program. START may return zero (0), which indicates failure to open the thread. This can occur by attempting to START a 65th thread, or by running out of memory, or by starting a thread when the system is modal.

Return Data Type: LONG

Example:

```
MainWin APPLICATION(´My Application´),SYSTEM,MAX,ICON(´MyIcon.ICO´),STATUS |
   ,HVSCROLL,RESIZE
        MENUBAR
          MENU(´&File´),USE(?FileMenu)
            ITEM(´Selection &1...´),USE(?MenuSelection1)
            ITEM(´Selection &2...´),USE(?MenuSelection2)
          END
        END
       END

SaveThread1   LONG  !Declare thread number save variable
SaveThread2   LONG  !Declare thread number save variable
  CODE
  OPEN(MainWin)                    !Open the APPLICATION
  ACCEPT                          !Handle Global events
    CASE ACCEPTED()
    OF ?MenuSelection1
      SaveThread1 = START(NewProc1)     !Start a new thread
    OF ?MenuSelection2
      SaveThread2 = START(NewProc2)     !Start a new thread
    OF ?Exit
      RETURN
    END
```

THREAD (return current execution thread)

THREAD( )

The **THREAD** function returns the currently executing thread number. The returned thread number number can be used by procedures and functions whose action may be performed on any execution thread, such as SETTARGET.

The maximum number of simultaneously available execution threads in a single application is 64. The first execution thread in any program is the main program code, which is always thread number one (1). Therefore, THREAD always returns a value in the range of one (1) to sixty-four (64).

Return Data Type: LONG

Example:

```
MainWin APPLICATION(´My Application´),SYSTEM,MAX,ICON(´MyIcon.ICO´),STATUS |
   ,HVSCROLL,RESIZE
        MENUBAR
          MENU(´&File´),USE(?FileMenu)
            ITEM(´Selection &1...´),USE(?MenuSelection1)
            ITEM(´Selection &2...´),USE(?MenuSelection2)
          END
        END
       END

SaveThread    LONG  !Declare thread number save variable
SaveThread1   LONG  !Declare thread number save variable
SaveThread2   LONG  !Declare thread number save variable
  CODE
  SaveThread = THREAD()             !Save thread number
  OPEN(MainWin)                    !Open the APPLICATION
  ACCEPT                          !Handle Global events
    CASE ACCEPTED()
    OF ?MenuSelection1
      SaveThread1 = START(NewProc1)   !Start a new thread
    OF ?MenuSelection2
      SaveThread2 = START(NewProc2)    !Start a new thread
    OF ?Exit
      RETURN
    END
  END
```

Window Procedures

- CHANGE (change control field value)
- CLOSE (close window)
- CREATE (create new control)
- DISABLE (dim a control)
- DISPLAY (write USE variables to screen)
- ENABLE (re-activate dimmed control)
- ERASE (clear screen control and USE variables)
- GETFONT (get font information)
- GETPOSITION (get control position)
- HELP (help window access)
- HIDE (blank a control)
- OPEN (open window for processing)
- SELECT (select next control to process)
- SET3DLOOK (set 3D window look)
- SETCURSOR (set temporary mouse cursor)
- SETFONT (specify font)
- SETPOSITION (specify new control position)
- SETTARGET (set current window or report)
- UNHIDE (show hidden control)
- UPDATE (write from screen to USE variables)

CHANGE (change control field value)

CHANGE(*control,value***)**

CHANGE	Changes the *value* displayed in a *control* in an APPLICATION or WINDOW structure.
control	Field number or field equate label of a window control field.
value	A constant or variable containing the *control´s* new value.

The **CHANGE** statement changes the *value* displayed in a *control* in an APPLICATION or WINDOW structure. CHANGE updates the *control´s* USE variable with the *value*, and then displays that new *value* in the control field.

Example:

```
Screen WINDOW,PRE(Scr)
         ENTRY(@N3),USE(Ctl:Code)
         ENTRY(@S30),USE(Ctl:Name)
         BUTTON(´OK´),USE(?OkButton),KEY(EnterKey)
         BUTTON(´Cancel´),USE(?CanxButton),KEY(EscKey)
       END
  CODE
  OPEN(Screen)
  ACCEPT
    CASE EVENT()
    OF EVENT:Selected
      CASE SELECTED()
      OF ?Ctl:Code
        CHANGE(?Ctl:Code,4)        !Change Ctl:Code to 4 and display it
      OF ?Ctl:Name
        CHANGE(?Ctl:Name,´ABC Company´)
                              !Change Ctl:Name to ABC Company and display
      END
    OF EVENT:Accepted
      CASE ACCEPTED()
      OF ?OkButton
        BREAK
      OF ?CanxButton
        CLEAR(Ctl:Record)
        BREAK
      END
    END
```

CLOSE (close window)

CLOSE(*label***)**

CLOSE Closes the active APPLICATION or WINDOW structure.

label The label of an APPLICATION or WINDOW structure.

CLOSE terminates processing on the active APPLICATION or WINDOW structure. Memory used by the active window is released when it is closed and the underlying screen is automatically re-drawn.

When a window is closed, if it is not the top-most window on its execution thread, all windows opened subsequent to the window being closed are automatically closed first. This occurs in the reverse order from which they were opened.

An APPLICATION or WINDOW that is declared local to (within) a PROCEDURE or FUNCTION is automatically closed when the program RETURNs from the procedure.

Example:

```
CLOSE(MenuScr)          !Close the  menu screen
CLOSE(CustEntry)        !Close customer data entry screen
```

CREATE (create new control)

CREATE(*control* ,*type* [,*parent*] **)**

CREATE	Creates a new control.
control	A field number or field equate label for the control to create.
type	An integer constant, expression, EQUATE, or variable that specifies the type of control to create.
parent	A field number or field equate label. This specifies an OPTION, GROUP, or MENU to contain the new *control*.

CREATE dynamically creates a new control in the currently active APPLICATION or WINDOW. When first created, the new *control* is initially hidden, so its properties can be set using the runtime property assignment syntax, SETPOSITION, and SETFONT. It appears on screen only by issuing an UNHIDE statement for the *control*. To place the new control on the toolbar, add CREATE:TOOLBAR to the equate for the new control´s *type*.

EQUATE statements for the *type* parameter are contained in the EQUATES.CLW file. The following list is a comprehensive sample of these (see EQUATES.CLW for the complete list):

CREATE:sstring	STRING(picture),USE(variable)
CREATE:string	STRING(constant)
CREATE:image	IMAGE()
CREATE:region	REGION()
CREATE:line	LINE()
CREATE:box	BOX()
CREATE:ellipse	ELLIPSE()
CREATE:entry	ENTRY()
CREATE:button	BUTTON()
CREATE:prompt	PROMPT()
CREATE:option	OPTION()
CREATE:radio	RADIO()
CREATE:check	CHECK()
CREATE:group	GROUP()
CREATE:list	LIST()
CREATE:combo	COMBO()
CREATE:spin	SPIN()
CREATE:text	TEXT()
CREATE:custom	CUSTOM()
CREATE:droplist	LIST(),DROP()
CREATE:dropcombo	COMBO(),DROP()
CREATE:menu	MENU()
CREATE:item	ITEM()

Example:

```
Screen WINDOW,PRE(Scr)
         ENTRY(@N3),USE(Ctl:Code)
         ENTRY(@S30),USE(Ctl:Name)
         BUTTON(´OK´),USE(?OkButton),KEY(EnterKey)
         BUTTON(´Cancel´),USE(?CanxButton),KEY(EscKey)
       END

X      SHORT
Y      SHORT
```

```
Width    SHORT
Height SHORT

Code4Entry  STRING(10)
?Code4Entry EQUATE(100)

  CODE
  OPEN(Screen)
  ACCEPT
    CASE ACCEPTED()
    OF ?Ctl:Code
      IF Ctl:Code = 4
        CREATE(?Code4Entry,CREATE:entry)          !Create the control
        ?Code4Entry{PROP:use} = ´Code4Entry´       !Set USE variable
        ?Code4Entry{PROP:text} = ´@s10´           !Set entry picture
        GETPOSITION(?Ctl:Code,X,Y,Width,Height)
        ?Code4Entry{PROP:at,1} = X + Width + 40    !Set x position
        ?Code4Entry{PROP:at,2} = Y               !Set y position
        UNHIDE(?Code4Entry)                      !Display the new control
      END
    OF ?OkButton
      BREAK
    OF ?CanxButton
      CLEAR(Ctl:Record)
      BREAK
    END
  END
  CLOSE(Screen)
  RETURN
```

DESTROY (remove a control)

DESTROY(*first control* [,*last control*] **)**

DESTROY	Removes window controls.
first control	Field number or field equate label of a control, or the first control in a range of controls.
last control	Field number or field equate label of the last control in a range of controls.

The **DESTROY** statement removes a control, or range of controls, from an APPLICATION or WINDOW structure. When removed, the controls resources are returned to the operating system.

DESTROYing a GROUP, OPTION, MENU, TAB, or SHEET control also destroys all controls contained within it.

Example:

```
Screen WINDOW,PRE(Scr)
         ENTRY(@N3),USE(Ctl:Code)
         ENTRY(@S30),USE(Ctl:Name)
         BUTTON('OK'),USE(?OkButton),KEY(EnterKey)
         BUTTON('Cancel'),USE(?CanxButton),KEY(EscKey)
       END
  CODE
  OPEN(Screen)
  DESTROY(?Ctl:Code)              !Remove a control
  DESTROY(?Ctl:Code,?Ctl:Name)    !Remove range of controls
  DESTROY(2)                      !Remove the second control
```

See Also:

CREATE

DISABLE (dim a control)

DISABLE(*first control* [,*last control*] **)**

DISABLE Dims controls on the window.

first control Field number or field equate label of a control, or the first control in a range of controls.

last control Field number or field equate label of the last control in a range of controls.

The **DISABLE** statement disables a control or a range of controls on an APPLICATION or WINDOW structure. When disabled, the control appears dimmed on screen.

Example:

```
Screen WINDOW,PRE(Scr)
         ENTRY(@N3),USE(Ctl:Code)
         ENTRY(@S30),USE(Ctl:Name)
         BUTTON(´OK´),USE(?OkButton),KEY(EnterKey)
         BUTTON(´Cancel´),USE(?CanxButton),KEY(EscKey)
       END
  CODE
  OPEN(Screen)
  DISABLE(?Ctl:Code)             !Disable a control
  DISABLE(?Ctl:Code,?Ctl:Name)    !Disable range of controls
  DISABLE(2)                     !Disable the second control
```

See Also:

ENABLE

HIDE

UNHIDE

DISPLAY (write USE variables to screen)

DISPLAY([*first control*] [,*last control*] **)**

DISPLAY	Writes the contents of USE variables to their associated controls.
first control	Field number or field equate label of a control, or the first control in a range of controls.
last control	Field number or field equate label of the last control in a range of controls.

DISPLAY writes the contents of the USE variables to their associated controls on the active window. DISPLAY with no parameters writes the USE variables for all controls on the screen. Using *first control* alone, as the parameter of DISPLAY, writes a specific USE variable to the screen. Both *first control* and *last control* parameters are used to display the USE variables for an inclusive range of controls on the screen.

The current contents of the USE variables of all controls are automatically displayed on screen each time the ACCEPT loop cycles. This eliminates the need to explicitly issue a DISPLAY statement to update the video display. Of course, if your application performs some operation that takes a long time and you want to indicate to the user that something is happening without cycling back to the top of the ACCEPT loop, you should DISPLAY some variable that you have updated.

Example:

```
DISPLAY                 !Display all controls on the screen
DISPLAY(2)              !Display control number 2
DISPLAY(3,7)            !Display controls 3 through 7
DISPLAY(?MenuControl)     !Display the menu control
DISPLAY(?TextBlock,?Ok)  !Display range of controls
```

See Also:

Field Equate Labels

UPDATE

ERASE

ENABLE (re-activate dimmed control)

ENABLE(*first control* [,*last control*] **)**

ENABLE	Reactivates disabled controls.
first control	Field number or field equate label of a control, or the first control in a range of controls.
last control	Field number or field equate label of the last control in a range of controls.

The **ENABLE** statement reactivates a control, or range of controls, that were dimmed by the DISABLE statement, or were declared with the DISABLE attribute. Once reactivated, the control is again available to the operator for selection.

Example:

```
CODE
OPEN(Screen)
DISABLE(?Control2)             !Control2 is deactivated
IF Ctl:Password = ´Supervisor´
  ENABLE(?Control2)             !Re-activate Control2
END
```

See Also:

DISABLE

HIDE

UNHIDE

ERASE (clear screen control and USE variables)

ERASE([*first control*] [,*last control*] **)**

ERASE	Blanks controls and clears their USE variables.
first control	Field number or field equate label of a control, or the first control in a range of controls.
last control	Field number or field equate label of the last control in a range of controls.

The **ERASE** statement erases the data from controls in the window and clears their corresponding USE variables. ERASE with no parameters erases all controls in the window. Using *first control* alone, as the parameter of ERASE, clears a specific USE variable and its associated control. Both *first control* and *last control* parameters are used to clear the USE variables and associated controls for an inclusive range of controls in the window.

Example:

```
ERASE(?)              !Erase the currently selected control
ERASE                   !Erase all controls on the screen
ERASE(3,7)              !Erase controls 3 through 7
ERASE(?Name,?Zip)         !Erase controls from name through zip
ERASE(?City,?City+2)      !Erase City and 2 controls following City
```

See Also:

Field equate Labels

GETFONT (get font information)

GETFONT(*control* ,*typeface* , *size* ,*color* ,*style***)**

GETFONT	Gets display font information.
control	A field number or field equate label for the control from which to get the information. If *control* is zero (0), it specifies the WINDOW.
typeface	A string variable to receive the name of the font.
size	An integer variable to receive the size (in points) of the font.
color	A LONG integer variable to receive the red, green, and blue values for the color of the font in the low-order three bytes. If the value is negative, the *color* represents a system color.
style	An integer variable to receive the strike weight and style of the font.

GETFONT gets the display font information for the *control*. If the *control* parameter is zero (0), GETFONT gets the default display font for the window.

Example:

```
TypeFace  STRING(20)
Size     BYTE
Color      LONG
Style      LONG

  CODE
  OPEN(Screen)
  GETFONT(0,TypeFace,Size,Color,Style)         !Get font info for the window
```

See Also:

SETFONT

GETPOSITION (get control position)

GETPOSITION(*control* ,*x* , *y* ,*width* ,*height***)**

GETPOSITION Gets the position and size of an APPLICATION, WINDOW, or control.

control A field number or field equate label for the control from which to get the information. If *control* is zero (0), it specifies the window.

x An integer variable to receive the horizontal position of the top left corner.

y An integer variable to receive the vertical position of the top left corner.

width An integer variable to receive the width.

height An integer variable to receive the height.

GETPOSITION gets the position and size of an APPLICATION, WINDOW, or control. The position and size values are dependent upon the presence or absence of the SCROLL attribute on the *control*. If SCROLL is present, the values are relative to the virtual window. If SCROLL is not present, the values are relative to the top left corner of the currently visible portion of the window. This means the values returned always match those specified in the AT attribute or most recent SETPOSITION.

The values in the *x*, *y*, *width*, and *height* parameters are measured in dialog units. Dialog units are defined as one-quarter the average character width by one-eighth the average character height. The size of a dialog unit is dependent upon the size of the default font for the window. This measurement is based on the font specified in the FONT attribute of the window, or the system default font specified by Windows.

Example:

```
Screen WINDOW,PRE(Scr)
         ENTRY(@N3),USE(Ctl:Code)
         ENTRY(@S30),USE(Ctl:Name)
         BUTTON(´OK´),USE(?OkButton),KEY(EnterKey)
         BUTTON(´Cancel´),USE(?CanxButton),KEY(EscKey)
       END

X      SHORT
Y      SHORT
Width    SHORT
Height SHORT
  CODE
  OPEN(Screen)
  GETPOSITION(?Ctl:Code,X,Y,Width,Height)
```

See Also:

SETPOSITION

HELP (help window access)

HELP([*helpfile*] [,*window-id*]**)**

HELP	Opens a help file and activates a help window.
helpfile	A string constant or the label of a STRING variable that has the DOS directory file specification for the help file. If the file specification does not contain a complete path and filename, the help file is assumed to be in the current directory. If the file extension is omitted, ".HLP" is assumed. If the *helpfile* parameter is omitted, a comma is required to hold its position.
window-id	A string constant or the label of a STRING variable that contains the key used to access the help system. This may be either a help keyword or a "context string."

The **HELP** statement opens a designated *helpfile*, and activates the window named by the *window-id*. While an ASK or ACCEPT is controlling program execution, the active help window is displayed when the operator presses F1 (the "Help" key).

If the *window-id* parameter is omitted, the *helpfile* is nominated but not opened. If the *helpfile* parameter is omitted, the current help file is opened, and the window identified by *window-id* is activated. If both parameters are omitted, the current *helpfile* is opened at the current topic.

The *window-ID* may contain a Help keyword. This is a keyword that is displayed in the Help Search dialog. When the user presses F1, if only one topic in the help file specifies this keyword, the help file is opened at that topic; if more than one topic specifies the keyword, the search dialog is opened for the user.

A "context string" is identified by a leading tilde (~) in the *window-ID*, followed by a unique identifier associated with exactly one help topic. If the tilde is missing, the *window-ID* is assumed to be a help keyword. When the user presses F1, the help file is opened at the specific topic associated with that "context string."

Help windows are also activated by the HLP attribute of an APPLICATION, WINDOW, or control.

Example:

```
HELP(´C:\HLPDIR\LEDGER.HLP´)  !Open the gen ledger help file
HELP(,´~CustUpd´)             !Activate customer update help window
HELP                        !Display the help window
```

See Also:

ASK

ACCEPT

HLP

HIDE (blank a control)

HIDE(*first control* [,*last control*] **)**

HIDE	Hides window controls.
first control	Field number or field equate label of a control, or the first control in a range of controls.
last control	Field number or field equate label of the last control in a range of controls.

The **HIDE** statement hides a control, or range of controls, on an APPLICATION or WINDOW structure. When hidden, the control does not appear on screen.

Example:

```
Screen WINDOW,PRE(Scr)
         ENTRY(@N3),USE(Ctl:Code)
         ENTRY(@S30),USE(Ctl:Name)
         BUTTON(´OK´),USE(?OkButton),KEY(EnterKey)
         BUTTON(´Cancel´),USE(?CanxButton),KEY(EscKey)
       END
  CODE
  OPEN(Screen)
  HIDE(?Ctl:Code)               !Hide a control
  HIDE(?Ctl:Code,?Ctl:Name)      !Hide range of controls
  HIDE(2)                      !Hide the second control
```

See Also:

UNHIDE

ENABLE

DISABLE

OPEN (open window for processing)

OPEN(*label***)**

OPEN Opens a window.

label The label of an APPLICATION or WINDOW structure.

OPEN activates an APPLICATION or WINDOW for processing. However, nothing is displayed until a DISPLAY statement or the ACCEPT loop is encountered. This allows an opportunity to execute pre-display code to customize the display.

Example:

```
OPEN(MenuScr)        !Open the menu screen
OPEN(CustEntry)        !Open customer data entry screen
```

SELECT (select next control to process)

SELECT([*control*] [,*position*] [,*endposition*] **)**

SELECT	Sets the next control to receive input focus.
control	A field number or field equate label of the next control to process. If omitted, the SELECT statement initiates AcceptAll mode.
position	Specifies a position within the *control* to place the cursor. For an ENTRY or TEXT, SPIN, or COMBO control this is a character position, or a beginning character position for a marked block. For an OPTION structure, this is the selection number within the OPTION. For a LIST control, this is the QUEUE entry number.
endposition	Specifies an ending character position within an ENTRY, TEXT, SPIN, or COMBO *control*. The character position specified by *position* and *endposition* are marked as a block, available for cut and paste operations.

SELECT overrides the normal TAB key sequence control selection order of an APPLICATION or WINDOW. Its action affects the next ACCEPT statement that executes. The *control* parameter determines which control the ACCEPT loop will process next. If *control* specifies a control which cannot rceive focus because a DISABLE or HIDE statment has been issued, focus goes to the next control following it in the window´s source code that can receive focus.

SELECT with *position* and *endposition* parameters specifies a marked block in the *control* which is available for cut and paste operations.

SELECT with no parameters initiates AcceptAll mode. This is a field edit mode in which each control in the window is processed in TAB key sequence by generating EVENT:Accepted for each. This allows data entry validation code to execute for all controls, including those that the user has not touched. AcceptAll mode terminates when any of the following conditions is met:

A SELECT(?) statement selects the same control for the user to edit. This code usually indicates the value it contains is invalid and the user must re-enter data.

The Window{PROP:AcceptAll} property is set to zero (0). This property contains one (1) when AcceptAll mode is active. Assigning values to this property can also be used to initiate and terminate AcceptAll mode.

A control with the REQ attribute is blank or zero. AcceptAll mode terminates with the control highlighted for user entry, without processing any more fields in the TAB key sequence.

When all controls have been processed, EVENT:Completed is posted to the window.

Example:

```
Screen WINDOW,PRE(Scr)
         ENTRY(@N3),USE(Ctl:Code)
         ENTRY(@S30),USE(Ctl:Name)
         BUTTON(´OK´),USE(?OkButton),KEY(EnterKey)
         BUTTON(´Cancel´),USE(?CanxButton),KEY(EscKey)
       END
  CODE
  OPEN(Screen)
  SELECT(?Ctl:Code)                      !Start with Ctl:Code
  ACCEPT
    CASE ACCEPTED()
```

```
  OF ?Ctl:Code
    IF Ctl:Code > 150                  !If data entered is invalid
      BEEP                           ! alert the user and
      SELECT(?)                       ! make them re-enter the data
    END
  OF ?Ctl:Name
    SELECT(?Ctl:Name,1,5)               !Mark first five characters as a block
  OF ?OkButton
    SELECT                           !Initiate AcceptAll mode
  END
  IF EVENT() = EVENT:Completed THEN BREAK.
                                     !AcceptAll mode terminated
END
```

See Also:

ACCEPT

SET3DLOOK (set 3D window look)

SET3DLOOK([*switch*] **)**

SET3DLOOK Toggles three-dimensional look and feel.

switch An integer constant switching the 3D look off (0) and on (1). If omitted, the default is one (1).

The **SET3DLOOK** procedure sets up the program to display a three-dimensional look and feel. The default program setting is 3D enabled. On a WINDOW, the GRAY attribute causes the controls to display with a three-dimensional appearance. Controls in the TOOLBAR are always displayed with the three-dimensional look, unless disabled by SET3DLOOK. When three-dimensional look is disabled by SET3DLOOK, the GRAY attribute has no effect.

SET3DLOOK(0) turns off the three-dimensional look and feel. SET3DLOOK(1) turns on the three-dimensional look and feel. Values other than zero or one are reserved for future use.

Example:

```
MainWin APPLICATION(´My Application´),SYSTEM,MAX,ICON(´MyIcon.ICO´),STATUS |
   ,HVSCROLL,RESIZE
        MENUBAR
          MENU(´&File´),USE(?FileMenu)
            ITEM(´&Open...´),USE(?OpenFile)
            ITEM(´&Close´),USE(?CloseFile),DISABLE
            ITEM(´Turn off 3D Look´),USE(?Toggle3D),CHECK
            ITEM(´E&xit´),USE(?MainExit)
          END
        END
      END
  CODE
  OPEN(MainWin)
  ACCEPT
    CASE ACCEPTED()
    OF ?Toggle3D
      IF MainWin$?Toggle3D{PROP:text} = ´Turn off 3D Look´                !If on
        SET3DLOOK(0)                                          !Turn off
        MainWin$?Toggle3D{PROP:text} = ´Turn on 3D Look´                        ! and
change text
      ELSE                                                !Else
        SET3DLOOK(1)                                      !Turn on
        MainWin$?Toggle3D{PROP:text} = ´Turn off 3D Look´                       ! and
change text
      END
    OF ?OpenFile
      START(OpenFileProc)
    OF ?MainExit
      BREAK
    END
  END
  CLOSE(MainWin)
```

SETCURSOR (set temporary mouse cursor)

SETCURSOR([*cursor*] **)**

SETCURSOR Specifies a temporary mouse cursor to display.

cursor An EQUATE naming a Windows-standard mouse cursor. If omitted, turns off the temporary cursor.

The **SETCURSOR** statement specifies a temporary mouse *cursor* to display until a SETCURSOR statement without a *cursor* parameter turns it off. This cursor overrides all CURSOR attributes. When SETCURSOR without a *cursor* parameter is encountered, all CURSOR attributes once again take effect. SETCURSOR is generally used to display the hourglass while your program is doing some "behind the scenes" work that the user should not break into.

EQUATE statements for the Windows-standard mouse cursors are contained in the EQUATES.CLW file. The following list is a representative sample of these (see EQUATES.CLW for the complete list):

CURSOR:None	No mouse cursor
CURSOR:Arrow	Normal windows arrow cursor
CURSOR:IBeam	Capital "I" like a steel I-beam
CURSOR:Wait	Hourglass
CURSOR:Cross	Large plus sign
CURSOR:UpArrow	Vertical arrow
CURSOR:Size	Four-headed arrow
CURSOR:Icon	Box within a box
CURSOR:SizeNWSE	Double-headed arrow slanting left
CURSOR:SizeNESW	Double-headed arrow slanting right
CURSOR:SizeWE	Double-headed horizontal arrow
CURSOR:SizeNS	Double-headed vertical arrow

Example:

```
MainWin APPLICATION(´My Application´),SYSTEM,MAX,ICON(´MyIcon.ICO´),STATUS |
   ,HVSCROLL,RESIZE
        MENUBAR
          ITEM(´Batch Update´),USE(?Batch)
        END
      END
  CODE
  OPEN(MainWin)
  ACCEPT
    CASE ACCEPTED()
    OF ?Batch
      SETCURSOR(CURSOR:Wait)   !Turn on hourglass mouse cursor
      BatchUpdate             ! and call the batch update procedure
    END
  END
```

SETFONT (specify font)

SETFONT(*control* ,*typeface* , *size* ,*color* ,*style***)**

SETFONT	Dynamically sets the display font for a control.
control	A field number or field equate label for the control to affect. If *control* is zero (0), it specifies the WINDOW.
typeface	A string constant or variable containing the name of the font. If omitted, the system font is used.
size	An integer constant or variable containing the size (in points) of the font. If omitted, the system default font size is used.
color	A LONG integer constant or variable containing the red, green, and blue values for the color of the font in the low-order three bytes, or an EQUATE for a standard Windows color value. If omitted, black is used.
style	An integer constant, constant expression, EQUATE, or variable specifying the strike weight and style of the font. If omitted, the weight is normal.

SETFONT dynamically specifies the display font for the *control*, overriding any FONT attribute previously specified. If the *control* parameter is zero (0), SETFONT specifies the default display font for the window.

SETFONT allows you to specify all parameters of a font change at once, instead of one at a time as runtime property assignment allows. This has the advantage of implementing all changes at once, whereas runtime property assignment would change each individually, displaying each separate change as it occurs.

The *typeface* may name any font registered in the Windows system. The EQUATES.CLW file contains EQUATE values for standard *style* values. A *style* on the range zero (0) to one thousand (1000) specifies the strike weight of the font. You may also add values that indicate italic, underline, or strikeout text. The following EQUATES are in EQUATES.CLW:

FONT:thin	EQUATE (100)
FONT:regular	EQUATE (400)
FONT:bold	EQUATE (700)
FONT:italic	EQUATE (01000H)
FONT:underline	EQUATE (02000H)
FONT:strikeout	EQUATE (04000H)

Example:

```
SETFONT(0,´Arial´,14,,FONT:thin+FONT:Italic)      !14 pt. Arial black thin italic
```

See Also:

GETFONT

SETPOSITION (specify new control position)

SETPOSITION(*control* ,*x* , *y* ,*width* ,*height***)**

SETPOSITION Dynamically specifies the position and size of an APPLICATION, WINDOW, or control.

control A field number or field equate label for the control to affect. If *control* is zero (0), it specifies the window.

x An integer constant, expression, or variable that specifies the horizontal position of the top left corner. If omitted, the *x* position is not changed.

y An integer constant, expression, or variable that specifies the vertical position of the top left corner. If omitted, the *y* position is not changed.

width An integer constant, expression, or variable that specifies the width. If omitted, the *width* is not changed.

height An integer constant, expression, or variable that specifies the height. If omitted, the *height* is not changed.

SETPOSITION dynamically specifies the position and size of an APPLICATION, WINDOW, or control. If any parameter is omitted, the value is not changed.

The values contained in the *x*, *y*, *width*, and *height* parameters are measured in dialog units. Dialog units are defined as one-quarter the average character width by one-eighth the average character height. The size of a dialog unit is dependent upon the size of the default font for the window. This measurement is based on the font specified in the FONT attribute of the window, or the system default font specified by Windows.

Using SETPOSITION produces a "smoother" control appearance change than using property expressions to change the AT attribute´s parameter values. This is because SETPOSITION changes all four parameters at once. Property expressions must change one parameter at a time. Since each individual parameter change would be immediately visible on screen, this would cause the control to appear to "jump."

Example:

```
CREATE(?Code4Entry,CREATE:entry,?Ctl:Code)  !Create a control
?Code4Entry{PROP:use} = ´Code4Entry´        !Set USE variable
?Code4Entry{PROP:text} = ´@s10´           !Set entry picture
GETPOSITION(?Ctl:Code,X,Y,Width,Height)      !Get Ctl:Code position
SETPOSITION(?Code4Entry,X+Width+40,Y)       !Set x 40 past Ctl:Code
UNHIDE(?Code4Entry)                       !Display the new control
```

See Also:

GETPOSITION

SETTARGET (set current window or report)

SETTARGET([*target*] [,*thread*] **)**

SETTARGET Sets the current window (or report) for drawing graphics and other window-interaction statements.

target The label of an APPLICATION, WINDOW or REPORT structure. If omitted, the last window opened and not yet closed in the specified *thread* is used.

thread The number of the execution thread in which the *target* structure is contained in the topmost procedure or function. If omitted, the current execution thread is used.

The **SETTARGET** procedure makes the *target* the structure which is current for drawing with the graphics primitives functions. SETTARGET also sets the *target* for runtime property assignment, and the CREATE, SETPOSITION, GETPOSITION, SETFONT, GETFONT, DISABLE, HIDE, CONTENTS, DISPLAY, ERASE, and UPDATE statements. Using these statements with SETTARGET allows you to manipulate the window display in the topmost window of any execution thread.

This *target* will receive any graphics drawn with the graphics procedures and functions described in the Graphics Commands section. This allows you to draw graphics to the topmost window, or report, in any execution thread.

SETTARGET sets the "built-in" variable, TARGET (also set when a window is opened), which may be used in any statement which requires the label of the current window or report. A REPORT data structure is never the default *target*. Therefore, SETTARGET must be used before using the graphics primitives functions to draw graphics on a REPORT.

SETTARGET does not change procedures, and it does not change which ACCEPT loop receives the events generated by Windows. SETTARGET without any parameters resets to the procedure and execution thread with the currently active ACCEPT loop.

Example:

```
Report REPORT
        !Report structure controls
      END
  CODE
  OPEN(Report)
  SETTARGET(Report)              !Make the report the current target
  TARGET{PROP:Landscape} = 1      ! and turn on landscape mode
```

UNHIDE (show hidden control)

UNHIDE(*first control* [,*last control*] **)**

UNHIDE	Displays previously hidden controls.
first control	Field number or field equate label of a control, or the first control in a range of controls.
last control	Field number or field equate label of the last control in a range of controls.

The **UNHIDE** statement reactivates a control or range of controls, that were hidden by the HIDE statement. Once un-hidden, the control is again visible on screen.

Example:

```
CODE
OPEN(Screen)
HIDE(?Control2)                 !Control2 is hidden
IF Ctl:Password = ´Supervisor´
  UNHIDE(?Control2)             !Unhide Control2
END
```

See Also:

HIDE

ENABLE

DISABLE

UPDATE (write from screen to USE variables)

UPDATE([*first control*] [,*last control*] **)**

UPDATE	Writes the contents of a control to its USE variable.
first control	Field number or field equate label of a control, or the first control in a range of controls.
last control	Field number or field equate label of the last control in a range of controls.

UPDATE writes the contents of a screen control to its USE variable. This takes the value displayed on screen and places it in the variable specified by the control´s USE attribute.

USE variables are updated automatically by ACCEPT as each control is accepted. However, certain events (such as an ALERTed key press) do not automatically update USE variables. This is the purpose of the UPDATE statement.

UPDATE	Updates all controls on the screen.
UPDATE(*first control*)	Updates a specific USE variable from its associated screen control.
UPDATE(*first control*,*last control*)	Updates the USE variables of an inclusive range of screen controls.

Example:

```
UPDATE(?)              !Update the currently selected control
UPDATE                !Update all controls on the screen
UPDATE(?Address)       !Update the address control
UPDATE(3,7)            !Update controls 3 through 7
UPDATE(?Name,?Zip)     !Update controls from name through zip
UPDATE(?City,?City+2)    !Update city and 2 controls following
```

See Also:

Field equate Labels

Window Functions

- ACCEPTED (return control just completed)
- CHOICE (return relative item position)
- CONTENTS (return contents of USE variable)
- FIELD (return control with focus)
- FIRSTFIELD (return first window control)
- FOCUS (return control with focus)
- INCOMPLETE (return empty REQ control)
- LASTFIELD (return last window control)
- MESSAGE (return message box response)
- MOUSEX (return mouse horizontal position)
- MOUSEY (return mouse vertical position)
- SELECTED (return control that has received focus)

ACCEPTED (return control just completed)

ACCEPTED( )

The **ACCEPTED** function returns the field number of the control on which an EVENT:Accepted event occurred. ACCEPTED returns zero (0) for all other events.

Positive field numbers are assigned by the compiler to all WINDOW controls, in the order their declarations occur in the WINDOW structure. Negative field numbers are assigned to all APPLICATION controls. In executable code statements, field numbers are usually represented by field equate labels--the label of the USE variable preceded by a question mark (?FieldName).

Return Data Type: SHORT

Example:

```
CASE ACCEPTED()         !Process post-edit code
OF ?Cus:Company
  !Edit field value
OF ?Cus:CustType
  !Edit field value
END
```

See Also:

ACCEPT

EVENT

CHOICE (return relative item position)

CHOICE([*control*] **)**

CHOICE Retuns a user selection number.

control A field equate label of a LIST, COMBO, or OPTION control.

The **CHOICE** function returns the sequence number of a selected item in an OPTION structure, LIST box, or COMBO control. With no parameter, CHOICE returns the sequence number of the selected item in the last control (LIST, OPTION, or COMBO) that generated a Field-specific event to cycle the ACCEPT loop. CHOICE(*control*) returns the current selection number of any LIST, OPTION, or COMBO in the currently active window.

CHOICE returns the sequence number of the selected RADIO control within an OPTION structure. The sequence number is determined by relative position within the OPTION. The first control listed in the OPTION structure´s code is relative position 1, the second is 2, etc.

CHOICE returns the memory QUEUE entry number of the selected item when a LIST or COMBO box is completed.

Return Data Type: LONG

Example:

```
CODE
ACCEPT
  EXECUTE CHOICE()   !Perform menu option
    AddRec          ! procedure to add record
    PutRec          ! procedure to change record
    DelRec          ! procedure to delete record
    RETURN          ! return to caller
  END
END
```

CONTENTS (return contents of USE variable)

CONTENTS(*control***)**

CONTENTS Returns the value in the USE variable of a control.

control A field number or field equate label.

The **CONTENTS** function returns a string containing the value in the USE variable of an ENTRY, OPTION RADIO, or TEXT control.

A USE variable may be longer than its associated control field display picture OR may contain fewer characters than its total capacity. The CONTENTS function always returns the full length of the USE variable.

Return Data Type: STRING

Example:

```
IF CONTENTS(?LastName) = ´´ AND CONTENTS(?FirstName) = ´´
                                    !If first and last name are blank
  MessageField = ´Must Enter a First or Last Name´
                                    ! display error message
END
```

FIELD (return control with focus)

FIELD( )

The **FIELD** function returns the field number of the control which has focus at the time of any field-specific event. This includes both the EVENT:Selected and EVENT:Accepted events. FIELD returns zero (0) for field-independent events.

Positive field numbers are assigned by the compiler to all WINDOW controls, in the order their declarations occur in the WINDOW structure. Negative field numbers are assigned to all APPLICATION controls. In executable code statements, field numbers are usually represented by field equate labels--the label of the USE variable preceded by a question mark (?FieldName).

Return Data Type: LONG

Example:

```
Screen WINDOW
         ENTRY(@N4),USE(Control1)
         ENTRY(@N4),USE(Control2)
         ENTRY(@N4),USE(Control3)
         ENTRY(@N4),USE(Control4)
       END
  CODE
  ACCEPT
    IF NOT ACCEPTED() THEN CYCLE.
    CASE FIELD()             !Control edit control
    OF ?Control1             ! Field number 1
      IF Control1 = 0          ! if no entry
        BEEP               !   sound alarm
        SELECT(?)            !   stay on control
      END
    OF ?Control2             ! Field number 2
      IF Control2 > 4          ! if status is more than 4
        Scr:Message = ´Control must be less than 4´
        ERASE(?)             !   clear control
        SELECT(?)            !   edit the control again
      ELSE                 ! value is valid
        CLEAR(Scr:Message)   !   clear message
      END
    OF ?Control4             ! Field number 4
      BREAK                  ! exit processing loop
  . .                      ! end case, end loop
```

FIRSTFIELD (return first window control)

FIRSTFIELD( )

The **FIRSTFIELD** function returns the lowest field number in the currently active window.

Return Data Type: LONG

Example:

```
DISABLE(FIRSTFIELD(),LASTFIELD())    !Dim all control fields
```

FOCUS (return control with focus)

FOCUS( )

The **FOCUS** function returns the field number of the control which has input focus at any time any event occurs.

Positive field numbers are assigned by the compiler to all WINDOW controls, in the order their declarations occur in the WINDOW structure. Negative field numbers are assigned to all APPLICATION controls. In executable code statements, field numbers are usually represented by field equate labels--the label of the USE variable preceded by a question mark (?FieldName).

Return Data Type: LONG

Example:

```
Screen WINDOW
         ENTRY(@N4),USE(Control1)
         ENTRY(@N4),USE(Control2)
         ENTRY(@N4),USE(Control3)
         ENTRY(@N4),USE(Control4)
       END
  CODE
  ACCEPT
    CASE EVENT()
    OF EVENT:LoseFocus
    OROF EVENT:CloseWindow
      CASE FOCUS()          !Control edit control
      OF ?Control1          ! Field number 1
        UPDATE(?Control1)
      OF ?Control2          ! Field number 2
        UPDATE(?Control2)
      OF ?Control4          ! Field number 4
        UPDATE(?Control4)
      END
    END
  END
```

See Also:

ACCEPTED

SELECTED

FIELD

EVENT

INCOMPLETE (return empty REQ control)

INCOMPLETE( )

The **INCOMPLETE** function returns the field number of the first control with the REQ attribute in the currently active window that has been left zero or blank, and gives input focus to that control. If all REQ controls in the window contain data, INCOMPLETE returns zero (0) and leaves input focus on the control that already had it.

The INCOMPLETE function duplicates the action performed by the REQ attribute on a BUTTON control.

Return Data Type: LONG

Example:

```
CODE
OPEN(Screen)
ACCEPT
  CASE ACCEPTED()
  OF ?OkBUtton
    IF INCOMPLETE()            !Any REQ fields empty?
      SELECT(INCOMPLETE())     ! if so, go to it
      CYCLE
    ELSE
      BREAK                  !If not, go on
    END
  END
END
```

LASTFIELD (return last window control)

LASTFIELD( )

The **LASTFIELD** function returns the highest field number in the currently active window.

Return Data Type: LONG

Example:

```
DISABLE(FIRSTFIELD(),LASTFIELD())      !Dim all control fields
```

MESSAGE (return message box response)

MESSAGE(*text* [,*caption*] [,*icon*] [,*buttons*] [,*default*] [,*style*] **)**

MESSAGE	Displays a message dialog box and returns the button the user pressed.
text	A string constant or variable containing the text to display in the message box.
caption	The dialog box title. If omitted, the dialog has no title.
icon	A string constant, variable, or EQUATE for a Windows standard icon. If omitted, no icon is displayed on the dialog box.
buttons	An integer constant, variable, EQUATE, or expression which indicates which Windows standard buttons to place on the dialog box. This may indicate multiple buttons. If omitted, the dialog displays an Ok button.
default	An integer constant, variable, EQUATE, or expression which indicates the default button on the dialog box. If omitted, the first button is the default.
style	An integer constant or variable which specifies the window is Application Modal (0) or System Modal (1). If omitted, the window is Application Modal.

The **MESSAGE** function displays a Windows-standard message box, typically requiring only a Yes or No response, or no specific response at all. The function returns the number of the button the user presses to exit the dialog box.

The EQUATES.CLW file contains symbolic constants for the *icon*, *buttons*, and *default* parameters. The *style* parameter determines whether the message window is Application Modal or System Modal. An Application Modal window must be closed before the user is allowed to do anything else in the application, but does not prevent the user from switching to another Windows application. A System Modal window must be closed before the user is allowed to do anything else in Windows.

Return Data Type: USHORT

Example:

```
CASE MESSAGE(´Quit?´,´Editor´,ICON:Question,BUTTON:Yes+BUTTON:No,BUTTON:No,1)
                    !A ? icon with Yes and No buttons, the default button is No
                    ! the window is System Modal
OF BUTTON:No
OF BUTTON:Yes
  MESSAGE('Goodbye')    !A message with only an Ok button.
  RETURN
END
```

MOUSEX (return mouse horizontal position)

MOUSEX( )

The **MOUSEX** function returns a numeric value corresponding to the current horizontal position of the mouse cursor at the time of the event. The position is relative to the origin of that window.

The return value is in dialog units.

Return Data Type: SHORT

Example:

```
SaveMouseX = MOUSEX()      !Save mouse position
```

MOUSEY (return mouse vertical position)

MOUSEY( )

The **MOUSEY** function returns a numeric value corresponding to the current vertical position of the mouse cursor at the time of the event. The position is relative to the origin of that window.

The return value is in dialog units.

Return Data Type: SHORT

Example:

```
SaveMouseY = MOUSEY()      !Save mouse position
```

POPUP (return popup menu selection)

POPUP(*selections* [, *x*] [, *y*] **)**

POPUP	Returns an integer indicating the user's choice from the menu.
selections	A string constant, variable, or expression containing the text for the menu choices.
x	An integer constant, variable, or expression that specifies the horizontal position of the top left corner. If omitted, the menu appear at the current cursor position.
y	An integer constant, variable, or expression that specifies the vertical position of the top left corner. If omitted, the menu appear at the current cursor position.

The **POPUP** function returns an integer indicating the user's choice from the popup menu that appears when the function is invoked. If the user CLICKS outside the menu or presses ESC (indicating no choice), POPUP returns zero.

Within the *selections* string, each choice in the popup menu must be delimited by a vertical bar (|) character. A set of vertical bars containing only a hyphen (|-|) defines a separator between groups of menu choices. A menu choice immediately preceded by a tilde (~) is disabled (it appears dimmed out in the popup menu). A menu choice immediately preceded by a plus sign (+) appears with a check mark to its left in the popup menu. A menu choice immediately followed by a set of choices contained within curly braces (|SubMenu{{SubChoice 1|SubChoice 2}|) defines a sub-menu within the popup menu (the two beginning curly braces are required by the compiler to differentiate your sub-menu from a string repeat count).

Each menu selection is numbered in ascending sequence according to its position within the *selections* string, beginning with one (1). Separators and selections that call a sub-menu are not included in the numbering sequence (which makes an EXECUTE structure the most efficient code structure to use with this function). When the user CLICKS or presses ENTER on a choice, the function terminates, returning the position number of the selected menu item.

Return Data Type: SHORT

Example:

```
PopupString = 'First|+Second|Sub menu{{One|Two}|-|Third|~Disabled'
ToggleChecked = 1
ACCEPT
  CASE EVENT()
  OF EVENT:AlertKey
    IF KEYCODE() = MouseRight
      EXECUTE POPUP(PopupString)
        FirstProc              !Call proc for selection 1
        BEGIN                  !Code to execute for toggle selection 2
          IF ToggleChecked = 1   !Check toggle state
            SecondProc(Off)      !Call proc to turn off something
            PopupString = 'First|Second|Sub menu{{One|Two}|-|Third|~Disabled'
                                 !Reset string so the check mark does not appear
            ToggleChecked = 0    !Set toggle flag
          ELSE
            SecondProc(On)       !Call proc to turn off something
            PopupString = 'First|+Second|Sub menu{{One|Two}|-|Third|~Disabled'
                                 !Reset string so the check mark does appear
            ToggleChecked = 1    !Set toggle flag
          END
        END                    !End Code to execute for toggle selection 2
        OneProc                 !Call proc for selection 3
```

```
      TwoProc                !Call proc for selection 4
      ThirdProc              !Call proc for selection 5
      DisabledProc   !Selection 6 is dimmed so it cannot execute this procedure
    END
  END
 END
END
```

SELECTED (return control that has received focus)

SELECTED( )

The **SELECTED** function returns the field number of the control receiving input focus when an EVENT:Selected event occurs. SELECTED returns zero (0) for all other events.

Positive field numbers are assigned by the compiler to all WINDOW controls, in the order their declarations occur in the WINDOW structure. Negative field numbers are assigned to all APPLICATION controls. In executable code statements, field numbers are usually represented by field equate labels--the label of the USE variable preceded by a question mark (?FieldName).

Return Data Type: SHORT

Example:

```
CASE SELECTED()           !Process pre-edit code
OF ?Cus:Company
  !Pre-load field value
OF ?Cus:CustType
  !Pre-load field value
END
```

See Also:

ACCEPT

SELECT

Keyboard Procedures

ALIAS (set alternate keycode)

ALIAS([*keycode*,[*new keycode*]] **)**

ALIAS	Changes the keycode generated when the original key is pressed.
keycode	A numeric keycode or keycode EQUATE. If both parameters are omitted, all ALIASed keys are reset to their original values.
new keycode	A numeric keycode or keycode EQUATE. If omitted, the *keycode* is reset to its original value.

ALIAS changes the *keycode* to generate the *new keycode* when the user presses the original key. ALIAS does not affect keypresses generated by PRESSKEY. The effect of ALIAS is global, throughout all execution threads, no matter where the ALIAS statement executes. Therefore, to only change the *keycode* locally, you must reset ALIASed keys when the window loses focus.

Keycode values 0800h through 0FFFFh are unassigned and may be used as a *new keycode*. The practical effect of this is to disable the original key if your program does not test for the *new keycode.*

Example:

```
ALIAS(EnterKey,TabKey)   !Allow user to press enter instead of tab
ALIAS(F3Key,F1Key)      !Move help to F3
ALIAS                   !  Clear all aliased keys
```

ASK (get one keystroke)

ASK

ASK reads a single keystroke from the keyboard buffer. Program execution stops to wait for a keystroke. If there is already a keystroke in the keyboard buffer, ASK gets one keystroke without waiting.

The ASK statement also allows any TIMER attribute events to generate and cycle their own ACCEPT loop. This means any batch processing code can allow other threads to execute their TIMER attribute tasks during the batch process.

Example:

```
ASK                         !Wait for a keystroke
LOOP WHILE KEYBOARD()          !Empty the keyboard buffer
  ASK                       ! without processing keystrokes
END
```

PRESS (put characters in the buffer)

PRESS(*string***)**

PRESS Places characters in the keyboard input buffer.

string A string constant, variable, or expression.

PRESS places characters in the Windows keyboard input buffer. The entire *string* is placed in the buffer. Once placed in the keyboard buffer, the *string* is processed just as if the user had typed in the data.

Example:

```
IF Action = ´AddRecord´           !On the way into a memo on adding a record
  TempString = FORMAT(TODAY(),@D1) & ´ ´ & FORMAT(CLOCK(),@T4)
  PRESS(TempString)               !Pre-load first line of memo with date and time
END
```

PRESSKEY (put a keystroke in the buffer)

PRESSKEY(*keycode***)**

PRESSKEY Places one keystroke in the keyboard input buffer.

keycode An integer constant or keycode EQUATE label.

PRESSKEY places one keystroke in the Windows keyboard input buffer. Once placed in the keyboard buffer, the *keycode* is processed just as if the user had pressed the key. ALIAS does not transform a PRESSKEY *keycode*.

Example:

```
IF Action = ´Add´         !On the way into a memo control on an add record
  Cus:MemoControl = FORMAT(TODAY(),@D1) & ´ ´ & FORMAT(CLOCK(),@T4)
                      !Pre-load first line of memo with date and time
  PRESSKEY(EnterKey)    ! and position user on second line
END
```

SETKEYCODE (specify keycode)

SETKEYCODE(*keycode***)**

SETKEYCODE Sets the keycode returned by the KEYCODE function.

keycode An integer constant or keycode EQUATE label.

SETKEYCODE sets the internal keycode returned by the KEYCODE function. The keycode is not put into the keyboard buffer.

Example:

```
SETKEYCODE(0800h)       !Set up the keycode function to return 0800h
```

See Also:

KEYCODE

Keycode Equate Labels

Keyboard Functions

KEYBOARD (return keystroke waiting)

KEYBOARD( )

The **KEYBOARD** function returns the keycode of the first keystroke in the keyboard buffer. It is used to determine if there are keystrokes waiting to be processed by an ASK or ACCEPT statement.

Return Data Type: LONG

Example:

```
LOOP UNTIL KEYBOARD()                !Wait for any key
  ASK
  IF KEYCODE() = EscKey THEN BREAK.    !On esc key, break the loop
END
```

See Also:

ASK

ACCEPT

Keycode Equate Labels

KEYCHAR (return ASCII code)

KEYCHAR( )

The **KEYCHAR** function returns the ASCII value of the last key pressed at the time the event occurred.

Return Data Type: LONG

Example:

```
ACCEPT                !Wait for an event
  CASE KEYCHAR()       !Process the last keystroke
  OF ´A´ TO ´Z´          ! upper case?
    DO ProcessUpper
  OF ´a´ TO ´z´          ! lower case?
    DO ProcesLower
  END
END
```

See Also:

ASK

ACCEPT

SELECT

Keycode Equate labels

KEYCODE (return last keycode)

KEYCODE( )

The **KEYCODE** function returns the keycode of the last key pressed at the time the event occurred, or the last keycode value set by the SETKEYCODE procedure.

Return Data Type: LONG

Example:

```
ACCEPT                !Loop on the display
  CASE KEYCODE()        !Process the keystroke
  OF UpKey            ! up arrow
    DO GetRecordUp      !   get a record
  OF DownKey            ! down arrow
    DO GetRecordDn      !   get a record
  END
END
```

See Also:

ASK

ACCEPT

SELECT

Keycode Equate labels

KEYSTATE (return keyboard status)

KEYSTATE( )

The **KEYSTATE** function returns a bitmap containing the status of the SHIFT, CTRL, ALT, any extended key, CAPS LOCK, NUM LOCK, SCROLL LOCK, and INSERT keys for the last KEYCODE function return value. The bitmap is contained in the high-order byte of the returned SHORT.

```
x . . . . . .  insert key  (8000h)
. x . . . . . .  scroll lock (4000h)
. . x . . . . .  num lock  (2000h)
. . . x . . . .  caps lock   (1000h)
. . . . x . . .  extended  (0800h)
. . . . . x . .  alt       (0400h)
. . . . . . x .  ctrl      (0200h)
. . . . . . . x  shift       (0100h)
```

Return Data Type: SHORT

Example:

```
ACCEPT                          !Loop on the display
  CASE KEYCODE()                  !Process the keystroke
  OF EnterKey                     ! up arrow
    IF BAND(KEYSTATE(),0800h)       !Detect enter on numeric keypad
      PRESSKEY(TabKey)              ! press tab for the user
    END
  END
END
```

See Also:

KEYCODE

BAND

Windows Standard Dialog Functions

- COLORDIALOG (return chosen color)
- FILEDIALOG (return chosen file)
- FONTDIALOG (return chosen font)
- PRINTERDIALOG (return chosen printer)

COLORDIALOG (return chosen color)

COLORDIALOG([*title*] [,*rgb*] **)**

COLORDIALOG

Displays the Windows standard color choice dialog box to allow the user to choose a color.

title A string constant or variable containing the title to place on the color choice dialog. If omitted, a default *title* is supplied by Windows.

rgb A LONG integer variable to receive the selected color.

The **COLORDIALOG** function displays the Windows standard color choice dialog box and returns the color chosen by the user in the *rgb* parameter. Any existing value in the *rgb* parameter sets the default color choice presented to the user in the color choice dialog.

COLORDIALOG returns zero (0) if the user pressed the Cancel button, or one (1) if the user pressed the Ok button on the color choice dialog.

Return Data Type: SHORT

Example:

```
MDIChild1 WINDOW(´Child One´),AT(0,0,320,200),MDI,MAX,HVSCROLL
          !window controls
        END

ColorNow LONG

  CODE
  IF NOT COLORDIALOG(´Choose Box Color´,ColorNow)
    ColorNow = 000000FFh          !Default to Blue if user pressed Cancel
  END
  OPEN(MDIChild1)
  BOX(100,50,100,50,ColorNow)      !User-defined color for box
```

FILEDIALOG (return chosen file)

FILEDIALOG([*title*] [,*file*] [,*extensions*] [,*flag*] **)**

FILEDIALOG

Displays the Windows standard file choice dialog box to allow the user to choose a file.

title A string constant or variable containing the title to place on the file choice dialog. If omitted, a default *title* is supplied by Windows.

file A string variable to receive the selected filename.

extensions A string constant or variable containing the available file extension selections for the List Files of Type drop list. If omitted, the default is all files (*.*).

flag An integer constant or variable to indicate type of file action to perform. If omitted, or zero (0), the Open... dialog is displayed and the user is warned if the file they choose does not exist (the file is not automatically opened). If one (1), the Save... dialog is displayed and the user is warned if the file does exist (the file is not automatically saved).

The **FILEDIALOG** function displays the Windows standard file choice dialog box and returns the file chosen by the user in the *file* parameter. Any existing value in the *file* parameter sets the default file choice presented to the user in the file choice dialog.

The *extensions* parameter string must contain a description followed by the file mask. All elements in the string must be delimited by the vertical bar (|) character. For example, the *extensions* string ´All Files | *.* | Clarion Source | *.CLW´ defines two selections for the List Files of Type drop list. The first extension listed in the *extensions* string is the default.

FILEDIALOG returns zero (0) if the user pressed the Cancel button, or one (1) if the user pressed the Ok button on the file choice dialog.

Return Data Type: SHORT

Example:

```
ViewTextFile  PROCEDURE

ViewQue       QUEUE                !LIST control display queue
               STRING(255)
              END

FileName      STRING(64),STATIC       !Filename variable

ViewFile      FILE,DRIVER(´ASCII´),NAME(FileName),PRE(Vew)
Record          RECORD
                  STRING(255)
                END
              END

MDIChild1 WINDOW(´View Text File´),AT(0,0,320,200),MDI,SYSTEM,HVSCROLL
           LIST,AT(0,0,320,200),USE(?L1),FROM(ViewQue),HVSCROLL
          END

  CODE
  IF NOT FILEDIALOG(´Choose File to View´,FileName,´Text|*.TXT|Source|*.CLW´,0)
    RETURN                        !Return if no file chosen
  END
  OPEN(ViewFile)                   !Open the file
```

```
  IF ERRORCODE() THEN RETURN.       ! aborting on any error
SET(ViewFile)                     !Start at top of file
LOOP
  NEXT(ViewFile)                  !Reading each line of text
    IF ERRORCODE() THEN BREAK.      !Break loop at end of file
  ViewQue = Vew:Record             !Assign text to queue
  ADD(ViewQue)                    ! and add a queue entry
END
CLOSE(ViewFile)                    !Close the file
OPEN(MDIChild1)                    ! and open the window
ACCEPT                           !Allow the user to read the text and
END                              ! break out of ACCEPT loop only from
                                 ! system menu close option
FREE(ViewQue)                      !Free the queue memory
RETURN                           ! and return to caller
```

FONTDIALOG (return chosen font)

FONTDIALOG([*title*] [,*typeface*] [,*size*] [,*color*] [,*style*]**)**

FONTDIALOG

Displays the standard Windows font choice dialog box to allow the user to choose a font.

title A string constant or variable containing the title to place on the font choice dialog. If omitted, a default *title* is supplied by Windows.

typeface A string variable to receive the name of the chosen font.

size An integer variable to receive the size (in points) of the chosen font.

color A LONG integer variable to receive the red, green, and blue values for the color of the chosen font in the low-order three bytes.

style An integer variable to receive the strike weight and style of the chosen font.

The **FONTDIALOG** function displays the Windows standard font choice dialog box to allow the user to choose a font. When called, any values in the parameters set the default font values presented to the user in the font choice dialog. They also receive the user´s choice when the user presses the Ok button on the dialog.

FONTDIALOG returns zero (0) if the user pressed the Cancel button, or one (1) if the user pressed the Ok button.

Return Data Type: SHORT

Example:

```
MDIChild1   WINDOW(´View Text File´),AT(0,0,320,200),MDI,SYSTEM,HVSCROLL
            !window controls
          END
Typeface  STRING(20)
FontSize  LONG
FontColor   LONG
FontStyle   LONG
  CODE
  OPEN(MDIChild1)                     !open the window
  IF FONTDIALOG(´Choose Display Font´,Typeface,FontSize,FontColor,FontStyle)
    SETFONT(0,Typeface,FontSize,FontColor,FontStyle) !Set window font
  ELSE
    SETFONT(0,´Arial´,12)             !Set default font
  END
  ACCEPT
    !Window handling code
  END
```

PRINTERDIALOG (return chosen printer)

PRINTERDIALOG([*title*] **)**

PRINTERDIALOG

Displays the Windows standard printer choice dialog box to allow the user to choose a file.

title A string constant or variable containing the title to place on the file choice dialog. If omitted, a default *title* is supplied by Windows.

The **PRINTERDIALOG** function displays the Windows standard printer choice dialog box and returns the printer chosen by the user in the PRINTER "built-in" variable in the internal library. This sets the default printer used for the next REPORT opened. Properties of the chosen printer may be set and queried using the runtime property assignment syntax, specifying PRINTER as the target.

PRINTERDIALOG returns zero (0) if the user pressed the Cancel button, or one (1) if the user pressed the Ok button on the printer choice dialog.

Return Data Type: SHORT

Example:

```
CustRpt  REPORT,AT(1000,1000,6500,9000),THOUS,FONT(´Arial´,12),PRE(Rpt)
           !Report structures and controls
         END

  CODE
  IF NOT PRINTERDIALOG(´Choose Printer´)
    RETURN                          !Abort if user pressed Cancel
  END
  OPEN(CustRpt)
```

Drag and Drop Processing

Drag-and-drop is a very powerful Windows tool that allows a user to copy or move data from one control to another (or even within the same control). These controls may be in the same window, separate windows in the same application, or even separately executing Clarion applications.

Implementing drag-and-drop in a Clarion application involves two processes:

Specifying drag host and drop target controls.

Performing the data exchange when the user initiates drag-and-drop by handling the drag-and-drop events.

To specify a drag host, you place the DRAGID attribute on a LIST or REGION control with a set of "signatures" that verify valid drop targets for the data. To specify a drop target, you place the DROPID attribute on a control to list the set of valid drag "signatures" from which the control will accept data. Drag-and-drop operations only occur between controls with matching "signatures" in their respective DRAGID and DROPID attributes.

A successful drag-and-drop operation occurs when the user drags information from a control with the DRAGID attribute to a control with the DROPID attribute and both controls have at least one identical *signature* string in their respective DRAGID and DROPID attributes. When the user initiates drag-and-drop, EVENT:Dragging is posted to the host control whenever the mouse is over a potential target control (valid or not). EVENT:Drag is posted to the host control when the user releases the mouse button over a potential target control (valid or not). EVENT:Drop is posted to the target control only if it is a valid match.

The DRAGID() function detects the successful drop. The DROPID() function can also detect a successful drop, or can pass the exchanged data as a string, if its value is set by the SETDROPID procedure. The actual data exchange between the controls can be accomplished several ways:

If the two controls are in the same window, you can exchange data using local or global variables, the DROPID function can exchange the data, or you can use the Windows clipboard.

If the two controls are in the same application, you can exchange data using global variables, the DROPID function can exchange the data, or you can use the Windows clipboard.

If the controls are in separate Clarion applications, you must use SETDROPID to have the DROPID function exchange the data, or use the Windows clipboard.

You can copy or move the data to the target control, depending upon how you write the data exchange code. Also, you should write the data exchange code for the most difficult coding circumstance. Therefore, if the drag host might be an external program´s control, you could pass the data through the DROPID() function (using SETDROPID), or through the Windows clipboard. If the drag host could be a control in any window within the program, you should either pass the data through the DROPID() function, or use global variables. Only for those instances where the drag host and drop target are always going to be in the same procedure should you use local variables.

See also:

CLIPBOARD (return windows clipboard contents)

DRAGID (return matching drag-and-drop signature)

DROPID (return drag-and-drop string)

SETCLIPBOARD (set windows clipboard contents)

SETDROPID (set DROPID return string)

CLIPBOARD (return windows clipboard contents)

CLIPBOARD( )

The **CLIPBOARD** function returns the current contents of the Windows clipboard.

Return Data Type: STRING

Example:

```
Que1 QUEUE
       STRING(30)
     END

Que2 QUEUE
       STRING(30)
     END

WinOne WINDOW,AT(0,0,160,400)
        LIST,AT(120,0,20,20),USE(?List1),FROM(Que1),DRAGID(´List1´)
                                    !Allows drags, but not drops
        LIST,AT(120,120,20,20),USE(?List2),FROM(Que2),DROPID(´List1´,´~FILE´)
                                    !Allows drops from List1, but no drags
       END

  CODE
  OPEN(WinOne)
  ACCEPT
    CASE EVENT()
    OF EVENT:Drag                  !When a drag event is attempted
      IF DRAGID()                  ! check for success
        SETCLIPBOARD(Que1)           ! and setup info to pass
      END
    OF EVENT:Drop                  !When drop event is successful
      Que2 = CLIPBOARD()             ! get dropped info
      ADD(Que2)                    ! and add it to the queue
    END
  END
```

See Also:

SETCLIPBOARD

DRAGID (return matching drag-and-drop signature)

DRAGID([*thread*] [, *control*] **)**

DRAGID	Returns matching host and target signatures on a successful drag-and-drop operation.
thread	The label of a numeric variable to receive the thread number of the host control. If the host control is in an external program, *thread* receives zero (0).
control	The label of a numeric variable to receive the field equate label of the host control.

The **DRAGID** function returns the matching host and target control signatures on a successful drag-and-drop operation. If the user aborted the operation, DRAGID returns an empty string (´´), otherwise it returns the first signature that matched between the two controls.

Return Data Type: STRING

Example:

```
Que1 QUEUE
       STRING(30)
     END
Que2 QUEUE
       STRING(30)
     END
WinOne WINDOW,AT(0,0,160,400)
        LIST,AT(120,0,20,20),USE(?List1),FROM(Que1),DRAGID(´List1´)
                             !Allows drags, but not drops
        LIST,AT(120,120,20,20),USE(?List2),FROM(Que2),DROPID(´List1´)
                             !Allows drops from List1, but no drags
       END
  CODE
  OPEN(WinOne)
  ACCEPT
    CASE EVENT()
    OF EVENT:Drag                  !When a drag event is attempted
      IF DRAGID()                  ! check for success
        SETDROPID(Que1)              ! and setup info to pass
      END
    OF EVENT:Drop                  !When drop event is successful
      Que2 = DROPID()              ! get dropped info
      ADD(Que2)                    ! and add it to the queue
    END
  END
```

See Also:

DROPID

SETDROPID

DROPID (return drag-and-drop string)

DROPID([*thread*] [, *control*] **)**

DROPID Returns matching host and target signatures on a successful drag-and-drop operation.

thread The label of a numeric variable to receive the thread number of the target control. If the target control is in an external program, *thread* receives zero (0).

control The label of a numeric variable to receive the field equate label of the target control.

The **DROPID** function returns the matching host and target control signatures on a successful drag-and-drop operation (just as DRAGID does), or the specific string set by the SETDROPID procedure. The DROPID function returns a comma-delimited list of filenames dragged from the Windows File Manager when ´~FILE´ is the DROPID attribute.

Return Data Type: STRING

Example:

```
DragDrop    PROCEDURE
Que1 QUEUE
       STRING(90)
     END
Que2 QUEUE
       STRING(90)
     END
WinOne WINDOW(´Test Drag Drop´),AT(10,10,240,320),SYSTEM,MDI
         LIST,AT(12,0,200,80),USE(?List1),FROM(Que1),DRAGID(´List1´)
                         !Allows drags, but not drops
         LIST,AT(12,120,200,80),USE(?List2),FROM(Que2),DROPID(´List1´,´~FILE´)
                         !Allows drops from List1 or the Windows File Manager,
       END               ! but no drags
  CODE
  OPEN(WinOne)
  ACCEPT
    CASE EVENT()
    OF EVENT:Drag           !When a drag event is attempted
      IF DRAGID()           ! check for success
        GET(Que1,CHOICE())
        SETDROPID(Que1)       ! and setup info to pass
      END
    OF EVENT:Drop           !When drop event is successful
         IF INSTRING(´,´,DROPID(),1,1)        !Check for multiple files from File
Manager
       Que2 = SUB(DROPID(),1,INSTRING(´,´,DROPID(),1,1)-1)         ! and only get first
       ADD(Que2)            ! and add it to the queue
         ELSE
       Que2 = DROPID()        ! get dropped info, from List1 or File Manager
       ADD(Que2)            ! and add it to the queue
         END
    END
  END
```

See Also:

DRAGID

SETDROPID

SETCLIPBOARD (set windows clipboard contents)

SETCLIPBOARD(*string* **)**

SETCLIPBOARD

Puts information in the Windows clipboard.

string A string constant or variable containing the information to place in the Windows clipboard.

The **SETCLIPBOARD** procedure places the contents of the *string* into the Windows clipboard, overwriting any previous contents.

Example:

```
Que1 QUEUE
       STRING(30)
     END
Que2 QUEUE
       STRING(30)
     END
WinOne WINDOW,AT(0,0,160,400)
         LIST,AT(120,0,20,20),USE(?List1),FROM(Que1),DRAGID(´List1´)
                              !Allows drags, but not drops
         LIST,AT(120,120,20,20),USE(?List2),FROM(Que2),DROPID(´List1´)
                              !Allows drops from List1, but no drags
       END
  CODE
  OPEN(WinOne)
  ACCEPT
    CASE EVENT()
    OF EVENT:Drag                !When a drag event is attempted
      IF DRAGID()                ! check for success
        SETCLIPBOARD(Que1)         ! and setup info to pass
      END
    OF EVENT:Drop                !When drop event is successful
      Que2 = CLIPBOARD()           ! get dropped info
      ADD(Que2)                  ! and add it to the queue
    END
  END
```

See Also:

CLIPBOARD

SETDROPID (set DROPID return string)

SETDROPID(*string* **)**

SETDROPID Sets the DROPID function´s return value.

string A string constant or variable containing the value the DROPID function will return.

The **SETDROPID** procedure sets the DROPID function´s return value. This allows the DROPID function to pass the data in a drag-and-drop operation. When drag-and-drop operations are performed between separate Clarion applications, this is the mechanism to use to pass the data.

Example:

```
Que1 QUEUE
       STRING(30)
     END
Que2 QUEUE
       STRING(30)
     END
WinOne WINDOW,AT(0,0,160,400)
         LIST,AT(120,0,20,20),USE(?List1),FROM(Que1),DRAGID(´List1´)
                        !Allows drags, but not drops
         LIST,AT(120,120,20,20),USE(?List2),FROM(Que2),DROPID(´List1´)
                        !Allows drops from List1 or the Windows File Manager,
                        ! but no drags
       END
  CODE
  OPEN(WinOne)
  ACCEPT
    CASE EVENT()
    OF EVENT:Drag          !When a drag event is attempted
      IF DRAGID()          ! check for success
        SETDROPID(Que1)      ! and setup info to pass
      END
    OF EVENT:Drop          !When drop event is successful
      Que2 = DROPID()        ! get dropped info, from List1 or File Manager
      ADD(Que2)            ! and add it to the queue
    END
  END
```

See Also:

DRAGID

DROPID

Maintaining INI Files

GETINI (return INI file entry)

GETINI(*section* ,*entry* [,*default*] [,*file*] **)**

GETINI	Returns the value for an INI file entry.
section	A string constant or variable containing the name of the portion of the INI file which contains the *entry*.
entry	A string constant or variable containing the name of the specific setting for which to return the value.
default	A string constant or variable containing the default value to return if the *entry* does not exist. If omitted, GETINI returns an empty string.
file	A string constant or variable containing the name of the INI file to search. If omitted, GETINI searches the WIN.INI file.

The **GETINI** function returns the value of an *entry* in a Windows-standard INI file. A Windows-standard INI file is an ASCII text file with the following format:

```
[some section name]
entry=value
next entry=another value
```

For example, WIN.INI contains entries such as:

```
[intl]
sLanguage=enu
sCountry=United States
iCountry=1
```

The GETINI function searches the specified *file* for the *entry* within the *section* you specify. It returns everything on the *entry´s* line of text that appears to the right of the equal sign (=).

Return Data Type: STRING

Example:

```
Value   STRING(30)
  CODE
  Value = GETINI(´intl´,´sLanguage´)      !Get the language entry
```

See Also:

PUTINI

PUTINI (set INI file entry)

PUTINI(*section* ,*entry* [,*value*] [,*file*] **)**

PUTINI	Sets the value for an INI file entry.
section	A string constant or variable containing the name of the portion of the INI file which contains the *entry*.
entry	A string constant or variable containing the name of the specific entry to set.
value	A string constant or variable containing the setting to place in the *entry*. An empty string (´´) leaves the *entry* empty. If omitted, the *entry* is deleted.
file	A string constant or variable containing the name of the INI file to search. If omitted, PUTINI places the *entry* in the WIN.INI file.

The **PUTINI** procedure places the *value* into an *entry* in a Windows-standard .INI file. A Windows-standard .INI file is an ASCII text file with the following format:

```
[some section name]
entry=value
next entry=another value
```

For example, WIN.INI contains entries such as:

```
[windows]
spooler=yes
load=nwpopup.exe
[intl]
sLanguage=enu
sCountry=United States
iCountry=1
```

The PUTINI function searches the specified *file* for the *entry* within the *section* you specify. It replaces the current entry value with the *value* you specify. If necessary, the *section* and *entry* are created.

Example:

```
CODE
PUTINI(´MyApp´,´SomeSetting´,´Initialized´)    !Place setting in WIN.INI
PUTINI(´MyApp´,´ASetting´,´2´,´MYAPP.INI´)   !Place setting in   MYAPP.INI
```

See Also:

GETINI

Reports

Reports in Windows

Clarion Database Developer for Windows reports use a page-based printing paradigm instead of a line-based paradigm. Instead of printing each line as it´s values are generated, nothing is sent to the printer until an entire page is ready to print. This means that the "print engine" in the Clarion runtime library can do a lot of work for you, based on the attributes you specify in the REPORT structure.

Some of the things that the "print engine" in the Clarion runtime library does for you are:

- Prints "pre-printed" forms on each page, that are then filled in by the data
- Calculates totals (count, sum, average, minimum, maximum)
- Provides automatic page break handling, including page headers and footers
- Provides automatic group break handling, including group headers and footers
- Provides complete widow/orphan control.

This automatic functionality makes the executable code required to print a complex report very small, making your programming job easier.

Since the "print engine" is page-based, the concepts of headers and footers lose their context indicating both page positioning and print sequence, and only retain their meaning of print sequence. Headers are printed at the beginning of a print sequence, and footers are printed at the end--their actual positioning on the page is irrelevant. For example, you could position the page footer, containing page totals, to print at the top of the page.

See also:

- Page Overflow
- Report Structure

Page Overflow

Page Overflow occurs when the PRINT statement cannot fit a DETAIL structure on a page. This may be due to a lack of space, or the presence of the PAGEBEFORE or PAGEAFTER attribute on a DETAIL structure.

The following steps occur during page overflow, in this sequence:

1. If the REPORT has a page FOOTER, it is printed at the position specified by its AT attribute.
2. The page counter is incremented.
3. If the REPORT has a FORM structure, it is printed at the position specified by its AT attribute.
4. If the REPORT has a page HEADER, it is printed at the position specified by its AT attribute.

Report Structure

- REPORT (declare a report structure)
- AT (set detail print area)
- FONT (set report default font)
- PRE (set report label prefix)
- PREVIEW (set report output to metafiles)
- LANDSCAPE (set page orientation)
- THOUS, MM, POINTS (set report coordinate measure)

REPORT (declare a report structure)

```
label REPORT([jobname])AT( )[FONT( )][PRE( )][LANDSCAPE][PREVIEW][,PAPER][  | THOUS   | ]
                                                                            | MM      |
                                                                            | POINTS  |
                  [FORM
                      controls
                   END ]
                  [HEADER
                      controls
                   END ]
label             DETAIL
                      controls
                   END
label             [BREAK( )
                      group break structures
                   END ]
                  [FOOTER
                      controls
                   END ]
          END
```

REPORT	Declares the beginning of a report data structure.
label	The name by which the structure is addressed in executable code.
jobname	Names the print job for the Windows Print Manager. If omitted, the REPORTs *label* is used.
AT	Specifies the size and location, relative to the top left corner of the page, of the area devoted to printing report detail.
FONT	Specifies the default font for all controls in this report. If omitted, the printers default font is used.
PRE	Specifies the label prefix for the report or structure.
LANDSCAPE	Specifies printing in landscape mode. If omitted, printing defaults to portrait mode.
PREVIEW	Specifies report output to Windows metafiles (.WMF); one file per report page.
PAPER	Specifies the paper size for the report output. If omitted, the default printers paper size is used.
THOUS	Specifies thousandths of an inch as the measurement unit used for all attributes which use coordinates.
MM	Specifies millimeters as the measurement unit used for all attributes which use coordinates.
POINTS	Specifies points as the measurement unit used for all attributes which use coordinates. There are 72 points per inch, vertically and horizontally.
FORM	Page layout structure defining pre-printed items on every page.
controls	Report output controls.
HEADER	Page header structure, printed at the beginning of each page.
DETAIL	Report detail structure.
BREAK	A group break structure, defining the variable which causes a group break to occur when

its value changes.

group break structures
: Group break HEADER, FOOTER, and DETAIL structures, and/or other nested BREAK structures.

FOOTER Page footer structure, printed at the end of each page.

The **REPORT** statement declares the beginning of a report data structure. A REPORT structure must be terminated with a period or END statement. Within the REPORT, the FORM, HEADER, DETAIL, FOOTER, and BREAK structures are the components that format the output of the report. A REPORT must be explicitly opened with the OPEN statement.

A REPORT with the PREVIEW attribute sends the report output to Windows metafiles (.WMF) containing one report page per file. The PREVIEW attribute names a QUEUE to receive the names of the metafiles. You can then create a window to display the report in an IMAGE control, using the QUEUE field contents (the file names) to set the IMAGE controls {PROP:text} property. This allows the end user to view the report before printing.

Only DETAIL structures can (and must) be printed with the PRINT statement. All other report structures (HEADER, FOOTER, and FORM) are automatically printed for you at the appropriate place in the report.

The REPORTs AT attribute defines the area of each page devoted to printing DETAIL structures. This includes any HEADERs and FOOTERs that are contained within a BREAK structure (group headers and footers).

The FORM structure is printed on every page except pages containing DETAIL structures with the ALONE attribute. Its format is determined once at the beginning of the report. This makes it the logical place to design a pre-printed form template, which is filled in by the subsequent HEADER, DETAIL, and FOOTER structures.

The page HEADER and FOOTER structures are not within a BREAK structure. They are automatically printed whenever a page break occurs.

The BREAK structure defines a group break. It may contain its own HEADER, FOOTER, and DETAIL structures, and/or other nested BREAK structures. It may also contain multiple DETAIL structures. The HEADER and FOOTER structures that are within a BREAK structure are the group header and footer. They are automatically printed when the value in a specified group break variable changes.

A REPORT data structure never defaults as the current target for runtime property assignment the way the most recently opened WINDOW or APPLICATION structure does. Therefore, the REPORT *label* must be explicitly named as the target, or the SETTARGET statement must be used to make the REPORT the current target, before using runtime property assignment to a REPORT control. Since the graphics commands draw graphics only to the current target, the SETTARGET statement must be used to make the REPORT the current target before using the graphics functions on a REPORT.

Example:

```
CustRpt  REPORT,AT(1000,1000,6500,9000),THOUS,FONT(Arial,12),PRE(Rpt)
         FORM,AT(1000,1000,6500,9000)
           IMAGE(LOGO.BMP),AT(0,0,1200,1200),USE(?I1)
           STRING(@n3),AT(6000,500,500,500),PAGENO
         END
         HEADER,AT(1000,1000,6500,1000)
           STRING(ABC Company),AT(3000,500,1500,500),FONT(Arial,18)
         END
Break1   BREAK(Pre:Key1)
           HEADER,AT(0,0,6500,1000)
```

```
            STRING(Group Head),AT(3000,500,1500,500),FONT(Arial,18)
          END
Detail    DETAIL,AT(0,0,6500,1000)
            STRING(@N$11.2'),AT(6000,1500,500,500),USE(Pre:F1)
          END
          FOOTER,AT(0,0,6500,1000)
            STRING(Group Total:),AT(5500,500,1500,500)
            STRING(@N$11.2'),AT(6000,500,500,500),USE(Pre:F1),SUM,RESET(Break1)
          END
        END
        FOOTER,AT(1000,1000,6500,1000)
          STRING(Page Total:),AT(5500,1500,1500,500)
          STRING(@N$11.2'),AT(6000,1500,500,500),USE(Pre:F1),SUM,PAGE
        END
      END                                    !End report declaration

  CODE
  OPEN(CustReport)
  SET(DataFile)
  LOOP
    NEXT(DataFile)
      IF ERRORCODE() THEN BREAK.
    PRINT(Rpt:Detail)
  END
  CLOSE(CustReport)
```

AT (set detail print area)

AT([*x*] [,*y*] [,*width*] [,*height*])

AT	Defines the position and size of the area of the page devoted to printing report detail.
x	An integer constant or constant expression that specifies the horizontal position of the top left corner of the detail area.
y	An integer constant or constant expression that specifies the vertical position of the top left corner of the detail area.
width	An integer constant or constant expression that specifies the width of the detail area.
height	An integer constant or constant expression that specifies the height of the detail area.

The **AT** attribute on a REPORT structure defines the position and size of the area of the page devoted to printing report detail. This includes the area to print all DETAIL structures and any group HEADER and FOOTER structures contained within BREAK structures.

The values contained in the *x*, *y*, *width*, and *height* parameters default to dialog units unless the THOUS, MM, or POINTS attribute is also present. Dialog units are defined as one-quarter the average character width by one-eighth the average character height. The size of a dialog unit is dependent upon the size of the default font for the report. This measurement is based on the font specified in the FONT attribute of the report, or the printer´s system default font.

Example:

```
CustRpt1 REPORT,AT(1000,1000,6500,9000),THOUS !1" margins all around for detail
                                           ! area on 8.5" x 11" paper
           !report declarations
         END

CustRpt2 REPORT,AT(72,72,468,648),POINTS      !1" margins all around for detail
                                           ! area on 8.5" x 11" paper
           !report declarations
         END
```

FONT (set report default font)

FONT([*typeface*] [,*size*] [,*color*] [,*style*]**)**

FONT	Specifies the REPORT structure's default print font.
typeface	A string constant containing the name of the font. If omitted, the printer´s default font is used.
size	An integer constant containing the size (in points) of the font. If omitted, the printer´s default font size is used.
color	A LONG integer constant containing the red, green, and blue values for the color of the font in the low-order three bytes, or an EQUATE for a standard Windows color value. If omitted, black is used.
style	An integer constant or constant expression or EQUATE specifying the strike weight and style of the font. If omitted, the weight is normal.

The **FONT** attribute on a REPORT structure specifies the default print font for all controls in the REPORT. This font is used when the control does not have a FONT attribute or its own, and the print structure it is in also has no FONT attribute.

The *typeface* may name any font registered in the Windows system which the printer driver supports. This includes the TrueType fonts for most printers. The EQUATES.CLW file contains EQUATE values for standard *style* values. A *style* on the range zero (0) to one thousand (1000) specifies the strike weight of the font. You may add to that values that indicate italic, underline, or strikeout text. The following EQUATES are in EQUATES.CLW:

FONT:thin	EQUATE (100)
FONT:regular	EQUATE (400)
FONT:bold	EQUATE (700)
FONT:italic	EQUATE (01000H)
FONT:underline	EQUATE (02000H)
FONT:strikeout	EQUATE (04000H)

Example:

```
CustRpt  REPORT,AT(1000,1000,6500,9000),THOUS, |
                    FONT(´Arial´,12,,FONT:Bold+FONT:Italic)
         !report declarations
         END
```

PRE (set report label prefix)

PRE(*prefix***)**

PRE	Provides a label prefix for structures in the report.
prefix	A string constant containing the prefix for labels within the REPORT. Acceptable characters are alphabet letters, numerals 0 through 9, and the underscore character. A *prefix* must start with an alphabet character and must not be a reserved word. By convention, a *prefix* is 1-3 characters, although it can be longer.

The **PRE** attribute on a REPORT provides a label prefix for DETAIL and BREAK structures. It is used to distinguish between identical variable names that occur in different structures. When referenced in executable statements, the *prefix* is attached to a label by a colon (Pre:Label).

Example:

```
Report  REPORT,PRE(´Rpt´)
DetailOne   DETAIL           !Always referenced as Rpt:DetailOne
            !Report controls
          END              ! in executable code
        END
```

See Also:

Reserved Words

PREVIEW (set report output to metafiles)

PREVIEW(*queue***)**

PREVIEW	Specifies report output goes to Windows metafiles (.WMF) containing one report page per file.
queue	The label of a QUEUE or a field in a QUEUE to receive the names of the metafiles.

The **PREVIEW** attribute on a REPORT sends the report output to Windows metafiles (.WMF) containing one report page per file. The PREVIEW attribute names a *queue* to receive the names of the metafiles. The filenames are temporary filenames internally created by the Clarion library and are complete file specifications (up to 64 characters, including drive and path). These temporary files are deleted from disk when you CLOSE the REPORT.

You can create a window to display the report in an IMAGE control, using the *queue* containing the file names to set the IMAGE control´s {PROP:text} property. This allows the end user to view the report before printing. A runtime-only property, {PROP:flushpreview}, when set to ON, flushes the metafiles to the printer.

Example:

```
SomeReport  PROCEDURE

WMFQue     QUEUE                  !Queue to contain .WMF filenames
             STRING(64)
           END

NextEntry  BYTE(1)                !Queue entry counter variable

Report   REPORT,PREVIEW(WMFQue)    !Report with PREVIEW attribute
DetailOne   DETAIL
             !Report controls
           END
         END

ViewReport  WINDOW(´View Report´),AT(0,0,320,200),MDI,MAX,HVSCROLL
             IMAGE(´´),AT(0,0,320,180),USE(?ImageField)
             BUTTON(´View Next Page´),AT(0,180,60,20),USE(?NextPage),DEFAULT
             BUTTON(´Print Report´),AT(80,180,60,20),USE(?PrintReport)
             BUTTON(´Exit Without Printing´),AT(160,180,60,20),USE(?ExitReport)
           END

  CODE
  OPEN(Report)
  SET(SomeFile)                          !Code to generate the report
  LOOP
    NEXT(SomeFile)
      IF ERRORCODE() THEN BREAK.
    PRINT(DetailOne)
  END
  ENDPAGE(Report)
  OPEN(ViewReport)                       !Open report preview window
  GET(WMFQue,NextEntry)                   !Get first queue entry
  ?ImageField{PROP:text} = WMFQue          !Load first report page
  ACCEPT
    CASE ACCEPTED()
    OF ?NextPage
      NextEntry += 1                     !Increment entry counter
      IF NextEntry > RECORDS(WMFQue) THEN CYCLE. !Check for end of report
```

```
    GET(WMFQue,NextEntry)               !Get next queue entry
    ?ImageField{PROP:text} = WMFQue        !Load next report page
    DISPLAY                            ! and display it
  OF ?PrintReport
    Report{PROP:flushpreview} = ON       !Flush files to printer
    BREAK                              ! and exit procedure
  OF ?ExitReport
    BREAK                              !Exit procedure
  END
END
CLOSE(ViewReport)                       !Close window
FREE(WMFQue)                           !Free the queue memory
CLOSE(Report)                          !Close report (deleting all .WMF files)
RETURN                                ! and return to caller
```

LANDSCAPE (set page orientation)

LANDSCAPE

The **LANDSCAPE** attribute on a REPORT indicates the report is to print in landscape mode by default. If the LANDSCAPE attribute is omitted, printing defaults to portrait mode.

Example:

```
Report REPORT,PRE(´Rpt´),LANDSCAPE       !Defaults to landscape mode
         !Report structure declarations
       END
```

THOUS, MM, POINTS (set report coordinate measure)

THOUS
MM
POINTS

The **THOUS**, **MM**, and **POINTS** attributes specify the coordinate measures used to position controls on the REPORT.

THOUS specifies thousandths of an inch, **MM** specifies millimeters, and **POINTS** specifies points (there are seventy-two points per inch, both vertically and horizontally).

If all these attributes are omitted, the measurements default to dialog units. Dialog units are defined as one-quarter the average character width by one-eighth the average character height. The size of a dialog unit is dependent upon the size of the default font for the report. This measurement is based on the font specified in the FONT attribute of the REPORT, or the system default font specified by Windows.

Print Structures

BREAK (declare group break structure)

label **BREAK(***variable***)**
group break structures
END

BREAK Declares a group break structure.

label The name by which the structure is addressed in executable code.

variable The variable whose change in value signals the group break.

group break structures
Group break HEADER, FOOTER, and DETAIL structures, and/or other nested BREAK structures.

The **BREAK** structure declares the *variable* which signals a group break when the value in the *variable* changes. A BREAK structure must be terminated with a period or END statement. It may contain its own HEADER, FOOTER, and DETAIL structures, and/or other nested BREAK structures. Only one HEADER and FOOTER are allowed in a BREAK structure; it may contain multiple DETAIL and/or BREAK structures.

The HEADER and FOOTER structures that are within a BREAK structure are the group header and footer. They are automatically printed when the value in the group break *variable* changes.

Example:

```
CustRpt   REPORT                    !Declare customer report
Break1    BREAK(SomeVariable)
            HEADER                ! begin group header declaration
              !report controls
            END                   ! end header declaration
GroupDet    DETAIL
              !report controls
            END                   ! end detail declaration
            FOOTER                ! begin group footer declaration
              !report controls
            END                   ! end footer declaration
          END                     ! end group break declaration
        END                       !End report declaration
```

DETAIL (report detail line structure)

label **DETAIL** ,**AT( )** [,**FONT( )**] [,**ALONE**] [,**ABSOLUTE**] [,**PAGEBEFORE( )**] [,**PAGEAFTER( )**]
[,**WITHPRIOR( )**] [,**WITHNEXT( )**] [,**USE( )**]
controls
END

DETAIL	Declares items to be printed as the body of the report.
label	The name by which the structure is addressed in executable code.
AT	Specifies the offset and minimum width and height of the DETAIL, relative to the size of the area specified by the REPORT´s AT attribute.
FONT	Specifies the default font for all controls in this structure. If omitted, the REPORT´s FONT attribute (if present) is used, or else the printer´s default font is used.
ALONE	Declares the DETAIL structure must be printed on a page without FORM, (page) HEADER, or (page) FOOTER structures.
ABSOLUTE	Declares the DETAIL prints at a fixed position relative to the page.
PAGEBEFORE	Declares the DETAIL prints at the start of a new page, after normal page overflow actions.
PAGEAFTER	Declares the DETAIL prints, and then starts a new page by activating normal page overflow actions.
WITHPRIOR	Declares the DETAIL prints on the same page as the DETAIL, group HEADER, or FOOTER that immediately precedes it.
WITHNEXT	Declares the DETAIL prints on the same page as the DETAIL, group HEADER, or FOOTER that immediately follows it.
USE	A field equate label to reference the DETAIL structure in executable code.
controls	Report output control fields.

The **DETAIL** structure declares items to be printed as the body of the report. A DETAIL structure must be terminated with a period or END statement. A REPORT may have multiple DETAIL structures.

A DETAIL structure is never automatically printed, therefore DETAIL structures are always explicitly printed by the PRINT statement. This means that a *label* is **required** for each DETAIL you wish to PRINT.

The DETAIL structure may be printed whenever necessary. Since you may have multiple DETAIL structures, they provide the ability to optionally print alternate print formats. This is determined by the logic in the executable code which prints the report.

Example:

```
CustRpt     REPORT                !Declare customer report
            HEADER               ! begin page header declaration
              !structure elements
            END                  ! end header declaration
CustDetail1   DETAIL               ! begin detail declaration
              !structure elements
            END                  ! end detail declaration
```

```
CustDetail2  DETAIL                ! begin detail declaration
             !structure elements
           END                   ! end detail declaration
         END                     !End report declaration
  CODE
  OPEN(CustRpt)
  SET(SomeFile)
  LOOP
    NEXT(SomeFile)
      IF ERRORCODE() THEN BREAK.
    IF SomeCondition
      PRINT(CustDetail1)
    ELSE
      PRINT(CustDetail2)
    END
  END
  CLOSE(CustRpt)
```

See Also:

PRINT

FOOTER (page or group footer structure)

```
FOOTER ,AT( ) [,FONT( )] [,ABSOLUTE] [,PAGEBEFORE( )] [,PAGEAFTER( )]
       [,WITHPRIOR( )] [,WITHNEXT( )] [,ALONE] [,USE( )]
   controls
END
```

FOOTER	Declares a page or group footer structure.
AT	Specifies the size and location of the FOOTER.
FONT	Specifies the default font for all controls in this structure. If omitted, the REPORT´s FONT attribute (if present) is used, or else the printer´s default font is used.
ABSOLUTE	Declares the FOOTER prints at a fixed position relative to the page. Valid only on a FOOTER within a BREAK structure (page FOOTER position is always fixed).
PAGEBEFORE	Declares the FOOTER prints at the start of a new page, after normal page overflow actions. Valid only on a FOOTER within a BREAK structure.
PAGEAFTER	Declares the FOOTER prints, and then starts a new page by activating normal page overflow actions. Valid only on a FOOTER within a BREAK structure.
WITHPRIOR	Declares the FOOTER prints on the same page as the DETAIL, group HEADER, or FOOTER that immediately precedes it. Valid only on a FOOTER within a BREAK structure.
WITHNEXT	Declares the FOOTER prints on the same page as the DETAIL, group HEADER, or FOOTER that immediately follows it. Valid only on a FOOTER within a BREAK structure.
ALONE	Declares the (group) FOOTER structure must be printed on a page without FORM, (page) HEADER, or (page) FOOTER structures.
USE	A field equate label to reference the FOOTER structure in executable code.
controls	Report output control fields.

The **FOOTER** structure declares the output which prints at the end of each page or group. A FOOTER structure must be terminated with a period or END statement.

A FOOTER structure that is not within a BREAK structure is a page footer. Only one page FOOTER is allowed in a REPORT. The page FOOTER is automatically printed whenever a page break occurs, at the page-relative position specified by its AT attribute.

The BREAK structure defines a group break. It may contain its own HEADER, FOOTER, and DETAIL structures, and/or other nested BREAK structures. It may also contain multiple DETAIL structures. The HEADER and FOOTER structures that are within a BREAK structure are the group header and footer. They are automatically printed when the value in a specified group break variable changes, at the next position available in the detail print area (specified by the REPORT´s AT attribute). Only one FOOTER is allowed in a BREAK structure.

Example:

```
CustRpt   REPORT             !Declare customer report
          FOOTER           ! begin page FOOTER declaration
```

```
            !report controls
          END               ! end FOOTER declaration
Break1    BREAK(SomeVariable)
GroupDet    DETAIL
              !report controls
            END             ! end detail declaration
            FOOTER          ! begin group footer declaration
              !report controls
            END             ! end footer declaration
          END               ! end group break declaration
        END                 !End report declaration
```

FORM (page layout structure)

FORM ,**AT( )** [,**FONT( )**] [,**USE( )**]
controls
END

FORM	Declares a report structure which prints on each page.
AT	Specifies the size and location, relative to the top left corner of the page, of the FORM.
FONT	Specifies the default font for all controls in this report structure. If omitted, the REPORT´s FONT attribute (if present) is used, or else the printer´s default font is used.
USE	A field equate label to reference the FORM structure in executable code.
controls	Report output control fields.

FORM declares a report structure which prints on every page of the report (except pages containing DETAIL structures with the ALONE attribute). A FORM structure must be terminated with a period or END statement. Only one FORM is allowed in a REPORT structure. The FORM structure automatically prints during page overflow.

The printed output of the FORM is determined only once at the beginning of the report. The page positioning of the FORM does not affect the page positioning of any other report structure. Once printed, all other structures may "overwrite" the FORM. Therefore, FORM is most aften used to design pre-printed forms which are filled in by the subsequent HEADER, DETAIL, and FOOTER structures. It may also be used to generate "watermarks" or page border graphics.

Example:

```
CustRpt  REPORT          !Declare customer report
         FORM
            IMAGE(´LOGO.BMP´),AT(0,0,1200,1200),USE(?I1)
            STRING(@N3),AT(6000,500,500,500),PAGENO
         END
GroupDet DETAIL
            !report controls
         END
        END              !End report declaration
```

HEADER (page or group header structure)

```
HEADER ,AT( ) [,FONT( )] [,ABSOLUTE] [,PAGEBEFORE( )] [,PAGEAFTER( )]
       [,WITHPRIOR( )] [,WITHNEXT( )] [,ALONE] [,USE( )]
   controls
END
```

HEADER	Declares a page or group header structure.
AT	Specifies the size and location of the HEADER.
FONT	Specifies the default font for all controls in this structure. If omitted, the REPORT´s FONT attribute (if present) is used, or else the printer´s default font is used.
ABSOLUTE	Declares the HEADER prints at a fixed position relative to the page. Valid only on a HEADER within a BREAK structure (page HEADER position is always fixed).
PAGEBEFORE	Declares the HEADER prints at the start of a new page after normal page overflow actions. Valid only on a HEADER within a BREAK structure.
PAGEAFTER	Declares the HEADER prints, and then starts a new page by activating normal page overflow actions. Valid only on a HEADER within a BREAK structure.
WITHPRIOR	Declares the HEADER prints on the same page as the DETAIL, group HEADER, or FOOTER that immediately precedes it. Valid only on a HEADER within a BREAK structure.
WITHNEXT	Declares the HEADER prints on the same page as the DETAIL, group HEADER, or FOOTER that immediately follows it. Valid only on a HEADER within a BREAK structure.
ALONE	Declares the (group) HEADER structure must be printed on a page without FORM, (page) HEADER, or (page) FOOTER structures.
USE	A field equate label to reference the HEADER structure in executable code.
controls	Report output control fields.

The **HEADER** structure declares the output which prints at the beginning of each page or group. A HEADER structure must be terminated with a period or END statement.

A HEADER structure that is not within a BREAK structure is a page header. Only one page HEADER is allowed in a REPORT. The page HEADER is automatically printed whenever a page break occurs, at the page-relative position specified by its AT attribute.

The BREAK structure defines a group break. It may contain its own HEADER, FOOTER, and DETAIL structures, and/or other nested BREAK structures. It may also contain multiple DETAIL structures. The HEADER and FOOTER structures that are within a BREAK structure are the group header and footer. They are automatically printed when the value in a specified group break variable changes, at the next position available in the detail print area (specified by the REPORT´s AT attribute). Only one HEADER is allowed in a BREAK structure.

Example:

```
CustRpt  REPORT           !Declare customer report
         HEADER         ! begin page header declaration
           !report controls
         END            ! end header declaration
```

```
Break1   BREAK(SomeVariable)
           HEADER        ! begin group header declaration
             !report controls
           END           ! end header declaration
GroupDet   DETAIL
             !report controls
           END           ! end detail declaration
         END             ! end group break declaration
       END               !End report declaration
```

Print Structure Attributes

- ABSOLUTE (set fixed-position printing)
- ALONE (set to print without page header, footer, or form)
- AT (set print structure position and size)
- FONT (set print structure default font)
- PAGEAFTER (set page break after)
- PAGEBEFORE (set page break first)
- USE (set structure equate label)
- WITHNEXT (set widow elimination)
- WITHPRIOR (set orphan elimination)

ABSOLUTE (set fixed-position printing)

ABSOLUTE

The **ABSOLUTE** attribute ensures that the DETAIL, or group HEADER or FOOTER structure (contained within a BREAK structure), always prints at a fixed position on the page. When ABSOLUTE is present, the position specified by the *x* and *y* parameters of the structure´s AT attribute is relative to the top left corner of the page.

Example:

```
CustRpt     REPORT
            HEADER
              !structure elements
            END
CustDetail1   DETAIL
              !structure elements
            END
CustDetail2   DETAIL,ABSOLUTE           ! fixed position detail
              !structure elements
            END
            FOOTER
              !structure elements
            END
          END
```

ALONE (set to print without page header, footer, or form)

ALONE

The **ALONE** attribute specifies that the DETAIL, or group HEADER or FOOTER structure (contained within a BREAK structure), is to be printed on the page without any FORM, page HEADER or FOOTER (not within a BREAK structure). The normal use is for report title and grand total pages.

Example:

```
CustRpt   REPORT
TitlePage   DETAIL,ALONE          !Title page detail structure
            !structure elements
          END
CustDetail  DETAIL
            !structure elements
          END
          FOOTER
            !structure elements
          END
        END
```

AT (set print structure position and size)

AT([*x*] [,*y*] [,*width*] [,*height*])

AT	Defines the position and size at which the structure prints.
x	An integer constant or constant expression that specifies the horizontal position of the top left corner of the print structure.
y	An integer constant or constant expression that specifies the vertical position of the top left corner of the print structure.
width	An integer constant or constant expression that specifies the minumum width of the print structure.
height	An integer constant or constant expression that specifies the minimum height of the print structure.

The **AT** attribute on print structures performs two different functions, depending upon the structure on which it is placed.

When placed on a FORM, or page HEADER or FOOTER (not within a BREAK structure), the AT attribute defines the position and size on the page at which the structure is printed. The position specified by the *x* and *y* parameters is relative to the top left corner of the page.

When placed on a DETAIL, or group HEADER or FOOTER (contained within a BREAK structure) the print structure is printed according to the following rules (unless the ABSOLUTE attribute is also present):

The *width* and *height* parameters of the AT attribute specify the minimum print size of the structure.

The structure is actually printed at the next available position within the detail print area (specified by the REPORT´s AT attribute).

The position specified by the *x* and *y* parameters of the structure´s AT attribute is an offset from the next available print position within the detail print area.

The first print structure on the page is printed at the top left corner of the detail print area (at the offset specified by its AT attribute).

Next and subsequent print structures are printed relative to the ending position of the previous print structure:

If there is room to print the next structure beside the previous structure, it is printed there.

If not, it is printed below the previous.

The values contained in the *x*, *y*, *width*, and *height* parameters default to dialog units unless the THOUS, MM, or POINTS attribute is also present. Dialog units are defined as one-quarter the average character width by one-eighth the average character height. The size of a dialog unit is dependent upon the size of the default font for the report. This measurement is based on the font specified in the FONT attribute of the report, or the printer´s default font.

Example:

```
CustRpt     REPORT,AT(1000,2000,6500,7000),THOUS    !1" margins all around
            HEADER,AT(1000,1000,6500,1000)         !Page relative position
              !structure elements                !1" band across top of page
            END
CustDetail1   DETAIL,AT(0,0,6500,1000)             !Detail relative position
```

```
            !structure elements                !1" band across page
           END
CustDetail2  DETAIL,ABSOLUTE,AT(1000,8000,6500,1000)        !Page relative position
            !structure elements                  !1" band near page bottom
           END
           FOOTER,AT(1000,9000,6500,1000)         !Page relative position
            !structure elements                !1" band across page bottom
           END
          END
```

FONT (set print structure default font)

FONT([*typeface*] [,*size*] [,*color*] [,*style*]**)**

FONT	Specifies the default print font.
typeface	A string constant containing the name of the font. If omitted, the printer´s default font is used.
size	An integer constant containing the size (in points) of the font. If omitted, the printer´s default font size is used.
color	A LONG integer constant containing the red, green, and blue values for the color of the font in the low-order three bytes, or an EQUATE for a standard Windows color value. If omitted, black is used.
style	An integer constant or constant expression or EQUATE specifying the strike weight and style of the font. If omitted, the weight is normal.

The **FONT** attribute on FORM, DETAIL, HEADER, and FOOTER structures specifies the default print font for all controls in the structures that do not have a FONT attribute.

The *typeface* may name any font registered in the Windows system which the printer driver supports. This includes the TrueType fonts for most printers. The EQUATES.CLW file contains EQUATE values for standard *style* values. A *style* on the range zero (0) to one thousand (1000) specifies the strike weight of the font. You may add to that values that indicate italic, underline, or strikeout text. The following EQUATES are in EQUATES.CLW:

```
FONT:thin          EQUATE (100)
FONT:regular       EQUATE (400)
FONT:bold          EQUATE (700)
FONT:italic        EQUATE (01000H)
FONT:underline     EQUATE (02000H)
FONT:strikeout     EQUATE (04000H)
```

Example:

```
CustRpt   REPORT,FONT(´Arial´,12)           !Default font: 12 point Arial
          HEADER,FONT(´Arial´,18,,FONT:bold) !18 point bold Arial for the header
             !structure elements
          END
CustDetail1 DETAIL                        !Detail uses the default font
             !structure elements
          END
          FOOTER,FONT(´Arial´,12,00FF0000h)    !12 point Red Arial for the footer
             !structure elements
          END
        END
```

PAGEAFTER (set page break after)

PAGEAFTER([*newpage*] **)**

PAGEAFTER Specifies the structure is printed, then initiates page overflow.

newpage An integer constant or constant expression that specifies the page number to print on the next page. If omitted, the current page number is incremented during page overflow.

The **PAGEAFTER** attribute specifies that the DETAIL, or group HEADER or FOOTER structure (contained within a BREAK structure), initiates page overflow after it is printed. This means that the print structure on which the PAGEAFTER attribute is present is printed, followed by the page FOOTER, and then the FORM and page HEADER.

The *newpage* parameter, if present, resets automatic page numbering at the number specified.

Example:

```
CustRpt    REPORT
           HEADER
             !structure elements
           END
Break1     BREAK(SomeVariable)
             HEADER
               !structure elements
             END
CustDetail   DETAIL
             !structure elements
             END
             FOOTER,PAGEAFTER          !Group Footer, initiates page overflow
               !structure elements
             END
           END
           FOOTER
             !structure elements
           END
         END
```

PAGEBEFORE (set page break first)

PAGEBEFORE([*newpage*] **)**

PAGEBEFORE Specifies the structure is printed on a new page, after page overflow.

newpage An integer constant or constant expression that specifies the page number to print on the new page. If omitted, the current page number is incremented during page overflow.

The **PAGEBEFORE** attribute specifies that the DETAIL, or group HEADER or FOOTER structure (contained within a BREAK structure), is printed on a new page, after page overflow. This means that first, the page FOOTER is printed, then the FORM and page HEADER. The print structure on which the PAGEBEFORE attribute is present is printed only after these page overflow actions are complete.

The *newpage* parameter, if present, resets automatic page numbering at the number specified.

Example:

```
CustRpt   REPORT
          HEADER
            !structure elements
          END
Break1    BREAK(SomeVariable)
            HEADER,PAGERBEFORE     !Group Header, initiates page overflow
              !structure elements
            END
CustDetail    DETAIL
              !structure elements
            END
            FOOTER
              !structure elements
            END
          END
          FOOTER
            !structure elements
          END
        END
```

USE (set structure equate label)

USE(*label* [,*number*] **)**

USE	Specifies a field equate label for the structure.
label	A field equate label to reference the structure in executable code.
number	An integer constant that specifies the number the compiler equates to the field equate label for the structure.

The **USE** attribute on a FORM, DETAIL, HEADER, or FOOTER structure specifies a field equate label for the structure. This provides a mechanism for executable source code statements to reference the structure.

The print structures in a REPORT are treated just as controls in a WINDOW; they are automatically assigned positive numbers by the compiler.

The USE attribute´s *number* parameter allows you to specify the actual field number the compiler assigns to the structure. This *number* also is used as the new starting point for subsequent field equate numbering for all structures and controls without a *number* parameter in their USE attribute. Subsequent structures or controls without a *number* parameter in their USE attribute are incremented (or decremented) relative to the last *number* assigned.

Example:

```
BuildRpt PROCEDURE
CustRpt   REPORT
          HEADER,USE(?PageHeader)    !Page header
            !structure elements
          END
CustDetail  DETAIL,USE(?Detail)        !Line item detail
            !structure elements
          END       !
          FOOTER,USE(?PageFooter)      !Page footer
            !structure elements
          END
        END
  CODE
  PrintRpt(CustRpt,?Detail)          !Pass report and detail equate to print proc

PrintRpt PROCEDURE(RptToPrint,DetailNumber)
  CODE
  OPEN(RptToPrint)                !Open passed report
  PRINT(RptToPrint,DetailNumber)     !Print its detail
  CLOSE(RptToPrint)                !Close passed report
```

WITHNEXT (set widow elimination)

WITHNEXT([*siblings*] **)**

WITHNEXT Specifies the structure is always printed on the same page as print structures PRINTed immediately following it.

siblings An integer constant or constant expression that specifies the number of following print structures to print on the same page. If omitted, the default value is one.

The **WITHNEXT** attribute specifies that the DETAIL, or group HEADER or FOOTER structure (contained within a BREAK structure), is always printed on the same page as the specified number of print structures PRINTed immediately following it. This ensures that the structure is never printed on a page by itself, eliminating "widow" print structures. A "widow" print structure is defined as a group header, or first detail item in a related group of items, printed on the preceding page, separated from the rest of its related items.

The *siblings* parameter, if present, sets the number of following print structures that must be printed on the same page with the structure. To be counted, the following print structures must come from the same, or nested, BREAK structures. They must be related items. Any print structures not within the same, or nested, BREAK structures are printed but not counted as part of the required number of *siblings*.

Example:

```
CustRpt   REPORT
Break1    BREAK(SomeVariable)
            HEADER,WITHNEXT(2)      !Always print with 2 siblings
              !structure elements
            END
CustDetail    DETAIL,WITHNEXT()         !Always print with 1 sibling
              !structure elements
            END
            FOOTER
              !structure elements
            END
          END
        END
```

WITHPRIOR (set orphan elimination)

WITHPRIOR([*siblings*] **)**

WITHPRIOR Specifies the structure is always printed on the same page as print structures PRINTed immediately preceding it.

siblings An integer constant or constant expression that specifies the number of preceding print structures to print on the same page. If omitted, the default value is one.

The **WITHPRIOR** attribute specifies that the DETAIL, or group HEADER or FOOTER structure (contained within a BREAK structure), is always printed on the same page as the specified number of print structures PRINTed immediately preceding it. This ensures that the structure is never printed on a page by itself, eliminating "orphan" print structures. An "orphan" print structure is defined as a group footer, or last detail item in a related group of items, that is printed on the following page separated from the rest of its related items.

The *siblings* parameter, if present, sets the number of preceding print structures that must be printed on the same page with the structure. To be counted, the preceding print structures must come from the same, or nested, BREAK structures. They must be related items. Any print structures not within the same, or nested, BREAK structures are printed, but not counted as part of the required number of *siblings*.

Example:

```
CustRpt   REPORT
Break1    BREAK(SomeVariable)
            HEADER
             !structure elements
            END
CustDetail   DETAIL,WITHPRIOR()         !Always print with 1 sibling
             !structure elements
            END
            FOOTER,WITHPRIOR(2)           !Always print with 2 siblings
             !structure elements
            END
          END
        END
```

Report Controls

BOX (declare a report box control)

BOX ,**AT( )** [,**USE( )**] [,**COLOR( )**] [,**FILL( )**] [,**ROUND**] [,**HIDE**]

BOX	Places a rectangular box in the REPORT.
AT	Specifies the size and location of the control. If omitted, default values are selected by the runtime library.
USE	Specifies a field equate label for the control.
COLOR	Specifies the color for the border of the control. If omitted, the border is black.
FILL	Specifies the fill color for the control. If omitted, the box is not filled with color.
ROUND	Specifies the box corners are rounded. If omitted, the corners are square.
HIDE	Specifies the control is not printed unless UNHIDE is used to allow it to print.

The **BOX** control places a rectangular box in the REPORT at the position and size specified by its AT attribute, relative to the top left corner of the print structure containing the BOX.

Example:

```
CustRpt  REPORT,AT(1000,1000,6500,9000),THOUS
CustDetail  DETAIL,AT(0,0,6500,1000)
            BOX,AT(0,0,20,20),USE(?B1)        !Unfilled, black border
            BOX,AT(20,20,20,20),ROUND           !Unfilled, rounded, black border
            BOX,AT(40,40,20,20),FILL(COLOR:ACTIVEBORDER)
                                          !Filled, black border
            BOX,AT(60,60,20,20),COLOR(COLOR:ACTIVEBORDER)
                                        !Unfilled, active border color border
          END
        END
```

CHECK (declare a report checkbox control)

CHECK(*text*) ,**AT( )** [,**USE( )**] [,**FONT( )**] [,**HIDE**] [,**DISABLE**] [, | **LEFT** | **RIGHT** |]

CHECK	Places a check box in the REPORT.
text	A string constant containing the text to display for the check box.
AT	Specifies the size and location of the control. If omitted, default values are selected by the runtime library.
USE	The label of a numeric variable containing the value of the check box, zero (0 = OFF) or one (1 = ON).
FONT	Specifies the display font for the control.
HIDE	Specifies the control is not printed unless UNHIDE is used to allow it to print.
DISABLE	Specifies the control appears dimmed in the REPORT.
LEFT	Specifies that the text appears to the left of the check box.
RIGHT	Specifies that the text appears to the right of the check box (the default position).

The **CHECK** control places a check box in the REPORT at the position and size specified by its AT attribute, relative to the top left corner of the print structure containing the CHECK.

Example:

```
CustRpt   REPORT,AT(1000,1000,6500,9000),THOUS
CustDetail  DETAIL,AT(0,0,6500,1000)
            CHECK(´1´),AT(0,0,20,20),USE(C1)
            CHECK(´2´),AT(20,80,20,20),USE(C2),LEFT
            CHECK(´3´),AT(0,100,20,20),USE(C3),FONT(´Arial´,12)
          END
        END
```

CUSTOM (declare a report .VBX custom control)

CUSTOM(*text***) ,AT( ) [,CLASS( )] [,USE( )] [,DISABLE] [,FONT( )] [,META]**
[,*property(value)***]**

CUSTOM	Places a Visual Basic .VBX control on the REPORT.
text	A string constant containing the title for the control.
AT	Specifies the size and location of the control. If omitted, default values are selected by the library.
CLASS	Specifies the .VBX filename and type of control.
USE	The label of a variable to supply the value of the control.
DISABLE	Specifies the control appears dimmed in the REPORT.
FONT	Specifies the display font for the control.
META	Specifies printing as a Windows metafile (.WMF).
HIDE	Specifies the control is not printed unless UNHIDE is used to allow it to print.
property	A string constant containing the name of a custom property setting for the control.
value	A string constant containing the property value number or EQUATE for the *property*.

The **CUSTOM** control places a Visual Basic .VBX control in the report at the position and size specified by its AT attribute.

The *property* attribute allows you to specify any additional property settings the .VBX control may require. These are properties that need to be set for the .VBX control to properly function, and are not standard Clarion properties (such as AT or USE). The custom control should only receive values for these properties that are defined for that control. Valid properties and *values* for those properties would be defined in the custom control´s documentation. You may have multiple *property* attributes on a single CUSTOM control.

Example:

```
Report   REPORT
DetailOne   DETAIL
            CUSTOM,AT(0,0,120,320),CLASS(´graph.vbx´,´graph´),´graphstyle´(´2´)
          END
        END
```

ELLIPSE (declare a report ellipse control)

ELLIPSE ,**AT( )** [,**USE( )**] [,**COLOR( )**] [,**FILL( )**] [,**HIDE**]

ELLIPSE	Places a "circular" figure in the REPORT.
AT	Specifies the size and location of the control. If omitted, default values are selected by the runtime library.
USE	Specifies a field equate label for the control.
COLOR	Specifies the color for the border of the ellipse. If omitted, the ellipse has a black border.
FILL	Specifies the fill color for the control. If omitted, the ellipse is not filled with color.
HIDE	Specifies the control is not printed unless UNHIDE is used to allow it to print.

The **ELLIPSE** control places a "circular" figure in the REPORT at the position and size specified by its AT attribute. The ellipse is drawn inside a "bounding box" defined by the *x*, *y*, *width*, and *height* parameters of its AT attribute. The *x* and *y* parameters specify the starting point, relative to the top left corner of the print structure containing it, and the *width* and *height* parameters specify the horizontal and vertical size of the "bounding box."

Example:

```
CustRpt  REPORT,AT(1000,1000,6500,9000),THOUS
CustDetail  DETAIL,AT(0,0,6500,1000)
            ELLIPSE,AT(0,0,20,20)             !Unfilled, black border
            ELLIPSE,AT(0,20,20,20),USE(?Ellipse1),DISABLE
                                          !Unfilled, black border, dimmed
            ELLIPSE,AT(20,20,20,20),ROUND        !Unfilled, rounded, black border
            ELLIPSE,AT(40,40,20,20),FILL(COLOR:ACTIVEBORDER)
                                          !Filled, black border
            ELLIPSE,AT(60,60,20,20),COLOR(COLOR:ACTIVEBORDER)
                                       !Unfilled, active border color border
          END
         END
```

GROUP (declare a group of report controls)

GROUP(*text*) ,**AT( )** [,**USE( )**] [,**FONT( )**] [,**BOXED**] [,**HIDE**]
controls
END

GROUP	Declares a group of controls that may be referenced as one entity.
text	A string constant containing the prompt for the group of controls. The *text* is printed only if the BOXED attribute is also present.
AT	Specifies the size and location of the control. If omitted, default values are selected by the runtime library.
USE	Specifies a field equate label for the control.
FONT	Specifies the display font for the control and the default for all the controls in the GROUP.
BOXED	Specifies a single-track border around the group of controls with the text at the top of the border.
HIDE	Specifies the control is not printed unless UNHIDE is used to allow it to print.
controls	Control declarations that may be referenced as the GROUP.

The **GROUP** control declares a group of controls that may be referenced as one entity. This control allows you to design reports that look the same on paper as on the screen.

Example:

```
CustRpt   REPORT,AT(1000,1000,6500,9000),THOUS
CustDetail  DETAIL,AT(0,0,6500,1000)
            GROUP(´Group 1´),USE(!G1),AT(80,0,20,20),BOXED
              STRING(@S8),AT(80,0,20,20),USE(E5)
              STRING(@S8),AT(100,0,20,20),USE(E6)
            END
            GROUP(´Group 2´),USE(?G2),FONT(´Arial´,12)
              STRING(@S8),AT(120,0,20,20),USE(E7)
              STRING(@S8),AT(140,0,20,20),USE(E8)
            END
          END
        END
```

IMAGE (declare a report graphic image control)

IMAGE(*file***) ,AT( ) [,USE( )] [,HIDE]**

IMAGE	Places a graphic image on the REPORT.
file	A string constant containing the name of the file to print. The file is linked into the .EXE as a resource.
AT	Specifies the size and location of the control. If omitted, default values are selected by the runtime library.
USE	Specifies a field equate label for the control.
HIDE	Specifies the control is not printed unless UNHIDE is used to allow it to print.

The **IMAGE** control places a graphic image on the REPORT at the position and size specified by its AT attribute. This may be a bitmap (.BMP), icon (.ICO), PaintBrush (.PCX), Graphic Interchange Format (.GIF), JPEG (.JPG), or windows metafile (.WMF).

Example:

```
CustRpt   REPORT,AT(1000,1000,6500,9000),THOUS
CustDetail  DETAIL,AT(0,0,6500,1000)
            IMAGE(´PIC.BMP´),AT(0,0,20,20),USE(?I1)
            IMAGE(´PIC.WMF´),AT(40,0,20,20),USE(?I2)
            IMAGE(´PIC.ICO´),AT(60,0,20,20),USE(?I3)
          END
        END
```

LINE (declare a report line control)

LINE ,**AT( )** [,**USE( )**] [,**COLOR( )**] [,**HIDE**]

LINE	Places a straight line in the REPORT.
AT	Specifies the size and location of the control. If omitted, default values are selected by the runtime library.
USE	Specifies a field equate label for the control.
COLOR	Specifies the color for the line. If omitted, the color is black.
HIDE	Specifies the control is not printed unless UNHIDE is used to allow it to print.

The **LINE** control places a straight line in the REPORT at the position and size specified by its AT attribute.

The *x* and *y* parameters of the AT attribute specify the starting point of the line. The *width* and *height* parameters of the AT attribute specify the horizontal and vertical distance to the end point of the line. If these are both positive numbers, the line slopes to the right and down from its starting point. If the *width* parameter is negative, the line slopes left; if the *height* parameter is negative, the line slopes left. If either the *width* or *height* parameter is zero, the line is horizontal or vertical.

Width	Height	Result
positive	positive	right and down from start point
negative	positive	left and down from start point
positive	negative	right and up from start point
negative	negative	left and up from start point
zero	positive	vertical, down from start point
zero	negative	vertical, up from start point
positive	zero	horizontal, right from start point
negative	zero	horizontal, left from start point

Example:

```
CustRpt   REPORT,AT(1000,1000,6500,9000),THOUS
CustDetail  DETAIL,AT(0,0,6500,1000)
            LINE,AT(60,60,20,20),COLOR(COLOR:ACTIVEBORDER)   !Border color
            LINE,AT(480,180,20,20),USE(?L2)
          END
        END
```

LIST (declare a report list control)

```
LIST ,FROM( ) ,AT( ) [,FONT( )] [,USE( )] [,HIDE] [,   | FORMAT( )   | ]
                                                       | LEFT        |
                                                       | RIGHT       |
                                                       | CENTER      |
                                                       | DECIMAL     |
```

LIST	Places the current item of a list of data items in the REPORT.
FROM	Specifies the origin of the data displayed in the list.
AT	Specifies the size and location of the control. If omitted, the runtime library chooses a value.
FONT	Specifies the display font for the control.
USE	Specifies a field equate label for the control.
HIDE	Specifies the control is not printed unless UNHIDE is used to allow it to print.
FORMAT	Specifies the print format of the data.
LEFT	Specifies that the data is left justified within the LIST.
RIGHT	Specifies that the data is right justified within the LIST.
CENTER	Specifies that the data is centered within the LIST.
DECIMAL	Specifies that the data is aligned on the decimal point within the LIST.

The **LIST** control places the current item of a list of data items in the REPORT at the position and size specified by its AT attribute. LIST is valid only in a DETAIL structure. Its purpose is to allow the report format to duplicate the screen appearance of the LIST´s FORMAT setting. When the first instance of the DETAIL structure containing the LIST is printed, any headers in the FORMAT attribute are printed along with the current FROM attribute entry. When the last DETAIL structure containing the LIST is printed, the LIST footers are printed along with the current FROM attribute entry.

Example:

```
Q        QUEUE
F1         STRING(1)
F2         STRING(4)
         END

CustRpt   REPORT,AT(1000,1000,6500,9000),THOUS
CustDetail  DETAIL,AT(0,0,6500,1000)
              LIST,AT(80,0,20,20),USE(?L1),FROM(Q),FORMAT(´5C~List~15L~Box~´)
            END
         END
```

OPTION (declare a group of report RADIO controls)

OPTION(*text***) ,AT( ) [,USE( )] [,BOXED] [,HIDE]**
radios
END

OPTION	Prints a group of RADIO controls.
text	A string constant containing the prompt for the group of controls. The *text* is printed only if the BOXED attribute is also present.
AT	Specifies the size and location of the control. If omitted, default values are selected by the runtime library.
USE	The label of a string variable containing the value of the RADIO selected by the user.
BOXED	Specifies a single-track border around the RADIO controls with the text at the top of the border.
HIDE	Specifies the control is not printed unless UNHIDE is used to allow it to print.
radios	Multiple RADIO control declarations.

The **OPTION** control prints a group of RADIO controls that display a list of choices. The multiple RADIO controls in the OPTION structure define the choices. The selected choice is identified by a filled RADIO button.

No RADIO button selected is a valid option. This occurs only when the OPTION structure´s USE variable does not contain a value duplicated in a RADIO *text* parameter.

Example:

```
CustRpt   REPORT,AT(1000,1000,6500,9000),THOUS
CustDetail  DETAIL,AT(0,0,6500,1000)
             OPTION(´Option´),USE(OptVar),AT(80,0,20,20),BOXED
               RADIO(´Radio 1´),AT(80,0,20,20),USE(?R1)
               RADIO(´Radio 2´),AT(100,0,20,20),USE(?R2)
              END
            END
           END
```

RADIO (declare a report radio button control)

RADIO(*text*) ,**AT( )** [,**FONT( )**] [, | **LEFT** |] [,**USE( )**] [,**HIDE**]
| **RIGHT** |

RADIO	Places a radio button in the REPORT.
text	A string constant containing the text to display for the radio button.
AT	Specifies the size and location of the control. If omitted, default values are selected by the runtime library.
FONT	Specifies the display font for the control.
LEFT	Specifies that the text appears to the left of the radio button.
RIGHT	Specifies that the text appears to the right of the radio button (the default position).
USE	Specifies a field equate label for the control.
HIDE	Specifies the control is not printed unless UNHIDE is used to allow it to print.

The **RADIO** control places a radio button in the REPORT at the position and size specified by its AT attribute. A RADIO control may only be placed within an OPTION control. The RADIO selected by the user (the value in the OPTION´s USE variable) is displayed as a filled RADIO button.

Example:

```
CustRpt   REPORT,AT(1000,1000,6500,9000),THOUS
CustDetail  DETAIL,AT(0,0,6500,1000)
            OPTION(´Option´),USE(OptVar),AT(80,0,20,20),BOXED
              RADIO(´Radio 1´),AT(80,0,20,20),USE(?R1)
              RADIO(´Radio 2´),AT(100,0,20,20),USE(?R2)
              RADIO(´Radio 3´),AT(100,0,20,20),USE(?R2),LEFT
            END
          END
        END
```

STRING (declare a report string control)

```
STRING(text),AT( ) [,FONT( )] [,HIDE] [,TRN] [,USE( )]
        [,  | LEFT       | ] [,   | PAGENO                      | ]
            | RIGHT      |        | CNT    [, RESET( ) / PAGE ] |
            | CENTER     |        | SUM    [, RESET( ) / PAGE ] |
            | DECIMAL    |        | AVE    [, RESET( ) / PAGE ] |
                                    | MIN    [, RESET( ) / PAGE ] |
                                    | MAX    [, RESET( ) / PAGE ] |
```

STRING	Places the *text* in the REPORT.
text	A string constant containing the text to display, or a display picture token to format the variable specified in the USE attribute.
AT	Specifies the size and location of the control. If omitted, default values are selected by the runtime library.
FONT	Specifies the font used to display the text.
HIDE	Specifies the control is not printed unless UNHIDE is used to allow it to print.
TRN	Specifies the text or USE variable characters transparently print over the background.
USE	Specifies a variable whose contents are printed in the format of the picture token declared instead of string text.
LEFT	Specifies that the text is left justified within the area specified by the AT attribute.
RIGHT	Specifies that the text is right justified within the area specified by the AT attribute.
CENTER	Specifies that the text is centered within the area specified by the AT attribute.
DECIMAL	Specifies that the text is aligned on the decimal point within the area specified by the AT attribute.
PAGENO	Specifies the current page number is printed in the format of the picture token declared instead of string text.
CNT	Specifies the number of details printed is printed in the format of the picture token declared instead of string text.
SUM	Specifies the sum of the USE variable is printed in the format of the picture token declared instead of string text.
AVE	Specifies the average value of the USE variable is printed in the format of the picture token declared instead of string text.
MIN	Specifies the mimimum value of the USE variable is printed in the format of the picture token declared instead of string text.
MAX	Specifies the maximum value of the USE variable is printed in the format of the picture token declared instead of string text.
RESET	Specifies the CNT, SUM, AVE, MIN, or MAX is reset when the specified group break occurs.
PAGE	Specifies the CNT, SUM, AVE, MIN, or MAX is reset to zero when the page break occurs.

The **STRING** control places the *text* in the REPORT at the position and size specified by its AT attribute. If the *text* parameter is a picture token instead of a string constant or variable, the contents of the variable

named in the USE attribute are formatted to that display picture, at the position and size specified by the AT attribute.

A STRING with the TRN attribute prints characters transparently, without obliterating the background. This means only the dots required to create each character are printed. This allows the STRING to be placed directly on top of an IMAGE without destroying the background picture.

Example:

```
CustRpt  REPORT,AT(1000,1000,6500,9000),THOUS
Break1   BREAK(Pre:Key1)
           HEADER,AT(0,0,6500,1000)
             STRING(´Group Head´),AT(3000,500,1500,500),FONT(´Arial´,18)
           END
Detail     DETAIL,AT(0,0,6500,1000)
             STRING(@N$11.2´),AT(6000,1500,500,500),USE(Pre:F1)
           END
           FOOTER,AT(0,0,6500,1000)
             STRING(´Group Total:´),AT(5500,500,1500,500)
             STRING(@N$11.2´),AT(6000,500,500,500),USE(Pre:F1),SUM,RESET(Pre:Key1)
           END
         END
       END
```

TEXT (declare a multi-line text control)

```
TEXT ,AT( ) [,USE( )] ,FONT( )] [,   | CAP | ] [,   | LEFT   | ]  [,HIDE]
                                     | UPR |        | RIGHT  |
                                                   | CENTER |
```

TEXT	Places a multi-line print field in the REPORT.
AT	Specifies the size and location of the control. If omitted, default values are selected by the runtime library.
USE	The label of the variable that contains the value to print.
FONT	Specifies the display font for the control.
UPR / CAP	Specifies all upper case or proper name capitalization (First Letter Of Each Word Capitalized).
LEFT	Specifies that the text is left justified within the area specified by the AT attribute.
RIGHT	Specifies that the text is right justified within the area specified by the AT attribute.
CENTER	Specifies that the text is centered within the area specified by the AT attribute.
HIDE	Specifies the control is not printed unless UNHIDE is used to allow it to print.

The **TEXT** control places a multi-line print field in the REPORT at the position and size specified by its AT attribute. The variable specified in the USE attribute contains the data to print.

Example:

```
CustRpt   REPORT,AT(1000,1000,6500,9000),THOUS
Detail    DETAIL,AT(0,0,6500,1000)
            TEXT,AT(0,0,40,40),USE(E1)
            TEXT,AT(100,0,40,40),USE(E6),FONT(´Arial´,12)
            TEXT,AT(120,0,40,40),USE(E7),CAP
            TEXT,AT(140,0,40,40),USE(E8),UPR
            TEXT,AT(160,0,40,40),USE(E9),LEFT
            TEXT,AT(180,0,40,40),USE(E10),RIGHT
            TEXT,AT(200,0,40,40),USE(E11),CENTER
          END
        END
```

Control Attributes

AT (set control position and size in report)

AT([*x*] [,*y*] [,*width*] [,*height*])

AT	Defines the position and size of a control.
x	An integer constant or constant expression that specifies the initial horizontal position of the top left corner of the control, relative to the top left corner of the print structure containing it. If omitted, the runtime library provides a default value.
y	An integer constant or constant expression that specifies the initial vertical position of the top left corner of the control, relative to the top left corner of the print structure containing it. If omitted, the runtime library provides a default value.
width	An integer constant or constant expression that specifies the width of the control. If omitted, the runtime library provides a default value.
height	An integer constant or constant expression that specifies the height of the control. If omitted, the runtime library provides a default value.

The **AT** attribute defines the position and size of a control, relative to the top left corner of the print structure containing it. If any parameter is omitted, the runtime library provides a default value.

The values contained in the *x*, *y*, *width*, and *height* parameters default to dialog units unless the THOUS, MM, or POINTS attribute is also present. Dialog units are defined as one-quarter the average character width by one-eighth the average character height. The size of a dialog unit is dependent upon the size of the default font for the report. This measurement is based on the font specified in the FONT attribute of the REPORT, or the printer´s default font.

Example:

```
CustRpt   REPORT,AT(1000,1000,6500,9000),THOUS   !AT specifies detail print area
Detail    DETAIL,AT(0,0,6500,1000)            !AT specifies band size and
                                             ! relative position offset from
                                             ! last printed detail
            STRING(´String Constant´),AT(500,500,1500,500)
                                             !AT specifies control size and
                                             ! offset within the detail band
          END
        END
```

AVE (set total average)

AVE

The **AVE** attribute specifies the average (arithmetic mean) of the STRING controls´ USE variable is printed.

An AVE field in a DETAIL structure is calculated each time the DETAIL structure containing the control is PRINTed.

An AVE field in a group FOOTER structure is calculated each time any DETAIL structure in the BREAK structure containing the control is PRINTed.

An AVE field in a page FOOTER structure is calculated each time any DETAIL structure in any BREAK structure is PRINTed.

An AVE field in a HEADER is meaningless, since no DETAIL structures will have been printed at the time the HEADER is printed.

The average is reset only if the RESET or PAGE attribute is also specified. The STRING control using this attribute would usually be placed in a group or page FOOTER.

BOXED (set report controls group border)

BOXED

The **BOXED** attribute specifies a single-track border around a GROUP or OPTION structure. The *text* parameter of the GROUP or OPTION control appears in a gap at the top of the border box. If BOXED is omitted, the *text* parameter of the GROUP or OPTION control is not printed.

CAP, UPR (set print case)

CAP
UPR

The **CAP** and **UPR** attributes specify the automatic case of text printed in a TEXT control. UPR specifies all upper case; CAP specifies "Proper Name Capitalization," where the first letter of each word is capitalized and all other letters are lower case.

CNT (set total count)

CNT

The **CNT** attribute specifies an automatic count of the number of times DETAIL structures have been printed.

A CNT field in a DETAIL structure is incremented each time the DETAIL structure containing the control is PRINTed. This provides a "running" count.

A CNT field in a group FOOTER structure is incremented each time any DETAIL structure in the BREAK structure containing the control is PRINTed. This provides a total of the number of DETAIL structures printed in the group.

A CNT field in a page FOOTER structure is incremented each time any DETAIL structure in any BREAK structure is PRINTed. This provides a total of the number of DETAIL structures printed on the page (or report).

A CNT field in a HEADER is meaningless, since no DETAIL structures will have been printed at the time the HEADER is printed.

The CNT is reset only if the RESET or PAGE attribute is also specified.

COLOR (set color)

COLOR(*rgb***)**

COLOR	Specifies the print color of a BOX, LINE, or ELLIPSE control.
rgb	A LONG or ULONG integer constant containing the red, green, and blue components that create the color in the three low-order bytes (bytes 0, 1, and 2), or an EQUATE for a standard Windows color value.

The **COLOR** attribute specifies the print color of a BOX, LINE, or ELLIPSE control. On a BOX or ELLIPSE, the color specified is the color used for the border. EQUATEs for Windows´ standard colors are contained in the EQUATES.CLW file.

Example:

```
CustRpt   REPORT,AT(1000,1000,6500,9000),THOUS
CustDetail  DETAIL,AT(0,0,6500,1000)
              ELLIPSE,AT(60,60,200,200),COLOR(COLOR:ACTIVEBORDER) !Color EQUATE
              BOX,AT(360,60,200,200),COLOR(00FF0000h) !Pure Red
            END
          END
```

FILL (set print fill color)

FILL(*rgb*)

FILL	Specifies the print fill color of a BOX or ELLIPSE control.
rgb	A LONG or ULONG integer constant containing the red, green, and blue components that create the color in the three low-order bytes (bytes 0, 1, and 2) or an EQUATE for a standard Windows color value.

The **FILL** attribute specifies the print fill color of a BOX or ELLIPSE control. If omitted, the control is not filled with color.

Example:

```
CustRpt   REPORT,AT(1000,1000,6500,9000),THOUS
CustDetail  DETAIL,AT(0,0,6500,1000)
            ELLIPSE,AT(60,60,200,200),FILL(COLOR:ACTIVEBORDER)
                                                !Color EQUATE
            BOX,AT(360,60,200,200),FILL(00FF0000h)  !Pure Red
          END
        END
```

FONT (set default font)

FONT([*typeface*] [,*size*] [,*color*] [,*style*]**)**

FONT	Specifies the print font for the control.
typeface	A string constant containing the name of the font. If omitted, the print structure´s FONT attribute is used (if present), or the REPORT structure´s FONT attribute is used (if present), or else the printer´s default font is used.
size	An integer constant containing the size (in points) of the font. If omitted, the printer´s default font size is used.
color	A LONG integer constant containing the red, green, and blue values for the color of the font in the low-order three bytes, or an EQUATE for a standard Windows color value. If omitted, black is used.
style	An integer constant, constant expression, or EQUATE specifying the strike weight and style of the font. If omitted, the weight is normal.

The **FONT** attribute specifies the print font for the control, overriding any FONT specified on the REPORT or print structure.

The *typeface* may name any font registered in the Windows system which the printer driver supports. This includes the TrueType fonts for most printers. The EQUATES.CLW file contains EQUATE values for standard *style* values. A *style* on the range zero (0) to one thousand (1000) specifies the strike weight of the font. You may add to that values that indicate italic, underline, or strikeout text. The following EQUATES are in EQUATES.CLW:

```
FONT:thin                EQUATE (100)
FONT:regular             EQUATE (400)
FONT:bold                EQUATE (700)
FONT:italic              EQUATE (01000H)
FONT:underline           EQUATE (02000H)
FONT:strikeout           EQUATE (04000H)
```

Example:

```
CustRpt   REPORT,AT(1000,1000,6500,9000),THOUS
CustDetail  DETAIL,AT(0,0,6500,1000)
              STRING(´Text´),AT(0,0),FONT(´Arial´,14,00FF0000h)
              STRING(´Text´),AT(160,160),FONT(´Arial´,12,,FONT:italic)
            END
          END
```

FORMAT (set LIST print format)

FORMAT(*format string***)**

FORMAT Specifies the print format for the data.

format string A string constant specifying the column or multi-column print format.

The **FORMAT** attribute specifies the print format for the data in the LIST control. The *format string* contains the information for single or multi-column formatting of the data.

The *format string* contains "field-specifiers" which map to the fields of the QUEUE. Multiple "field-specifiers" may be grouped together as a "field-group" in square brackets (**[ ]**) to display as a single unit.

Only the fields in the QUEUE for which there are "field-specifiers" are printed. This means that, if there are two fields specified in the *format string* and three fields in the QUEUE, only the two specified in the *format string* are printed in the LIST control.

Field-specifier" format: *width justification* [**(***indent***)**] [*modifiers*]

width A required integer defining the width of the field. Specified in dialog units unless overridden by the THOUS, MM, or POINTS attribute.

justification A single capital letter (**L** , **R** , **C** , or **D**) that specifies **L**eft, **R**ight, **C**enter, or **D**ecimal justification. One is required.

indent An optional integer, enclosed in parentheses, that specifies the indent from the justification. This may be negative. With left (**L**) justification, *indent* defines a left margin; with right (**R**) or decimal (**D**), it defines a right margin; and with center (**C**), it defines an indent from the center of the field.

modifiers: Optional special characters (listed below) to modify the print format of the field or group. Multiple *modifiers* may be used on one field or group.

~header~ [justification [(indent)]]
A header string enclosed in tildes, followed by optional justification and/or indent, prints the header at the top of the list. The header uses the same justification and indent as the field, if not specifically overidden.

@picture@ The *picture* formats the field for printing. The trailing @ is required to define the end of the *picture*, so that display pictures like @N12~Kr~ can be used in the format string without creating ambiguity.

#number# The *number* enclosed in pound signs (#) indicates the QUEUE field to print. Following fields in the format string without an explicit #*number*# are taken in order from the fields following the #*number*# field. For example, #2# on the first field in the format string indicates starting with the second field in the QUEUE, skipping the first. If the number of fields specified in the format string are >= the number of fields in the QUEUE, the format "wraps around" to the start of the QUEUE.

_ An underscore underlines the field.

/ A slash causes the next field to appear on a new line (only used on a field within a group).

| A vertical bar places a vertical line to the right of the field.

"Field-group" format: **[** *multiple field-specifiers* **]** [**(***size***)**] [*modifiers*]

multiple field-specifiers
: A list of field-specifiers contained in square brackets (**[ ]**) that cause them to be treated as a single display unit.

size
: An optional integer, enclosed in parentheses, that specifies the default width of the group. If omitted, the size is calculated from the enclosed fields.

modifiers
: The "field-group" *modifiers* act on the entire group of fields. These are the same *modifiers* listed above.

Example:

```
TD       QUEUE,AUTO
FName       STRING(20)
LName       STRING(20)
Init      STRING(4)
Wage      REAL
         END
CustRpt   REPORT,AT(1000,1000,6500,9000),THOUS
CustDetail  DETAIL,AT(0,0,6500,1000)
            LIST,AT(0,34,366,146),FORMAT(´´),FROM(TD),USE(?Show)
           END
         END
  CODE
  OPEN(CustRpt)
  SETTARGET(CustRpt)
  IF SomeCondition
    ?Show{PROP:format} = ´80C~First Name~80C~Last Name~16L~Intls~60R~Wage~|´
  ELSE
    ?Show{PROP:format} = ´80C~First Name~80C~Last Name~16L~Intls~60D(10)~Wage~|´
  END
```

FROM (set report listbox data source)

FROM(*source*)

FROM Specifies the source of the data printed in a LIST control.

source The label of a QUEUE, or any variable (normally a GROUP) containing the data items to print in the LIST.

The **FROM** attribute specifies the source of the data elements printed in a LIST control. The data elements are formatted for display according to the information in the FORMAT attribute.

If the label of a QUEUE is specified as the *source*, all fields in the QUEUE are printed. If the label of one field in a QUEUE is specified as the *source*, only that field is printed. Only the current QUEUE entry in the queue´s data buffer is printed in the LIST.

If a string constant or variable is specified as the *source*, the entire string is printed in the LIST.

Example:

```
TD       QUEUE,AUTO
FName       STRING(20)
LName       STRING(20)
Init      STRING(4)
Wage      REAL
         END
CustRpt   REPORT,AT(1000,1000,6500,9000),THOUS
CustDetail  DETAIL,AT(0,0,6500,1000)
            LIST,AT(0,34,366,146),FORMAT(´80L80L16L60L´),FROM(TD),USE(?Show1)
            LIST,AT(0,200,100,146),FORMAT(´80L´),FROM(Fname),USE(?Show2)
          END
         END
```

HIDE (set control non-print)

HIDE

The **HIDE** attribute specifies the control does not print unless the UNHIDE statement is used to allow it to print.

LEFT, RIGHT, CENTER, DECIMAL (set print justification)

LEFT([*indent*] **)**
RIGHT([*indent*] **)**
CENTER([*indent*] **)**
DECIMAL([*indent*] **)**

---

indent An integer constant specifying the amount of margin left after justification. This is in dialog units unless overridden by the THOUS, MM, or POINTS attribute.

The **LEFT**, **RIGHT**, **CENTER**, and **DECIMAL** attributes specify the justification of data printed. LEFT specifies left justification, RIGHT specifies right justification, CENTER specifies centered text, and DECIMAL specifies numeric data aligned on the decimal point.

The *indent* parameter on the CENTER attribute specifies an offset from the center. On the DECIMAL attribute, *indent* specifies the position of the decimal point.

The following controls allow LEFT or RIGHT only (without an *indent* parameter):

CHECK
GROUP
OPTION
RADIO

The following controls allow LEFT(*indent*), RIGHT(*indent*), CENTER(*indent*), or DECIMAL(*indent*):

LIST
STRING

The TEXT control allows LEFT, RIGHT, and CENTER (without an *indent* parameter).

Example:

```
Rpt  REPORT,AT(1000,1000,6500,9000),THOUS
Detail DETAIL,AT(0,0,6500,1000)
         LIST,AT(0,20,100,146),FORMAT(´800L´),FROM(Fname),USE(?Show2),LEFT(100)
       END
     END
```

MAX (set total maximum)

MAX

The **MAX** attribute specifies printing the maximum value the STRING control´s USE variable has contained so far.

A MAX field in a DETAIL structure is evaluated each time the DETAIL structure containing the control is PRINTed. This provides a "running" maximum value.

A MAX field in a group FOOTER structure is evaluated each time any DETAIL structure in the BREAK structure containing the control is PRINTed. This provides the maximum value of the variable in the group.

A MAX field in a page FOOTER structure is evaluated each time any DETAIL structure in any BREAK structure is PRINTed. This is the maximum value of the variable in the page (or report to date).

A MAX field in a HEADER is meaningless, since no DETAIL structures will have been printed at the time the HEADER is printed.

The MAX value is reset only if the RESET or PAGE attribute is also specified.

META (set .VBX to print as .WMF)

META

The **META** attribute specifies printing a .VBX custom control as a .WMF windows metafile. This will print the control as a graphic image on the report.

MIN (set total minimum)

MIN

The **MIN** attribute specifies printing the minimum value the STRING control´s USE variable has contained so far.

A MIN field in a DETAIL structure is evaluated each time the DETAIL structure containing the control is PRINTed. This provides a "running" minimum value.

A MIN field in a group FOOTER structure is evaluated each time any DETAIL structure in the BREAK structure containing the control is PRINTed. This provides the minimum value of the variable in the group.

A MIN field in a page FOOTER structure is evaluated each time any DETAIL structure in any BREAK structure is PRINTed. This is the minimum value of the variable in the page (or report to date).

A MIN field in a HEADER is meaningless, since no DETAIL structures will have been printed at the time the HEADER is printed.

The MIN value is reset only if the RESET or PAGE attribute is also specified.

PAGE (set page total reset)

PAGE

The **PAGE** attribute specifies the CNT, SUM, AVE, MIN, or MAX is reset to zero (0) when page break occurs.

PAGENO (set page number print)

PAGENO

The **PAGENO** attribute specifies the STRING control prints the current page number.

RESET (set total reset)

RESET(*breaklevel***)**

RESET Resets the CNT, SUM, AVE, MIN, or MAX to zero (0).

breaklevel The label of a BREAK structure.

The **RESET** attribute specifies the group break at which the CNT, SUM, AVE, MIN, or MAX is reset to zero (0).

Example:

```
CustRpt  REPORT,AT(1000,1000,6500,9000),THOUS
Break1   BREAK(Pre:Key1)
           HEADER,AT(0,0,6500,1000)
             STRING(´Group Head´),AT(3000,500,1500,500),FONT(´Arial´,18)
           END
Detail     DETAIL,AT(0,0,6500,1000)
             STRING(@N$11.2´),AT(6000,1500,500,500),USE(Pre:F1)
           END
           FOOTER,AT(0,0,6500,1000)
             STRING(´Group Total:´),AT(5500,500,1500,500)
             STRING(@N$11.2´),AT(6000,500,500,500),USE(Pre:F1),SUM,RESET(Break1)
           END
         END
       END
```

ROUND (set round-cornered report BOX)

ROUND

The **ROUND** attribute specifies a BOX control with rounded corners.

SUM (set total)

SUM

The **SUM** attribute specifies printing the sum of the values contained in the STRING control´s USE variable.

A SUM field in a DETAIL structure is incremented each time the DETAIL structure containing the control is PRINTed. This provides a "running" total.

A SUM field in a group FOOTER structure is incremented each time any DETAIL structure in the BREAK structure containing the control is PRINTed. This provides the sum of the value contained in the variable in the group.

A SUM field in a page FOOTER structure is incremented each time any DETAIL structure in any BREAK structure is PRINTed. This is the sum of the values contained in the variable in the page.

A SUM field in a HEADER is meaningless, since no DETAIL structures will have been printed at the time the HEADER is printed.

The SUM value is reset only if the RESET or PAGE attribute is also specified.

TRN (set transparent report string)

TRN

The **TRN** attribute on a STRING control specifies the characters print transparently, without obliterating the background over which the STRING is placed. Only the dots required to create each character are printed. This allows the STRING to be placed directly on top of an IMAGE without destroying the background picture.

Example:

```
CustRpt  REPORT,AT(1000,1000,6500,9000),THOUS
         FORM,AT(0,0,6500,9000)
           IMAGE(´PIC.BMP´),USE(?I1)AT(0,0,6500,9000)                    !Full page
image
           STRING(´String Constant´),AT(10,0,20,20),USE(?S1),TRN
                                          !Transparent string on the image
         END
        END
```

USE (set code reference name)

USE(*variable* [,*number*] **)**

USE	Specifies the data to print in the control or a field equate label with which to reference the control.
variable	The label of a variable or a field equate label.
number	An integer constant that specifies the number the compiler equates to the field equate label for the control.

The **USE** attribute specifies the *variable* containing the data to print in the control or a field equate label with which to reference the control.

All controls in a REPORT are automatically assigned numbers by the compiler. Print structures receive a number if assigned a USE variable. These numbers start at one (1) and increment by one (1) for each control in the REPORT.

The USE attribute´s *number* parameter allows you to specify the actual field number the compiler assigns to the control. This *number* also is used as the new starting point for subsequent field numbering for fields without a *number* parameter in their USE attribute. Subsequent controls without a *number* parameter in their USE attribute are incremented relative to the last *number* assigned.

Two or more controls with exactly the same USE variable in a REPORT structure would create the same Field Equate Label for all, therefore, when the compiler encounters this condition, all Field Equate Labels for that USE variable are discarded. This makes it impossible to reference any of these controls in executable code, preventing confusion about which control you really want to reference. It also allows you to deliberately create this condition to print the contents of the variable in multiple controls.

Example:

```
CustRpt   REPORT,AT(1000,1000,6500,9000),THOUS
Detail    DETAIL,AT(0,0,6500,1000)
            STRING(´Group Total:´).AT(5500,500,1500,500),USE(?Constant)
                                                        !Field equate label
            STRING(@N$11.2´),AT(6000,1500,500,500),USE(Pre:F1)
                                                        !USE variable
          END
        END
```

Report Procedures

- CLOSE (close an active report structure)
- ENDPAGE (force page overflow)
- OPEN (open a report structure for processing)
- PRINT (print a report structure)

CLOSE (close an active report structure)

CLOSE(*report***)**

CLOSE Deactivates a REPORT structure.

report The label of a REPORT structure.

CLOSE prints the last page FOOTER, (unless the last structure printed has the ALONE attribute), and closes the REPORT. If the REPORT has the PREVIEW attribute, all the temporary metafiles are deleted.

RETURN from a procedure in which a REPORT is opened automatically closes the REPORT.

Example:

```
CLOSE(CustRpt)      !Close the report
```

ENDPAGE (force page overflow)

ENDPAGE(*report*)

ENDPAGE Forces page overflow.

report The label of a REPORT structure.

The **ENDPAGE** statement initiates page overflow and flushes the print engine´s print structure buffer. If the REPORT has the PREVIEW attribute, this has the effect of ensuring that the entire report is available to view.

Example:

```
SomeReport  PROCEDURE
WMFQue      QUEUE                    !Queue to contain .WMF filenames
              STRING(64)
            END
NextEntry     BYTE(1)                  !Queue entry counter variable
Report      REPORT,PREVIEW(WMFQue)     !Report with PREVIEW attribute
DetailOne       DETAIL
                !Report controls
            . .
ViewReport  WINDOW(´View Report´),AT(0,0,320,200),MDI,MAX,HVSCROLL
              IMAGE(´´),AT(0,0,320,180),USE(?ImageField)
              BUTTON(´View Next Page´),AT(0,180,60,20),USE(?NextPage),DEFAULT
              BUTTON(´Print Report´),AT(80,180,60,20),USE(?PrintReport)
              BUTTON(´Exit Without Printing´),AT(160,180,60,20),USE(?ExitReport)
            END
  CODE
  OPEN(Report)
  SET(SomeFile)                      !Code to generate the report
  LOOP
    NEXT(SomeFile)
    PRINT(DetailOne)
  END
  ENDPAGE(Report)                      !Flush the buffer
  OPEN(ViewReport)                   !Open report preview window
  GET(WMFQue,NextEntry)                !Get first queue entry
  ?ImageField{PROP:text} = WMFQue     !Load first report page
  ACCEPT
    CASE ACCEPTED()
    OF ?NextPage
      NextEntry += 1                             !Increment entry counter
      IF NextEntry > RECORDS(WMFQue) THEN CYCLE.   !Check for end of report
      GET(WMFQue,NextEntry)                       !Get next queue entry
      ?ImageField{PROP:text} = WMFQue                !Load next report page
      DISPLAY                                    ! and display it
    OF ?PrintReport
      Report{PROP:flushpreview} = ON               !Flush files to printer
      BREAK                                      ! and exit procedure
    OF ?ExitReport
      BREAK                                      !Exit procedure
  . .
  CLOSE(ViewReport)              !Close window
  FREE(WMFQue)                 !Free the queue memory
  CLOSE(Report)                !Close report (deleting all .WMF files)
  RETURN                     ! and return to caller
```

See Also:

Page Overflow

PREVIEW

OPEN (open a report structure for processing)

OPEN(*report***)**

OPEN Activates a REPORT structure.

report The label of a REPORT structure.

OPEN activates a REPORT structure. You must OPEN a REPORT before any of the structures may be printed.

Example:

```
OPEN(CustRpt)      !Open the report
```

PRINT (print a report structure)

PRINT(| *structure* | **)**
| *report* ,*number* |

PRINT Prints a report DETAIL, HEADER, or FOOTER structure.

structure The label of a DETAIL structure.

report The label of a REPORT structure.

number The number or EQUATE label of a report structure to print (only valid with a *report* parameter).

The **PRINT** statement prints a report structure to the destination specified by the user in the Windows Print... dialog. PRINT automatically activates group breaks and page overflow as needed.

Example:

```
BuildRpt PROCEDURE
CustRpt    REPORT
           HEADER,USE(?PageHeader)    !Page header
             !structure elements
           END
CustDetail  DETAIL,USE(?Detail)        !Line item detail
             !structure elements
           END       !
         END
  CODE
  PRINT(CustDetail)                   !Print order detail line
  PrintRpt(CustRpt,?PageHeader)        !Pass report and equate to print proc

PrintRpt PROCEDURE(RptToPrint,DetailNumber)
  CODE
  PRINT(RptToPrint,DetailNumber)       !Print its structure
```

See Also:

Page Overflow

BREAK

Graphics Commands

Graphics Overview

Clarion supplies the set of "graphics primitives" defined in this chapter to allow drawing in windows and reports.

Controls always appear on top of any graphics drawn to the window. This means the graphics appear to underly any controls in the window, so they don´t get in the way of the controls the user needs to access.

The Current Target

Graphics are always drawn to the "current target." Unless overridden with SETTARGET, the "current target" is the last window opened (and not yet closed) on the current execution thread and is the window with input focus. Drawings in a window are persistent--redraws are handled automatically by the runtime library.

Graphics can also be drawn to a report. To do this, SETTARGET must be used to nominate the REPORT as the "current target."

Every window or report has its own current pen width, color, and style. Therefore, to consistently use the same pen (which does not use the default settings) across multiple windows, the SETPENWIDTH, SETPENCOLOR, and SETPENSTYLE statements should be issued for each window.

Graphics Coordinates

The graphics coordinate system starts with the x,y coordinates (0,0) at the top left corner of the window. The coordinates are specified in dialog units (unless overridden by the THOUS, MM, or POINTS attributes when used on graphics placed in a REPORT). A dialog unit is defined as one-quarter the average character width and one-eighth the average character height of the font specified in the window´s FONT attribute (or the system font, if no FONT attribute is specified on the window).

Graphics drawn outside the currently visible portion of the window will appear if the window is scrolled. The size of the virtual screen over which the window may scroll automatically expands to include all graphics drawn to the window. Drawing graphics outside the visible portion of the window automatically causes the scroll bars to appear (if the window has the HSCROLL, VSCROLL, or HVSCROLL attribute).

Graphics Procedures

ARC (draw an arc of an ellipse)

ARC(*x* ,*y* ,*width* ,*height* ,*startangle* ,*endangle* **)**

ARC	Draws an arc of an ellipse on the current window or report.
x	An integer expression that specifies the horizontal position of the starting point.
y	An integer expression that specifies the vertical position of the starting point.
width	An integer expression that specifies the width.
height	An integer expression that specifies the height.
startangle	An integer expression that specifies the starting point of the arc, in tenths of degrees (10 = 1 degree) measured counter-clockwise from three o´clock.
endangle	An integer expression that specifies the ending point of the arc, in tenths of degrees (10 = 1 degree) measured counter-clockwise from three o´clock.

The **ARC** procedure places an arc of an ellipse on the current window or report.

The ellipse is drawn inside a "bounding box" defined by the *x*, *y*, *width*, and *height* parameters. The *x* and *y* parameters specify the starting point, and the *width* and *height* parameters specify the horizontal and vertical size of the "bounding box."

The *startangle* and *endangle* parameters specify what sector of the ellipse will be drawn, as an arc.

The border color is the current pen color set by SETPENCOLOR; the default color is the Windows system color for window text. The border width is the current width set by SETPENWIDTH; the default width is one pixel. The border style is the current pen style set by SETPENSTYLE; the default style is a solid line.

Example:

```
MDIChildWINDOW(´Child One´),AT(0,0,320,200),MDI,MAX,HVSCROLL
          !window controls
         END
  CODE
  OPEN(MDIChild)
  ARC(100,50,100,50,0,900)     !Draw 90 degree arc from 3 to 12 o´clock, as
                          ! the top-right quadrant of ellipse
```

BLANK (erase graphics)

BLANK([*x*] [,*y*] [,*width*] [,*height*] **)**

BLANK	Erases all graphics written to the specified area of the current window or report.
x	An integer expression that specifies the horizontal position of the starting point. If omitted, the default is zero.
y	An integer expression that specifies the vertical position of the starting point. If omitted, the default is zero.
width	An integer expression that specifies the width. If omitted, the default is the width of the window.
height	An integer expression that specifies the height. If omitted, the default is the height of the window.

The **BLANK** procedure erases all graphics written to the specified area of the current window or report. Controls are not erased. BLANK with no parameters erases the entire window or report.

Example:

```
MDIChildWINDOW(´Child One´),AT(0,0,320,200),MDI,MAX,HVSCROLL
          !window controls
        END
  CODE
  OPEN(MDIChild)
  ARC(100,50,100,50,0,900)      !Draw arc
  BLANK                      !Then erase it
```

BOX (draw a rectangle)

BOX(*x* ,*y* ,*width* ,*height* [,*fill*] **)**

BOX	Draws a rectangular box on the current window or report.
x	An integer expression that specifies the horizontal position of the starting point.
y	An integer expression that specifies the vertical position of the starting point.
width	An integer expression that specifies the width.
height	An integer expression that specifies the height.
fill	A LONG or ULONG integer constant, constant EQUATE, or variable containing the red, green, and blue components that create the color in the three low-order bytes (bytes 0, 1, and 2) or an EQUATE for a standard Windows color value.

The **BOX** procedure places a rectangular box on the current window or report. The position and size of the box are specified by *x*, *y*, *width*, and *height* parameters.

The *x* and *y* parameters specify the starting point, and the *width* and *height* parameters specify the horizontal and vertical size of the box. The box extends to the right and down from its starting point.

The border color is the current pen color set by SETPENCOLOR; the default color is the Windows system color for window text. The border width is the current width set by SETPENWIDTH; the default width is one pixel. The border style is the current pen style set by SETPENSTYLE; the default style is a solid line.

Example:

```
MDIChildWINDOW(´Child One´),AT(0,0,320,200),MDI,MAX,HVSCROLL
          !window controls
         END
  CODE
  OPEN(MDIChild)
  BOX(100,50,100,50,00FF0000h)    !Red box
```

CHORD (draw a section of an ellipse)

CHORD(*x* ,*y* ,*width* ,*height* ,*startangle* ,*endangle* [,*fill*] **)**

CHORD	Draws a closed sector of an ellipse on the current window or report.
x	An integer expression that specifies the horizontal position of the starting point.
y	An integer expression that specifies the vertical position of the starting point.
width	An integer expression that specifies the width.
height	An integer expression that specifies the height.
startangle	An integer expression that specifies the starting point of the chord, in tenths of degrees (10 = 1 degree) measured counter-clockwise from three o´clock.
endangle	An integer expression that specifies the ending point of the chord, in tenths of degrees (10 = 1 degree) measured counter-clockwise from three o´clock.
fill	A LONG or ULONG integer constant, constant EQUATE, or variable containing the red, green, and blue components that create the color in the three low-order bytes (bytes 0, 1, and 2) or an EQUATE for a standard Windows color value.

The **CHORD** procedure places a closed sector of an ellipse on the current window or report. The ellipse is drawn inside a "bounding box" defined by the *x*, *y*, *width*, and *height* parameters. The *x* and *y* parameters specify the starting point, and the *width* and *height* parameters specify the horizontal and vertical size of the "bounding box." The *startangle* and *endangle* parameters specify what sector of the ellipse will be drawn, as an arc. The two end points of the arc are also connected with a straight line.

The border color is the current pen color set by SETPENCOLOR; the default color is the Windows system color for window text. The border width is the current width set by SETPENWIDTH; the default width is one pixel. The border style is the current pen style set by SETPENSTYLE; the default style is a solid line.

Example:

```
MDIChildWINDOW(´Child One´),AT(0,0,320,200),MDI,MAX,HVSCROLL
          !window controls
        END
  CODE
  OPEN(MDIChild)
  CHORD(100,50,100,50,0,900,00FF0000h)    !Red 90 degree crescent
```

ELLIPSE (draw an ellipse)

ELLIPSE(*x* ,*y* ,*width* ,*height* [,*fill*] **)**

ELLIPSE	Draws an ellipse on the current window or report.
x	An integer expression that specifies the horizontal position of the starting point.
y	An integer expression that specifies the vertical position of the starting point.
width	An integer expression that specifies the width.
height	An integer expression that specifies the height.
fill	A LONG or ULONG integer constant, constant EQUATE, or variable containing the red, green, and blue components that create the color in the three low-order bytes (bytes 0, 1, and 2) or an EQUATE for a standard Windows color value.

The **ELLIPSE** procedure places an ellipse on the current window or report. The ellipse is drawn inside a "bounding box" defined by the *x*, *y*, *width*, and *height* parameters. The *x* and *y* parameters specify the starting point, and the *width* and *height* parameters specify the horizontal and vertical size of the "bounding box."

The border color is the current pen color set by SETPENCOLOR; the default color is the Windows system color for window text. The border width is the current width set by SETPENWIDTH; the default width is one pixel. The border style is the current pen style set by SETPENSTYLE; the default style is a solid line.

Example:

```
MDIChildWINDOW(´Child One´),AT(0,0,320,200),MDI,MAX,HVSCROLL
          !window controls
        END
  CODE
  OPEN(MDIChild)
  ELLIPSE(100,50,100,50,00FF0000h)   !Red ellipse
```

IMAGE (draw a graphic image)

IMAGE(*x* ,*y* ,*width* ,*height* ,*filename* **)**

IMAGE	Places a graphic image on the current window or report.
x	An integer expression that specifies the horizontal position of the starting point.
y	An integer expression that specifies the vertical position of the starting point.
width	An integer expression that specifies the width. This may be a negative number.
height	An integer expression that specifies the height. This may be a negative number.
filename	A string constant or variable containing the name of the file to display.

The **IMAGE** procedure places a graphic image on the current window or report at the position and size specified by its *x*, *y*, *width*, and *height* parameters. This may be a bitmap (.BMP), icon (.ICO), PaintBrush (.PCX), Graphic Interchange Format (.GIF), JPEG (.JPG), or Windows metafile (.WMF).

Example:

```
MDIChildWINDOW(´Child One´),AT(0,0,320,200),MDI,MAX,HVSCROLL
          !window controls
        END
  CODE
  OPEN(MDIChild)
  IMAGE(100,50,100,50,´LOGO.BMP´)    !Draw graphic image
```

LINE (draw a straight line)

LINE(*x* ,*y* ,*width* ,*height* **)**

LINE	Draws a straight line on the current window or report.
x	An integer expression specifying the horizontal position of the starting point.
y	An integer expression specifying the vertical position of the starting point.
width	An integer expression specifying the width. This may be a negative number.
height	An integer expression specifying the height. This may be a negative number.

The **LINE** procedure places a straight line on the current window or report. The starting position, slope, and length of the line are specified by *x*, *y*, *width*, and *height* parameters. The *x* and *y* parameters specify the starting point of the line. The *width* and *height* parameters specify the horizontal and vertical distance to the end point of the line. If these are both positive numbers, the line slopes to the right and down from its starting point. If the *width* parameter is negative, the line slopes left; if the *height* parameter is negative, the line slopes left. If either the *width* or *height* parameter is zero, the line is horizontal or vertical.

Width	Height	Result
positive	positive	right and down from start point
negative	positive	left and down from start point
positive	negative	right and up from start point
negative	negative	left and up from start point
zero	positive	vertical, down from start point
zero	negative	vertical, up from start point
positive	zero	horizontal, right from start point
negative	zero	horizontal, left from start point

The line color is the current pen color set by SETPENCOLOR; the default color is the Windows system color for window text. The width is the current width set by SETPENWIDTH; the default width is one pixel. The line´s style is the current pen style set by SETPENSTYLE; the default style is a solid line.

Example:

```
MDIChildWINDOW(´Child One´),AT(0,0,320,200),MDI,MAX,HVSCROLL
          !window controls
        END
  CODE
  OPEN(MDIChild)
  LINE(100,50,100,50)      !Draw line
```

PIE (draw a pie chart)

PIE(*x* ,*y* ,*width* ,*height* ,*slices* ,*colors* [,*depth*] [,*wholevalue*] [,*startangle*] **)**

PIE	Draws a pie chart on the current window or report.
x	An integer expression that specifies the horizontal position of the starting point.
y	An integer expression that specifies the vertical position of the starting point.
width	An integer expression that specifies the width.
height	An integer expression that specifies the height.
slices	A SHORT array of values that specify the relative size of each slice of the pie.
colors	A LONG array that specifies the fill color for each slice.
depth	An integer expression that specifies the depth of the three-dimensional pie chart. If omitted, the chart is two-dimensional.
wholevalue	A numeric constant or variable that specifies the total value required to create a complete pie chart. If omitted, the sum of the *slices* array is used.
startangle	A numeric constant or variable that specifies the starting point of the first slice of the pie, measured as a fraction of the *wholevalue*. If omitted (or zero), the first slice starts at the twelve o´clock position.

The **PIE** procedure creates a pie chart on the current window or report. The pie (an ellipse) is drawn inside a "bounding box" defined by the *x*, *y*, *width*, and *height* parameters. The *x* and *y* parameters specify the starting point, and the *width* and *height* parameters specify the horizontal and vertical size of the "bounding box."

The slices of the pie are created clockwise from the *startangle* parameter as a fraction of the *wholevalue*. Supplying a *wholevalue* parameter that is greater than the sum of all the *slices* array elements creates a pie chart with a piece missing.

The color of the lines is the current pen color set by SETPENCOLOR; the default color is the Windows system color for window text. The width of the lines is the current width set by SETPENWIDTH; the default width is one pixel. The line style is the current pen style set by SETPENSTYLE; the default style is a solid line.

Example:

```
MDIChildWINDOW(´Child One´),AT(0,0,320,200),MDI,MAX,HVSCROLL
          !window controls
        END

SliceSize   SHORT,DIM(4)
SliceColor  LONG,DIM(4)

  CODE
  SliceSize[1] = 90
  SliceColor[1] = 0           !Black
  SliceSize[2] = 90
  SliceColor[2] = 00FF0000h     !Red
  SliceSize[3] = 90
  SliceColor[3] = 0000FF00h     !Green
  SliceSize[4] = 90
  SliceColor[4] = 000000FFh     !Blue
  OPEN(MDIChild)
```

```
PIE(100,50,100,50,SliceSize,SliceColor)
                      !Draw pie chart containing
                      ! four equal slices, starting at 12 o´clock
                      ! drawn counter-clockwise
                      !- Black, Red, Green, and Blue
```

POLYGON (draw a multi-sided figure)

POLYGON(*array* [,*fill*] **)**

POLYGON	Draws a multi-sided figure on the current window or report.
array	An array of SHORT integers that specify the x and y coordinates of each "corner point" of the polygon.
fill	A LONG or ULONG integer constant, constant EQUATE, or variable containing the red, green, and blue components that create the color in the three low-order bytes (bytes 0, 1, and 2) or an EQUATE for a standard Windows color value.

The **POLYGON** procedure places a multi-sided figure on the current window or report. The polygon is always closed.

The *array* parameter contains the x and y coordinates of each "corner point" of the polygon. The polygon will have as many corner points as the total number of array elements divided by two. For each corner point in turn, its x coordinate is taken from the odd-numbered array element and the y coordinate from the immediately following even-numbered element.

The border color is the current pen color set by SETPENCOLOR; the default color is the Windows system color for window text. The border width is the current width set by SETPENWIDTH; the default width is one pixel. The line´s style is the current pen style set by SETPENSTYLE; the default style is a solid line.

Example:

```
MDIChildWINDOW(´Child One´),AT(0,0,320,200),MDI,MAX,HVSCROLL
          !window controls
         END
Corners   SHORT,DIM(8)

  CODE
  Corners[1] = 0       !1st x position
  Corners[2] = 90        !1st y position
  Corners[3] = 90        !2nd x position
  Corners[4] = 190     !2nd y position
  Corners[5] = 100     !3rd x position
  Corners[6] = 200     !3rd y position
  Corners[7] = 50        !4th x position
  Corners[8] = 60        !4th y position
  OPEN(MDIChild)
  POLYGON(Corners,000000FFh)    !Blue filled four-sided polygon
```

ROUNDBOX (draw a box with round corners)

ROUNDBOX(*x* ,*y* ,*width* ,*height* [,*fill*] **)**

ROUNDBOX	Draws a rectangular box with rounded corners on the current window or report.
x	An integer expression that specifies the horizontal position of the starting point.
y	An integer expression that specifies the vertical position of the starting point.
width	An integer expression that specifies the width.
height	An integer expression that specifies the height.
fill	A LONG or ULONG integer constant, constant EQUATE, or variable containing the red, green, and blue components that create the color in the three low-order bytes (bytes 0, 1, and 2) or an EQUATE for a standard Windows color value.

The **ROUNDBOX** procedure places a rectangular box with rounded corners on the current window or report. The position and size of the box are specified by *x*, *y*, *width*, and *height* parameters.

The *x* and *y* parameters specify the starting point, and the *width* and *height* parameters specify the horizontal and vertical size of the box. The box extends to the right and down from its starting point.

The border color is the current pen color set by SETPENCOLOR; the default color is the Windows system color for window text. The border width is the current width set by SETPENWIDTH; the default width is one pixel. The border style is the current pen style set by SETPENSTYLE; the default style is a solid line.

Example:

```
MDIChildWINDOW(´Child One´),AT(0,0,320,200),MDI,MAX,HVSCROLL
          !window controls
         END
  CODE
  OPEN(MDIChild)
  ROUNDBOX(100,50,100,50,00FF0000h)      !Red round-cornered box
```

SETPENCOLOR (set line draw color)

SETPENCOLOR([*color*] **)**

SETPENCOLOR Sets the current pen color.

color A LONG or ULONG integer constant, constant EQUATE, or variable containing the red, green, and blue components that create the color in the three low-order bytes (bytes 0, 1, and 2) or an EQUATE for a standard Windows color value. If omitted, the Windows system color for window text is set.

The **SETPENCOLOR** procedure sets the current pen color for use by all graphics procedures. The default color is the Windows system color for window text.

Every window has its own current pen color. Therefore, to consistently use the same pen (which does not use the default color setting) across multiple windows, the SETPENCOLOR statement should be issued for each window.

Example:

```
MDIChildWINDOW(´Child One´),AT(0,0,320,200),MDI,MAX,HVSCROLL
          !window controls
        END
  CODE
  OPEN(MDIChild)
  SETPENCOLOR(000000FFh)            !Set blue pen color
  ROUNDBOX(100,50,100,50,00FF0000h)  !Red round-cornered box with blue border
```

SETPENSTYLE (set line draw style)

SETPENSTYLE([*style*] **)**

SETPENSTYLE Sets the current pen style.

style An integer constant, constant EQUATE, or variable that specifies the pen´s style. If omitted, a solid line is set.

The **SETPENSTYLE** procedure sets the current line draw style for use by all graphics procedures. The default is a solid line.

Every window has its own current pen style. Therefore, to consistently use the same pen (which does not use the default style setting) across multiple windows, the SETPENSTYLE statement should be issued for each window.

EQUATE statements for the pen styles are contained in the EQUATES.CLW file. The following list is a representative sample of these (see EQUATES.CLW for the complete list):

PEN:solid	Solid line
PEN:dash	Dashed line
PEN:dot	Dotted line
PEN:dashdot	Mixed dashes and dots

Example:

```
MDIChildWINDOW(´Child One´),AT(0,0,320,200),MDI,MAX,HVSCROLL
          !window controls
        END
  CODE
  OPEN(MDIChild)
  SETPENCOLOR(000000FFh)          !Set blue pen color
  SETPENSTYLE(PEN:dash)           !Set dashes for line style
  ROUNDBOX(100,50,100,50,00FF0000h)
                              !Red round-cornered box with blue dashed border
```

SETPENWIDTH (set line draw thickness)

SETPENWIDTH([*width*] **)**

SETPENWIDTH Sets the current pen width.

width An integer expression that specifies the pen´s thickness, measured in dialog units unless overridden by the THOUS, MM, or POINTS attributes. If omitted, the default (one pixel) is set.

The **SETPENWIDTH** procedure sets the current line draw thickness for use by all graphics procedures. The default is one pixel, which may be set with a *width* of zero (0).

Every window has its own current pen width. Therefore, to consistently use the same pen (which does not use the default width setting) across multiple windows, the SETPENWIDTH statement should be issued for each window.

Example:

```
MDIChildWINDOW(´Child One´),AT(0,0,320,200),MDI,MAX,HVSCROLL
          !window controls
        END
  CODE
  OPEN(MDIChild)
  SETPENCOLOR(000000FFh)         !Set blue pen color
  SETPENSTYLE(PEN:dash)          !Set dashes for line style
  SETPENWIDTH(2)               !Set two dialog unit thickness
  BOX(100,50,100,50,00FF0000h)  !Red box with thick blue dashed border
```

SHOW (write to screen)

SHOW(*x* ,*y* ,*string* **)**

SHOW	Writes a *string* to the current window or report.
x	An integer expression that specifies the horizontal position of the starting point, in dialog units.
y	An integer expression that specifies the vertical position of the starting point, in dialog units.
string	A string constant, variable, or expression containing the formatted text to place on the current window or report.

SHOW writes the *string* text to the current window or report. The font used is the current font for the window or report.

Example:

```
MDIChildWINDOW(´Child One´),AT(0,0,320,200),MDI,MAX,HVSCROLL
          !window controls
         END
  CODE
  OPEN(MDIChild)
  SHOW(100,100,FORMAT(TODAY(),@D3))       !Display the date
  SHOW(20,20,´Press Any Key to Continue´)   !Display a message
```

TYPE (write string to screen)

TYPE(*string***)**

TYPE Writes a *string* to the current window or report.

string A string constant, variable, or expression.

TYPE writes a *string* to the current window or report. The *string* appears on the window or report at the current cursor position, and "wraps around" if the *string* length extends beyond the right edge. The font used is the current font for the window or report. The SHOW statement may be used to position the cursor before output from TYPE.

Example:

```
MDIChildWINDOW(´Child One´),AT(0,0,320,200),MDI,MAX,HVSCROLL
          !window controls
         END
  CODE
  OPEN(MDIChild)
  TYPE(Cus:Notes)         !Type the notes field
```

Graphics Functions

- PENCOLOR (return line draw color)
- PENSTYLE (return line draw style)
- PENWIDTH (return line draw thickness)

PENCOLOR (return line draw color)

PENCOLOR( )

The **PENCOLOR** function returns the current pen color set by SETPENCOLOR.

Return Data Type: LONG

Example:

```
Proc1       PROCEDURE
MDIChild1   WINDOW(´Child One´),AT(0,0,320,200),MDI,MAX,HVSCROLL
            !window controls
           END
  CODE
  OPEN(MDIChild1)
  SETPENCOLOR(000000FFh)       !Set blue pen color
  Proc2                       !Call another procedure

Proc2       PROCEDURE
MDIChild2   WINDOW(´Child Two´),AT(0,0,320,200),MDI,MAX,HVSCROLL
            !window controls
           END
ColorNow  LONG
  CODE
  ColorNow = PENCOLOR()         !Get current pen color
  OPEN(MDIChild2)
  SETPENCOLOR(ColorNow)         !Set same pen color
  SETPENSTYLE(PEN:dash)         !Set dashes for line style
  SETPENWIDTH(2)               !Set two dialog unit thickness
  BOX(100,50,100,50,00FF0000h)    !Red box with thick blue dashed border
```

PENSTYLE (return line draw style)

PENSTYLE( )

The **PENSTYLE** function returns the current line draw style set by SETPENSTYLE.

EQUATE statements for the pen styles are contained in the EQUATES.CLW file. The following list is a representative sample of these (see EQUATES.CLW for the complete list):

PEN:solid	Solid line
PEN:dash	Dashed line
PEN:dot	Dotted line
PEN:dashdot	Mixed dashes and dots

Return Data Type: LONG

Example:

```
Proc1     PROCEDURE
MDIChild1 WINDOW(´Child One´),AT(0,0,320,200),MDI,MAX,HVSCROLL
           !window controls
          END
  CODE
  OPEN(MDIChild1)
  SETPENCOLOR(000000FFh)       !Set blue pen color
  SETPENSTYLE(PEN:dash)        !Set dashes for line style
  Proc2                       !Call another procedure

Proc2     PROCEDURE
MDIChild2 WINDOW(´Child Two´),AT(0,0,320,200),MDI,MAX,HVSCROLL
           !window controls
          END
ColorNow LONG
StyleNow LONG
  CODE
  ColorNow = PENCOLOR()         !Get current pen color
  StyleNow = PENSTYLE()         !Get current pen style
  OPEN(MDIChild2)
  SETPENCOLOR(ColorNow)         !Set same pen color
  SETPENSTYLE(StyleNow)         !Set same pen style
  SETPENWIDTH(2)               !Set two dialog unit thickness
  BOX(100,50,100,50,00FF0000h)    !Red box with thick blue dashed border
```

PENWIDTH (return line draw thickness)

PENWIDTH( )

The **PENWIDTH** function returns the current line draw thickness set by SETPENWIDTH. The return value is in dialog units (unless overridden by the THOUS, MM, or POINTS attributes on a REPORT).

Return Data Type: LONG

Example:

```
Proc1     PROCEDURE
MDIChild1 WINDOW(´Child One´),AT(0,0,320,200),MDI,MAX,HVSCROLL
           !window controls
          END

  CODE
  OPEN(MDIChild1)
  SETPENCOLOR(000000FFh)       !Set blue pen color
  SETPENSTYLE(PEN:dash)        !Set dashes for line style
  SETPENWIDTH(2)              !Set two dialog unit thickness
  Proc2                       !Call another procedure

Proc2     PROCEDURE
MDIChild2 WINDOW(´Child Two´),AT(0,0,320,200),MDI,MAX,HVSCROLL
           !window controls
          END
ColorNow LONG
StyleNow LONG
WidthNow LONG
  CODE
  ColorNow = PENCOLOR()          !Get current pen color
  StyleNow = PENSTYLE()          !Get current pen style
  WidthNow = PENWIDTH()          !Get current pen width
  OPEN(MDIChild2)
  SETPENCOLOR(ColorNow)          !Set same pen color
  SETPENSTYLE(StyleNow)          !Set same pen style
  SETPENWIDTH(WidthNow)          !Set same pen width
  BOX(100,50,100,50,00FF0000h)    !Red box with thick blue dashed border
```

Data Files

Data File Structures

FILE (declare a data file structure)

```
label    FILE,DRIVER( ) [,CREATE] [,RECLAIM] [,OWNER( )] [,ENCRYPT] [,NAME()] [,PRE( )]
                                  [,BINDABLE] [,THREAD] [,EXTERNAL] [,DLL][,OEM]
label      [INDEX( )]
label      [KEY( )]
label      [MEMO( )]
[label]    RECORD
             fields
           END
         END
```

FILE	Declares a data file.
DRIVER	Specifies the data file type. The DRIVER attribute is required on all FILE structure declarations.
CREATE	Allows the file to be created with the CREATE statement during program execution.
RECLAIM	Specifies reuse of deleted record space.
OWNER	Specifies the password for data encryption.
ENCRYPT	Encrypt the data file.
NAME	Set DOS filename specification.
PRE	Declare a label prefix for the structure.
BINDABLE	Specify all variables in the RECORD structure may be used in dynamic expressions.
THREAD	Specify memory for the record buffer is separately allocated for each execution thread, when the file is opened on the thread.
External	Specify the file is defined, and the memory for its record buffer is allocated, in an external library.
DLL	Specify the file is defined in a .DLL. This is required in addition to the EXTERNAL attribute.
OEM	Specify string data is converted from OEM ASCII to ANSI when read from disk and ANSI to OEM ASCII before writing to disk.
INDEX	Declare a static file access index which must be built at run time.
KEY	Declare a dynamically updated file access index.
MEMO	Declare a variable length text field up to 64K in length.
RECORD	Declare a record structure for the *fields*. A RECORD structure is required in all FILE structure declarations.
fields	Data elements in the RECORD structure.

FILE declares a data file structure which is an exact description of a data file residing on disk. The label of the FILE structure is used in file processing statements and functions to effect operations on the disk file. The FILE structure must be terminated by a period or the END statement.

All attributes of the FILE, KEY, INDEX, MEMO, data declaration statements, and the data types which a FILE may contain, are dependent upon the support of the file driver. Anything in the FILE declaration which is not supported by the file system specified in the DRIVER attribute will cause a file driver error message when the FILE is opened. Attribute and/or data type exclusions for a specific file system are listed in the file driver´s documentation.

At run-time, the RECORD structure is assigned memory for a data buffer where records from the disk file may be processed by executable statements. A RECORD structure is required in a FILE structure. Memory for a data buffer for any MEMO fields is allocated only when the FILE is opened, and de-allocated when the FILE is closed.

A FILE with the BINDABLE attribute declares all the variables within the RECORD structure as available for use in a dynamic expression, without requiring a separate BIND statement for each (allowing BIND(file) to enable all the fields in the file). The contents of each variable´s NAME attribute is the logical name used in the dynamic expression. If no NAME attribute is present, the label of the variable (including prefix) is used. Space is allocated in the .EXE for the names of all of the variables in the structure. This creates a larger program that uses more memory than it normally would. Therefore, the BINDABLE attribute should only be used when a large proportion of the constituent fields are going to be used.

A FILE with the THREAD attribute declares a separate record buffer (and file control block) for each execution thread that OPENs the FILE. If the thread does not OPEN the file, no record buffer is allocated for the file on that thread.

Example:

```
Names  FILE,DRIVER(´Clarion´)   !Declare a file structure
Rec    RECORD                 !Required record structure
Name     STRING(20)             ! containing one or more data elements
     . .                      !End file and record declaration
```

CREATE (allow data file creation)

CREATE

The **CREATE** attribute of a FILE declaration allows a disk file to be created by the CREATE statement from within the PROGRAM where the FILE is declared. This adds some overhead, as all the file information must be contained in the excutable program.

Example:

```
Names  FILE,DRIVER(´Clarion´),CREATE      !Declare a file, allow create
Rec    RECORD
Name     STRING(20)
     . .
```

DRIVER (specify data file type)

DRIVER(*filetype* [,*driver string*]**)**

DRIVER	Specifies the file system the file uses.
filetype	A string constant containing the name of the file manager (Btrieve, Clarion, etc.).
driver string	A string constant or variable containing any additional instructions to the file driver.

The **DRIVER** attribute specifies which file driver is used to access the data file. DRIVER is a required attribute of all FILE declarations.

Clarion programs use file drivers for physical file access. A file driver acts as a translator between a Clarion program and the file system, eliminating different access commands for each file system. File drivers allow access to files from different file systems without changes in the Clarion syntax.

The specific implementation method of each Clarion file access command is dependent on the file driver. Some commands may not be available in a file driver due to limitations in the file system. Each file driver is documented separately. Any unsupported file access commands, FILE declaration attributes, data types, and/or file system idiosyncracies are listed there. See Also: Supported File Systems.

Example:

```
Names    FILE,DRIVER(´Clarion´)     !Begin file declaration
Record   RECORD
Name       STRING(20)
       . .
```

NAME (set filename)

NAME([| *constant* |])
| *variable* |

NAME Specifies the DOS filename of the file.

constant A string constant.

variable The label of a static string variable. This may be declared as Global data, Module data, or Local data with the STATIC attribute.

The **NAME** attribute on a FILE statement specifies the DOS filename for the file driver. If the *constant* or *variable* does not contain a drive and path, the current drive and directory are assumed. If the extension is omitted, the directory entry assumes the file driver´s default value.

Some file drivers require that KEYs, INDEXes, or MEMOs be in separate files. Therefore, a NAME may also be placed on a KEY, INDEX, or MEMO. A NAME attribute without a *constant* or *variable* defaults to the label of the declaration statement on which it is placed (including any specified prefix).

NAME(*constant*) may be used on any field declared within the RECORD structure. This provides the file driver with the name of a field as it may be used in that driver´s file system.

Example:

```
Cust   FILE,PRE(Cus),NAME(CustName)          !Filename in CustName variable
CustKey   KEY(´Name´),NAME(´c:\data\cust.idx´) !Declare key, cust.idx
Record  RECORD
Name      STRING(20)                       !Default NAME to ´Cus:Name´
       . .
```

See Also:

FILE

KEY

INDEX

ENCRYPT (encrypt data file)

ENCRYPT

The **ENCRYPT** attribute is used in conjunction with the OWNER attribute to disguise the information in a data file. ENCRYPT is only valid with an OWNER attribute. Even with a "hex-dump" utility, the data in an encrypted file is extremely difficult to decipher.

Example:

```
Names    FILE,DRIVER(´Clarion´),OWNER(´Clarion´),ENCRYPT
Record   RECORD
Name       STRING(20)
       . .
```

See Also:

OWNER

OWNER (declare password for data encryption)

OWNER(*password***)**

OWNER Specifies a file encryption password.

password A string constant or variable.

The **OWNER** attribute specifies the *password* which is used by the ENCRYPT attribute to encrypt the data.

An OWNER attribute without an accompanying ENCRYPT attribute is allowed by some file systems. Exact implementation of what is encrypted is file driver dependent. See Also: Supported File Systems.

Example:

```
Customer FILE,DRIVER(´Clarion´),OWNER(´abCdeF´),ENCRYPT
             !Encrypt data password "abCdeF"
Record     RECORD
Name         STRING(20)
         . .
```

See Also:

ENCRYPT

RECLAIM (reuse deleted record space)

RECLAIM

The **RECLAIM** attribute specifies that the file driver adds new records to the file in the space previously used by a record that has been deleted, if available. Otherwise, the record is added at the end of the file. Implementation of RECLAIM is file driver specific and may not be supported in all file systems. See Also: Supported File Systems.

Example:

```
Names    FILE,DRIVER(´Clarion´),RECLAIM  !Reuse deleted record space
Record   RECORD
Name       STRING(20)
       . .
```

PRE (set file label)

PRE(*prefix***)**

PRE	Provides a label prefix for complex data structures.
prefix	Acceptable characters are alphabet letters, numerals 0 through 9, and the underscore character. A *prefix* must start with an alpha character (or underscore) and must not be a reserved word. By convention, a *prefix* is 1-3 characters, although it can be longer.

The **PRE** attribute provides a label prefix for the file. It is used to distinguish between identical variable names that occur in different structures. When a data element from the file is referenced in executable statements, assignments, and parameter lists, the *prefix* is attached to its label by a colon (Pre:Label).

Example:

```
MasterFile  FILE,DRIVER(´Clarion´),PRE(Mst)    !Declare master file layout
Record      RECORD
AcctNumber      LONG
          . .
Detail     FILE,DRIVER(´Clarion´),PRE(Dtl)    !Declare detail file layout
Record       RECORD
AcctNumber      LONG
          . .
          GROUP,PRE(Mem)                  !Declare memory variables
Message       STRING(30)
          END
  CODE
  IF Dtl:AcctNumber <> Mst:AcctNumber          !Is it a new account
    Mem:Message = ´New Account´               ! display message
    DO MatchMaster                           ! get new record
  END
```

See Also:

Reserved Words

BINDABLE (set runtime expression string RECORD variables)

BINDABLE

The **BINDABLE** attribute on a FILE statement declares a RECORD structure whose constituent variables are all available for use in a dynamic expression. The contents of each variable´s NAME attribute is the logical name used in the dynamic expression. If no NAME attribute is present, the label of the variable (including prefix) is used. Space is allocated in the .EXE for the names of all of the variables in the structure. This creates a larger program that uses more memory than it normally would. Therefore, the BINDABLE attribute should only be used when a large proportion of the constituent fields are going to be used.

The BIND(group) form of the BIND statement must still be used in the executable code before the individual fields in the RECORD structure may be used.

Example:

```
Names      FILE,DRIVER(´Clarion´),BINDABLE    !Bindable Record structure
Record     RECORD
Name         STRING(20)
FileName     STRING(8),NAME(´FILE´)       !Dynamic name: FILE
Dot          STRING(1)                   !Dynamic name: Dot
Extension      STRING(3),NAME(´EXT´)        !Dynamic name: EXT
          . .
  CODE
  OPEN(Names)
  BIND(Names)
```

See Also:

BIND

UNBIND

EVALUATE

THREAD (set thread-specific record buffer)

THREAD

The **THREAD** attribute declares a FILE which is allocated memory for its record buffer (and file control block) separately for each execution thread in the program. This makes the values contained in the record buffer dependent upon which thread is executing.

Whenever a new execution thread is started, the FILE must be OPENed again to receive a new instance of the record buffer.

Example:

```
        PROGRAM
          MAP
            Thread1
            Thread2
          END
Names       FILE,DRIVER(´Clarion´),PRE(Nam),THREAD  !Threaded file
NbrNdx      INDEX(Nam:Number),OPT
Rec         RECORD
Name          STRING(20)
Number        SHORT
          . .
  CODE
  START(Thread1)
  START(Thread2)

Thread1   PROCEDURE
  CODE
  OPEN(Names)                  !OPEN creates new record buffer instance
  GET(Names,1)                 ! containing the 1st record in the file

Thread2   PROCEDURE
  CODE
  OPEN(Names)                  !OPEN creates another new record buffer instance
  GET(Names,5)                 ! containing the 5th record in the file
```

See Also:

START

Data Declarations and Memory Allocation

EXTERNAL (set file defined externally)

EXTERNAL(*member* **)**

EXTERNAL Specifies the FILE is defined in an external library.

member A string constant. Valid only on a FILE declaration. It contains the filename (without extension) of the MEMBER module containing the FILE definition without the EXTERNAL attribute. If the FILE is defined in a PROGRAM module, an empty *member* string () is required.

The **EXTERNAL** attribute specifies that the FILE on which it is placed is defined in an external library. Therefore, a FILE with the EXTERNAL attribute is declared and may be referenced in the Clarion code, but is not allocated memory for a record buffer. The memory for the FILEs record buffer is allocated by the external library. This allows the Clarion program access to FILEs declared as public in external libraries.

When using EXTERNAL(*member*) to declare a FILE shared by multiple libraries (.LIBs, or .DLLs and .EXE), only one library should define the FILE without the EXTERNAL attribute. All the other libraries (and the .EXE) should declare the FILE with the EXTERNAL attribute. This ensures that there is only one record buffer allocated for the FILE and all the libraries and the .EXE will reference the same memory when referring to data elements from that FILE.

The FILE declarations in all libraries (or .EXEs) that reference common files must be EXACTLY the same (with the appropriate addition of the EXTERNAL attribute). If they are not exactly the same, data corruption could occur. The actual consequence of incompatible FILE declarations is dependent upon the file driver for that file system. Any incompatibilities between libraries cannot be detected by the compiler or linker, therefore it is the programmers responsibility to ensure that consistency is maintained.

One suggested way of coding large systems using many .DLLs and/or .EXEs that share the same files would have one .DLL containing the actual FILE definition that only contains FILE and global variable definitions that are shared among all (or most) of the .DLLs and .EXEs. This makes one central library in which the actual file definitions are maintained. This one central .DLL is linked into all .EXEs that use those common files. All other .DLLs and/or .EXEs in the system would declare the common FILEs with the EXTERNAL attribute.

Example:

```
  PROGRAM
  MAP
   MODULE(LIB.LIB)
     AddCount                  !External library procedure
  . .

TotalCount  LONG,EXTERNAL       !A variable declared in an external library

Cust   FILE,PRE(Cus),EXTERNAL()     !A File defined in a PROGRAM module
CustKey   KEY(Name)                ! whose .LIB is linked into this program
Record  RECORD
Name      STRING(20)
       . .

Contact   FILE,PRE(Con),EXTERNAL(LIB01)   !A File defined in a MEMBER module
ContactKey  KEY(Name)                  ! whose .LIB is linked into this program
Record    RECORD
Name        STRING(20)
         . .

! The LIB.CLW file contains:
```

```
  PROGRAM
  MAP
   MODULE(LIB01)
     AddCount  !
  . .

TotalCount  LONG                      !The TotalCount variable definition

Cust     FILE,PRE(Cus)                !The Cust File definition where the
CustKey     KEY(Cus:Name)             ! record buffer is allocated
Record    RECORD
Name        STRING(20)
         . .

  CODE
  !Executable code ...

! The LIB01.CLW file contains:
 MEMBER(LIB)

Contact     FILE,PRE(Con)        !The Contact File definition where the
ContactKey    KEY(Con:Name)    ! record buffer is allocated
Record      RECORD
Name          STRING(20)
           . .

AddCount PROCEDURE
  CODE
  TotalCount += 1
```

DLL (set file defined externally in .DLL)

DLL([*flag*] **)**

DLL Declares a FILE defined externally in a .DLL.

flag A numeric constant, equate, or Project system define which specifies the attribute as active or not. If the *flag* is zero, the attribute is not active, just as if it were not present. If the *flag* is any value other than zero, the attribute is active.

The **DLL** attribute specifies that the FILE on which it is placed is defined in a .DLL. A FILE with DLL attribute must also have the EXTERNAL attribute. The DLL attribute is required for 32-bit applications because .DLLs are relocatable in a 32-bit flat address space, which requires one extra dereference by the compiler to address the FILE.

The FILE declarations in all libraries (or .EXEs) that reference common FILEs must be EXACTLY the same (with the appropriate addition of the EXTERNAL and DLL attributes). If they are not exactly the same, data corruption could occur. Any incompatibilities between libraries cannot be detected by the compiler or linker, therefore it is the programmers responsibility to ensure that consistency is maintained.

When using EXTERNAL and DLL to declare a FILE shared by .DLLs and .EXE, only one .DLL should define the FILE without the EXTERNAL and DLL attributes. All the other .DLLs (and the .EXE) should declare the FILE with the EXTERNAL and DLL attributes. This ensures that there is only one memory allocation for the FILE and all the .DLLs and the .EXE will reference the same memory when referring to that FILE.

One suggested way of coding large systems using many .DLLs and/or .EXEs that share the same FILEs would have one .DLL containing the actual file definition that only contains FILE and global data definitions that are shared among all (or most) of the .DLLs and .EXEs. This makes one central library in which the actual file definitions are maintained. This one central .DLL is linked into all .EXEs that use those common files. All other .DLLs and/or .EXEs in the system would declare the common variables with the EXTERNAL and DLL attributes.

Example:

```
Cust   FILE,PRE(Cus),EXTERNAL(),DLL    !File defined in PROGRAM module of a .DLL
CustKey   KEY(Cus:Name)
Record  RECORD
Name      STRING(20)
       . .
```

See Also: EXTERNAL

OEM (set international string support)

OEM

The **OEM** attribute specifies that the FILE on which it is placed contains non-English language string data. These strings are automatically translated from the OEM ASCII character set data contained in the file to the ANSI character set for display in Windows. All string data in the record is automatically translated from the ANSI character set to the OEM ASCII character set before the record is written to disk.

The specific OEM ASCII character set used for the translation comes from the DOS code page loaded by the *country*.SYS file. This makes the data file specific to the language used for that code page, and means the data may not be useable on a computer with a different code page loaded.

Example:

```
Cust   FILE,DRIVER(TopSpeed),PRE(Cus),OEM   !Contains international strings
CustKey   KEY(Cus:Name)
Record  RECORD
Name      STRING(20)
       . .

Screen WINDOW(Window)
         ENTRY(@S20),USE(Cus:Name)
         BUTTON(&OK),USE(?Ok),DEFAULT
         BUTTON(&Cancel),USE(?Cancel)
       END

  CODE
  OPEN(Cust)                       !Open Cust file
  SET(Cust)
  NEXT(Cust)                       !Get record, ASCII strings are automatically
                                  ! translated to ANSI character set
  OPEN(Screen)                     !Open window and display ANSI data
  ACCEPT
    CASE FIELD()
    OF ?Ok
      CASE EVENT()
      OF EVENT:Accepted
        PUT(Cust)                  !Put record, ANSIstrings are automatically
                                  ! translated to the OEM ASCII character set
                                  ! per the loaded DOS code page
        BREAK
      END
    END
  END
  CLOSE(Screen)
  CLOSE(Cust)
```

File Structure Statements

INDEX (declare static file access index)

label **INDEX(**[**-/+**][*field*],...,[**-/+**][*field*]**)** [,**NAME( )**] [,**NOCASE**] [,**OPT**]

INDEX	Declares a static index into the data file.
-/+	The - (*minus sign*) preceding an index component *field* specifies descending order for that component. If omitted, or + (*plus sign*) the component is sorted in ascending order.
field	The label of a field in the RECORD structure of the FILE in which the INDEX is declared. The *field* is an index component. A field declared with the DIM attribute (an array) may not be used as an index component.
NAME	Specifies the disk file specification for the INDEX.
OPT	Excludes, from the INDEX, those records with null values (zero or blank) in all index component fields.
NOCASE	Specifies case insensitive sort order.

INDEX declares a "static key" for a FILE structure. An INDEX is updated only by the BUILD statement. It is used to access records in a different logical order than the "physical order" of the file. An INDEX may be used for either sequential file processing or direct random access. An INDEX always allows duplicate entries. An INDEX may have more than one component *field*. The order of the components determines the sort sequence of the index. The first component is the most general, and the last component is the most specific. Generally, a data file may have up to 255 indexes (and/or keys) and each index may be up to 255 bytes, but the exact numbers are file driver dependent. See Also: Supported File Systems.

An INDEX declared without a *field* creates a "dynamic index." A dynamic index may use any field (or fields) in the RECORD as components (except arrays). The component fields of a dynamic index are defined at run time in the second parameter of the BUILD statement. The same dynamic index declaration may be built and re-built using different component fields each time.

Example:

```
Names      FILE,DRIVER('Clarion'),PRE(Nam)
NameNdx      INDEX(Nam:Name),NOCASE       !Declare the name index
NbrNdx     INDEX(Nam:Number),OPT        !Declare the number index
DynamicNdx  INDEX()                     !Declare a dynamic index
Rec        RECORD
Name         STRING(20)
Number       SHORT
          . .
```

See Also:

KEY

BUILD

KEY (declare dynamic file access index)

label **KEY([-/+]***field*,...,[**-/+**][*field*]**)** [,**DUP**] [,**NAME( )**] [,**NOCASE**] [,**OPT**] [,**PRIMARY**]

KEY	Declares a dynamically maintained index into the data file.
-/+	The - (*minus sign*) preceding a key component *field* specifies descending order for that component. If omitted, or + (*plus sign*), the component is sorted in ascending order.
field	The label of a field in the RECORD structure of the FILE in which the KEY is declared. The *field* is a key component. A field declared with the DIM attribute (an array) may not be used as a key component.
NAME	Specifies the disk file specification of the KEY.
DUP	Allows multiple records with duplicate values in their key component fields.
OPT	Excludes, from the KEY, those records with null (zero or blank) values in all key component fields.
NOCASE	Specifies case insensitive sort order.

A **KEY** is an index into the data file which is automatically updated whenever records are added, changed, or deleted. It is used to access records in a different logical order than the "physical order" of the file. A KEY may be used for either sequential file processing or direct random access.

A KEY may have more than one component *field*. The order of the components determines the sort sequence of the key. The first component is the most general, and the last component is the most specific. Generally, a data file may have up to 255 keys (and indexes) and each key may be up to 255 bytes, but the exact numbers are file driver dependent. See Also: Supported File Systems.

Example:

```
Names    FILE,DRIVER(´Clarion´),PRE(Nam)
NameKey    KEY(Nam:Name),NOCASE,DUP       !Declare the name key
NbrKey   KEY(Nam:Number),OPT            !Declare the number key
Rec      RECORD
Name       STRING(20)
Number     SHORT
       . .
  CODE
  Nam:Name = ´Clarion Software´       !Initialize key field
  GET(Names,Nam:NameKey)            !Get the record
  SET(Nam:NbrKey)                   !Set sequential by number
```

See Also:

SET

GET

INDEX

MEMO (declare a text field)

label **MEMO(***length***)** [,**BINARY**] [,**NAME( )**]

MEMO	Declares a fixed-length string which is stored variable-length on disk.
length	A numeric constant that determines the maximum number of characters. The range is from 1 to 65,520 bytes.
BINARY	Declares the MEMO a storage area for binary data.
NAME	Specifies the disk filename for the MEMO field. (Use of this parameter is file driver dependent.)

MEMO declares a fixed-length string field which is stored variable-length on disk. The *length* parameter defines the maximum size of a memo. A MEMO must be declared before the RECORD structure. Memory is allocated for a MEMO field´s buffer when the file is opened, and is de-allocated when the file is closed.

Generally, up to 255 MEMO fields may be declared in a FILE structure. The exact number of MEMO fields and their manner of storage on disk is file driver dependent. See Also: Supported File Systems.

MEMO fields are usually displayed in TEXT controls in WINDOW and REPORT structures.

Example:

```
Names    FILE,DRIVER(´Clarion´),PRE(Nam)
NameKey    KEY(Nam:Name)
NbrKey   KEY(Nam:Number)
Notes      MEMO(4800)             !Memo, 4800 bytes
Rec      RECORD
Name       STRING(20)
Number     SHORT
       . .
```

RECORD (declare record structure)

[label] **RECORD** [,**PRE( )**] [,**NAME( )**]
fields
END

RECORD	Declares the beginning of the data structure within the FILE declaration.
fields	Multiple variable declarations.
PRE	Specify a label prefix for the structure.
NAME	Specifies an external name for the RECORD structure. (Use of this parameter is file driver dependent.)

The **RECORD** statement declares the beginning of the data structure within the FILE declaration. A RECORD structure is required in a FILE declaration. Each *field* is an element of the RECORD structure. The length of a RECORD structure is the sum of the length of its fields. When the label of a RECORD structure is used in an assignment statement, expression, or parameter list, it is treated as a GROUP data type.

At run time, static memory is allocated as a data buffer for the RECORD structure. The *fields* in the record buffer are available whether the file is open or closed.

If the *fields* contain variable declarations with initial values, that initial value is only used to determine the size of the variable, the record buffer is not initialized to the value. For example, a STRING(´abc´) field declaration creates a three-byte string, but it´s value is not automatically initialized to ´abc´ unless the program´s executable code assigns it that value.

Records from the data file on disk are read into the data buffer with the NEXT, PREVIOUS, or GET statements. Data in the *fields* are processed, then written to the data file as a single RECORD unit by the ADD, PUT, or DELETE statements.

Example:

```
Names    FILE,DRIVER(´Clarion´)     !Declare a file structure
Record   RECORD                  ! begin record declaration
Name       STRING(20)              !   declare name field
Number     SHORT                   !   declare number field
       . .                       !End file, end record declaration
```

INDEX, KEY and MEMO Attributes

BINARY (MEMO contains binary data)

BINARY

The **BINARY** attribute of a MEMO declaration specifies the MEMO field will receive data that is not just ASCII characters. This attribute is normally used to store small graphic images for display in an IMAGE field on screen.

Example:

```
Names    FILE,DRIVER(´Clarion´),PRE(Nam)
NameKey    KEY(Nam:Name)
NbrKey   KEY(Nam:Number)
Picture    MEMO(48000),BINARY       !Binary memo - 48,000 bytes
Rec      RECORD
Name       STRING(20)
Number     SHORT
       . .
```

See Also:

MEMO

IMAGE

DUP (allow duplicate KEY entries)

DUP

The **DUP** attribute of a KEY declaration allows multiple records with the same key value to occur in a file. If the DUP attribute is omitted, attempting to ADD or PUT records with duplicate key values will cause the "Creates Duplicate Key" error, an` the record will not be written to the file. During sequential processing using the KEY, records with duplicate key values are accessed in the physical order their entries appear in the KEY file. The GET and SET statements access the first record in a set of duplicates. The DUP attribute is unnecessary on INDEX declarations because an INDEX always allows duplicate entries.

Example:

```
Names    FILE,DRIVER(´Clarion´),PRE(Nam)
NameKey    KEY(Nam:Name),DUP          !Declare name key, allow duplicate names
NbrKey   KEY(Nam:Number)              !Declare number key, no duplicates allowed
Rec      RECORD
Name       STRING(20)
Number     SHORT
       . .
```

NOCASE (case insensitive KEY or INDEX)

NOCASE

The **NOCASE** attribute of a KEY or INDEX declaration makes the sorted sequence of alphabetic characters insensitive to the ASCII upper/lower case sorting convention. All alphabetic characters in key fields are converted to upper case as they are written to the KEY. This case conversion has no affect on the case of the stored data. The NOCASE attribute has no effect on non-alphabetic characters.

Example:

```
Names    FILE,DRIVER(´Clarion´),PRE(Nam)
NameKey    KEY(Nam:Name),NOCASE       !Declare name key, make case insensitive
NbrKey   KEY(Nam:Number)            !Declare number key
Rec      RECORD
Name       STRING(20)
Number     SHORT
       . .
```

See Also:

INDEX

KEY

PRIMARY (set relational primary key)

PRIMARY

The **PRIMARY** attribute specifies the KEY is unique, includes all records in the file, and does not allow "null" values in any of the fields comprising the KEY. This is the definition of a file's "Primary Key" per the relational database theory as expressed by E. F. Codd.

Example:

```
Names    FILE,DRIVER('TopSpeed'),PRE(Nam)    !Declare a file structure
NameKey    KEY(Nam:Name),OPT              !Declare name key, exclude blanks
NbrKey   KEY(Nam:Number),OPT,PRIMARY         !Declare number key, exclude zeroes
Rec      RECORD
Name       STRING(20)
Number     SHORT
       . .
```

See Also: KEY

OPT (exclude null KEY or INDEX entries)

OPT

The **OPT** attribute excludes entries in the KEY or INDEX for records with "null" values in all fields comprising the KEY or INDEX. For the purpose of this attribute, a "null" value is defined as zero in a numeric field or all blank spaces in a string field.

Example:

```
Names    FILE,DRIVER(´Clarion´),PRE(Nam)   !Declare a file structure
NameKey    KEY(Nam:Name),OPT              !Declare name key, exclude blanks
NbrKey   KEY(Nam:Number),OPT            !Declare number key, exclude zeroes
Rec      RECORD
Name       STRING(20)
Number     SHORT
       . .
```

See Also:

INDEX

KEY

NAME (set external name)

NAME([| *constant* |])
| *variable* |

NAME Specifies an "external" name for the file driver.

constant A string constant.

variable The label of a static string variable. This may be declared as Global data, Module data, or Local data with the STATIC attribute.

The **NAME** attribute on a KEY or INDEX or MEMO statement specifies an "external" name for the key or memo for the file driver. Some file drivers require that KEYs, INDEXes, or MEMOs be in separate files, which is specified in the NAME attribute. See Also: Supported File Systems.

NAME(*constant*) may be used on any field declared within the RECORD structure. This provides the file driver with the name of a field as it may be used in that driver´s file system.

A NAME attribute without a *constant* or *variable* defaults to the label of the declaration statement on which it is placed (including any specified prefix).

Example:

```
Cust   FILE,PRE(Cus),NAME(CustName)            !Filename in CustName variable
CustKey   KEY(´Cus:Name´),NAME(´c:\data\cust.idx´)  !Declare key, cust.idx
Record  RECORD
Name      STRING(20)
       . .
```

See Also:

FILE

INDEX

KEY

File Commands

BUILD (build keys and indexes)

```
          | key     |
BUILD(    |index    | [,components])
          | file    |
```

BUILD Builds keys and indexes.

key The label of a KEY declaration.

index The label of an INDEX declaration.

file The label of a FILE declaration.

components A string constant or variable containing the list of the component fields on which to BUILD the dynamic INDEX. If the file has the CREATE attribute, field labels may be used in the *components* parameter. Without the CREATE attribute, the contents of each field´s NAME attribute must be used. The fields must be separated by commas, with leading plus (+) or minus (-) to indicate ascending or descending sequence (if supported by the file driver).

The **BUILD** statement builds keys and indexes. BUILD(*key*), BUILD(*index*), and BUILD(*file*) require exclusive access to the file. Therefore, the file must LOCKed or opened with *access mode* set to 12h (Read/Write Deny All) or 22h (Read/Write Deny Write). BUILD(*index*,*components*) does not require exclusive access to the file.

BUILD(*key*) or BUILD(*index*)
Builds only that KEY or INDEX. The file must be closed, LOCKed, or opened with *access mode* set to 12h or 22h.

BUILD(*file*) Builds all the KEYs declared for the file. The file must be closed, LOCKed, or opened with *access mode* set to 12h or 22h.

BUILD(*index*,*components*)
Allows you to BUILD a dynamic INDEX. This form of BUILD does not require exclusive access to the file, however, the file must be open (with any valid *access mode*). The dynamic INDEX is created as a temporary file, exclusive to the user who BUILDs it. The temporary file is automatically deleted when the file is closed.

Errors Posted: 37 File Not Open
40 Creates Duplicate Key
63 Exclusive Access Required
76 Invalid Index String

Example:

```
Names   FILE,DRIVER(´Clarion´),PRE(Nam)    !Declare a file structure
NameKey   KEY(Nam:Name),OPT               !Declare name key
NbrNdx   INDEX(Nam:Number),OPT           !Declare number index
DynNdx   INDEX()                        !Declare a dynamic index
Rec      RECORD
Name       STRING(20),NAME(´Nam:Name´)
Number     SHORT,NAME(´Nam:Number´)
       . .

  CODE
  OPEN(Names,12h)             !Open file, exclusive read/write
```

```
BUILD(Names)              !Build all keys on Names file

BUILD(Nam:NbrNdx)           !Build the number index

BUILD(Nam:DynNdx,´+Nam:Number,+Nam:Name´)
                        !Build dynamic index ascending number, ascending name
```

See Also:

OPEN

SHARE

CLOSE (close a data file)

CLOSE(*file***)**

CLOSE	Closes a FILE.
file	The label of a FILE.

The **CLOSE** statement closes a FILE. Generally, this flushes DOS buffers and frees any memory used by the open file other than the RECORD structure´s data buffer. The exact action CLOSE takes is file driver dependent. See Also: Supported File Systems.

Example:

```
CLOSE(Customer)      !Close the customer file
```

COPY (copy a data file)

COPY(*file*,*new file*)

COPY	Duplicates a FILE.
file	The label of the FILE to copy.
new file	A string constant or a STRING variable containing a DOS directory file specification. If the file specification does not contain a drive and path, the current drive and directory are assumed. If only the path is specified, the filename and extension of the original *file* are used for the *new file*.

The **COPY** statement duplicates a FILE and enters the specification for the *new file* in the DOS directory. The *file* to be copied must be closed, or the "File Already Open" error is posted. If the file specification of the *new file* is identical to the original *file*, the COPY statement is ignored.

Since some file drivers use multiple physical disk files for one logical FILE structure, the default filename and extension assumptions are file driver dependent. See Also: Supported File Systems. If any error is posted, the file is not copied.

Errors Posted:
02 File Not Found
03 Path Not Found
05 Access Denied
52 File Already Open

Example:

```
COPY(Names,´A:\´)              !Copy Names file to floppy
COPY(CompText,Filename)        !Copy the text file to another file
```

CREATE (create an empty data file)

CREATE(*file*)

CREATE Creates an empty data file.

file The label of the FILE to be created.

The **CREATE** statement adds an empty data file to the DOS directory. If the *file* already exists, it is deleted and recreated as an empty file. The *file* must be closed, or the "File Already Open" error is posted. CREATE does not open the file for access.

Errors Posted:
- 03 Path Not Found
- 04 Too Many Open Files
- 05 Access Denied
- 52 File Already Open
- 54 No Create Attribute

Example:

```
CREATE(Master)      !Create a new master file
CREATE(Detail)      !Create a new detail file
```

EMPTY (empty a data file)

EMPTY(*file***)**

EMPTY Deletes all records from a FILE.

file The label of a FILE.

EMPTY deletes all records from the specified *file*. EMPTY requires exclusive access to the file. Therefore, the file must be opened with *access mode* set to 12h (Read/Write Deny All) or 22h (Read/Write Deny Write).

Errors Posted: 63 Exclusive Access Required

Example:

```
OPEN(Master,18)       !Open the master file
EMPTY(Master)        ! and start a new one
```

See Also:

OPEN

SHARE

FLUSH (flush DOS buffers)

FLUSH(*file*)

FLUSH Terminates a STREAM operation, flushing the DOS buffers.

file The label of a FILE.

The **FLUSH** statement terminates a STREAM operation. It flushes the DOS buffers, which updates the DOS directory entry for that *file*. Support for this statement is dependent upon the file system and its specific action is described in the file driver section. See Also: Supported File Systems.

Example:

```
STREAM(History)           !Use DOS buffering
SET(Current)             !Set to top of current file
LOOP
  NEXT(Current)
  IF ERRORCODE() THEN BREAK.
  His:Record = Cur:Record
  ADD(History)
END
FLUSH(History)           !End streaming, flush buffers
```

See Also:

STREAM

LOCK (exclusive file access)

LOCK(*file* [,*seconds*]**)**

LOCK Locks a data file.

file The label of a FILE opened for shared access.

seconds A numeric constant or variable which specifies the maximum wait time in seconds.

The **LOCK** statement locks a *file* against access by other workstations in a multi-user environment. Generally, this excludes other users from writing to or reading from the *file*. The specific action LOCK takes is file driver dependent. See Also: Supported File Systems.

LOCK(*file*) Attempts to lock the *file* until it is successful. If it is already locked by another workstation, LOCK will wait until the other workstation unlocks it.

LOCK(*file*,*seconds*)
Posts the "File Is Already Locked" error after unsuccessfully trying to lock the file for the specified number of *seconds*.

The most common problem to avoid when locking files is referred to as "deadly embrace." This condition occurs when two workstations attempt to lock the same set of files in two different orders and both are using the LOCK(*file*) form of LOCK. One workstation has already locked a file that the other is trying to LOCK, and vice versa. This problem may be avoided by using the LOCK(*file*,*seconds*) form of LOCK, and always locking files in the same order.

Errors Posted: 32 File Is Already Locked

Example:

```
LOOP                    !Loop to avoid "deadly embrace"
  LOCK(Master,1)           !Lock the master file, try 1 second
  IF ERRORCODE() = 32      !If someone else has it
    BEEP(0,100)           ! pause for 1 second
    CYCLE                 ! and try again
  END
  LOCK(Detail,1)           !Lock the detail file, try 1 second
  IF ERRORCODE() = 32      !If someone else has it
    UNLOCK(Master)        ! unlock the locked file
    BEEP(0,100)           ! pause for 1 second
    CYCLE                 ! and try again
. .
```

OPEN (open a data file)

OPEN(*file* [,*access mode*]**)**

OPEN Opens a FILE structure for processing.

file The label of a FILE declaration.

access mode A numeric constant, variable, or expression which determines the level of access granted to both the user opening the file, and other users in a multi-user system. If omitted, the default value is 22h (Read/Write + Deny Write).

The **OPEN** statement opens a FILE structure for processing and sets the *access mode*. Support for various *access modes* are file driver dependent. All files must be explicitly opened before they may be accessed. The *access mode* is a bitmap which tells the operating system what access to grant the user opening the file and what access to deny to others using the file. The actual values for each access level are:

```
                    Dec.   Hex.   Access
User Access:        0      0h     Read Only
                    1      1h     Write Only
                    2      2h     Read/Write

Other´s Access:     0      0h     Any Access (FCB compatibility
mode)
                    16     10h    Deny All
                    32     20h    Deny Write
                    48     30h    Deny Read
                    64     40h    Deny None
```

Errors Posted:
02 File Not Found
04 Too Many Open Files
05 Access Denied
52 File Already Open
75 Invalid Field Type Descriptor

Example:

```
ReadOnly  EQUATE(0)       !Access mode equates
WriteOnly    EQUATE(1)
ReadWrite    EQUATE(2)
DenyAll     EQUATE(10h)
DenyWrite    EQUATE(20h)
DenyRead  EQUATE(30h)
DenyNone  EQUATE(40h)
  CODE
  OPEN(Names,ReadWrite+DenyNone)    !Open fully shared access
```

See Also:

SHARE

PACK (remove deleted records)

PACK(*file***)**

PACK Removes deleted records from a data file and rebuilds its keys.

file The label of a FILE declaration.

The **PACK** statement removes deleted records from a data file and rebuilds its keys. The resulting data files are as compact as possible. PACK requires at least twice the disk space that the file, keys, and memos occupy to perform the process. New files are created from the old, and the old files are deleted only after the process is complete. PACK requires exclusive access to the file. Therefore, the file must be opened with *access mode* set to 12h (Read/Write Deny All) or 22h (Read/Write Deny Write).

Errors Posted: 63 Exclusive Access Required

Example:

```
OPEN(Trans,12h)          !Open the file in exclusive mode
PACK(Trans)            ! and pack it
```

See Also:

OPEN

SHARE

REMOVE (erase the data file)

REMOVE(*file*)

REMOVE Deletes a FILE.

file The label of the FILE to be removed.

The **REMOVE** statement erases a file specification from the DOS directory in the same manner as the DOS Delete command. The *file* must be closed, or the "File Already Open" error is posted. If any error is posted, the file is not removed.

Errors Posted: 02 File Not Found
05 Access Denied
52 File Already Open

Example:

```
REMOVE(OldFile)      !Delete the old file
REMOVE(Changes)      !Delete the changes file
```

RENAME (change data file directory name)

RENAME(*file*,*new file***)**

RENAME	Renames a FILE.
file	The label of the FILE to be renamed.
new file	A string constant or a STRING variable containing a DOS directory file specification. If the file specification does not contain a drive and path, the current drive and directory are assumed. If only the path is specified, the filename and extension of the original *file* are used for the *new file*. Files cannot be renamed to a new drive.

The **RENAME** statement changes the file specification to the specification for the *new file* in the directory. The *file* to be renamed must be closed, or the "File Already Open" error is posted. If the file specification of the *new file* is identical to the original *file*, the RENAME statement is ignored. If any error is posted, the file is not renamed.

Since some file drivers use multiple physical disk files for one logical FILE structure, the default filename and extension assumptions are file driver dependent. See Also: Supported File Systems.

Errors Posted:
- 02 File Not Found
- 03 Path Not Found
- 05 Access Denied
- 52 File Already Open

Example:

```
RENAME(Text,'text.bak')      !Make it the backup
RENAME(Master,'\newdir')     !Move it to another directory
```

SHARE (open a data file)

SHARE(*file* [,*access mode*]**)**

SHARE Opens a FILE structure for processing.

file The label of a FILE declaration.

access mode A numeric constant, variable, or expression which determines the level of access granted to both the user opening the file, and other users in a multi-user system. If omitted, the default value is 42h (Read/Write, Deny None).

The **SHARE** statement opens a FILE structure for processing and sets the *access mode*. The SHARE statement is the same as the OPEN statement, with the exception of the default value of *access mode*. The *access mode* is a bitmap which tells the operating system what access to grant the user opening the file and what access to deny to others using the file. The actual values for each access level are:

```
                    Dec.   Hex.   Access
User Access:        0      0h     Read Only
                    1      1h     Write Only
                    2      2h     Read/Write

Other´s Access:     0      0h     Any Access (FCB compatibility
mode)
                    16     10h    Deny All
                    32     20h    Deny Write
                    48     30h    Deny Read
                    64     40h    Deny None
```

Errors Posted:
02 File Not Found
04 Too Many Open Files
05 Access Denied
52 File Already Open
75 Invalid Field Type Descriptor

Example:

```
ReadOnly  EQUATE(0)         !Access mode equates
WriteOnly    EQUATE(1)
ReadWrite    EQUATE(2)
DenyAll     EQUATE(10h)
DenyWrite    EQUATE(20h)
DenyRead  EQUATE(30h)
DenyNone  EQUATE(40h)
  CODE
  SHARE(Master,ReadOnly+DenyWrite)    !Open read only mode
```

See Also:

OPEN

STREAM (enable DOS buffering)

STREAM(*file***)**

STREAM Disables automatic FILE flushing.

file The label of a FILE.

Some file systems flush the DOS buffers on each disk write. The **STREAM** statement disables this automatic flushing operation. DOS buffers are allocated by the BUFFERS= statement in the Config.Sys file. They store disk writes until the buffers are full, then write the buffers to disk all at once. The directory entries for the *file* are updated only when the buffers are written to disk (flushed). A STREAM operation is terminated by closing the file, which automatically flushes the buffers, or by issuing a FLUSH statement.

Support for this statement is dependent upon the file system and is described in its file driver´s documentation. See Also: Supported File Systems.

Example:

```
STREAM(History)            !Use DOS buffering
SET(Current)              !Set to top of current file
LOOP
  NEXT(Current)
  IF ERRORCODE() THEN BREAK.
  His:Record = Cur:Record
  ADD(History)
END
FLUSH(History)            !End streaming, flush buffers
```

See Also:

FLUSH

UNLOCK (unlock a locked data file)

UNLOCK(*file*)

UNLOCK Unlocks a previously locked data file.

file The label of a FILE declaration.

The **UNLOCK** statement unlocks a previously LOCKed data file. It will not unlock a file locked by another user. If the *file* is not locked, or is locked by another user, UNLOCK is ignored. UNLOCK posts no errors. The specific action UNLOCK takes is file driver dependent. See Also: Supported File Systems.

Example:

```
LOOP                    !Loop to avoid "deadly embrace"
  LOCK(Master,1)          !Lock the master file, try for 1 second
  IF ERRORCODE() = 32       !If someone else has it
    BEEP(0,100)           ! pause for 1 second
    CYCLE                 ! and try again
  END
  LOCK(Detail,1)          !Lock the detail file, try for 1 second
  IF ERRORCODE() = 32       !If someone else has it
    UNLOCK(Master)        ! unlock the locked file
    BEEP(0,100)           ! pause for 1 second
    CYCLE                 ! and try again
. .                     !End if, end loop
```

Record Access Commands

ADD (add a new file record)

ADD(*file* [,*length*]**)**

ADD	Writes a new record to a FILE.
file	The label of a FILE declaration.
length	An integer constant, variable, or expression which contains the number of bytes to write to the *file*. The *length* must be greater than zero and not greater than the length of the RECORD. If omitted or out of range, *length* defaults to the length of the RECORD structure.

The **ADD** statement writes a new record from the RECORD structure data buffer to the data file. All KEYs associated with the *file* are also updated during each ADD. If an error is posted, no record is added to the file. The specific action ADD takes is file driver dependent. See Also: Supported File Systems.

If there is no room for the record on disk, the "Access Denied" error is posted.

Errors Posted:
05 Access Denied
37 File Not Open
40 Creates Duplicate Key

Example:

```
ADD(Customer)                       !Add a new customer record
  IF ERRORCODE() THEN STOP(ERROR()).   ! and check for errors
```

APPEND (add a new file record)

APPEND(*file* [,*length*]**)**

APPEND	Writes a new record to a FILE.
file	The label of a FILE declaration.
length	An integer constant, variable, or expression which contains the number of bytes to write to the *file*. The *length* must be greater than zero and not greater than the length of the RECORD. If omitted or out of range, *length* defaults to the length of the RECORD structure.

The **APPEND** statement writes a new record from the RECORD structure data buffer to the data file. No KEYs associated with the *file* are updated during an APPEND. After APPENDing records, the KEYs must be rebuilt with the BUILD command. APPEND is usually used in batch adding a number of records at one time.

If an error is posted, no record is added to the file. The specific action APPEND takes is file driver dependent. See Also: Supported File Systems.

If there is no room for the record on disk, the "Access Denied" error is posted.

Errors Posted: 05 Access Denied
37 File Not Open

Example:

```
LOOP                                !Process an input file
  NEXT(InFile)                        ! getting each record in turn
    IF ERRORCODE() THEN BREAK.          ! break loop on error
  Cus:Record = Inf:Record               !Copy the data to Customer file
  APPEND(Customer)                    ! and APPEND a customer record
    IF ERRORCODE() THEN STOP(ERROR()).   !  check for errors
END
BUILD(Customer)                         !Re-build Keys
```

See Also:

BUILD

DELETE (delete a file record)

DELETE(*file*)

DELETE Removes a record from a FILE.

file The label of a FILE declaration.

The **DELETE** statement removes the last record accessed by NEXT, PREVIOUS, GET, ADD, or PUT. The key entries for that record are also removed from the KEYs. DELETE does not clear the record buffer. Therefore, data values from the record just deleted still exist and are available for use until the record buffer is overwritten.

If no record was previously accessed, or the record is held by another workstation, DELETE posts the "Record Not Available" error and no record is deleted. The specific action DELETE takes is file driver dependent. See Also: Supported File Systems.

Errors Posted: 05 Access Denied
33 Record Not Available

Example:

```
Customer FILE,DRIVER(´Clarion´),PRE(Cus)
NameKey     KEY(Cus:Name),OPT
NbrKey    KEY(Cus:Number),OPT
Rec       RECORD
Name        STRING(20)
Number      SHORT
        . .
  CODE
  Cus:Number = 12345             !Initialize key field
  GET(Customer,Cus:NbrKey)        !Get that record
    IF ERRORCODE() THEN STOP(ERROR()).
  DELETE(Customer)               !Delete the customer record
```

See Also:

ADD

GET

HOLD

NEXT

PREVIOUS

PUT

GET (read a file record by direct access)

```
        | file,key                  |
GET(    | file,filepointer [, length ] |)
        | key,keypointer            |
```

GET Retrieves a specific record from a FILE.

file The label of a FILE declaration.

key The label of a KEY or INDEX declaration.

filepointer A numeric constant, variable, or expression for the value returned by the POINTER(*file*) function. The specific value is file driver dependent.

keypointer A numeric constant, variable, or expression for the value returned by the POINTER(*key*) function. The specific value is file driver dependent.

length An integer constant, variable, or expression which contains the number of bytes to read from the *file*. The *length* must be greater than zero and not greater than the RECORD length. If omitted or out of range, *length* defaults to the length of the RECORD structure.

The **GET** statement locates a specific record in the data file and reads it into the RECORD structure data buffer. Direct access to the record is achieved by relative record position within the file, or by matching key values.

GET(*file,key*) Gets the first record from the file (as listed in the *key*) which contains values matching the values in the component fields of the *key*.

GET(*file,filepointer* [,*length*])
Gets a record from the file based on the *filepointer* relative position within the *file*. If *filepointer* is zero, the current record pointer is cleared and no record is retrieved.

GET(*key,keypointer*)
Gets a record from the file based on the *keypointer* relative position within the *key*.

The values for *filepointer* and *keypointer* are file driver dependent. They could be: record number; relative byte position within the file; or, some other kind of "seek position" within the file. See Also: Supported File Systems.

If the *filepointer* or *keypointer* value is out of range, or there are no matching *key* values in the data file, the "Record Not Found" error is posted.

Errors Posted: 35 Record Not Found
37 File Not Open
43 Record Is Already Held

Example:

```
Customer FILE,DRIVER(´Clarion´),PRE(Cus)
NameKey     KEY(Cus:Name),OPT
NbrKey    KEY(Cus:Number),OPT
Rec       RECORD
Name        STRING(20)
Number      SHORT
          . .
  CODE

  Cus:Name = ´Clarion´            !Initialize key field
```

```
GET(Customer,Cus:NameKey)         ! get record with matching value
  IF ERRORCODE() THEN STOP(ERROR()).

GET(Customer,3)                  !Get 3rd rec in physical file order
  IF ERRORCODE() THEN STOP(ERROR()).

GET(Cus:NameKey,3)              !Get 3rd rec in keyed order
  IF ERRORCODE() THEN STOP(ERROR()).
```

See Also:

POINTER

DUPLICATE

HOLD (exclusive file record access)

HOLD(*file* [,*seconds*]**)**

HOLD Arms record locking.

file The label of a FILE opened for shared access.

seconds A numeric constant or variable which specifies the maximum wait time in seconds.

The **HOLD** statement arms record locking for a following GET, NEXT, or PREVIOUS statement in a multi-user environment. The GET, NEXT, or PREVIOUS flags the record as "held" when it successfully gets the record. Generally, this excludes other users from writing to, but not reading, the record. The specific action HOLD takes is file driver dependent. See Also: Supported File Systems.

HOLD(*file*) Arms the process so that the following GET, NEXT, or PREVIOUS attempts to hold the record until it is successful. If it is held by another workstation, GET, NEXT, or PREVIOUS will wait until the other workstation releases it.

HOLD(*file*,*seconds*)
Arms the process for the following GET, NEXT, or PREVIOUS to post the "Record Is Already Held" error after unsuccessfully trying to hold the record for *seconds*.

A user may HOLD one record at a time in each file. If a second record is accessed in the same file, the previously held record in that file is automatically released. A common problem to avoid is "deadly embrace." This occurs when two workstations attempt to hold the same set of records in two different orders and both are using the HOLD(*file*) form of HOLD. One workstation has already held a record that the other is trying to HOLD, and vice versa. You can avoid this problem by using the HOLD(*file*,*seconds*) form of HOLD, and trapping for the "Record Is Already Held" error.

Example:

```
LOOP                    !Loop to avoid "deadly embrace"
  HOLD(Master,1)          !Arm Hold on master file, try for 1 second
  GET(Master,1)           !Get and hold the record
  IF ERRORCODE() = 43       !If someone else has it
    BEEP(0,100); CYCLE    ! pause for 1 second and try again
  END
  HOLD(Detail,1)          !Lock the detail file, try for 1 second
  GET(Detail,1)           !Get and hold the record
  IF ERRORCODE() = 43       !If someone else has it
    RELEASE(Master)         ! release the held record
    BEEP(0,100); CYCLE    ! pause for 1 second and try again
  END
  BREAK
END
```

See Also:

RELEASE

GET

NEXT

PREVIOUS

NEXT (read next file record in sequence)

NEXT(*file*)

NEXT	Reads the next record in sequence from a FILE.
file	The label of a FILE declaration.

NEXT reads the next record in sequence from a data file and places it in the RECORD structure data buffer. The SET statement determines the sequence in which records are read. The first NEXT following a SET reads the record at the position specified by the SET statement. Subsequent NEXT statements read subsequent records in that sequence. The sequence is not effected by any GET, ADD, PUT, or DELETE.

Executing NEXT without a preceding SET, or attempting to read past the end of file posts the "Record Not Available" error.

Errors Posted:
33 Record Not Available
37 File Not Open
43 Record Is Already Held

Example:

```
SET(Cus:NameKey)                !Beginning of file in keyed sequence
LOOP                           !Read all records through end of file
  NEXT(Customer)                ! read a record sequentially
    IF ERRORCODE() THEN BREAK.     ! break on end of file
  DO PostTrans                  ! call transaction posting routine
END
```

See Also:

SET

PREVIOUS

EOF

HOLD

NOMEMO (read file record without reading memo)

NOMEMO(*file*)

NOMEMO Arms "memoless" record retrieval.

file The label of a FILE.

The **NOMEMO** statement arms "memoless" record retrieval for the next GET, NEXT, or PREVIOUS statement encountered. The following GET, NEXT, or PREVIOUS gets the record but does not get any associated MEMO field(s) for the record. Generally, this speeds up access to the record when the contents of the MEMO field(s) are not needed by the procedure.

Example:

```
SET(Master)
LOOP
  NOMEMO(Master)              !Arm "memoless" access
  NEXT(Master)                !Get record without memo
    IF ERRORCODE() THEN BREAK.
  Queue = Mst:Record          !Fill memory queue
  ADD(Queue)
    IF ERRORCODE() THEN STOP(ERROR()).
. .
DISPLAY(?ListBox)              !Display the queue
```

See Also:

GET

NEXT

PREVIOUS

PREVIOUS (read previous file record in sequence)

PREVIOUS(*file***)**

PREVIOUS Reads the previous record in sequence from a FILE.

file The label of a FILE declaration.

PREVIOUS reads the previous record in sequence from a data file and places it in the RECORD structure data buffer. The SET statement determines the sequence in which records are read. The first PREVIOUS following a SET reads the record at the position specified by the SET statement. Subsequent PREVIOUS statements read subsequent records in reverse sequence. The sequence is not effected by any GET, ADD, PUT, or DELETE.

Executing PREVIOUS without a preceding SET, or attempting to read past the beginning of file posts the "Record Not Available" error.

Errors Posted: 33 Record Not Available
37 File Not Open
43 Record Is Already Held

Example:

```
SET(Trn:DateKey)                !End/Beginning of file in keyed sequence
LOOP                           !Read all records in reverse order
  PREVIOUS(Trans)                 ! read a record sequentially
    IF ERRORCODE() THEN BREAK.     ! break at beginning of file
  DO LastInFirstOut               ! call last in first out routine
END
```

See Also:

SET

NEXT

BOF

HOLD

PUT (write record back to file)

PUT(*file* [,*filepointer*] [,*length*]**)**

PUT	Writes a record back to a FILE.
file	The label of a FILE declaration.
filepointer	A numeric constant, variable, or expression for the value returned by the POINTER(*file*) function. The specific value is file driver dependent. See Also: Supported File Systems.
length	An integer constant, variable, or expression containing the number of bytes to write to the *file*. This must be greater than zero and not greater than the RECORD length. If omitted or out of range, the RECORD length is used.

The **PUT** statement writes the current values in the RECORD structure data buffer to a previously accessed record in the *file*.

PUT(*file*) Writes back the last record accessed with NEXT, PREVIOUS, GET, or ADD. If the values in the key variables were changed, the KEYs are updated.

PUT(*file,filepointer*)
Writes the record to the *filepointer* location in the *file* and the KEYs are updated.

PUT(*file,filepointer,length*)
Writes *length* bytes to the *filepointer* location in the *file* and the KEYs are updated.

If a record was not accessed with NEXT, PREVIOUS, GET, ADD, or was deleted, the "Record Not Available" error is posted. PUT also posts the "Creates Duplicate Key" error. If any error is posted, the record is not written to the file.

Errors Posted: 05 Access Denied
33 Record Not Available
40 Creates Duplicate Key

Example:

```
SET(Trn:DateKey)                !End/Beginning of file in keyed sequence
LOOP                           !Read all records in reverse order
  PREVIOUS(Trans)                 ! read a record sequentially
    IF ERRORCODE() THEN BREAK.     ! break at beginning of file
  DO LastInFirstOut               !Call last in first out routine
  PUT(Trans)                     !Write transaction record back to the file
    IF ERRORCODE() THEN STOP(ERROR()).
END
```

See Also:

GET

NEXT

PREVIOUS

ADD

RELEASE (release a held file record)

RELEASE(*file***)**

RELEASE Releases the held record.

file The label of a FILE declaration.

The **RELEASE** statement releases a previously held record. It will not release a record held by another user. If the record is not held, or is held by another user, RELEASE is ignored.

Example:

```
LOOP                    !Loop to avoid "deadly embrace"
  HOLD(Master,1)          !Arm Hold on master file, try for 1 second
  GET(Master,1)           !Get and hold the record
  IF ERRORCODE() = 43       !If someone else has it
    BEEP(0,100)           ! pause for 1 second
    CYCLE                 ! and try again
  END
  HOLD(Detail,1)          !Hold the detail file, try for 1 second
  GET(Detail,1)           !Get and hold the record
  IF ERRORCODE() = 43       !If someone else has it
    RELEASE(Master)         ! release the held record
    BEEP(0,100)           ! pause for 1 second
    CYCLE                 ! and try again
  END
  BREAK
END
```

See Also:

HOLD

REGET (reget file record)

REGET(*file,string***)**

REGET	Regets a specific record in the FILE.
file	The label of a FILE declaration.
string	The string returned by the POSITION function.

The **REGET** reads the record identified by the *string* returned by the POSITION function. The value contained in the *string* returned by the POSITION function, and its length, are file driver dependent. See Also: Supported File Systems.

Errors Posted: 33 Record Not Available

Example:

```
RecordQue     QUEUE,PRE(Dsp)
QueFields       LIKE(Trn:Record),PRE(Dsp)
            END
SavPosition   STRING(260)
  CODE
  SET(Trn:DateKey)                        !Top of file in keyed sequence
  LOOP                                   !Read all records in file
    NEXT(Trans)                           ! read a record sequentially
      IF ERRORCODE() THEN BREAK.
    RecordQue = Trn:Record                  !Move record into queue
    ADD(RecordQue)                        ! and add it
      IF ERRORCODE() THEN STOP(ERROR()).
    IF RECORDS(RecordQue) = 20 OR EOF(Trans) !20 records in queue or end of file?
      SavPosition = POSITION(Trn:DateKey)     !Save record position
      DO DisplayQue                       !Display the queue
      FREE(RecordQue)                       ! and free it
      REGET(Trans,SavPosition)              ! and get the record again
  . .
```

See Also:

POSITION

RESET

RESET (reset file record sequence position)

RESET(*sequence,string***)**

RESET	Resets the sequential processing pointer to a specific record in the FILE.
sequence	The label of a FILE, KEY, or INDEX declaration.
string	The string returned by the POSITION function.

RESET restores the record pointer to the record identified by the *string* returned by the POSITION function. Once RESET has restored the record pointer, either NEXT or PREVIOUS will read that record.

The value contained in the *string* returned by the POSITION function, and its length, are file driver dependent. See Also: Supported File Systems.

RESET is used in conjunction with POSITION to temporarily suspend and resume sequential file processing.

Example:

```
RecordQue     QUEUE,PRE(Dsp)
QueFields       LIKE(Trn:Record),PRE(Dsp)
            END
SavPosition     STRING(260)
  CODE
  SET(Trn:DateKey)                        !Top of file in keyed sequence
  LOOP                                   !Read all records in file
    NEXT(Trans)                           ! read a record sequentially
      IF ERRORCODE() THEN BREAK.
    RecordQue = Trn:Record                  !Move record into queue
    ADD(RecordQue)                        ! and add it
      IF ERRORCODE() THEN STOP(ERROR()).
    IF RECORDS(RecordQue) = 20 OR EOF(Trans)  !20 records in queue or end of file?
      SavPosition = POSITION(Trn:DateKey)     !Save record position
      DO DisplayQue                       !Display the queue
      FREE(RecordQue)                       ! and free it
      RESET(Trn:DateKey,SavPosition)          !Reset the record pointer
      NEXT(Trans)                         ! and get the record again
  . .
```

See Also:

POSITION

NEXT

PREVIOUS

SET (initiate sequential file processing)

```
          | file                |
SET(      | file,key            | )
          | file,filepointer    |
          | key                 |
          | key,key             |
          | key,keypointer      |
          | key,key,filepointer |
```

SET Initializes sequential processing of a FILE.

file The label of a FILE declaration. This parameter specifies processing in the physical order in which records occur in the data file.

key The label of a KEY or INDEX declaration. When used in the first parameter position, *key* specifies processing in the sort sequence of the KEY or INDEX.

filepointer A numeric constant, variable, or expression for the value returned by the POINTER(file) function. The specific value is file driver dependent.

keypointer A numeric constant, variable, or expression for the value returned by the POINTER(key) function. The specific value is file driver dependent.

SET initializes sequential processing of a data file. SET does not get a record, but only sets up processing order and starting point for the following NEXT or PREVIOUS statements. The first parameter determines the order in which records are processed. The second and third parameters determine the starting point within the file. If omitted, processing begins at the beginning (or end) of the file.

SET(*file*) Specifies physical record order processing and positions to the beginning (SET...NEXT) or end (SET...PREVIOUS) of the file.

SET(*file,key*) Specifies physical record order processing and positions to the first record which contains values matching the values in the component fields of the *key*.

SET(*file,filepointer*)
Specifies physical record order processing and positions to the *filepointer* record within the *file*.

SET(*key*) Specifies keyed sequence processing and positions to the beginning (SET...NEXT) or end (SET...PREVIOUS) of the file in that sequence.

SET(*key,key*) Specifies keyed sequence processing and positions to the first or last record which contains values matching the values in the component fields of the *key*. Both *key* parameters must be the same.

SET(*key,keypointer*)
Specifies keyed sequence processing and positions to the *keypointer* record within the *key*.

SET(*key*,*key*,*filepointer*)
Specifies keyed sequence processing and positions to a record which contains values matching the values in the component fields of the *key* at the exact record number specified by *filepointer*. Both *key* parameters must be the same.

When *key* is the second parameter, processing begins at the first or last record containing values matching the values in the component fields of the KEY or INDEX. If an exact match is found, NEXT will read the first matching record while PREVIOUS will read the last matching record. If no exact match is found, the record with the next greater value is read by NEXT, the record with next lesser value is read by

PREVIOUS.

The values for *filepointer* and *keypointer* are file driver dependent. They could be a record number, the relative byte position within the file, or some other kind of "seek position" within the file. See Also: Supported File Systems. These parameters are used to begin processing at a specific record within the file.

For all file drivers, an attempt to SET past the end of the file will set the EOF function to true, and an attempt to SET before the beginning of the file will set the BOF function to true.

Example:

```
SET(Customer)             !Physical file order, beginning of file

Cus:Name = ´Smith´
SET(Customer,Cus:NameKey)    !Physical file order, first record where Name = ´Smith´

SavePtr = POINTER(Customer)
SET(Customer,SavePtr)      !Physical file order, physical record number = SavePtr

SET(Cus:NameKey)           !NameKey order, beginning of file (relative to the key)

SavePtr = POINTER(Cus:NameKey)
SET(Cus:NameKey,SavePtr)   !NameKey order, key-relative record number = SavePtr

Cus:Name = ´Smith´
SET(Cus:NameKey,Cus:NameKey)
                        !NameKey order, first record where Name = ´Smith´

Cus:Name = ´Smith´
SavePtr = POINTER(Customer)
SET(Cus:NameKey,Cus:NameKey,SavePtr)
                        !NameKey order, Name = ´Smith´ and rec number = SavePtr
```

See Also:

NEXT

PREVIOUS

KEY

RECORD

POINTER

SKIP (bypass file records in sequence)

SKIP(*file*,*count***)**

SKIP	Bypasses records during sequential file processing.
file	The label of a FILE declaration.
count	A numeric constant or variable. The *count* specifies the number of records to bypass. If the value is positive, records are skipped in forward (NEXT) sequence. If *count* is negative, records are skipped in reverse (PREVIOUS) sequence.

The **SKIP** statement is used to bypass records during sequential file processing. It bypasses records, in the sequence specified by the SET statement, by moving the file pointer *count* records. SKIP is more efficient than NEXT or PREVIOUS for skipping past records because it does not move records into the RECORD structure data buffer.

If SKIP reads past the end or beginning of file, the EOF( ) and BOF( ) functions return true. If no SET has been issued, SKIP is ignored.

Example:

```
SET(Itm:InvoiceKey)                !Start at beginning of Items file
LOOP                              !Process all records
  NEXT(Items)                     ! Get a record
    IF ERRORCODE() THEN BREAK.
  IF Itm:InvoiceNo <> SavInvNo      ! Check for first item in order
    Hea:InvoiceNo = Itm:InvoiceNo     !   Initialize key field
    GET(Header,Hea:InvoiceKey)      !   Get the associated header record
      IF ERRORCODE() THEN STOP(ERROR()).
    IF Hea:InvoiceStatus = ´Cancel´   !   Is it a canceled order?
      SKIP(Items,Hea:ItemCount-1)     !     SKIP rest of the items
      CYCLE                         !     and process next order
  . .
  DO ItemProcess                    ! process the item
  SavInvNo = Itm:InvoiceNo            ! save the invoice number
END
```

WATCH (automatic file concurrency check)

WATCH(*file***)**

WATCH Arms automatic optimistic concurrency checking.

file The label of a FILE declaration.

The **WATCH** statement arms automatic optimistic concurrency checking by the file driver for a following GET, NEXT, or PREVIOUS statement in a multi-user environment. Generally, the file driver retains a copy of the retrieved record on the GET, NEXT, or PREVIOUS when it successfully gets the record. When the retrieved record is PUT to the *file*, the record on disk is compared to the original record retrieved. An error is returned by the PUT statement if the record has been changed by another user. The specific action WATCH takes is file driver dependent. See Also: Supported File Systems.

Example:

```
SET(Itm:InvoiceKey)                     !Start at beginning of Items file
LOOP                                  !Process all records
  WATCH(Items)                          !Arm concurrency check
  NEXT(Items)                           ! Get a record
    IF ERRORCODE() THEN BREAK.
  DO ItemProcess                        ! process the item
  PUT(Items)                            ! and put it back
    IF ERRORCODE() THEN STOP(ERROR()).      !Stop on any error, including
                                        ! record changed by another user
END
```

File Functions

BOF (beginning of file function)

BOF(*file*)

BOF Flags the beginning of the FILE during sequential processing.

file The label of a FILE declaration.

The **BOF** function returns a non-zero value (true) when the first record in relative file sequence has been read by PREVIOUS or passed by SKIP. Otherwise, the return value is zero (false).

The BOF function is most often used as a LOOP UNTIL condition. Since a LOOP condition is evaluated at the top of the LOOP, BOF returns true after the last record has been read and processed in reverse order.

The BOF function may not be supported by all file drivers (or may be inefficient). Check the driver documentation before using this function.

Return Data Type: LONG

Example:

```
SET(Trn:DateKey)                !End/Beginning of file in keyed sequence
LOOP UNTIL BOF(Trans)             !Process file backwards
  PREVIOUS(Trans)                 ! read a record sequentially
    IF ERRORCODE() THEN STOP(ERROR()).
  DO LastInFirstOut               ! call last in first out routine
END
```

See Also:

PREVIOUS

SKIP

LOOP

BYTES (return size in bytes)

BYTES(*file***)**

BYTES Returns number of bytes in FILE, or most recently read.

file The label of a FILE.

The **BYTES** function returns the size of a FILE in bytes or the number of bytes in the last record accessed. Following an OPEN statement, BYTES returns the size of the file. After the *file* has been accessed by GET, NEXT, ADD, or PUT, the BYTES function returns the number of bytes accessed in the RECORD. The BYTES function may be used to return the number of bytes read in a variable length record.

Return Data Type: LONG

Example:

```
OPEN(DosFile)                          !Open the file
IF (BYTES(DosFile) % 80) > 0             !Check for short record
  SavPtr = INT(BYTES(DosFile) % 80) + 1     ! compute short record pointer
ELSE
  SavPtr = BYTES(DosFile) / 80           ! compute last record pointer
END
GET(DosFile,SavPtr)                      !Get the last record
LastRec = BYTES(DosFile)                 !Save size of the short record
```

DUPLICATE (check for duplicate key entries)

DUPLICATE(| *key* | **)**
| *file* |

DUPLICATE Checks duplicate entries in unique keys.

key The label of a KEY declaration.

file The label of a FILE declaration.

The **DUPLICATE** function returns a non-zero value (true) if writing the current record to the data file would post the "Creates Duplicate Key" error. With a *key* parameter, the specified KEY is checked. With a *file* parameter, all KEYs declared without a DUP attribute are checked.

The DUPLICATE function assumes that the contents of the RECORD structure data buffer are duplicated at the current record pointer location. Therefore, when using DUPLICATE prior to ADDing a record, the record pointer should be cleared with: GET(*file*,0).

Return Data Type: LONG

Example:

```
IF Action = ´ADD´ THEN GET(Vendor,0).          !If adding, clear the file pointer
IF DUPLICATE(Vendor)                          !If this vendor already exists
  SCR:MESSAGE = ´Vendor Number already assigned´ ! display message
  SELECT(?)                                   ! and  stay on the field
END
```

See Also:

GET

EOF (end of file function)

EOF(*file*)

EOF Flags the end of the FILE during sequential processing.

file The label of a FILE declaration.

The **EOF** function returns a non-zero value (true) when the last record in relative file sequence has been read by NEXT or passed by SKIP. Otherwise, the return value is zero (false). The EOF function is most often used as a LOOP UNTIL condition. Since a LOOP condition is evaluated at the top of the LOOP, EOF returns true after the last record has been read and processed.

The EOF function may not be supported by all file drivers (or may be inefficient). Check the driver documentation before using this function.

Return Data Type: LONG

Example:

```
SET(Trn:DateKey)        !Beginning of file in keyed sequence
LOOP UNTIL EOF(Trans)     !Process all records
  NEXT(Trans)           ! read a record sequentially
    IF ERRORCODE() THEN STOP(ERROR()).
  DO LastInFirstOut       ! call last in first out routine
END
```

See Also:

NEXT

SKIP

LOOP

POINTER (return relative record position)

POINTER(| *file* | **)**
| *key* |

POINTER Returns relative record position.

file The label of a FILE declaration. This specifies physical record order within the file.

key The label of a KEY or INDEX declaration. This specifies the entry order within the KEY or INDEX file.

POINTER returns the relative record position within the data file (in *file* sequence), or the relative record position within the KEY or INDEX file (in *key* sequence) of the last record accessed. The value returned by the POINTER function is file driver dependent. It may be a record number, the relative byte position within the file, or some other kind of "seek position" within the file. See Also: Supported File Systems.

Return Data Type: LONG

Example:

```
SavePtr# = POINTER(Customer)   !Save file pointer
```

See Also:

SET

POSITION (return file record sequence position)

POSITION(*sequence*)

POSITION Identifies a record´s unique position in the FILE.

sequence The label of a FILE, KEY, or INDEX declaration.

POSITION returns a STRING which identifies a record´s unique position within the *sequence*. POSITION returns the position of the last record accessed in the file (the record currently in the file´s record buffer). POSITION is used in conjunction with RESET to temporarily suspend and resume sequential file processing.

The value contained in the returned STRING and the length of that STRING are file driver dependent. See Also: Supported File Systems. As a general rule, for file systems that have record numbers, the size of the STRING returned by POSITION(file) is 4 bytes. The return string from POSITION(key) is 4 bytes plus the sum of the sizes of the fields in the key. For file systems that do not have record numbers the size of the STRING returned by POSITION(file) is the sum of the sizes of the fields in the Primary Key (the first KEY on the FILE that does not have the DUP or OPT attribute). The return string from POSITION(key) is the sum of the sizes of the fields in the Primary Key plus the sum of the sizes of the fields in the key.

Return Data Type: STRING

Example:

```
RecordQue     QUEUE,PRE(Dsp)
QueFields       LIKE(Trn:Record),PRE(Dsp)
            END
SavPosition   STRING(260)
  CODE
  SET(Trn:DateKey)                    !Top of file in keyed sequence
  LOOP                               !Read all records in file
    NEXT(Trans)                       ! read a record sequentially
      IF ERRORCODE() THEN BREAK.
    RecordQue = Trn:Record              !Move record into queue
    ADD(RecordQue)                    ! and add it
      IF ERRORCODE() THEN STOP(ERROR()).
    IF RECORDS(RecordQue) = 20 OR EOF(Trans)
                                      !20 records in queue?
      SavPosition = POSITION(Trn:DateKey)  !Save record position
      DO DisplayQue                   !Display the queue
      FREE(RecordQue)                   ! and free it
      RESET(Trn:DateKey,SavPosition)      !Reset the record pointer
      NEXT(Trans)                     ! and get record
  . .
```

See Also:

RESET

RECORDS (return number of file or key records)

RECORDS(| *file* | **)**
| *key* |

RECORDS Returns the number of records.

file The label of a FILE declaration.

key The label of a KEY or INDEX declaration.

The **RECORDS** function returns the number of records in a *file* or *key*. Since the OPT attribute of a KEY or INDEX excludes "null" entries, RECORDS may return a smaller number for the KEY or INDEX than the FILE.

Return Data Type: LONG

Example:

```
SaveCount = RECORDS(Master)          !Save the record count
SaveNameCount = RECORDS(Nam:NameKey) !Number of records with names filled in
```

See Also:

INDEX

KEY

OPT

SEND (send message to file driver)

SEND(*file*,*message***)**

SEND	Sends a message to the file driver.
file	The label of a FILE declaration. The FILE´s DRIVER attribute identifies the file driver to receive the *message*.
message	A string constant or variable containing the information to supply to the file driver.

The **SEND** function allows the program to pass any parameters specific to a file driver during program execution. Specific examples of valid SEND *messages* are listed in the file driver´s documentation.

Return Data Type: STRING

Example:

```
FileCheck = SEND(ClarionFile,´RECOVER=120´)
                          !Arm recovery process for a Clarion data file
```

See also Driver Strings for:

ASCII

Basic

Btrieve

Clarion

Clipper

dBase III

dBase IV

DOS Files

FoxPro and FoxBase

TopSpeed

Transaction Processing

- COMMIT (terminate successful transaction)
- LOGOUT (begin transaction)
- ROLLBACK (terminate unsuccessful transaction)

COMMIT (terminate successful transaction)

COMMIT

The **COMMIT** statement terminates an active transaction. Execution of a COMMIT statement assumes that the transaction was completely successful and no ROLLBACK is necessary. Once COMMIT has been executed, ROLLBACK of the transaction is impossible.

COMMIT informs the file driver involved in the transaction that the temporary files containing the information necessary to restore the database to its previous state may be deleted. The file driver then performs the actions necessary to its file system to successfully terminate a transaction. See Also: Supported File Systems.

Example:

```
 LOGOUT(.1,OrderHeader,OrderDetail)   !Begin Transaction
   DO ErrHandler                     ! always check for errors
 ADD(OrderHeader)                    !Add Parent record
   DO ErrHandler                     ! always check for errors
 LOOP X# = 1 TO RECORDS(DetailQue)    !Process stored detail records
   GET(DetailQue,X#)                  ! Get one from the QUEUE
     DO ErrHandler                   ! always check for errors
   Det:Record = DetailQue             ! Assign to record buffer
   ADD(OrderDetail)                  ! and add it to the file
     DO ErrHandler                   ! always check for errors
 END
 COMMIT                            !Terminate successful transaction

ErrHandler ROUTINE                   !Error routine
 IF NOT ERRORCODE() THEN EXIT.          !Bail out if no error
 ROLLBACK                          !Rollback the aborted transaction
 BEEP                              !Alert the user
 MESSAGE('Transaction Error - ' & ERROR())
 RETURN                            ! and get out
```

See Also:

LOGOUT

ROLLBACK

LOGOUT (begin transaction)

LOGOUT(*timeout* ,*file* [,*file*,...,*file*]**)**

LOGOUT	Initiates transaction processing.
timeout	A numeric constant or variable which specifies the number of seconds to attempt to begin the transaction for a *file* before aborting the transaction and returning an error.
file	The label of a FILE declaration. There may be multiple *file* parameters, separated by commas, in the parameter list. All *files* that will be in the transaction set must be listed.

The **LOGOUT** statement initiates transaction processing for a specified set of *files*. All *files* in the transaction set must have the same file driver. LOGOUT informs the file driver that a transaction is beginning. The file driver then performs the actions necessary to that file system to initiate transaction processing for the specified set of *files*. If the file system requires that the *files* be locked for transaction processing, LOGOUT automatically locks the *files*.

Only one LOGOUT transaction may be active at a time. A second LOGOUT statement without a prior COMMIT or ROLLBACK halts the program with an error message, returning the user to DOS.

Errors Posted: 32 File Is Already Locked

Example:

```
 LOGOUT(.1,OrderHeader,OrderDetail)     !Begin Transaction
   DO ErrHandler                       ! always check for errors
 ADD(OrderHeader)                      !Add Parent record
   DO ErrHandler                       ! always check for errors
 LOOP X# = 1 TO RECORDS(DetailQue)      !Process stored detail records
   GET(DetailQue,X#)                    ! Get one from the QUEUE
     DO ErrHandler                     ! always check for errors
   Det:Record = DetailQue               ! Assign to record buffer
   ADD(OrderDetail)                    ! and add it to the file
     DO ErrHandler                     ! always check for errors
 END
 COMMIT                               !Terminate successful transaction

ErrHandler ROUTINE                     !Error routine
  IF NOT ERRORCODE() THEN EXIT.           !Bail out if no error
  ROLLBACK                            !Rollback the aborted transaction
  BEEP                                !Alert the user
  MESSAGE('Transaction Error - ' & ERROR())
  RETURN                              ! and get out
```

See Also:

COMMIT

ROLLBACK

ROLLBACK (terminate unsuccessful transaction)

ROLLBACK

The **ROLLBACK** statement terminates an active transaction. Execution of a ROLLBACK statement assumes that the transaction was unsuccessful and the database must be restored to the state it was in before the transaction began.

ROLLBACK informs the file driver involved in the transaction that the temporary files containing the information necessary to restore the database to its previous state must be used to restore the database. The file driver then performs the actions necessary to its file system to roll back the transaction. See Also: Supported File Systems.

Example:

```
 LOGOUT(.1,OrderHeader,OrderDetail)     !Begin Transaction
   DO ErrHandler                       ! always check for errors
 ADD(OrderHeader)                      !Add Parent record
   DO ErrHandler                       ! always check for errors
 LOOP X# = 1 TO RECORDS(DetailQue)      !Process stored detail records
   GET(DetailQue,X#)                    ! Get one from the QUEUE
     DO ErrHandler                     ! always check for errors
   Det:Record = DetailQue               ! Assign to record buffer
   ADD(OrderDetail)                    ! and add it to the file
     DO ErrHandler                     ! always check for errors
 END
 COMMIT                               !Terminate successful transaction

ErrHandler ROUTINE                     !Error routine
 IF NOT ERRORCODE() THEN EXIT.           !Bail out if no error
 ROLLBACK                            !Rollback the aborted transaction
 BEEP                                !Alert the user
 MESSAGE('Transaction Error - ' & ERROR())
 RETURN                              ! and get out
```

See Also:

LOGOUT

COMMIT

Null Data Processing

The concept of a null "value" in a field of a FILE or VIEW indicates that the user has never entered data into the field. Null actually means "value not known" for the field. This is completely different from a blank or zero value, and makes it possible to detect the difference between a field which has never had data, and a field which has a (true) blank or zero value.

In expressions, null does not equal blank or zero. Therefore, any expression which compares the value of a field from a FILE or VIEW with another value will always evaluate as unknown if the field is null. This is true even if the value of both elements in the expression are unknown (null) values. For example, the conditional expression Pre:Field1 = Pre:Field2 will evaluate as true only if both fields contain known values. If both fields are null, the result of the expression is also unknown.

```
Known = Known        !Evaluates as True or False
Known = Unknown      !Evaluates as unknown
Unknown = Unknown    !Evaluates as unknown
Unknown <> 10       !Evaluates as unknown
1 + Unknown         !Evaluates as unknown
```

The only four exceptions to this rule are boolean expressions using OR and AND where only one portion of the entire expression in unknown and the other protion of the expression meets the expression criteria:

```
Unknown OR True       !Evaluates as True
True OR Unknown       !Evaluates as True
Unknown AND False     !Evaluates as False
False AND Unknown     !Evaluates as False
```

Support for null "values" in a FILE or VIEW is entirely dependent upon the file driver. Some file drivers support the null field concept (SQL drivers, for the most part), while others do not. Consult the documentation for the specific file driver to determine whether or not your file system´s driver supports nulls. See Also: Supported File Systems.

See also:

NULL (return null file field)

SETNULL (set file field null)

SETNONNULL (set file field non-null)

NULL (return null file field)

NULL(*field*)

NULL Determines null "value" of a *field*.

field The label (including prefix) of a field in a FILE or VIEW structure.

The **NULL** function returns a non-zero value (true) if the *field* is null, and zero (false) if the *field* contains any known value (including blank or zero). Support for null "values" in a FILE or VIEW is entirely dependent upon the file driver. See Also: Supported File Systems.

Return Data Type: LONG

Example:

```
Customer FILE,DRIVER('Clarion'),PRE(Cus)   !Declare customer file layout
AcctKey     KEY(Cus:AcctNumber)
Record    RECORD
AcctNumber    LONG
OrderNumber   LONG
Name          STRING(20)
Addr          STRING(20)
CSZ           STRING(35)
         . .

Header   FILE,DRIVER('Clarion'),PRE(Hea)   !Declare header file layout
AcctKey     KEY(Hea:AcctNumber)
Record    RECORD
AcctNumber    LONG
OrderNumber   LONG
ShipToName    STRING(20)
ShipToAddr    STRING(20)
ShipToCSZ     STRING(35)
         . .

  CODE
  OPEN(Header)
  OPEN(Customer)
  SET(Hea:AcctKey)
  LOOP
    NEXT(Header)
      IF ERRORCODE() THEN BREAK.
    IF NULL(Hea:ShipToName)             !Check for null ship-to address
      Cus:AcctNumber = Hea:AcctNumber
      GET(Customer,Cus:AcctKey)           !Get Customer record
        IF ERRORCODE() THEN CLEAR(Cus:Record).
      Hea:ShipToName = Cus:Name         ! and assign customer address
      Hea:ShipToAddr = Cus:Addr         ! as the ship-to address
      Hea:ShipToCSZ  = Cus:CSZ
    END
    PUT(Header)                        !Put Header record back
  END
```

SETNULL (set file field null)

SETNULL(*field***)**

SETNULL Assigns null "value" to a *field*.

field The label (including prefix) of a field in a FILE or VIEW structure.

The **SETNULL** statement assigns a null "value" to a *field* in a FILE or VIEW structure. Support for null "values" in a FILE or VIEW is entirely dependent upon the file driver.

Return Data Type: LONG

Example:

```
Customer FILE,DRIVER(´Clarion´),PRE(Cus) !Declare customer file layout
AcctKey     KEY(Cus:AcctNumber)
Record    RECORD
AcctNumber    LONG
OrderNumber   LONG
Name         STRING(20)
Addr         STRING(20)
CSZ          STRING(35)
         . .

Header   FILE,DRIVER(´Clarion´),PRE(Hea)   !Declare header file layout
AcctKey     KEY(Hea:AcctNumber)
OrderKey  KEY(Hea:OrderNumber)
Record    RECORD
AcctNumber    LONG
OrderNumber   LONG
ShipToName    STRING(20)
ShipToAddr    STRING(20)
ShipToCSZ     STRING(35)
         . .

  CODE
  OPEN(Header)
  OPEN(Customer)
  SET(Hea:AcctKey)
  LOOP
    NEXT(Header)
      IF ERRORCODE() THEN BREAK.
    Cus:AcctNumber = Hea:AcctNumber
    GET(Customer,Cus:AcctKey)                !Get Customer record
      IF ERRORCODE() THEN CLEAR(Cus:Record).
    IF NOT NULL(Hea:ShipToName) AND Hea:ShipToName = Cus:Name
                                        !Check ship-to address
      SETNULL(Hea:ShipToName)              ! and assign null "values"
      SETNULL(Hea:ShipToAddr)              ! to ship-to address
      SETNULL(Hea:ShipToCSZ)
    END
    PUT(Header)                           !Put Header record back
  END
```

SETNONNULL (set file field non-null)

SETNONNULL(*field*)

SETNONNULL Assigns non-null value (blank or zero) to a *field*.

field The label (including prefix) of a field in a FILE or VIEW structure.

The **SETNONNULL** statement assigns a non-null value (blank or zero) to a *field* in a FILE or VIEW structure. Support for null "values" in a FILE or VIEW is entirely dependent upon the file driver. See Also: Supported File Systems.

Return Data Type: LONG

Example:

```
Customer  FILE,DRIVER(´Clarion´),PRE(Cus)   !Declare customer file layout
AcctKey      KEY(Cus:AcctNumber)
Record     RECORD
AcctNumber     LONG
OrderNumber    LONG
Name           STRING(20)
Addr           STRING(20)
          . .

Header    FILE,DRIVER(´Clarion´),PRE(Hea)   !Declare header file layout
AcctKey      KEY(Hea:AcctNumber)
OrderKey   KEY(Hea:OrderNumber)
Record     RECORD
AcctNumber     LONG
OrderNumber    LONG
ShipToName     STRING(20)
ShipToAddr     STRING(20)
          . .

  CODE
  OPEN(Header)
  OPEN(Customer)
  SET(Hea:AcctKey)
  LOOP
    NEXT(Header)
      IF ERRORCODE() THEN BREAK.
    Cus:AcctNumber = Hea:AcctNumber
    GET(Customer,Cus:AcctKey)                    !Get Customer record
      IF ERRORCODE() THEN CLEAR(Cus:Record).
    IF NULL(Hea:ShipToName) OR Hea:ShipToName = Cus:Name
                                              !Check same ship-to address
      Hea:ShipToName = ´Same as Customer Address´ ! flag the record
      SETNONNULL(Hea:ShipToAddr)               ! and blank out ship-to address
      SETNONNULL(Hea:ShipToCSZ)
    END
    PUT(Header)                               !Put Header record back
  END
```

See Also:

NULL

SETNULL

Internationalization

Environment Files

An environment file contains internationalization settings for an application. On program initialization, the Clarion run-time library attempts to locate an environment file with the same name and location as your application's program file (*appname*.ENV). If an environment file is not found, the run-time library defaults to standard English/ASCII. You can also use these settings to specify internationalization issues for the Clarion environment by creating a CW.ENV file (the Database Manager uses these settings when displaying data files).

The .ENV file is compatible with the .INI files used by Clarion for DOS (both versions 3 and 3.1) if the CLACHARSET is set to OEM, because Clarion for DOS .INI files are generally written using OEM ASCII, not the ANSI character set.

The LOCALE procedure can be used to load environment files at run-time to dynamically change the international settings. LOCALE can also be used to set individual entries. International support is dependent on support in the File Driver (generally for the OEM attribute). All file drivers included with Clarion fior Windows, except ODBC, support the OEM attribute.

The following settings can be set in an environment file:

CLACHARSET=WINDOWS
CLACHARSET=OEM

This determines the character set used by the entries in the .ENV file. WINDOWS is the default if this setting is omitted from the environment file. Use the OEM setting if you are using a DOS editor to edit the .ENV file, or if it has to be compatible with Clarion for DOS. Otherwise, specify WINDOWS or omit the entry. This shoud always be the first setting in the environment file.

CLACOLSEQ=WINDOWS
CLACOLSEQ="*string*"

Specifies a specific collating sequence for use at run-time. This collating sequence is used for building KEY and INDEX files, as well as for sorting QUEUEs and all string/character comparisons.

If the WINDOWS setting is used, then the default collation sequence is defined by Windows' Country setting (in the Control Panel). If this entry is omitted from the environment file, then the default ANSI ordering is used, not the windows default.

Using the WINDOWS setting, the ordering can 'interleave' characters of differing case (AaBbCc ...), so code such as

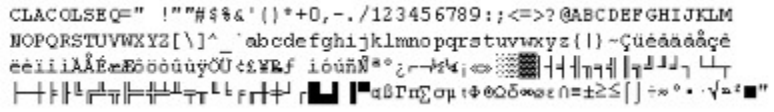
```
CASE SomeString[1]
     OF 'A' TO 'Z'
```

includes 'a' TO 'y' as well. Use the ISUPPER and ISLOWER functions in preference to this kind of code if WINDOWS (or other non-default) collation sequences are used.

In addition to the WINDOWS setting, you may specify a *string* of characters (in double quotes) to explicitly define the collation sequence to use. Only those characters that need to have their sort order specified need be included; all other characters not listed remain in their same relative order. For example, if CLACOLSEQ="CA" is specified for the standard English sort (ABCD ...) the resulting sort order is "CBAD." This is a change from the Clarion for DOS versions of this setting that needed exactly 222 characters, but it is backward compatible.

NOTE: You should always read and write files using the same collation sequence. Using a different sequence may result in keys becoming out of order and records becoming inaccessible. Specifying CLACOLSEQ=WINDOWS means that the collation sequence may change if the user changes the Country in Windows' Control Panel.

example:



CLAAMPM=WINDOWS
CLAAMPM="*AMstring*","*PMstring*"

This specifies the text used to indicate AM or PM as a part of a time display field. The WINDOWS setting specifies use of the AM/PM strings set up in the Windows Control Panel. The *AMstring* and *PMstring* settings are the same as in Clarion for DOS, except that they take notice of the setting of CLACHARSET.

CLAMONTH="Month1","Month2", ... ,"Month12"

Specifies the text returned by functions and picture formats involving the month full name.

CLAMON="AbbrevMonth1","AbbrevMonth2", ... ,"AbbrevMonth12"

Specifies the text returned by functions and picture formats involving the abbreviated month name.

CLADIGRAPH="*DigraphChar1Char2, ...* "

This allows *Digraph* characters to collate correctly. A *Digraph* is a single logical character that is a combination of two characters (*Char1* and *Char2*). The *Digraph* is collated as the two characters that combine to create it. They are more common in non-English languages. For example, with CLADIGRAPH="ÆAe,æae" specified, the word "Jæger" sorts before "Jager" (since "Jae" comes before "Jag").

Multiple *DigraphChar1Char2* combinations may be defined, separated by commas. This setting takes notice of the CLACHARSET setting.

CLACASE=WINDOWS
CLACASE="*UpperString*","*LowerString*"

Allows you to specify upper and lower case letter pairs.

The WINDOWS setting uses the default upper/lower case pair sets as defined by the Windows Country setting (in the Control Panel). If this entry is omitted from the environment file, then the default ANSI ordering is used, not the windows default.

The *UpperString* and *LowerString* parameters specify a set of uppercase characters and each one's lowercase equivalent. The length of the *UpperString* and *LowerString* parameters must be equal. CLACASE takes notice of the setting of CLACHARSET. ANSI characters less than 127 are not affected.

CLABUTTON="OK","&Yes","&No","&Abort","&Ignore","&Retry","Cancel","&Help"

This defines the text used by the buttons of the MESSAGE function. The text is specified as a list of comma separated strings in the following order: OK, YES, NO, ABORT, RETRY, IGNORE, CANCEL, HELP. The default is as specified above.

CLAMSG*errornumber*="*ErrorMessage*"

This allows run-time error messages to be overridden with translated strings. The

errornumber is a standard Clarion error code number appended to CLAMSG. *ErrorMessage* is the string value used to replace that error number's default message. For example, CLAMSG2="No File Found" makes "No File Found" the return value of the ERROR() function when ERRORCODE() = 2.

Example:

```
CLACHARSET=WINDOWS
CLACOLSEQ="AÄÅÆaàáâäåæBbCÇcçDdEÉeèéêëFfGgHhIiìíîïJjKkLlMmNÑnñOÖoòóôöPpQqRrSsßTtUÜuùúûü
VvWwXxYyZzÿ"
CLAAMPM="AM","PM"
CLAMONTH="January","February","March","April","May","June","July","August","September"
,"October","November","December"
CLAMON="Jan","Feb","Mar","Apr","May","Jun","Jul","Aug","Sep","Oct","Nov","Dec"
CLADIGRAPH="ÆAe,æae"
CLACASE="ÄÅÆÇÉÑÖÜ","äåæçéñöü"
CLABUTTON="OK","&Si","&No","&Abortar","&Ignora","&Volveratratar","Cancelar","&Ayuda"
CLAMSG2="No File Found"
```

Tip: By default, the Clarion environment and Clarion applications utilize the ASCII character set for all file I/O -- not the ANSI set, which is the Windows default character set. If you develop international applications, you can use the Internationalization functions or the OEM FILE attribute.

CONVERTANSITOOEM (convert ANSI strings to ASCII)

CONVERTANSITOOEM(*string* **)**

CONVERTANSITOOEM

Translates ANSI strings to OEM ASCII.

string The label of the string to convert. This may be a single variable or a any structure that is treated as a GROUP (RECORD, QUEUE, etc.).

The **CONVERTANSITOOEM** statement translates either a single string or the strings within a GROUP from the ANSI (Windows display) character set into the OEM character set (ASCII with extra characters defined by the active code page).

This procedure is not required on data files if the OEM attribute is set on the file.

Example:

```
Customer  FILE,DRIVER('Clarion'),PRE(Cus)   !Declare file without OEM attribute
AcctKey       KEY(Cus:AcctNumber)
Record      RECORD
AcctNumber      LONG
OrderNumber     LONG
Name            STRING(20)
Addr            STRING(20)
              . .
WinWINDOW,SYSTEM
      STRING(@s20),USE(Cus:Name)
    END
  CODE
  OPEN(Customer)
  SET(Customer)
  NEXT(Customer)
  CONVERTOEMTOANSI(Cus:Record) !Convert all strings from ASCII to ANSI
  OPEN(Win)
  ACCEPT
    !Process window controls
  END
  CONVERTANSITOOEM(Cus:Record) !Convert back to ASCII from ANSI
  PUT(Customer)
```

See Also:

CONVERTOEMTOANSI

CONVERTOEMTOANSI (convert ASCII strings to ANSI)

CONVERTOEMTOANSI(*string* **)**

CONVERTOEMTOANSI

Translates OEM ASCII strings to ANSI.

string The label of the string to convert. This may be a single variable or a any structure that is treated as a GROUP (RECORD, QUEUE, etc.).

The **CONVERTOEMTOANSI** statement translates either a single string or the strings within a GROUP from the the OEM character set (ASCII with extra characters defined by the active code page) into ANSI (Windows display) character set.

This procedure is not required on data files if the OEM attribute is set on the file.

Example:

```
Customer  FILE,DRIVER('Clarion'),PRE(Cus)   !Declare file without OEM attribute
AcctKey       KEY(Cus:AcctNumber)
Record      RECORD
AcctNumber      LONG
OrderNumber     LONG
Name            STRING(20)
Addr            STRING(20)
              . .
WinWINDOW,SYSTEM
      STRING(@s20),USE(Cus:Name)
    END
  CODE
  OPEN(Customer)
  SET(Customer)
  NEXT(Customer)
  CONVERTOEMTOANSI(Cus:Record) !Convert all strings from ASCII to ANSI
  OPEN(Win)
  ACCEPT
    !Process window controls
  END
  CONVERTANSITOOEM(Cus:Record) !Convert back to ASCII from ANSI
  PUT(Customer)
```

See Also:

CONVERTANSITOOEM

ISALPHA (return alphabetic string)

ISALPHA(*string* **)**

ISALPHA Returns whether the *string* passed to it contains an alphabetic character.

string The label of the character string to test. If the *string* contains more than one character, only the first character is tested.

The ISALPHA function returns TRUE if the *string* passed to it is alphabetic (an upper or lower case letter) and false otherwise. This is independent of the language and collation sequence.

Return Data Type: LONG

Example:

```
SomeString  STRING(1)
  CODE
  SomeString = 'A'   !ISALPHA returns true
  IF ISALPHA(SomeString)
    X#= MESSAGE('Alpha string')
  END
  SomeString = '1'   !ISALPHA returns false
  IF ISALPHA(SomeString)
    X#= MESSAGE('Alpha string')
  ELSE
    X#= MESSAGE('Not Alpha string')
  END
```

See Also:

ISUPPER

ISLOWER

ISLOWER (return lower case alphabetic string)

ISLOWER(*string* **)**

ISLOWER Returns whether the *string* passed to it contains a lower case alphabetic character.

string The label of the string to test. If the *string* contains more than one character, only the first character is tested.

The ISLOWER function returns TRUE if the *string* passed to it is a lower case letter and false otherwise. This is independent of the language and collation sequence.

Return Data Type: LONG

Example:

```
SomeString  STRING(1)
  CODE
  SomeString = 'a'   !ISLOWER returns true
  IF ISLOWER(SomeString)
    X#= MESSAGE('lower case string')
  END
  SomeString = 'A'   !ISLOWER returns false
  IF ISLOWER(SomeString)
    X#= MESSAGE('lower case string')
  ELSE
    X#= MESSAGE('Not lower case string')
  END
```

See Also:

ISUPPER

ISALPHA

ISUPPER (return upper case alphabetic string)

ISUPPER(*string* **)**

ISUPPER Returns whether the *string* passed to it contains an upper case alphabetic character.

string The label of the string to test. If the *string* contains more than one character, only the first character is tested.

The ISUPPER function returns TRUE if the *string* passed to it is an upper case letter and false otherwise. This is independent of the language and collation sequence.

Return Data Type: LONG

Example:

```
SomeString  STRING(1)
  CODE
  SomeString = 'A'   !ISUPPER returns true
  IF ISUPPER(SomeString)
    X#= MESSAGE('Upper case string')
  END
  SomeString = 'a'   !ISUPPER returns false
  IF ISUPPER(SomeString)
    X#= MESSAGE('Upper case string')
  ELSE
    X#= MESSAGE('Not upper case string')
  END
```

See Also:

ISLOWER

ISALPHA

LOCALE (load environment file)

LOCALE(| *file* **|)**
| *setting*, *value* |

LOCALE Allows the user to load a specific environment file (.ENV) at run-time and also to set individual environment settings.

file A string constant or variable containing the name (including extension) of the environment file (.ENV) to load, or the keyword WINDOWS. This may be a fully-qualified DOS pathname.

setting A string constant or variable containing the name of the environment variable to set. Valid choices are listed under the *Environment Files* section.

value A string constant or variable containing the environment variable setting.

The **LOCALE** procedure allows the user to load a specific environment file (.ENV) at run-time and also to set individual environment settings. This allows an application to load another file to override the default *appname*.ENV file, or to specify individual environment file settings when no environment file exists.

The WINDOWS keyword as the *file* parameter specifies use of Windows' default values for CLACOLSEQ, CLACASE and CLAAMPM. When specifying individual *settings*, the *value* parameter does not require double quotes around each individual item in the *value* string, unlike the syntax required in an .ENV file.

Example:

```
LOCALE('MY.ENV')                    !Load an environment file
LOCALE('WINDOWS')                     !Set default CLACOLSEQ, CLACASE and CLAAMPM
LOCALE('CLABUTTON','OK,&Si,&No,&Abortar,&Ignora,&Volveratratar,Cancelar,&Ayuda')
                                    !Set CLABUTTON to Spanish
LOCALE('CLACOLSEQ','AÄÅÆaàáâãåæBbCÇcçDdEÉeèéêëFfGgHhIiìíîïJjKkLlMmNÑnñOÖoòóôöPpQqRrSsß
TtUÜuùúûüVvWwXxYyZzÿ')
                                    !Set the collating sequence
LOCALE('CLACASE','ÄÅÆÇÉÑÖÜ,äåæçéñòü')   !Set upper/lower case pairs
LOCALE('CLAMSG2','No File Found')     !Set ERROR() message for ERRORCODE()=2
```

See Also: Environment Files

File Views

View Structures

VIEW (declare a "virtual" file)

```
label    VIEW(primary file) [,FILTER( )]
             [PROJECT( )]
             [JOIN( )
                 [PROJECT( )]
                 [JOIN( )
                     [PROJECT( )]
                  END]
              END]
         END
```

VIEW	Declares a "virtual" file as a composite of related files.
label	The name of the VIEW.
primary file	The label of the primary FILE of the VIEW.
FILTER	Declares an expression used to filter valid records for the VIEW.
PROJECT	Specifies the fields from the *primary file,* or the secondary related file specified by a JOIN structure, that the VIEW will retrieve. If omitted, all fields from the file are retrieved.
JOIN	Declares a secondary related file.

VIEW declares a "virtual" file as a composite of related data files. The data elements declared in a VIEW do not physically exist in the VIEW, because the VIEW structure is a logical construct. VIEW is a separate method of addressing data physically residing in multiple, related FILE structures. At run-time, the VIEW structure is not assigned memory for a data buffer, so the fields used in the VIEW are placed in their respective FILE structure´s record buffer.

A VIEW structure must be explicitly OPENed before use, and all primary and secondary related files used in the VIEW must have been previously OPENed. File I/O operations that operate directly on the primary or any secondary related file in the VIEW are not permitted while the VIEW is OPEN.

The VIEW data structure allows sequential access, only. A SET statement on the VIEW´s primary file must be issued before the OPEN(view) to set the VIEW´s processing order and starting point, then NEXT(view) or PREVIOUS(view) allow sequential access to the VIEW. The REGET statement is also available for VIEW, but only to specify the primary and secondary related file records that should be current in their respective record buffers after the VIEW is CLOSEd. If no REGET statement is issued immediately before the CLOSE(view) statement, the primary and secondary related file record buffers are set to no current record. The processing sequence of the primary and secondary related files is undefined after the VIEW is CLOSEd. Therefore, SET or RESET must be used to establish sequential file processing order, if necessary, after closing the VIEW.

The VIEW data structure is designed to facilitate database access on client-server systems. It accomplishes two relational operations at once: the relational "Join" and "Project" operations. On client-server systems, these operations are performed on the file server, and only the result of the operation is sent to the client. This can dramatically improve performance of network applications.

A relational "Join" retrieves data from multiple files, based upon the relationships defined between the files. The JOIN structure in a VIEW structure defines the relational "Join" operation. There may be multiple JOIN structures within a VIEW, and they may be nested within each other to perform multiple-level "Join" operations.

A relational "Project" operation retrieves only specified data elements from the files involved, not their entire record structure. Only those fields explicitly declared in PROJECT statements in the VIEW structure

are retrieved. Therefore, the relational "Project" operation is automatically implemented by the VIEW structure. The contents of fields that are not contained in the PROJECT are undefined.

The FILTER attribute restricts the VIEW to a sub-set of records. The FILTER expression may include any of the fields explicitly declared in the VIEW structure and restrict the VIEW based upon the contents of any of the fields. This makes the FILTER operate across all levels of the "Join" operation.

Example:

```
Customer FILE,DRIVER(´Clarion´),PRE(Cus) !Declare customer file layout
AcctKey     KEY(Cus:AcctNumber)
Record      RECORD
AcctNumber     LONG
OrderNumber    LONG
Name           STRING(20)
Addr           STRING(20)
City           STRING(20)
State            STRING(20)
Zip            STRING(20)
         . .

Header   FILE,DRIVER(´Clarion´),PRE(Hea) !Declare header file layout
AcctKey     KEY(Hea:AcctNumber)
OrderKey   KEY(Hea:OrderNumber)
Record     RECORD
AcctNumber    LONG
OrderNumber   LONG
ShipToName    STRING(20)
ShipToAddr    STRING(20)
ShipToCity    STRING(20)
ShipToState   STRING(20)
ShipToZip     STRING(20)
         . .

Detail   FILE,DRIVER(´Clarion´),PRE(Dtl) !Declare detail file layout
OrderKey   KEY(Dtl:OrderNumber)
Record     RECORD
OrderNumber   LONG
Item         LONG
Quantity     SHORT
         . .

Product    FILE,DRIVER(´Clarion´),PRE(Pro) !Declare product file layout
ItemKey      KEY(Pro:Item)
Record     RECORD
Item         LONG
Description   STRING(20)
Price          DECIMAL(9,2)
         . .

ViewOrder VIEW(Customer),PRE(Vew)                !Declare VIEW structure
           PROJECT(Cus:AcctNumber,Cus:Name)
           JOIN(Hea:AcctKey,Cus:AcctNumber)      !Join Header file
             PROJECT(Hea:OrderNumber)
             JOIN(Dtl:OrderKey,Hea:OrderNumber)     !Join Detail file
               PROJECT(Det:Item,Det:Quantity)
               JOIN(Pro:ItemKey,Dtl:Item)          !Join Product file
                 PROJECT(Pro:Description,Pro:Price)
               END
             END
```

```
   END
END
```

FILTER (set view filter expression)

FILTER(*expression***)**

FILTER Specifies a filter *expression* used to evaluate records to include in the VIEW.

expression A string constant containing a logical expression.

The **FILTER** attribute specifies a filter *expression* used to evaluate records to include in the VIEW. The *expression* may reference any field in the VIEW, at all levels of JOIN structures. The entire *expression* must evaluate as true for a record to be included in the VIEW.

Example:

```
 !Get only orders for customer 9999 since order number 100
ViewOrder VIEW(Customer),FILTER(´Cus:AcctNumber = 9999 AND Hea:OrderNumber > 100´)
          PROJECT(Cus:AcctNumber,Cus:Name)
          JOIN(Hea:AcctKey,Cus:AcctNumber)       !Join Header file
            PROJECT(Hea:OrderNumber)
            JOIN(Dtl:OrderKey,Hea:OrderNumber)     !Join Detail file
              PROJECT(Det:Item,Det:Quantity)
              JOIN(Pro:ItemKey,Dtl:Item)          !Join Product file
                PROJECT(Pro:Description,Pro:Price)
              END
            END
          END
        END
```

PROJECT (set view fields)

PROJECT(*fields* **)**

PROJECT Declares the fields retrieved for the VIEW.

fields A comma delimited list of fields (including prefixes) from the primary file of the VIEW, or the secondary related file named in the JOIN structure, containing the PROJECT declaration.

The **PROJECT** statement in a VIEW structure declares *fields* retrieved for a relational "Project" operation. A relational "Project" operation retrieves only the specified *fields* from the file, not the entire record structure. Only those fields explicitly declared in PROJECT declarations in the VIEW structure are retrieved.

A PROJECT statement may be declared in the VIEW, or within one of its component JOIN structures. If there is no PROJECT declaration in the VIEW or JOIN structure, all fields in the relevant file are retrieved.

Example:

```
Detail   FILE,DRIVER(´Clarion´),PRE(Dtl) !Declare detail file layout
OrderKey  KEY(Dtl:OrderNumber)
Record    RECORD
OrderNumber  LONG
Item        LONG
Quantity    SHORT
Description  STRING(20)    !Line item comment
         . .

Product   FILE,DRIVER(´Clarion´),PRE(Pro) !Declare product file layout
ItemKey     KEY(Pro:Item)
Record    RECORD
Item        LONG
Description  STRING(20)    !Product description
Price        DECIMAL(9,2)
         . .

ViewOrder VIEW(Detail)
          PROJECT(Det:OrderNumber,Det:Item,Det:Description)
          JOIN(Pro:ItemKey,Det:Item)
            PROJECT(Pro:Description,Pro:Price)
          END
         END
```

JOIN (declare a "join" operation)

```
JOIN(secondary key,linking fields)
        [PROJECT( )]
        [JOIN( )
                [PROJECT( )]
         END]
END
```

JOIN	Declares a secondary file for a relational "Join" operation.
secondary key	The label of a KEY which defines the secondary FILE and its access key.
linking fields	A comma-delimited list of fields in the related file that contain the values the *secondary key* uses to access records.
PROJECT	Specifies the fields from the secondary related file specified by a JOIN structure that the VIEW will retrieve. If omitted, all fields from the file are retrieved.

The **JOIN** structure declares a secondary file for a relational "Join" operation. A relational "Join" retrieves data from multiple files, based upon the relationships defined between the files. There may be multiple JOIN structures within a VIEW, and they may be nested within each other to perform multiple-level "Join" operations.

The *secondary key* defines the access key for the secondary file. The *linking fields* name the fields in the file to which the secondary file is related, that contain the values used to retrieve the related records. For a JOIN directly within the VIEW, these fields come from the VIEW´s primary file. For a JOIN nested within another JOIN, these fields come from the secondary file of the JOIN in which it is nested. Non-linking fields in the *secondary key* are allowed as long as they appear in the list of the key´s component fields after all the linking fields.

When data is retrieved, if there are no matching secondary file records for a primary file record, null values are supplied in the fields specified in the PROJECT. This type of relational "Join" operation is known as an "outer join." The FILTER attribute of the VIEW can be used to accomplish all other forms of the relational "Join" operation.

Example:

```
Header    FILE,DRIVER(´Clarion´),PRE(Hea) !Declare header file layout
AcctKey      KEY(Hea:AcctNumber)
OrderKey   KEY(Hea:AcctNumber,Hea:OrderNumber)
Record     RECORD
AcctNumber     LONG
OrderNumber    LONG
ShipToName     STRING(20)
ShipToAddr     STRING(20)
ShipToCity     STRING(20)
ShipToState    STRING(20)
ShipToZip      STRING(20)
         . .
Detail    FILE,DRIVER(´Clarion´),PRE(Dtl) !Declare detail file layout
OrderKey   KEY(Dtl:AcctNumber,Dtl:OrderNumber)
Record     RECORD
AcctNumber     LONG
OrderNumber    LONG
Item         LONG
Quantity     SHORT
         . .
Product    FILE,DRIVER(´Clarion´),PRE(Pro) !Declare product file layout
```

```
ItemKey       KEY(Pro:Item)
Record      RECORD
Item          LONG
Description     STRING(20)
Price           DECIMAL(9,2)
          . .
ViewOrder   VIEW(Header)   !Declare VIEW structure
            PROJECT(Hea:AcctNumber,Hea:OrderNumber)
            JOIN(Dtl:OrderKey,Hea:AcctNumber,Hea:OrderNumber) !Join Detail file
              PROJECT(Dtl:ItemDtl:Quantity)
              JOIN(Pro:ItemKey,Dtl:Item) !Join Product file
                PROJECT(Pro:Description,Pro:Price)
              END
            END
          END
```

View Commands

- CLOSE (close a VIEW)
- OPEN (open a VIEW)
- DELETE (delete a view primary file record)
- HOLD (exclusive view record access)
- NEXT (read next view record in sequence)
- NOMEMO (read view record without reading memos)
- PREVIOUS (read previous view record in sequence)
- PUT (write VIEW primary file record back)
- REGET (reget view record)
- RELEASE (release a held view record)
- RESET (reset view record sequence position)
- SKIP (bypass view records in sequence)
- WATCH (automatic view concurrency check)

CLOSE (close a VIEW)

CLOSE(*view***)**

CLOSE Closes a VIEW.

view The label of a VIEW.

The **CLOSE** statement closes a VIEW. A VIEW declared within a procedure is implicitly closed upon RETURN from the procedure, if it has not already been explicitly CLOSEd.

If the CLOSE(*view*) statement is not immediately preceded by a REGET statement, the primary and secondary related files in the VIEW are set to no current record. This means the contents of their record buffers are undefined and a SET or RESET statement must be issued before performing sequential processing on the file.

Example:

```
Customer  FILE,DRIVER('Clarion'),PRE(Cus) !Declare customer file layout
AcctKey     KEY(Cus:AcctNumber)
Record    RECORD
AcctNumber    LONG
OrderNumber   LONG
Name          STRING(20)
Addr          STRING(20)
City          STRING(20)
State           STRING(20)
Zip           STRING(20)
          . .
ViewCust  VIEW(Customer)  !Declare VIEW structure
            PROJECT(Cus:AcctNumber,Cus:Name)
          END
  CODE
  OPEN(Customer,22h)
  SET(Cus:AcctKey)
  OPEN(ViewCust)        !Open the customer view
   !executable statements
  CLOSE(ViewCust)         !and close it again
```

OPEN (open a VIEW)

OPEN(*view***)**

OPEN Opens a VIEW structure for processing.

view The label of a VIEW declaration.

The **OPEN** statement opens a VIEW structure for processing. A VIEW must be explicitly opened before it may be accessed. The files used in the VIEW must already be open.

Immediately before the OPEN(*view*) statement, you must issue a SET statement on the VIEW structure´s primary file to setup sequential processing for the VIEW. You cannot issue a SET statement to the primary file while the VIEW is OPEN; you must CLOSE(*view*) then issue the SET before a subsequent OPEN(*view*).

Example:

```
Header    FILE,DRIVER(´Clarion´),PRE(Hea)   !Declare header file layout
AcctKey     KEY(Hea:AcctNumber)
OrderKey    KEY(Hea:OrderNumber)
Record      RECORD
AcctNumber     LONG
OrderNumber    LONG
ShipToName     STRING(20)
ShipToAddr     STRING(20)
ShipToCity     STRING(20)
ShipToState    STRING(20)
ShipToZip      STRING(20)
          . .
Detail    FILE,DRIVER(´Clarion´),PRE(Dtl) !Declare detail file layout
OrderKey    KEY(Dtl:OrderNumber)
Record      RECORD
OrderNumber    LONG
Item         LONG
Quantity     SHORT
          . .
Product     FILE,DRIVER(´Clarion´),PRE(Pro) !Declare product file layout
ItemKey       KEY(Pro:Item)
Record      RECORD
Item         LONG
Description    STRING(20)
Price          DECIMAL(9,2)
          . .
ViewOrder   VIEW(Header)                        !Declare VIEW structure
            PROJECT(Hea:OrderNumber)
            JOIN(Dtl:OrderKey,Hea:OrderNumber)      !Join Detail file
              PROJECT(Det:Item,Det:Quantity)
              JOIN(Pro:ItemKey,Dtl:Item)           !Join Product file
                PROJECT(Pro:Description,Pro:Price)
          . . .
  CODE
  OPEN((Header,22h)
  OPEN(Detail,22h)
  OPEN(Product,22h)
  SET(Cus:AcctKey)
  OPEN(ViewOrder)  !Open
```

DELETE (delete a view primary file record)

DELETE(*view***)**

DELETE Removes a primary file record from a VIEW.

view The label of a VIEW declaration.

The **DELETE** statement removes the last VIEW primary file record that was accessed by a NEXT or PREVIOUS statement. The key entries for that record are also removed from the KEYs. DELETE does not remove records from any secondary JOIN files in the VIEW.

DELETE only deletes the primary file record in the VIEW because the VIEW structure performs both relational Project and Join operations at the same time. Therefore, it is possible to create a VIEW structure that, if all its component files were updated, would violate the Referential Integrity rules set for the database. The common solution to this problem in SQL-based database products is to write only to the Primary file. Therefore, Clarion has adopted this same industry standard solution.

If no record was previously accessed, or the record is held by another workstation, DELETE posts the "Record Not Available" error and no record is deleted. The specific disk action DELETE performs in the file is file driver dependent. See Also: Supported File Systems.

Errors Posted: 05 Access Denied
33 Record Not Available

Example:

```
Customer  FILE,DRIVER('Clarion'),PRE(Cus)   !Declare customer file layout
AcctKey      KEY(Cus:AcctNumber)
Record     RECORD
AcctNumber     LONG
Name           STRING(20)
Addr           STRING(20)
City           STRING(20)
State            STRING(20)
Zip            STRING(20)
          . .
CustView  VIEW(Customer)                !Declare VIEW structure
            PROJECT(Cus:AcctNumber,Cus:Name)
          END
  CODE
  OPEN(Customer)
  Cus:AcctNumber = 12345          !Initialize key field
  SET(Cus:AcctKey,Cus:AcctKey)
  OPEN(CustView)
  NEXT(CustView)                 !Get that record
    IF ERRORCODE() THEN STOP(ERROR()).
  DELETE(CustView)               !Delete the customer record
```

See Also:

HOLD

NEXT

PREVIOUS

PUT

HOLD (exclusive view record access)

HOLD(*view* [,*seconds*]**)**

HOLD Arms record locking.

view The label of a VIEW opened for shared access.

seconds A numeric constant or variable which specifies the maximum wait time in seconds.

The **HOLD** statement arms record locking for the primary file in the VIEW in a multi-user environment. The following NEXT or PREVIOUS statement flags the primary file record as "held" when it successfully gets the record. Generally, this excludes other users from writing to the record, although it does not prevent them from reading the record. The specific action HOLD takes is file driver dependent. See Also: Supported File Systems.

HOLD(*view*) Arms the process so that the following NEXT or PREVIOUS attempts to hold the record until it is successful. If it is held by another workstation, GET, NEXT, or PREVIOUS will wait until the other workstation releases it.

HOLD(*view*,*seconds*)
Arms the process so that the following NEXT or PREVIOUS statement posts the "Record Is Already Held" error after unsuccessfully trying to hold the record for *seconds*.

A user may only HOLD one record at a time in the VIEW. If a second record is to be accessed in the same file, the previously held record must be released (see RELEASE).

As with LOCK, a common problem to avoid when holding records is "deadly embrace." This condition occurs when two workstations attempt to hold the same set of records in two different orders and both are using the HOLD(*view*) form of HOLD. One workstation has already held a record that the other is trying to HOLD, and vice versa. This problem may be avoided by using the HOLD(*view*,*seconds*) form of HOLD, and trapping for the "Record Is Already Held" error.

Example:

```
ViewOrder   VIEW(Customer)   !Declare VIEW structure
            PROJECT(Cus:AcctNumber,Cus:Name)
            JOIN(Hea:AcctKey,Cus:AcctNumber) !Join Header file
              PROJECT(Hea:OrderNumber)
              JOIN(Dtl:OrderKey,Hea:OrderNumber) !Join Detail file
                PROJECT(Det:Item,Det:Quantity)
                JOIN(Pro:ItemKey,Dtl:Item) !Join Product file
                  PROJECT(Pro:Description,Pro:Price)
                END
              END
            END
          END
  CODE
  OPEN(Customer,22h)
  OPEN((Header,22h)
  OPEN(Detail,22h)
  OPEN(Product,22h)
  SET(Cus:AcctKey)
  OPEN(ViewOrder)
  LOOP                      !Process records Loop
    LOOP                    !Loop to avoid "deadly embrace"
      HOLD(ViewOrder,1)        !Arm Hold on view, try for 1 second
```

```
    NEXT(ViewOrder)           !Get and hold the record
    IF ERRORCODE() = 43       !If someone else has it
      CYCLE                  ! try again
    ELSE
      BREAK                  !Break if not held
    END
  END
  IF ERRORCODE() THEN BREAK.      !Check for end of file
    !Process the records
  RELEASE(ViewOrder)          !release held records
END
CLOSE(ViewOrder)
```

See Also:

RELEASE

NEXT

PREVIOUS

NEXT (read next view record in sequence)

NEXT(*view***)**

NEXT Reads the next record(s) in sequence for a VIEW.

view The label of a VIEW declaration.

NEXT reads the next record(s) in sequence from a VIEW and places the appropriate fields in the VIEW structure component files´ data buffer(s). If the VIEW contains JOIN structures, NEXT retrieves the appropriate next set of related records.

The SET statement issued on the VIEW´s primary file before the OPEN(view) statement determines the sequence in which records are read. The first NEXT(*view*) reads the record at the position specified by the SET statement. Subsequent NEXT statements read subsequent records in that sequence. The sequence is not affected by PUT or DELETE statements.

Executing NEXT without a preceding SET, or attempting to read past the end of the primary file in the VIEW posts the "Record Not Available" error.

Errors Posted: 33 Record Not Available
37 File Not Open
43 Record Is Already Held

Example:

```
ViewOrder   VIEW(Customer)   !Declare VIEW structure
            PROJECT(Cus:AcctNumber,Cus:Name)
            JOIN(Hea:AcctKey,Cus:AcctNumber) !Join Header file
              PROJECT(Hea:OrderNumber)
              JOIN(Dtl:OrderKey,Hea:OrderNumber) !Join Detail file
                PROJECT(Det:Item,Det:Quantity)
                JOIN(Pro:ItemKey,Dtl:Item) !Join Product file
                  PROJECT(Pro:Description,Pro:Price)
                END
              END
            END
          END
  CODE
  OPEN(Customer,22h)
  OPEN((Header,22h)
  OPEN(Detail,22h)
  OPEN(Product,22h)
  SET(Cus:AcctKey)
  OPEN(ViewOrder)
   LOOP                            !Read all records through end of primary file
    NEXT(ViewOrder)                  ! read a record sequentially
      IF ERRORCODE() THEN BREAK.       ! break on end of file
    DO PostTrans                    ! call transaction posting routine
  END                              !End loop
```

See Also:

PREVIOUS

HOLD

NOMEMO (read view record without reading memos)

NOMEMO(*view***)**

NOMEMO Arms "memoless" record retrieval.

view The label of a VIEW.

The **NOMEMO** statement arms "memoless" record retrieval for the next NEXT or PREVIOUS statement encountered. The following NEXT or PREVIOUS gets the record but does not get any associated MEMO field(s) for the record. Generally, this speeds up access to the record when the contents of the MEMO field(s) are not needed by the procedure.

Example:

```
Customer  FILE,DRIVER('Clarion'),PRE(Cus) !Declare customer file layout
AcctKey       KEY(Cus:AcctNumber)
Notes         MEMO(1024)
Record      RECORD
AcctNumber      LONG
Name          STRING(20)
Addr          STRING(20)
City          STRING(20)
State           STRING(20)
Zip           STRING(20)
          . .
CustView  VIEW(Customer)   !Declare VIEW structure
          END
  CODE
  OPEN(Customer)
  Cus:AcctNumber = 12345     !Initialize key field
  SET(Cus:AcctKey,Cus:AcctKey)
  OPEN(CustView)
  LOOP
    NOMEMO(CustView)
    NEXT(CustView) !Get that record
      IF ERRORCODE() THEN BREAK.
    !Process the record
  END
  CLOSE(CustView)
```

See Also:

GET

NEXT

PREVIOUS

PREVIOUS (read previous view record in sequence)

PREVIOUS(*view*)

PREVIOUS Reads the previous record in sequence from a VIEW.

view The label of a VIEW declaration.

PREVIOUS reads the previous record(s) in sequence from a VIEW and places the appropriate fields in the VIEW structure component files´ data buffer(s). If the VIEW contains JOIN structures, PREVIOUS retrieves the appropriate previous set of related records.

The SET statement issued on the VIEW´s primary file before the OPEN(view) statement determines the sequence in which records are read. The first PREVIOUS(*view*) reads the record at the position specified by the SET statement. Subsequent PREVIOUS statements read subsequent records in that sequence. The sequence is not affected by PUT or DELETE statements.

Executing PREVIOUS without a preceding SET, or attempting to read past the beginning of the primary file in the VIEW posts the "Record Not Available" error.

Errors Posted: 33 Record Not Available
37 VIEW Not Open
43 Record Is Already Held

Example:

```
ViewOrder   VIEW(Header)
            PROJECT(Hea:OrderNumber)
            JOIN(Dtl:OrderKey,Hea:OrderNumber)      !Join Detail file
              PROJECT(Det:Item,Det:Quantity)
              JOIN(Pro:ItemKey,Dtl:Item)           !Join Product file
                PROJECT(Pro:Description,Pro:Price)
              END
            END
          END
  CODE
  OPEN((Header,22h)
  OPEN(Detail,22h)
  OPEN(Product,22h)
  SET(Cus:AcctKey)
  OPEN(ViewOrder)
   LOOP                        !Read all records through beginning of primary file
    PREViOUS(ViewOrder)           ! read a record sequentially
      IF ERRORCODE() THEN BREAK.   ! break on end of file
    DO PostTrans                ! call transaction posting routine
  END                           !End loop
```

See Also:

NEXT

HOLD

PUT (write VIEW primary file record back)

PUT(*view***)**

PUT Writes the VIEW´s primary file record back to disk.

view The label of a VIEW declaration.

The **PUT** statement writes the current values in the VIEW structure´s primary file´s data buffer to a previously accessed primary file record in the *view*. If the record was held, it is automatically released. PUT writes to the last record accessed with the NEXT or PREVIOUS statements. If the values in the key variables were changed, then the KEYs are updated.

PUT only writes to the primary file in the VIEW because the VIEW structure performs both relational Project and Join operations at the same time. Therefore, it is possible to create a VIEW structure that, if all its component files were updated, would violate the Referential Integrity rules set for the database. The common solution to this problem in SQL-based database products is to write only to the Primary file. Therefore, Clarion has adopted this same industry standard solution.

If a record was not accessed with NEXT or PREVIOUS statements, or was deleted, then the "Record Not Available" error is posted. PUT also posts the "Creates Duplicate Key" error. If any error is posted, then the record is not written to disk.

Errors Posted: 05 Access Denied
33 Record Not Available
40 Creates Duplicate Key

Example:

```
ViewOrder   VIEW(Header)
            JOIN(Dtl:OrderKey,Hea:OrderNumber) !Join Detail file
              PROJECT(Det:Item,Det:Quantity)
            END
          END
  CODE
  OPEN((Header,22h)
  OPEN(Detail,22h)
  SET(Cus:AcctKey)
  OPEN(ViewOrder)
  LOOP                          !Read all records in reverse order
    PREVIOUS(ViewOrder)            ! read a record sequentially
    IF ERRORCODE() THEN BREAK.       ! break at beginning of file
    DO LastInFirstOut               !Call last in first out routine
    PUT(ViewOrder)                 !Write transaction record back to the file
    IF ERRORCODE() THEN STOP(ERROR()).
  END                           !End loop
```

See Also:

NEXT

PREVIOUS

REGET (reget view record)

REGET(*view,string***)**

REGET	Re-gets a specific record in the VIEW.
view	The label of a VIEW declaration.
string	A string constant or variable containing the string returned by the POSITION function.

The **REGET** reads the VIEW record identified by the *string* returned by the POSITION(view) function. The value contained in the *string* returned by the POSITION function, and its length, are file driver dependent. See Also: Supported File Systems.

If the VIEW contains JOIN structures, REGET retrieves the appropriate set of related records.

REGET re-loads all the VIEW component files´ record buffers with complete records. It does not perform the relational "Project" operation. REGET(*view*) is explicitly designed to reset the record buffers to the appropriate records immediately prior to a CLOSE(*view*) statement. However, the processing sequence of the files must be reset with a SET or RESET statement.

Errors Posted: 33 Record Not Available

Example:

```
ViewOrder   VIEW(Customer)   !Declare VIEW structure
            PROJECT(Cus:AcctNumber,Cus:Name)
            JOIN(Hea:AcctKey,Cus:AcctNumber) !Join Header file
              PROJECT(Hea:OrderNumber)
              JOIN(Dtl:OrderKey,Hea:OrderNumber) !Join Detail file
                PROJECT(Det:Item,Det:Quantity)
                JOIN(Pro:ItemKey,Dtl:Item) !Join Product file
                  PROJECT(Pro:Description,Pro:Price)
                END
              END
            END
          END
RecordQue   QUEUE,PRE(Que)
AcctNumber    LIKE(Cus:AcctNumber)
Name        LIKE(Cus:Name)
OrderNumber     LIKE(Hea:OrderNumber)
Item          LIKE(Det:Item)
Quantity    LIKE(Det:Quantity)
Description   LIKE(Pro:Description)
Price         LIKE(Pro:Price)
SavPosition   STRING(260)
          END
  CODE
  OPEN(Customer,22h)
  OPEN((Header,22h)
  OPEN(Detail,22h)
  OPEN(Product,22h)
  SET(Cus:AcctKey)
   OPEN(ViewOrder)
  LOOP                                  !Read all records in file
    NEXT(ViewOrder)                        ! read a record sequentially
      IF ERRORCODE()
        DO DisplayQue
        BREAK
```

```
    END
  RecordQue :=: Cus:Record              !Move record into queue
  RecordQue :=: Hea:Record              !Move record into queue
  RecordQue :=: Dtl:Record              !Move record into queue
  RecordQue :=: Pro:Record              !Move record into queue
  SavPosition = POSITION(ViewOrder)        !Save record position
  ADD(RecordQue)                       ! and add it
    IF ERRORCODE() THEN STOP(ERROR()).
END
ACCEPT
  CASE ACCEPTED()
  OF ?ListBox
    GET(RecordQue,CHOICE())
    REGET(ViewOrder,Que:SavPosition)     !Reset the record buffers
    CLOSE(ViewOrder)                   ! and get the record again
    FREE(RecordQue)
    UpdateProc                         !Call Update Procedure
    BREAK
  END
END
```

See Also:

POSITION

RESET

RELEASE (release a held view record)

RELEASE(*view*)

RELEASE Releases the held record(s).

view The label of a VIEW declaration.

The **RELEASE** statement releases a previously held record in a VIEW. It will not release a record held by another user in a multi-user environment. If the record is not held, or is held by another user, RELEASE is ignored.

Example:

```
ViewOrder   VIEW(Customer)   !Declare VIEW structure
            PROJECT(Cus:AcctNumber,Cus:Name)
            JOIN(Hea:AcctKey,Cus:AcctNumber) !Join Header file
              PROJECT(Hea:OrderNumber)
              JOIN(Dtl:OrderKey,Hea:OrderNumber) !Join Detail file
                PROJECT(Det:Item,Det:Quantity)
                JOIN(Pro:ItemKey,Dtl:Item) !Join Product file
                  PROJECT(Pro:Description,Pro:Price)
                END
              END
            END
          END
  CODE
  OPEN(Customer,22h)
  OPEN((Header,22h)
  OPEN(Detail,22h)
  OPEN(Product,22h)
  SET(Cus:AcctKey)
  OPEN(ViewOrder)
   LOOP                          !Process records Loop
    LOOP                        !Loop to avoid "deadly embrace"
      HOLD(ViewOrder,1)            !Arm Hold on view, try for 1 second
      NEXT(ViewOrder)              !Get and hold the record
      IF ERRORCODE() = 43          !If someone else has it
        CYCLE                     ! and try again
      ELSE
        BREAK                     !Break if not held
      END
    END
    IF ERRORCODE() THEN BREAK.      !Check for end of file
      !Process the records
    RELEASE(ViewOrder)            !release held records
  END
```

See Also:

HOLD

RESET (reset view record sequence position)

RESET(*view,string***)**

RESET	Resets the sequential processing pointer to a specific record in the VIEW.
view	The label of a VIEW.
string	The string returned by the POSITION function.

RESET restores the record pointer to the record identified by the *string* that was returned by the POSITION function. Once RESET has restored the record pointer, either NEXT or PREVIOUS will read that record.

The value contained in the *string* (returned by the POSITION function) and its length, are file driver dependent. See Also: Supported File Systems.

RESET is used in conjunction with POSITION to temporarily suspend and resume sequential VIEW processing.

Example:

```
ViewOrder   VIEW(Customer)   !Declare VIEW structure
            PROJECT(Cus:AcctNumber,Cus:Name)
            JOIN(Hea:AcctKey,Cus:AcctNumber) !Join Header file
              PROJECT(Hea:OrderNumber)
              JOIN(Dtl:OrderKey,Hea:OrderNumber) !Join Detail file
                PROJECT(Det:Item,Det:Quantity)
                JOIN(Pro:ItemKey,Dtl:Item) !Join Product file
                  PROJECT(Pro:Description,Pro:Price)
                END
              END
            END
          END
RecordQue   QUEUE,PRE(Que)
AcctNumber    LIKE(Cus:AcctNumber)
Name        LIKE(Cus:Name)
OrderNumber    LIKE(Hea:OrderNumber)
Item          LIKE(Det:Item)
Quantity    LIKE(Det:Quantity)
Description   LIKE(Pro:Description)
Price         LIKE(Pro:Price)
          END
SavPosition STRING(260)
  CODE
  OPEN(Customer,22h)
  OPEN((Header,22h)
  OPEN(Detail,22h)
  OPEN(Product,22h)
  SET(Cus:AcctKey)
  OPEN(ViewOrder)                    !Top of file in keyed sequence
  LOOP                             !Read all records in file
    NEXT(ViewOrder)                  ! read a record sequentially
      IF ERRORCODE()
        DO DisplayQue
        BREAK
      END
    RecordQue :=: Cus:Record         !Move record into queue
    RecordQue :=: Hea:Record         !Move record into queue
    RecordQue :=: Dtl:Record         !Move record into queue
```

```
   RecordQue :=: Pro:Record        !Move record into queue
   ADD(RecordQue)                 ! and add it
     IF ERRORCODE() THEN STOP(ERROR()).
   IF RECORDS(RecordQue) = 20        !20 records in queue?
    DO DisplayQue                 !Display the queue
 . .                             !End loop

DisplayQue  ROUTINE
     SavPosition = POSITION(ViewOrder)      !Save record position
     DO ProcessQue                       !Display the queue
     FREE(RecordQue)                       ! and free it
     RESET(ViewOrder,SavPosition)          !Reset the record pointer
     NEXT(ViewOrder)                       ! and get the record again
```

See Also:

POSITION

NEXT

PREVIOUS

SKIP (bypass view records in sequence)

SKIP(*view*,*count*)

SKIP	Bypasses records during sequential VIEW processing.
view	The label of a VIEW declaration.
count	A numeric constant or variable. The *count* specifies the number of records to bypass. If the value is positive, records are skipped in forward (NEXT) sequence; if *count* is negative, records are skipped in reverse (PREVIOUS) sequence.

The **SKIP** statement is used to bypass records during sequential VIEW processing. It bypasses records (in the sequence specified by the SET statement) by moving the file pointer *count* records. SKIP is more efficient than NEXT or PREVIOUS for skipping past records because it does not move records into the data buffer(s).

If SKIP reads past the end or beginning of VIEW, the EOF and BOF functions return true (if supported by the file system in use). If no SET has been issued, SKIP is ignored.

Example:

```
ViewOrder   VIEW(Customer)   !Declare VIEW structure
            PROJECT(Cus:AcctNumber,Cus:Name)
            JOIN(Hea:AcctKey,Cus:AcctNumber) !Join Header file
              PROJECT(Hea:OrderNumber)
             JOIN(Dtl:OrderKey,Hea:OrderNumber) !Join Detail file
               PROJECT(Det:Item,Det:Quantity)
               JOIN(Pro:ItemKey,Dtl:Item) !Join Product file
                 PROJECT(Pro:Description,Pro:Price)
               END
              END
            END
          END
SavOrderNo  LONG
  CODE
  OPEN(Customer,22h)
  OPEN((Header,22h)
  OPEN(Detail,22h)
  OPEN(Product,22h)
  SET(Cus:AcctKey)
  OPEN(ViewOrder)                      !Top of file in keyed sequence
  LOOP                               !Process all records
    NEXT(ViewOrder)                    ! Get a record
      IF ERRORCODE() THEN BREAK.
    IF Hea:OrderNumber <> SavOrderNo   ! Check for first item in order
      IF Hea:OrderStatus = ´Cancel´    !   Is it a canceled order?
        SKIP(Items,Vew:ItemCount-1)    !    SKIP rest of the items
        CYCLE                        !    and process next order
    . .                              ! end ifs
    DO ItemProcess                     ! process the item
    SavInvNo = Hea:OrderNUmber         ! save the invoice number
  END                                !End loop
```

WATCH (automatic view concurrency check)

WATCH(*view***)**

WATCH	Arms automatic optimistic concurrency checking.
view	The label of a VIEW declaration.

The **WATCH** statement arms automatic optimistic concurrency checking by the file driver for a following NEXT or PREVIOUS statement in a multi-user environment. Generally, the file driver retains a copy of the retrieved fields from each file on the NEXT or PREVIOUS when it successfully gets the *view*. When the fields are PUT to the *view*, the fields on disk are compared to the original data retrieved. An error is returned by the PUT statement if the data has been changed by another user. The specific action WATCH performs is file driver dependent. See Also: Supported File Systems.

Example:

```
Customer  FILE,DRIVER(´Clarion´),PRE(Cus)   !Declare customer file layout
AcctKey      KEY(Cus:AcctNumber)
Record     RECORD
AcctNumber     LONG
OrderNumber    LONG
Name           STRING(20)
Addr           STRING(20)
City           STRING(20)
State            STRING(20)
Zip            STRING(20)
          . .
CustView  VIEW(Customer)               !Declare VIEW structure
          END
  CODE
  OPEN(Customer,22h)
  SET(Cus:AcctKey)
  OPEN(ViewOrder)                        !Top of file in keyed sequence
  LOOP                                 !Process all records
    WATCH(ViewOrder)                    !Arm concurrency check
    NEXT(ViewOrder)                       ! Get a record
      IF ERRORCODE() THEN BREAK.
    DO ItemProcess                      ! process the item
    PUT(ViewOrder)                      ! and put it back
      IF ERRORCODE() THEN STOP(ERROR()).     !Stop on any error, including
                                       ! record changed by another user
  END
```

View Functions

POSITION (return view record sequence position)

POSITION (return view record sequence position)

POSITION(*sequence***)**

POSITION Identifies a record´s unique position in the VIEW.

sequence The label of a VIEW declaration.

POSITION returns a STRING which identifies a record´s unique position within the *sequence*. POSITION returns the position of the last record accessed in the VIEW. The POSITION function is used with RESET to temporarily suspend and resume sequential processing.

The return string for POSITION(view) contains the sequence set by the SET statement on the primary file issued immediately before the OPEN(view) statement. It also contains the file system´s POSITION return value for the primary file key and all secondary file linking keys. This allows POSITION(view) to accurately define a position for all related records in the VIEW.

As a general rule, for file systems that have record numbers, the size of the STRING returned by POSITION(file) is 4 bytes. The return string from POSITION(key) is 4 bytes plus the sum of the sizes of the fields in the key. For file systems that do not have record numbers, the size of the STRING returned by POSITION(file) is usually the sum of the sizes of the fields in the Primary Key (the first KEY on the FILE that does not have the DUP or OPT attribute). The return string from POSITION(key) is the sum of the sizes of the fields in the Primary Key plus the sum of the sizes of the fields in the key.

Return Data Type: STRING

Example:

```
ViewOrder   VIEW(Customer)   !Declare VIEW structure
            PROJECT(Cus:AcctNumber,Cus:Name)
            JOIN(Hea:AcctKey,Cus:AcctNumber) !Join Header file
              PROJECT(Hea:OrderNumber)
              JOIN(Dtl:OrderKey,Hea:OrderNumber) !Join Detail file
                PROJECT(Det:Item,Det:Quantity)
                JOIN(Pro:ItemKey,Dtl:Item) !Join Product file
                  PROJECT(Pro:Description,Pro:Price)
                END
              END
            END
          END
RecordQue   QUEUE,PRE(Que)
AcctNumber    LIKE(Cus:AcctNumber)
Name        LIKE(Cus:Name)
OrderNumber     LIKE(Hea:OrderNumber)
Item          LIKE(Det:Item)
Quantity    LIKE(Det:Quantity)
Description   LIKE(Pro:Description)
Price         LIKE(Pro:Price)
          END
SavPosition STRING(260)
  CODE
  OPEN(Customer,22h)
  OPEN((Header,22h)
  OPEN(Detail,22h)
  OPEN(Product,22h)
  SET(Cus:AcctKey)
  OPEN(ViewOrder)                          !Top of file in keyed sequence
  LOOP                                   !Read all records in file
    NEXT(ViewOrder)                        ! read a record sequentially
```

```
    IF ERRORCODE()
      DO DisplayQue                     !Display the queue
      BREAK
    END
   RecordQue :=: Cus:Record               !Move record into queue
   RecordQue :=: Hea:Record               !Move record into queue
   RecordQue :=: Dtl:Record               !Move record into queue
   RecordQue :=: Pro:Record               !Move record into queue
   ADD(RecordQue)                       ! and add it
     IF ERRORCODE() THEN STOP(ERROR()).
   IF RECORDS(RecordQue) = 20             !20 records in queue?
     DO DisplayQue                      !Display the queue
  . .

DisplayQue  ROUTINE
     SavPosition = POSITION(ViewOrder)       !Save record position
     DO ProcessQue                      !Display the queue
     FREE(RecordQue)                      ! and free it
     RESET(ViewOrder,SavPosition)         !Reset the record pointer
     NEXT(ViewOrder)                      ! and get the record again
```

See Also:

RESET

Memory Queues

Queue Structure

QUEUE (declare a memory QUEUE structure)

label **QUEUE** [,**PRE**] [,**STATIC**] [,**THREAD**] [,**TYPE**] [,**BINDABLE**] [,**EXTERNAL**] [,**DLL**]
fieldlabel *variable* [,**NAME( )**]
END

QUEUE	Declares a memory queue structure.
label	The name of the QUEUE.
PRE	Declare a *fieldlabel* prefix for the structure.
STATIC	Declares a QUEUE, local to a PROCEDURE or FUNCTION, whose buffer is allocated in static memory.
THREAD	Specify memory for the queue is allocated once for each execution thread. Must be used with the STATIC attribute on Procedure Local data.
TYPE	Specify the QUEUE is a type definition for QUEUEs passed as parameters.
BINDABLE	Specify all variables in the queue may be used in dynamic expressions.
EXTERNAL	Specify the QUEUE is defined, and its memory is allocated, in an external library.
DLL	Specify the QUEUE is defined in a .DLL. This is required in addition to the EXTERNAL attribute.
fieldlabel	The name of the *variables* in the queue.
variable	Data declaration. The sum of the memory required for all declared *variables* in the QUEUE must not be greater than 65,000 bytes in 16-bit applications and 4MB in 32-bit applications.

QUEUE declares a memory QUEUE structure. The *label* of the QUEUE structure is used in queue processing statements and functions. When used in assignment statements, expressions, or parameter lists, a QUEUE is treated like a GROUP data type.

A QUEUE may be thought of as a memory file which is internally implemented as a dynamic array of the QUEUE entries. When a QUEUE is declared, a data buffer is allocated (just as with a file). Each entry in the QUEUE occupies exactly the same amount of memory as the data buffer with no per-entry overhead (also no data compression or clipping).

The data buffer for a Procedure local QUEUE (declared in the data section of a PROCEDURE or FUNCTION) is allocated on the stack (unless it has the STATIC attribute or is too large). The memory allocated to the entries in a procedure-local QUEUE without the STATIC attribute is allocated only until you FREE the QUEUE, or you RETURN from the PROCEDURE or FUNCTIONthe QUEUE is automatically FREEd upon RETURN.

For a Global data, Module data, or Local data QUEUE with the STATIC attribute, the data buffer is allocated static memory and the data in the buffer is persistent between procedure calls. The memory allocated to the entries in the QUEUE remains allocated until you FREE the QUEUE.

The *variables* in the QUEUEs data buffer are not automatically initialized to any value, they must be explicitly assigned values. Do not assume that they contain blanks or zero before your programs first assignment to them.

As entries are added to the QUEUE, memory for the entry is dynamically allocated and the data is copied from the buffer to the entry. As entries are deleted from the QUEUE, the memory used by the deleted entry is freed. The maximum number of entries in a QUEUE is 1,000,000. The memory used by each

entry in the QUEUE is equal to the total of the field sizes.

A QUEUE with the BINDABLE attribute makes all the variables within the QUEUE available for use in a dynamic expression, without requiring a separate BIND statement for each (allowing BIND(queue) to enable all the fields in the queue). The contents of each variables NAME attribute is the logical name used in the dynamic expression. If no NAME attribute is present, the label of the variable (including prefix) is used. Space is allocated in the .EXE for the names of all of the variables in the structure. This creates a larger program that uses more memory than it normally would. Therefore, the BINDABLE attribute should only be used when a large proportion of the constituent fields are going to be used.

A QUEUE with the TYPE attribute is not allocated any memory; it is only a type definition for QUEUEs that are passed as parameters to PROCEDUREs or FUNCTIONs. This allows the receiving procedure to directly address component fields in the passed QUEUE. The parameter declaration on the PROCEDURE or FUNCTION statement instantiates a local prefix for the passed QUEUE as it names the passed QUEUE for the procedure. For example, PROCEDURE(LOC:PassedGroup) declares the procedure uses the LOC: prefix (along with the individual field names used in the type declaration) to directly address component fields of the QUEUE actually passed as the parameter.

Example:

```
NameQue    QUEUE,PRE(Nam)                    !Declare a queue
Name       STRING(20)
Zip        DECIMAL(5,0),NAME(SortField)
         END                                !End queue structure
```

See Also:

PRE

STATIC

NAME

FREE

THREAD

PRE (set label prefix)

PRE(*prefix***)**

PRE	Provides a label prefix for variables declared in the QUEUE.
prefix	Acceptable characters are alphabet letters, numerals 0 through 9, and the underscore character. A *prefix* must start with an alphabet character and must not be a reserved word.

The **PRE** attribute provides a label prefix for variables declared in the QUEUE. It is used to distinguish between identical variable names that occur in different structures. When referenced in executable statements, assignments, and parameter lists, a *prefix* is attached to a label by a colon (Pre:Label).

Example:

```
SaveQueue   QUEUE,PRE(Sav)
Field1      LONG              !Referenced as Sav:Field1
Field2      STRING            !Referenced as Sav:Field2
          END
```

See Also:

Reserved Words

STATIC (set local queue static)

STATIC

The **STATIC** attribute allows the data buffer of a QUEUE declared within a PROCEDURE or FUNCTION to be allocated static memory instead of stack memory. This makes any value contained in the data buffer "persistent" from one instance of the procedure to the next.

Example:

```
SomeProc  PROCEDURE
SaveQueue   QUEUE,STATIC       !Static QUEUE data buffer
Field1      LONG             !Value retained between
Field2      STRING           ! procedure calls
          END
```

See Also:

Data Declarations and Memory Allocation

THREAD (set thread-specific static queue)

THREAD

The **THREAD** attribute declares a static QUEUE which is allocated memory separately for each execution thread in the program. This makes the values contained in the QUEUE dependent upon which thread is executing. Whenever a new execution thread is begun, a new instance of the QUEUE, specific to that thread, is created.

The THREAD attribute implies a static QUEUE, so the STATIC attribute is unnecessary on a Procedure Local QUEUE. This attribute creates a lot of runtime "overhead," particularly on Global or Module data. Therefore, it should be used only when absolutely necessary.

Example:

```
SomeProc  PROCEDURE
SaveQueue   QUEUE,THREAD       !Static QUEUE data buffer
Field1      LONG            !Thread-specific QUEUE
Field2      STRING
          END
```

See Also:

Data Declarations and Memory Allocation

NAME (set queue variable external name)

NAME([*name*] **)**

NAME Specifies an "external" name for queue processing.

name A string constant containing the "external" name of the variable.

The **NAME** attribute on a variable declared in a QUEUE structure specifies an "external" *name* for queue processing. The *name* is an alternate method of addressing the variables in the QUEUE used by the SORT, GET, PUT, and ADD statements.

Example:

```
SortQue   QUEUE,PRE(Que)
Field1    STRING(10),NAME(´FirstField´)      !QUEUE SORT NAME
Field2    LONG,NAME(´SecondField´)          !QUEUE SORT NAME
        END
```

See Also:

QUEUE

SORT

GET

PUT

ADD

TYPE (QUEUE type definition)

TYPE

The **TYPE** attribute creates a QUEUE that is not allocated any memory; it is only a type definition for QUEUEs that are passed as parameters to PROCEDUREs or FUNCTIONs. This allows the receiving procedure to directly address component fields in the passed QUEUE. The parameter declaration on the PROCEDURE or FUNCTION statement instantiates a local prefix for the passed QUEUE as it names the passed QUEUE for ther procedure. For example, PROCEDURE(LOC:PassedGroup) declares the procedure uses the LOC: prefix (along with the individual field names used in the type definition) to directly address component fields of the QUEUE passed as the parameter.

Example:

```
PassQue   QUEUE,TYPE              !Type-definition for passed QUEUE parameters
F1        STRING(20)             ! first field
F2        STRING(1)              ! middle field
F3        STRING(20)             ! last field
        END

   MAP
     MyProc1(PassQue)            !Passes a QUEUE defined the same as PassGroup
   END

NameQue   QUEUE,PRE(Nme)           !Name queue
First       STRING(20)             ! first name
Middle    STRING(1)              ! middle initial
Last      STRING(20)             ! last name
        END                     !End queue declaration

  CODE
  MyProc1(NameQue)               !Call proc passing NameQue as parameter

MyProc1     PROCEDURE(LOC:PassedGroup)   !Proc to receive QUEUE parameter
LocalVar  STRING(20)
  CODE
  LocalVar = LOC:F1                !Assign value in Nme:First to LocalVar
                                !  from passed parameter
```

EXTERNAL (set queue defined externally)

EXTERNAL

The **EXTERNAL** attribute specifies that the QUEUE on which it is placed is defined in an external library. Therefore, a QUEUE with the EXTERNAL attribute is declared and may be referenced in the Clarion code, but is not allocated memory. The memory for the QUEUE is allocated by the external library. This allows the Clarion program access to QUEUEs declared as public in external libraries.

When using EXTERNAL to declare a QUEUE shared by multiple libraries (.LIBs, or .DLLs and .EXE), only one library should define the QUEUE without the EXTERNAL attribute. All the other libraries (and the .EXE) should declare the QUEUE with the EXTERNAL attribute. This ensures that there is only one memory allocation for the QUEUE and all the libraries and the .EXE will reference the same memory when referring to that QUEUE.

The QUEUE declarations in all libraries (or .EXEs) that reference common QUEUEs must be EXACTLY the same (with the appropriate addition of the EXTERNAL attribute). If they are not exactly the same, data corruption could occur. Any incompatibilities between libraries cannot be detected by the compiler or linker, therefore it is the programmers responsibility to ensure that consistency is maintained.

One suggested way of coding large systems using many .DLLs and/or .EXEs that share the same variables would have one .DLL containing the actual data definition that only contains FILE and global variable and QUEUE definitions that are shared among all (or most) of the .DLLs and .EXEs. This makes one central library in which the actual file definitions are maintained. This one central .DLL is linked into all .EXEs that use those common files. All other .DLLs and/or .EXEs in the system would declare the common FILEs, QUEUEs, and variables with the EXTERNAL attribute.

Example:

```
Names     QUEUE,EXTERNAL                !A queue declared in an external library
Name      STRING(20)
FileName  STRING(8),NAME(FName)      !Dynamic name: FName
Dot       STRING(1)                  !Dynamic name: Dot
Extension   STRING(3),NAME(EXT)        !Dynamic name: EXT
        END
```

DLL (set queue defined externally in .DLL)

DLL([*flag*] **)**

DLL Declares a QUEUE defined externally in a .DLL.

flag A numeric constant, equate, or Project system define which specifies the attribute as active or not. If the *flag* is zero, the attribute is not active, just as if it were not present. If the *flag* is any value other than zero, the attribute is active.

The **DLL** attribute specifies that the QUEUE on which it is placed is defined in a .DLL. A QUEUE with DLL attribute must also have the EXTERNAL attribute. The DLL attribute is required for 32-bit applications because .DLLs are relocatable in a 32-bit flat address space, which requires one extra dereference by the compiler to address the QUEUE.

The QUEUE declarations in all libraries (or .EXEs) that reference common QUEUEs must be EXACTLY the same (with the appropriate addition of the EXTERNAL and DLL attributes). If they are not exactly the same, data corruption could occur. Any incompatibilities between libraries cannot be detected by the compiler or linker, therefore it is the programmers responsibility to ensure that consistency is maintained.

When using EXTERNAL and DLL to declare a QUEUE shared by .DLLs and .EXE, only one .DLL should define the QUEUE without the EXTERNAL and DLL attributes. All the other .DLLs (and the .EXE) should declare the QUEUE with the EXTERNAL and DLL attributes. This ensures that there is only one memory allocation for the QUEUE and all the .DLLs and the .EXE will reference the same memory when referring to that QUEUE.

One suggested way of coding large systems using many .DLLs and/or .EXEs that share the same QUEUEs would have one .DLL containing the actual data definition that only contains FILE and global QUEUE and variable definitions that are shared among all (or most) of the .DLLs and .EXEs. This makes one central library in which the actual file definitions are maintained. This one central .DLL is linked into all .EXEs that use those common files. All other .DLLs and/or .EXEs in the system would declare the common variables with the EXTERNAL and DLL attributes.

Example:

```
DLLQueue    QUEUE,PRE(Que),EXTERNAL,DLL     !A queue declared in an external .DLL
  TotalCount    LONG
            END
```

See Also: EXTERNAL

Queue Procedures

ADD (add an entry)

```
                | pointer          |
ADD(queue [,    | [+]key,...,[-]key] | ] )
                | name             |
```

ADD Writes a new entry to the QUEUE.

queue The label of a QUEUE structure, or the label of a passed QUEUE parameter.

pointer A numeric constant, variable, or numeric expression. The *pointer* must be in the range from 1 to the number of entries in the memory queue.

+ - The leading plus or minus sign specifies the *key* is sorted in ascending or descending sequence.

key The label of a field declared within the QUEUE structure. If the QUEUE has a PRE attribute, the *key* must include the prefix.

name A string constant, variable, or expression containing the NAME attribute of QUEUE fields, separated by commas, and optional leading + or - signs for each attribute. This parameter is case sensitive.

ADD writes a new entry from the QUEUE structure data buffer to the QUEUE. If there is not enough memory to ADD a new entry, the "Insufficient Memory" error is posted.

ADD(*queue*) Appends a new entry to the end of the QUEUE.

ADD(*queue,pointer*)
Places a new entry at the relative position specified by the *pointer* parameter. If there is an entry already at the relative *pointer* position, it is "pushed down" to make room for the new entry. All following pointers are readjusted to account for the new entry. For example, an entry added at position 10 pushes entry 10 to position 11, entry 11 to position 12, etc. If *pointer* is zero or greater than the number of entries in the QUEUE, the entry is added at the end.

ADD(*queue,key*) Inserts a new entry in a sorted memory queue. Multiple *key* parameters may be used (up to 16), separated by commas, with optional leading plus or minus signs to indicate ascending or descending sequence. The entry is inserted immediately after all other entries with matching *key* values. If there are no entries, ADD(*queue,key*) may be used to build the QUEUE in sorted order.

ADD(*queue,name*)
Inserts a new queue entry in a sorted memory queue. The *name* string must contain the NAME attributes of the fields, separated by commas, with optional leading plus or minus signs to indicate ascending or descending sequence. The entry is inserted immediately after all other entries with matching field values. If there are no entries, ADD(*queue,name*) may be used to build the QUEUE in sorted order.

Errors Posted: 08 Insufficient Memory
75 Invalid Field Type Descriptor

Example:

```
NameQue   QUEUE,PRE(Que)
Name      STRING(20),NAME('FirstField')
Zip       DECIMAL(5,0),NAME('SecondField')
        END
```

```
CODE
ADD(NameQue)                    !Add an entry at the end of the QUEUE
ADD(NameQue,1)                  !Add an entry at position 1
Que:Name = ´Jones´              !Initialize fields
Que:Zip = 12345
ADD(NameQue,+Que:Name,-Que:Zip)    !Ascending name, descending zip order
Que:Name = ´Smith´              !Initialize fields
Que:Zip = 12345
ADD(NameQue,´+FirstField,-SecondField´)
                               !Add in name, descending zip order
```

DELETE (delete an entry)

DELETE(*queue***)**

DELETE Removes a QUEUE entry.

queue The label of a QUEUE structure, or the label of a passed QUEUE parameter.

DELETE removes the QUEUE entry at the position of the last successful GET or ADD and de-allocates its memory. If no previous GET or ADD was executed, the "Queue Entry Not Found" error is posted. All forward and backward chain pointers are automatically adjusted to compensate for the deleted entry.

Errors Posted: 08 Insufficient Memory
30 Entry Not Found

Example:

```
Que:Name = ´Jones´        !Initialize the key
GET(NameQue,Que:Name)       !Get the matching record
DELETE(NameQue)             !Delete the entry
```

FREE (delete all entries)

FREE(*queue***)**

FREE Deletes all entries from a QUEUE.

queue The label of a QUEUE structure, or the label of a passed QUEUE parameter.

FREE deletes all entries from a QUEUE and de-allocates the memory they occupied. It also de-allocates the memory used by the QUEUE´s "overhead." FREE does not clear the QUEUE´s data buffer.

Errors Posted: 08 Insufficient Memory

Example:

```
FREE(Location)          !Free the location queue
FREE(NameQue)           !Free the name queue
```

GET (read an entry)

```
                | pointer          |
GET(queue , | [+]key,...,[-]key] | )
                | name             |
```

GET Retrieves a specific QUEUE entry.

queue The label of a QUEUE structure, or the label of a passed QUEUE parameter.

pointer A numeric constant, variable, or numeric expression. The *pointer* must be in the range from 1 to the number of entries in the memory queue.

+ - The leading plus or minus sign specifies the *key* is sorted in ascending or descending sequence.

key The label of a field declared within the QUEUE structure. If the QUEUE has a PRE attribute, the *key* must include the prefix.

name A string constant, variable, or expression containing the NAME attribute of QUEUE fields, separated by commas, and optional leading + or - signs for each attribute. This parameter is case sensitive.

GET reads an entry into the QUEUE structure data buffer for processing. If GET does not find a match, the "Queue Entry Not Found" error is posted.

GET(*queue,pointer*)
Retrieves the entry at the relative entry position specified by the *pointer* value.

GET(*queue,key*) Searches for a QUEUE entry that matches the value in the *key* field(s). Multiple *key* parameters may be used (up to 16), separated by commas. The QUEUE must already be sorted on the field(s) used as the *key* parameter(s).

GET(*queue,name*)
Searches for a QUEUE entry that matches the value in the *name* field(s). The *name* string must contain the NAME attributes of the fields, separated by commas, with optional leading plus or minus signs to indicate ascending or descending sequence. The QUEUE must already be sorted on the field(s) listed in the *name* parameter.

Errors Posted: 08 Insufficient Memory
30 Entry Not Found
75 Invalid Field Type Descriptor

Example:

```
NameQue   QUEUE,PRE(Que)
Name      STRING(20),NAME('FirstField')
Zip       DECIMAL(5,0),NAME('SecondField')
        END
  CODE
  DO BuildQue                          !Call routine to build the queue
  GET(NameQue,1)                       !Get the first entry
    IF ERRORCODE() THEN STOP(ERROR()).

  Que:Name = 'Jones'                   !Initialize key field
  GET(NameQue,Que:Name)                 !Get the matching record
    IF ERRORCODE() THEN STOP(ERROR()).

  Que:Name = Fil:Name                   !Initialize to value in Fil:Name
```

```
GET(NameQue,Que:Name)                   !Get the matching record
  IF ERRORCODE() THEN STOP(ERROR()).

Que:Name = ´Smith´                     !Initialize the key fields
Que:Zip = 12345
GET(NameQue,´FirstField,SecondField´)     !Get the matching record
  IF ERRORCODE() THEN STOP(ERROR()).
```

See Also:

SORT

PUT (write an entry)

PUT(*queue* , | [**+**]*key*,...,[**-**]*key*] | **)**
| *name* |

PUT Writes an entry back to the QUEUE.

queue The label of a QUEUE structure, or the label of a passed QUEUE parameter.

+ - The leading plus or minus sign specifies the *key* is sorted in ascending or descending sequence.

key The label of a field declared within the QUEUE structure. If a the QUEUE has a PRE attribute, the *key* must include the prefix.

name A string constant, variable, or expression containing the NAME attribute of QUEUE fields, separated by commas, and optional leading + or - signs for each attribute. This parameter is case sensitive.

PUT writes the contents of the data buffer back to the QUEUE after a successful GET or ADD. If no previous GET or ADD was executed, the "Queue Entry Not Found" error is posted.

PUT(*queue*) Writes the data buffer back to the same relative position within the QUEUE of the last successful GET or ADD.

PUT(*queue,key*) Returns an entry to a sorted memory queue after a successful GET or ADD, maintaining the sort order if any *key* fields have changed value. Multiple *key* parameters may be used (up to 16), separated by commas, with optional leading plus or minus signs to indicate ascending or descending sequence. The entry is inserted immediately after all other entries with matching *key* values.

PUT(*queue,name*)
Returns an entry to a sorted memory queue after a successful GET or ADD, maintaining the sort order if any key fields have changed value. The *name* string must contain the NAME attributes of the fields, separated by commas, with optional leading plus or minus signs to indicate ascending or descending sequence. The entry is inserted immediately after all other entries with matching field values.

Errors Posted: 08 Insufficient Memory
30 Entry Not Found
75 Invalid Field Type Descriptor

Example:

```
NameQue   QUEUE,PRE(Que)
Name      STRING(20),NAME('FirstField')
Zip       DECIMAL(5,0),NAME('SecondField')
        EBD
  CODE
  DO BuildQue                          !Call routine to build the queue

  Que:Name = 'Jones'                 !Initialize key field
  GET(NameQue,Que:Name)                !Get the matching record
    IF ERRORCODE() THEN STOP(ERROR()).
  Que:Zip = 12345                      !Change the zip
  PUT(NameQue)                       !Write the changes to the queue
    IF ERRORCODE() THEN STOP(ERROR()).
```

```
Que:Name = ´Jones´                !Initialize key field
GET(NameQue,Que:Name)               !Get the matching record
  IF ERRORCODE() THEN STOP(ERROR()).
Que:Name = ´Smith´                !Change key field
PUT(NameQue,Que:Name)               !Write changes to the queue
  IF ERRORCODE() THEN STOP(ERROR()).

Que:Name = ´Smith´                  !Initialize key field
GET(NameQue,´FirstField´)             !Get the matching record
  IF ERRORCODE() THEN STOP(ERROR()).
Que:Name = ´Jones´                !Change key field
PUT(NameQue,´FirstField´)             !Write changes to the queue
  IF ERRORCODE() THEN STOP(ERROR()).
```

SORT (sort entries)

SORT(*queue*, | [**+**]*key*,...,[**-**]*key*] | **)**
| *name* |

SORT Reorders entries in a QUEUE.

queue The label of a QUEUE structure, or the label of a passed QUEUE parameter.

+ - The leading plus or minus sign specifies the *key* will be sorted in ascending or descending sequence.

key The label of a field declared within the QUEUE structure. If the QUEUE has a PRE attribute, the *key* must include the prefix.

name A string constant, variable, or expression containing the NAME attribute of QUEUE fields, separated by commas, and optional leading + or - signs for each attribute. This parameter is case sensitive.

SORT reorders the entries in a QUEUE. QUEUE entries with identical key values maintain their relative position. SORT does not move data, it rearranges the pointers between the entries.

SORT(*queue,key*) Reorders the QUEUE in the sequence specified by the *key*. Multiple *key* parameters may be used (up to 16), separated by commas, with optional leading plus or minus signs to indicate ascending or descending sequence.

SORT(*queue,name*)
Reorders the QUEUE in the sequence specified by the *name* string. The *name* string must contain the NAME attributes of the fields, separated by commas, with leading plus or minus signs to indicate ascending or descending sequence.

Errors Posted: 08 Insufficient Memory
75 Invalid Field Type Descriptor

Example:

```
Location    QUEUE,PRE(Loc)
Name        STRING(20),NAME('FirstField')
City        STRING(10),NAME('SecondField')
State         STRING(2)
Zip         DECIMAL(5,0)
          END
  CODE
  SORT(Location,Loc:State,Loc:City,Loc:Zip)      !Sort by zip in city in state
  SORT(Location,+Loc:State,-Loc:Zip)           !Sort descending by zip in state
  SORT(Location,'FirstField,-SecondField')     !Sort descending by city in name
```

Queue Functions

POINTER (return last entry position)

POINTER(*queue***)**

POINTER Returns the entry number of the last entry accessed in a QUEUE.

queue The label of a QUEUE structure, or the label of a passed QUEUE parameter.

The **POINTER** function returns a LONG integer specifying the entry number of the last QUEUE entry accessed by ADD or GET.

Return Data Type: LONG

Example:

```
Que:Name = ´Jones´                    !Initialize key field
GET(NameQue,Que:Name)                   !Get the entry
  IF ERRORCODE() THEN STOP(ERROR()).       ! and check for errors
SavPoint = POINTER(NameQue)               !Save the pointer
```

RECORDS (return number of entries)

RECORDS(*queue***)**

RECORDS Returns the number of entries in a QUEUE.

queue The label of a QUEUE structure, or the label of a passed QUEUE parameter.

The **RECORDS** function returns a LONG integer containing the number of entries in the QUEUE.

Return Data Type: LONG

Example:

```
Entries# = RECORDS(Location)      !Determine number of entries
LOOP I# = 1 TO Entries#             !Loop through QUEUE
  GET(Location,I#)                ! getting each entry
    IF ERRORCODE() THEN STOP(ERROR()).
  DO SomeProcess                 ! process the entry
END
```

Miscellaneous Procedures and Functions

Mathematical Functions

ABS (return absolute value)

ABS(*expression***)**

ABS Returns absolute value.

expression A constant, variable, or expression.

The **ABS** function returns the absolute value of an *expression*. The absolute value of a number is always positive (or zero).

Return Data Type: REAL or DECIMAL

Example:

```
C = ABS(A - B)              !C is absolute value of the difference
IF B < 0 THEN B = ABS(B).     !If b is negative make it positive
```

See Also:

BCD Operations and Functions

INRANGE (check number within range)

INRANGE(*expression*,*low*,*high***)**

INRANGE	Return number in valid range.
expression	A numeric constant, variable, or expression.
low	A numeric constant, variable, or expression of the lower boundary of the range.
high	A numeric constant, variable, or expression of the upper boundary of the range.

The **INRANGE** function compares a numeric *expression* to an inclusive range of numbers. If the value of the *expression* is within the range, the function returns the value 1 for "true." If the *expression* is greater than the *high* parameter, or less than the *low* parameter, the function returns a zero for "false."

Return Data Type: LONG

Example:

```
IF INRANGE(Date % 7,1,5)   !If this is a week day
  DO WeekdayRate          ! use the weekday rate
ELSE                     !Otherwise
  DO WeekendRate            ! use the weekend rate
END
```

INT (truncate fraction)

INT(*expression***)**

INT Return integer.

expression A numeric constant, variable, or expression.

The **INT** function returns the integer portion of a numeric expression. No rounding is performed, and the sign remains unchanged.

Return Data Type: REAL or DECIMAL

Example:

```
    !INT(8.5)      returns 8
    !INT(-5.9)    returns -5

x = INT(y)          !Return integer portion of y variable contents
```

See Also:

BCD Operations and Functions

LOGE (return natural logarithm)

LOGE(*expression***)**

LOGE Returns the natural logarithm.

expression A numeric constant, variable, or expression. If the value of the *expression* is less than zero, the return value is zero. The natural logarithm is undefined for values less than zero.

The **LOGE** (pronounced "log-e") function returns the natural logarithm of a numeric *expression*. The natural logarithm of a value is the power to which **e** must be raised to equal that value. The value of **e** is 2.71828182846.

Return Data Type: REAL

Example:

```
  !LOGE(2.71828182846)  returns 1
  !LOGE(1)              returns 0

LogVal = LOGE(Val)    !Get the natural log of Val
```

LOG10 (return base 10 logarithm)

LOG10(*expression***)**

LOG10 Returns base 10 logarithm.

expression A numeric constant, variable, or expression. If the value of the *expression* is zero or less, the return value will be zero. The base 10 logarithm is undefined for values less than or equal to zero.

The **LOG10** (pronounced "log ten") function returns the base 10 logarithm of a numeric *expression*. The base 10 logarithm of a value is the power to which 10 must be raised to equal that value.

Return Data Type: REAL

Example:

```
  !LOG10(10)    returns 1
  !LOG10(1)       returns 0

LogStore = LOG10(Var)        !Store the log 10 of var
```

RANDOM (return random number)

RANDOM(*low*,*high***)**

RANDOM	Returns random integer.
low	A numeric constant, variable, or expression for the lower boundary of the range.
high	A numeric constant, variable, or expression for the upper boundary of the range.

The **RANDOM** function returns a random integer between the *low* and *high* parameter values, inclusively. The *low* and *high* parameters may be any numeric expression, but only their integer portion is used for the inclusive range.

Return Data Type: LONG

Example:

```
LOOP X# = 1 TO 6
  LottoNbr[X#] = RANDOM(1,49)       !Pick numbers for Lotto
END
```

ROUND (return rounded number)

ROUND(*expression*,*order*)

ROUND	Returns rounded value.
expression	A numeric constant, variable, or expression.
order	A numeric expression with a value equal to a power of ten, such as 1, 10, 100, 0.1, 0.001, etc. If the value is not an even power of ten, the next lowest power is used; 0.55 will use 0.1 and 155 will use 100.

The **ROUND** function returns the value of an *expression* rounded to a power of ten. If the *order* is a LONG or DECIMAL Base Type, then rounding is performed as a BCD operation. Note that if you want to round a real number larger than 1e30, you should use ROUND(num,1.0e0), and not ROUND(num,1). The ROUND function is very efficient ("cheap") as a BCD operation and should be used to compare REALs to DECIMALs at decimal width.

Return Data Type: DECIMAL or REAL

Example:

```
  !ROUND(5163,100)        returns 5200
  !ROUND(657.50,1)        returns 658
  !ROUND(51.63594,.01)    returns 51.64

Commission = ROUND(Price / Rate,.01)   !Round the commission to the nearest cent
```

See Also:

BCD Operations and Functions

SQRT (return square root)

SQRT(*expression***)**

SQRT Returns square root.

expression A numeric constant, variable, or expression. If the value of the expression is less than zero, the return value is zero.

The **SQRT** function returns the square root of the *expression*. If X represents any positive real number, the square root of X is a number that, when multiplied by itself, produces a product equal to X.

Return Data Type: REAL

Example:

```
Length = SQRT(X^2 + Y^2) !Find the distance from 0,0 to x,y (pythagorean theorem)
```

Trigonometric Functions

Trigonometric functions return values representing angles and ratios of the sides of a right triangle (a triangle containing a 90-degree angle). The hypotenuse is the side of the triangle opposite the right (90-degree) angle. For either of the other two angles, the adjacent side forms the angle with the hypotenuse, and the opposite side is opposite the angle. (See any good Trigonometry text for further explanation of these terms.)

Angles are expressed in radians. PI is a constant which represents the ratio of the circumference and radius of a circle. There are 2*PI radians (or 360 degrees) in a circle.

The following equates provide high precision constants for PI and the conversion factors between degrees and radians.

```
PI EQUATE(3.1415926535898)      !The value of PI
Rad2Deg EQUATE(57.295779513082)   !Number of degrees in a radian
Deg2Rad EQUATE(.0174532925199)    !Number of radians in a degree
```

SIN (return sine)

COS (return cosine)

TAN (return tangent)

ASIN (return arcsine)

ACOS (return arccosine)

ATAN (return arctangent)

SIN (return sine)

SIN(*radians***)**

SIN Returns sine.

radians A numeric constant, variable or expression for the angle expressed in radians.

The **SIN** function returns the trigonometric sine of an angle measured in *radians*. The sine is the ratio of the length of the angle´s opposite side divided by the length of the hypotenuse.

Return Data Type: REAL

Example:

```
Angle = 45 * Deg2Rad       !Translate 45 degrees to Radians
SineAngle = SIN(Angle)     !Get the sine of 45 degree angle
```

COS (return cosine)

COS(*radians***)**

COS Returns cosine.

radians A numeric constant, variable or expression for the angle in radians.

The **COS** function returns the trigonometric cosine of an angle measured in *radians*. The cosine is the ratio of the length of the angle´s adjacent side divided by the length of the hypotenuse.

Return Data Type: REAL

Example:

```
Angle = 45 * Deg2Rad          !Translate 45 degrees to Radians
CosineAngle = COS(Angle)      !Get the cosine of 45 degree angle
```

TAN (return tangent)

TAN(*radians***)**

TAN Returns tangent.

radians A numeric constant, variable or expression for the angle in radians.

The **TAN** function returns the trigonometric tangent of an angle measured in *radians*. The tangent is the ratio of the angle´s opposite side divided by its adjacent side.

Return Data Type: REAL

Example:

```
Angle = 45 * Deg2Rad          !Translate 45 degrees to Radians
TangentAngle = TAN(Angle)      !Get the tangent of 45 degree angle
```

ASIN (return arcsine)

ASIN(*expression***)**

ASIN Returns inverse sine.

expression A numeric constant, variable, or expression for the value of the sine.

The **ASIN** function returns the inverse sine. The inverse of a sine is the angle that produces the sine. The return value is the angle in radians.

Return Data Type: REAL

Example:

```
InvSine = ASIN(SineAngle)     !Get the Arcsine
```

See Also:

SIN

ACOS (return arccosine)

ACOS(*expression***)**

ACOS Returns inverse cosine.

expression A numeric constant, variable, or expression for the value of the cosine.

The **ACOS** function returns the inverse cosine. The inverse of a cosine is the angle that produces the cosine. The return value is the angle in radians.

Return Data Type: REAL

Example:

```
InvCosine = ACOS(CosineAngle)    !Get the Arccosine
```

See Also:

COS

ATAN (return arctangent)

ATAN(*expression***)**

ATAN Returns inverse tangent.

expression A numeric constant, variable, or expression for the value of the tangent.

The **ATAN** function returns the inverse tangent. The inverse of a tangent is the angle that produces the tangent. The return value is the angle in radians.

Return Data Type REAL

Example:

```
InvTangent = ATAN(TangentAngle)    !Get the Arctangent
```

See Also:

TAN

String Functions

ALL (return repeated characters)

ALL(*string* [,*length*]**)**

ALL	Returns repeated characters.
string	A string expression containing the character sequence to be repeated.
length	The length of the return string. If omitted the *length* of the return string is 255 characters.

The **ALL** function returns a string containing repetitions of the character sequence *string*.

Return Data Type: STRING

Example:

```
Starline = ALL(´*´,25)       !Get 25 asterisks
Dotline = ALL(´.´)          !Get 255 dots
```

CENTER (return centered string)

CENTER(*string* [,*length*]**)**

CENTER Returns centered string.

string A string constant, variable or expression.

length The length of the return string. If omitted, the length of the *string* parameter is used.

The **CENTER** function first removes leading and trailing spaces from a *string*, then pads it with leading and trailing spaces to center it within the *length*, and returns a centered string.

Return Data Type: STRING

Example:

```
!CENTER(´ABC´,5)    returns ´ ABC ´
!CENTER(´ABC ´)   returns ´ ABC ´
!CENTER(´ ABC´)     returns ´ ABC ´

 Message = CENTER(Message)       !Center the message
Rpt:Title = CENTER(Name,60)     !Center the name
```

CHR (return character from ASCII)

CHR(*code***)**

CHR Returns the display character.

code A numeric expression containing a numeric ASCII character code.

The **CHR** function returns the character represented by the ASCII character *code* parameter.

Return Data Type: STRING

Example:

```
Stringvar = CHR(122)        !Get lower case z
Stringvar = CHR(65)         !Get upper case A
```

CLIP (return string without trailing spaces)

CLIP(*string***)**

CLIP Removes trailing spaces.

string A string expression.

The **CLIP** function removes trailing spaces from a *string*. The return string is a substring with no trailing spaces. CLIP is frequently used with the concatenation operator in string expressions.

Return Data Type: STRING

Example:

```
Name = CLIP(Last) & ´, ´ & CLIP(First) & Init & ´.´ !Full name in military order
```

DEFORMAT (remove formatting from numeric string)

DEFORMAT(*string* [,*picture*]**)**

DEFORMAT Removes formatting characters from a numeric string.

string A string expression containing a numeric string.

picture A picture token or the label of a CSTRING, STRING, or PSTRING variable containing a picture token (CSTRING is more efficient than STRING or PSTRING). If omitted, the picture for the *string* parameter is used. If the *string* parameter was not declared with a picture token, the return value will contain only characters that are valid for a numeric constant.

The **DEFORMAT** function removes formatting characters from a numeric string, returning only the numbers contained in the string.

Return Data Type: STRING

Example:

```
DialString = ´ATDT1´ & DEFORMAT(Phone,@P(###)###-####P) & ´<13,10>´
                                    !Get phone number for modem to dial
ClarionDate = DEFORMAT(dBaseDate,@D1)   !Clarion Standard date from mm/dd/yy string
```

FORMAT (format numbers into a picture)

FORMAT(*value*,*picture***)**

FORMAT	Returns a formatted numeric string.
value	A numeric expression for the *value* to be formatted.
picture	A picture token or the label of a STRING, CSTRING, or PSTRING variable containing a picture token (CSTRING is more efficient than STRING or PSTRING).

The **FORMAT** function returns a numeric string formatted according to the *picture* parameter.

Return Data Type: STRING

Example:

```
Rpt:SocSecNbr = FORMAT(Emp:SSN,@P###-##-####P)        !Format the soc-sec-nbr
Phone = FORMAT(DEFORMAT(Phone,@P###-###-####P),@P(###)###-####P)
                                  !Change phone format from dashes to parens
DateString = FORMAT(DateLong,@D1)      !Format a date as a string
```

INLIST (search for entry in list)

INLIST(*searchstring*,*liststring*,*liststring* [,*liststring...*]**)**

INLIST Returns item in a list.

searchstring A constant, variable, or expression that contains the value for which to search. If the value is numeric, it is converted to a string before comparisons are made.

liststring The label of a variable or constant value to compare against the *searchstring*. If the value is numeric, it is converted to a string before comparisons are made. There may be any number of *liststring* parameters, but there must be at least two.

The **INLIST** function compares the contents of the *searchstring* against the values contained in each *liststring* parameter. If a matching value is found, the function returns the number of the *liststring* parameter containing the matching value (relative to the first *liststring* parameter). If the *searchstring* is not found in any *liststring* parameter, **INLIST** returns zero.

Return Data Type: LONG

Example:

```
!INLIST(´D´,´A´,´B´,´C´,´D´,´E´) returns 4
!INLIST(´B´,´A´,´B´,´C´,´D´,´E´) returns 2

 EXECUTE INLIST(Emp:Status,´Fulltime´,´Parttime´,´Retired´,´Consultant´)
   Scr:Message = ´All Benefits´         !Full timer
   Scr:Message = ´Holidays Only´          !Part timer
   Scr:Message = ´Medical/Dental Only´      !Retired
   Scr:Message = ´No Benefits´            !Consultant
 END
```

INSTRING (search for substring)

INSTRING(*substring*,*string* [,*step*] [,*start*]**)**

INSTRING	Searches for a substring in a string.
substring	A string constant, variable, or expression that contains the string for which to search.
string	A string constant, or the label of the STRING, CSTRING, or PSTRING variable to be searched.
step	A numeric constant, variable, or expression which specifies the step length of the search. A *step* of 1 searches for the *substring* beginning at every character in the *string*, a *step* of 2 starts at every other character, and so on. If *step* is omitted, the step length defaults to the length of the *substring*.
start	A numeric constant, variable, or expression which specifies where to begin the search of the *string*. If omitted, the search starts at the first character position.

The **INSTRING** function *steps* through a *string*, searching for the occurrence of a *substring*. If the *substring* is found, the function returns the *step* number on which the *substring* was found. If the *substring* is not found in the *string*, **INSTRING** returns zero.

Return Data Type: LONG

Example:

```
!INSTRING(´DEF´,´ABCDEFGHIJ´,1,1) returns 4
!INSTRING(´DEF´,´ABCDEFGHIJ´,2,1) returns 0
!INSTRING(´DEF´,´ABCDEFGHIJ´,2,2) returns 2
!INSTRING(´DEF´,´ABCDEFGHIJ´,3,1) returns 2
Extension = SUB(FileSpec,INSTRING(´.´,FileSpec) + 1,3)
                                !Extract extension from file spec
IF INSTRING(Search,Cus:Notes,1,1)  !If search variable found
  Scr:Message = ´Found´           ! display message
END
```

LEFT (return left justified string)

LEFT(*string* [,*length*]**)**

LEFT	Left justifies a string.
string	A string constant, variable, or expression.
length	A numeric constant, variable, or expression for the length of the return string. If omitted, *length* defaults to the length of the *string*.

The **LEFT** function returns a left justified string. Leading spaces are removed from the *string*.

Return Data Type: STRING

Example:

```
CompanyName = LEFT(CompanyName)    !Left justify the company name
```

LEN (return length of string)

LEN(*string***)**

LEN Returns length of a string.

string A string constant, variable, or expression.

The **LEN** function returns the length of a *string*. If the *string* parameter is the label of a variable, the function will return the declared length of the variable. Numeric variables are automatically converted to STRING intermediate values.

Return Data Type: LONG

Example:

```
IF LEN(CLIP(Title) & ´ ´ & CLIP(First) & ´ ´ & CLIP(Last)) > 30
                                                !If full name won´t fit
  Rpt:Name = CLIP(Title) & ´ ´ & SUB(First,1,1) & ´. ´ & Last
                                                ! use first initial
ELSE
  Rpt:Name = CLIP(Title) & ´ ´ & CLIP(First) & ´ ´ & CLIP(Last)
                                                ! else use full name
END
Rpt:Title = CENTER(Cus:Name,LEN(Rpt:Title))         !Center the name in the title
```

LOWER (return lower case)

LOWER(*string***)**

LOWER Converts a string to all lower case.

string A string constant, variable, or expression for the *string* to be converted.

The **LOWER** function returns a string with all letters converted to lower case.

Return Data Type: STRING

Example:

```
Name = SUB(Name,1,1) & LOWER(SUB(Name,2,19))
                                  !Make the rest of the name lower case
```

NUMERIC (check numeric string)

NUMERIC(*string***)**

NUMERIC Validates all numeric string.

string A string constant, variable, or expression.

The **NUMERIC** function returns the value 1 (true) if the *string* contains a valid numeric value. It returns zero (false) if the *string* contains non-numeric characters. Valid numeric characters are the digits 0 through 9, a leading minus sign, and a decimal point.

Return Data Type: LONG

Example:

```
IF NOT NUMERIC(PartNumber)      !If part number is not numeric
  DO ChkValidPart               ! check for valid part number
END                           !End if
```

RIGHT (return right justified string)

RIGHT(*string* [,*length*]**)**

RIGHT	Right justifies a string.
string	A string constant, variable, or expression.
length	A numeric constant, variable, or expression for the length of the return string. If omitted, the *length* is set to the length of the *string*.

The **RIGHT** function returns a right justified string. Trailing spaces are removed, then the string is right justified and returned with leading spaces.

Return Data Type: STRING

Example:

```
Message = RIGHT(Message)      !Right justify the message
```

SUB (return substring of string)

SUB(*string*,*position*,*length***)**

SUB	Returns a portion of a string.
string	A string constant, variable or expression.
position	A integer constant, variable, or expression. If positive, it points to a character position relative to the beginning of the *string*. If negative, it points to the character position relative to the end of the *string* (i.e., a *position* value of -3 points to a position 3 characters from the end of the *string*).
length	A numeric constant, variable, or expression of number of characters to return.

The **SUB** function parses out a sub-string from a *string* by returning *length* characters from the *string*, starting at *position*.

The SUB function is similar to the "string slicing" operation on STRING, CSTRING, and PSTRING variables, but is less flexible and efficient. "String slicing" is more flexible because it may be used on both the destination and source sides of an assignment statement, while the SUB function can only be used as the source. It is more efficient because it takes less memory than individual character assignments or the SUB function.

To take a "slice" of a string, the beginning and ending character numbers are separated by a colon (:) and placed in the implicit array dimension position within the square brackets ([]) of the string. The position numbers may be integer constants, variables, or expressions. If variables are used, there must be at least one blank space between the variable name and the colon separating the beginning and ending number (to prevent PREfix confusion).

Return Data Type: STRING

Example:

```
 !SUB(´ABCDEFGHI´,1,1)  returns ´A´
 !SUB(´ABCDEFGHI´,-1,1) returns ´I´
 !SUB(´ABCDEFGHI´,4,3)  returns ´DEF´
  Extension = SUB(FileName,INSTRING(´.´,FileName,1,1)+1,3)
                         !Get the file extension using SUB function
Extension = FileName[(INSTRING(´.´,FileName,1,1)+1):(INSTRING(´.´,FileName,1,1)+3)]
                         !The same operation using string slicing
```

See Also:

INSTRING

STRING

CSTRING

PSTRING

String Slicing

UPPER (return upper case)

UPPER(*string***)**

UPPER Returns all upper case string.

string A string constant, variable, or expression for the *string* to be converted.

The **UPPER** function returns a string with all letters converted to upper case.

Return Data Type: STRING

Example:

```
Name = UPPER(Name)    !Make the name upper case
```

VAL (return ASCII value)

VAL(*character***)**

VAL Returns ASCII code.

character A one-byte string containing a character.

The **VAL** function returns the ASCII code of a *character*.

Return Data Type: LONG

Example:

```
  !VAL(´A´)     returns 65
  !VAT(´z´)     returns 122

CharVal = VAL(StrChar)  !Get the ASCII value of the string character
```

Bit Manipulation Functions

BAND (return bitwise AND)

BAND(*value*,*mask***)**

BAND	Performs bitwise AND operation.
value	A numeric constant, variable, or expression for the bit *value* to be compared to the bit *mask*. The *value* is converted to a LONG data type prior to the operation, if necessary.
mask	A numeric constant, variable, or expression for the bit *mask*. The *mask* is converted to a LONG data type prior to the operation, if necessary.

The **BAND** function compares the *value* to the *mask*, performing a Boolean AND operation on each bit. The return value is a LONG integer with a one (1) in the bit positions where the *value* and the *mask* both contain one (1), and zeroes in all other bit positions.

BAND is usually used to determine whether an individual bit, or multiple bits, are on (1) or off (0) within a variable.

Return Data Type: LONG

Example:

```
!BAND(0110b,0010b) returns 0010b  !0110b = 6, 0010b = 2

RateType  BYTE           !Type of rate
Female  EQUATE(0001b)    !Female mask
Male      EQUATE(0010b)    !Male mask
Over25  EQUATE(0100b)    !Over age 25 mask
  CODE
  IF BAND(RateType,Female)   |    !If female
     AND BAND(RateType,Over25)     ! and over 25
    DO BaseRate                  ! use base premium
  ELSIF BAND(RateType,Male)        !If male
    DO AdjBase                   ! adjust base premium
  END
```

BOR (return bitwise OR)

BOR(*value*,*mask***)**

BOR	Performs bitwise OR operation.
value	A numeric constant, variable, or expression for the bit *value* to be compared to the bit *mask*. The *value* is converted to a LONG data type prior to the operation, if necessary.
mask	A numeric constant, variable, or expression for the bit *mask*. The *mask* is converted to a LONG data type prior to the operation, if necessary.

The **BOR** function compares the *value* to the *mask*, performing a Boolean OR operation on each bit. The return value is a LONG integer with a one (1) in the bit positions where the *value*, or the *mask*, or both, contain a one (1), and zeroes in all other bit positions.

BOR is usually used to unconditionally turn on (set to one), an individual bit, or multiple bits, within a variable.

Return Data Type: LONG

Example:

```
!BOR(0110b,0010b) returns 0110b  !0110b = 6, 0010b = 2

RateType  BYTE           !Type of rate
Female   EQUATE(0001b)    !Female mask
Male     EQUATE(0010b)    !Male mask
Over25   EQUATE(0100b)    !Over age 25 mask
  CODE
  RateType = BOR(RateType,Over25)   !Turn on over 25 bit
  RateType = BOR(RateType,Male)     !Set rate to male
```

BXOR (return bitwise exclusive OR)

BXOR(*value*,*mask***)**

BXOR	Performs bitwise exclusive OR operation.
value	A numeric constant, variable, or expression for the bit *value* to be compared to the bit *mask*. The *value* is converted to a LONG data type prior to the operation, if necessary.
mask	A numeric constant, variable, or expression for the bit *mask*. The *mask* is converted to a LONG data type prior to the operation, if necessary.

The **BXOR** function compares the *value* to the *mask*, performing a Boolean XOR operation on each bit. The return value is a LONG integer with a one (1) in the bit positions where either the *value* or the *mask* contain a one (1), but not both. Zeroes are returned in all bit positions where the bits in the *value* and *mask* are alike.

BXOR is usually used to toggle on (1) or off (0) an individual bit, or multiple bits, within a variable.

Return Data Type: LONG

Example:

```
!BXOR(0110b,0010b) returns 0100b  !0110b = 6, 0100b = 4, 0010b = 2

RateType  BYTE           !Type of rate
Female   EQUATE(0001b)    !Female mask
Male     EQUATE(0010b)    !Male mask
Over25   EQUATE(0100b)    !Over age 25 mask
Over65   EQUATE(1100b)    !Over age 65 mask
  CODE
  RateType = BXOR(RateType,Over65)   !Toggle over 65 bits
```

BSHIFT (return shifted bits)

BSHIFT(*value*,*count***)**

BSHIFT	Perfroms bit shift operation.
value	A numeric constant, variable, or expression. The *value* is converted to a LONG data type prior to the operation, if necessary.
count	A numeric constant, variable, or expression for the number of bit positions to be shifted. If *count* is positive, *value* is shifted left. If *count* is negative, *value* is shifted right.

The **BSHIFT** function shifts a bit *value* by a bit *count*. The bit value may be shifted left (toward the high order), or right (toward the low order). Zero bits are supplied to fill vacated bit positions when shifting.

Return Data Type: LONG

Example:

```
  !BSHIFT(0110b,1)    returns 1100b
  !BSHIFT(0110b,-1)   returns 0011b

Varswitch = BSHIFT(20,3)           !Multiply by eight
Varswitch = BSHIFT(Varswitch,-2)     !Divide by four
```

Date / Time Procedures and Functions

Standard Date

A Clarion standard date is the number of days that have elapsed since December 28, 1800. The range of accessible dates is from January 1, 1801 (standard date 4) to December 31, 2099 (standard date 109,211). Date functions will not return correct values outside the limits of this range. The standard date calendar also adjusts for each leap year within the range of accessible dates. Dividing a standard date by modulo 7 gives you the day of the week: zero = Sunday, one = Monday, etc.

The LONG data type with a date format (@D) display picture is normally used for a standard date. The DATE data type is a data format used in the Btrieve Record Manager. A DATE field is internally converted to LONG containing the Clarion standard date before any mathematical or date function operation is performed. Therefore, DATE should be used for external Btrieve file compatibility, and LONG should normally be used for other dates.

Standard Time

A Clarion standard time is the number of hundredths of a second that have elapsed since midnight, plus one (1). The valid range is from 1 (defined as midnight) to 8,640,000 (defined as 11:59:59:99). A standard time of one is exactly equal to midnight (which allows a zero value to be used to detect no time entered). Although time is expressed to the nearest hundredth of a second, the system clock is only updated 18.2 times a second (approximately every 5.5 hundredths of a second).

The LONG data type with a time format (@T) display picture is normally used for a standard time. The TIME data type is a data format used in the Btrieve Record Manager. A TIME field is internally converted to LONG containing the Clarion standard time before any mathematical or time function operation is performed. Therefore, TIME should be used for external Btrieve file compatibility, and LONG should normally be used for other times.

TODAY (return system date)

TODAY( )

The **TODAY** function returns the DOS system date as a standard date. The range of possible dates is from January 1, 1801 (standard date 4) to December 31, 2099 (standard date 109,211).

Return Data Type: LONG

Example:

```
OrderDate = TODAY()     !Set the order date to system date
```

SETTODAY (set system date)

SETTODAY(*date*)

SETTODAY Sets the DOS system date.

date A numeric constant, variable, or expression for a standard date.

The **SETTODAY** statement sets the DOS system date.

Example:

```
SETTODAY(TODAY() + 1)    !Set the date ahead one day
```

CLOCK (return system time)

CLOCK( )

The **CLOCK** function returns the time of day from the DOS system time in standard time (expressed as hundredths of a second since midnight). Although the time is expressed to the nearest hundredth of a second, the system clock is only updated 18.2 times a second (approximately every 5.5 hundredths of a second).

Return Data Type: LONG

Example:

```
Time = CLOCK()      !Save the system time
```

SETCLOCK (set system time)

SETCLOCK(*time***)**

SETCLOCK Sets the DOS system clock.

time A numeric constant, variable, or expression for a standard time (expressed as hundredths of a second since midnight plus one).

The **SETCLOCK** statement sets the DOS system time of day.

Example:

```
SETCLOCK(1)        !Set clock to midnight
```

DATE (return standard date)

DATE(*month*,*day*,*year***)**

DATE	Return standard date.
month	A numeric constant, variable, or expression for the *month*.
day	A numeric constant, variable, or expression for the *day* of the month.
year	A numeric constant, variable or expression for the *year*. The valid range for a *year* value is 00 through 99 (which assumes the range 1900 - 1999), or 1801 through 2099.

The **DATE** function returns a standard date for a given *month*, *day*, and *year*. The *month* and *day* parameters allow out-of-range values. A *month* value of 13 is interpreted as January of the next year. A *day* value of 32 in January is interpreted as the first of February. Consequently, DATE(12,32,87), DATE(13,1,87), and DATE(1,1,88) all produce the same result.

Return Data Type: LONG

Example:

```
HireDate = DATE(Hir:Month,Hir:Day,Hir:Year) !Compute hire date
```

See Also:

Standard Date

DAY (return day of month)

DAY(*date***)**

DAY	Returns day of month.
date	A numeric constant, variable, expression, or the label of a STRING, CSTRING, or PSTRING variable declared with a date picture token. The *date* must be a standard date. A variable declared with a date picture token is automatically converted to a standard date intermediate value.

The **DAY** function computes the day of the month (1 to 31) for a given standard date.

Return Data Type: LONG

Example:

```
OutDay = DAY(TODAY())     !Get the day from today´s date
DueDay = DAY(TODAY()+2)   !Calculate the return day
```

See Also:

Standard Date

MONTH (return month of date)

MONTH(*date***)**

MONTH	Returns month in year.
date	A numeric constant, variable, expression, or the label of a STRING, CSTRING, or PSTRING variable declared with a date picture token. The *date* must be a standard date. A variable declared with a date picture token is automatically converted to a standard date intermediate value.

The **MONTH** function returns the month of the year (1 to 12) for a given standard date.

Return Data Type: LONG

Example:

```
PayMonth = MONTH(DueDate)     !Get the month from the date
```

See Also:

Standard Date

YEAR (return year of date)

YEAR(*date***)**

YEAR	Returns the year.
date	A numeric constant, variable, expression, or the label of a string variable declared with a date picture, containing a standard date. A variable declared with a date picture is automatically converted to a standard date intermediate value.

The **YEAR** function returns a four digit number for the year of a standard *date* (1801 to 2099).

Return Data Type: LONG

Example:

```
IF YEAR(LastOrd) < YEAR(TODAY())   !If last order date not from this year
  DO StartNewYear      ! start new year to date totals
END
```

See Also:

Standard Date

AGE (return age from base date)

AGE(*birthdate* [,*base date*]**)**

AGE	Returns elapsed time.
birthdate	A numeric expression for a standard date.
base date	A numeric expression for a standard date. If this parameter is omitted, the system date from DOS is used for the computation.

The **AGE** function returns a string containing the time elapsed between two dates. The age return string is in the following format:

```
1 to 60 days  - ´nn DAYS´
61 days to 24 months - ´nn MOS´
2 years to 999 years - ´nnn YRS´
```

Return Data Type: STRING

Example:

```
Message = Emp:Name & ´is ´ & AGE(Emp:DOB,TODAY()) & ´ old today.´
```

DOS Procedures and Functions

COMMAND (return command line)

COMMAND(*flag* **)**

COMMAND Returns command line parameters.

flag A string constant or variable containing the parameter for which to search, or the number of the command line parameter to return.

The **COMMAND** function returns the value of the *flag* parameter from the command line. If the *flag* is not found, COMMAND returns an empty string. If the *flag* is multiply defined, only the first occurrence encountered is returned.

COMMAND searches the command line for *flag*=*value* and returns *value*. There must be no blanks between *flag*, the equal sign, and *value*. The returned *value* terminates at the first comma or blank space. If a blank or comma is desired in a command line parameter, everything to the right of the equal sign must be enclosed in double quotes (*flag*="*value*").

COMMAND will also search the command line for a *flag* containing a leading slash (/). If found, COMMAND returns the value of *flag* without the slash. If the *flag* only contains a number, COMMAND returns the parameter at that numbered position on the command line. A *flag* of ´0´ returns the minimum path DOS used to find the command. This minimum path always includes the command (without command line parameters) but may not include the path (if DOS found it in the current directory). A *flag* containing ´1´ returns the first command line parameter. If *flag* is an empty string (´´), all command parameters are returned as entered on the command line, appended to a leading space.

Return Data Type: STRING

Example:

```
IF COMMAND(´/N´)             !Was /N on the command line?
  DO SomeProcess
END
CommandString = COMMAND(´´)     !Get all command parameters
CommandItself = COMMAND(´0´)    !Get the command itself
SecondParm = COMMAND(´2´)       !Get second parameter from command line
```

See Also:

SETCOMMAND

PATH (return current DOS directory)

PATH( )

PATH returns a string containing the current drive and directory.

Return Data Type: STRING

Example:

```
IF PATH() = ´C:\´                          !If in the root
  MESSAGE(´You are in the Root Directory´)     ! display message
END
```

See Also:

SETPATH

RUNCODE (return DOS exit code)

RUNCODE( )

The **RUNCODE** function returns the exit code passed to DOS from the command executed by the RUN statement. This is the exit code passed by the HALT statement in Clarion programs and is the same as the DOS ERRORLEVEL. RUNCODE returns a LONG integer which may be any value that is returned to DOS as an exit code by the child program.

The child program may only supply DOS with a BYTE value as an exit code, therefore negative values are not possible as DOS exit codes. This fact allows RUNCODE to reserve these values to handle situations in which an exit code is not available:

```
 0 normal termination
-1 program terminated with Ctrl-C
-2 program terminated with Critical error
-3 TSR exit
-4 program did not run (check ERROR())
```

Return Data Type: LONG

Example:

```
RUN(´Nextprog.exe´)                  !Run next program
IF RUNCODE() = -4
  IF ERROR() = ´Not Enough Memory´     !If program didn´t run for lack of memory
    MESSAGE(´Insufficient memory´)     ! display a message
    RETURN                          ! and terminate the procedure
  ELSE
    STOP(ERROR())                      ! terminate program
. .
```

See Also:

RUN

HALT

SETCOMMAND (set command line parameters)

SETCOMMAND(*commandline***)**

SETCOMMAND Internally sets command line parameters.

commandline A string constant, variable, or expression containing the new command line parameters.

SETCOMMAND allows the program to internally specify command line parameters that may be read by the COMMAND function. SETCOMMAND overwrites any previous command line flag of the same value. To turn off a leading slash flag, append an equal sign (=) to it in the *commandline*.

SETCOMMAND may not be used to set system level switches which must be specified when the program is loaded. Switches like virtual memory (CLAVM=), or the program´s configuration file (CLAINI=) must be set at load time and may not be set with SETCOMMAND. The temporary files directory switch (CLATMP=) may be set with SETCOMMAND.

Example:

```
SETCOMMAND(´ /N´)       !Add /N parameter
SETCOMMAND(´ /N=´)    !Turn off /N parameter
```

See Also:

COMMAND

SETPATH (change current drive and directory)

SETPATH(*path***)**

SETPATH Changes the current DOS drive and directory.

path A string constant or the label of a STRING, CSTRING, or PSTRING variable containing a new drive and/or directory specification.

SETPATH changes the current DOS drive and directory. If the *drive and path* entry is invalid, the "Path Not Found" error is posted, and the current directory is not changed.

If the drive letter and colon are omitted from the *path*, the current drive is assumed. If only a drive letter and colon are in the *path*, SETPATH changes to the DOS current directory of that drive.

Errors Posted: 03 Path Not Found

Example:

```
SETPATH(´C:\LEDGER´)       !Change to the ledger directory
SETPATH(UserPath)          !Change to the user´s directory
```

Error Reporting Functions

ERROR (return error message)

ERROR( )

The **ERROR** function returns a string containing a description of any error that was posted. If no error was posted, ERROR returns an empty string.

Return Data Type: STRING

Example:

```
PUT(NameQueue)                       !Write the record
IF ERROR() = ´Queue Entry Not Found´     !If not found
  ADD(NameQueue)                     ! add new entry
    IF ERRORCODE() THEN STOP(ERROR()). !Check for unexpected error
END
```

ERRORCODE (return error code number)

ERRORCODE( )

The **ERRORCODE** function returns the code number for any error that was posted. If no error was posted, ERRORCODE returns zero.

Return Data Type: LONG

Example:

```
ADD(Location)                !Add new entry
IF ERRORCODE() = 8           !If not enough memory
  MESSAGE(´Out of Memory´)     ! display message
END
```

ERRORFILE (return error filename)

ERRORFILE( )

The **ERRORFILE** function returns the name of the file for which an error was posted. If the file is open, the full DOS file specification is returned. If the file is not open, the contents of the FILE statement´s NAME attribute is returned. If the file is not open and the file has no NAME attribute, the label of the FILE statement is returned. If no error was posted, or the posted error did not involve a file, ERRORFILE returns an empty string.

Return Data Type: STRING

Example:

```
ADD(Location)     !Add new entry
IF ERRORCODE()
  MESSAGE(´Error with ´ & ERRORFILE()) !Display error filename
END
```

FILEERROR (return file driver error message)

FILEERROR(*file* **)**

The **FILEERROR** function returns a string containing the "native" error message from the file system (file driver) being used to access a data file. If no error was posted, FILEERROR returns an empty string.

Return Data Type: STRING

Example:

```
PUT(NameFile)     !Write the record
IF FILEERRORCODE()
  MESSAGE(FILERROR())
  RETURN
END
```

See Also:

FILEERRORCODE

FILEERRORCODE (return file driver error code number)

FILEERRORCODE( )

The **FILEERRORCODE** function returns a string containing the code number for the "native" error message from the file system (file driver) being used to access a data file. If no error was posted, FILEERRORCODE returns an empty string.

Return Data Type: STRING

Example:

```
PUT(NameFile)     !Write the record
IF FILEERRORCODE()
  MESSAGE(FILERROR())
  RETURN
END
```

See Also:

FILEERROR

Other Procedures and Functions

ADDRESS (return a memory address)

ADDRESS(| *segment,offset* | **)**
| *variable* |
| *procedure* |

ADDRESS Returns memory address of a variable.

segment The label of a data element, or an integer variable or constant containing the segment portion of a segment:offset real-mode absolute memory address.

offset An integer variable or constant containing the offset portion of a segment:offset real-mode absolute memory address.

variable The label of a data element.

procedure The label of a PROCEDURE or FUNCTION.

The **ADDRESS** function returns a LONG integer containing a memory address in selector:offset format, where the selector is a reference into the protected mode lookup table.

ADDRESS(*segment,offset*)
Returns the protected mode selector:offset for the real mode address specified by the *segment* and *offset* parameters. This allows protected mode direct memory access without incurring a protection violation.

ADDRESS(*variable*)
Returns the protected mode address of the data element specified by the *variable* parameter.

ADDRESS(*procedure*)
Returns the protected mode address of the PROCEDURE or FUNCTION specified by the *procedure* parameter.

Tha ADDRESS function allows you to pass the address of a *variable* or *procedure* to external libraries written in other languages.

Return Data Type: LONG

Example:

```
 MAP
   ClarionProc                !A Clarion language procedure
   MODULE(´External.Obj´)        !An external library
     ExternVarProc(LONG)       !C function receiving variable address
     ExternProc(LONG)          !C function receiving procedure address
  . .

Var1 CSTRING(10)  !Define a null-terminated string

  CODE
  ExternVarProc(ADDRESS(Var1))       !Pass address of Var1 to external procedure
  ExternProc(ADDRESS(ClarionProc))   !Pass address of ClarionProc

ClarionProc PROCEDURE              !A Clarion language procedure
  CODE
  RETURN
```

BEEP (sound tone on speaker)

BEEP([*sound*] **)**

BEEP Generates a sound through the system speaker.

sound A numeric constant, variable, expression, or EQUATE for the Windows sound to issue.

The **BEEP** statement generates a sound through the system speaker. These are standard Windows sounds available through the [sounds] section of the WIN.INI file. Standard EQUATE values are listed in the EQUATES.CLW file.

Example:

```
IF ERRORCODE()       !If unexpected error
  BEEP(-1)         ! sound a standard beep
  STOP(ERROR())      ! stop for the error
END
```

CALL (call procedure from a DLL)

CALL(*file, procedure* **)**

CALL Calls a procedure that has not been prototyped in the application´s MAP structure from a Windows standard .DLL.

file A string constant, variable, or expression containing the name (including extension) of the .DLL to open. This may include a full path.

procedure A string constant, variable, or expression containing the name of the *procedure* to call (which may not receive parameters or return a value). This can also be the ordinal number indicating the *procedure´s* position within the .DLL.

The **CALL** function calls a *procedure* from a Windows standard .DLL. The *procedure* does not need to be prototyped in the application´s MAP structure. If it is not already loaded by Windows, the .DLL *file* is loaded into memory.

CALL returns zero (0) for a successful *procedure* call, or one of the following error values:

```
2  File not found
3  Path not found
5  Attempted to load a task, not a .DLL
6  Library requires separate data segments for each task
10 Wrong Windows version
11 Invalid .EXE file (DOS file or error in program header)
12 OS/2 application
13 DOS 4.0 application
14 Unknown .EXE type
15 Attempt to load an .EXE created for an earlier version of Windows.
   This error will not occur if Windows is run in Real mode.
16 Attempt to load a second instance of an .EXE file containing
   multiple, writeable data segments.
17 EMS memory error on the second loading of a .DLL
18 Attempt to load a protected-mode-only application while Windows is
   running in Real mode
```

Return Data Type: LONG

Example:

```
X# = CALL(´CUSTOM.DLL´,´1´)     !Call first procedure in CUSTOM.DLL
IF X# THEN STOP(X#).           !Check for successful execution
```

MAXIMUM (return maximum subscript value)

MAXIMUM(*variable*,*subscript***)**

MAXIMUM	Returns maximum subscript value.
variable	The label of a variable declared with a DIM attribute.
subscript	A numeric constant, variable, or expression for the subscript number. The *subscript* identifies which array dimension is passed to the function.

The **MAXIMUM** function returns the maximum subscript value for an explicitly dimensioned variable. MAXIMUM does not operate on the implicit array dimension of STRING, CSTRING, or PSTRING variables. This is usually used to determine the size of an array passed as a parameter to a procedure or function.

Return Data Type: LONG

Example:

```
Array BYTE,DIM(10,12)        !Define a two-dimensional array

   !For the above Array:    MAXIMUM(Array,1) returns 10
   !                       MAXIMUM(Array,2) returns 12

  CODE
  LOOP X# = 1 TO MAXIMUM(Array,1)        !Loop until end of 1st dimension
    LOOP Y# = 1 TO MAXIMUM(Array,2)      ! Loop until end of 2nd dimension
      Array[X#,Y#] = 27               !   Initialize each element to default
  . .
```

See Also:

DIM

Arrays as Parameters of PROCEDUREs and FUNCTIONs

NAME (return DOS file or device name)

NAME(*label***)**

NAME Returns name of a file.

label The label of a FILE declaration.

The **NAME** function returns a string containing the DOS device name for the structure identified by the *label*. For FILE structures, if the file is OPEN, the complete DOS file specification (drive, path, name, and extension) is returned. If the FILE is closed, the contents of the NAME attribute on the FILE are returned.

Return Data Type: STRING

Example:

```
OpenFile = NAME(Customer)    !Save the name of the open file
```

OMITTED (check omitted parameters)

OMITTED(*position***)**

OMITTED Tests for unpassed parameters.

position An integer constant or variable which specifies the parameter to test.

The **OMITTED** function tests whether a parameter of a PROCEDURE or FUNCTION was actually passed. The return value is 1 (true) if the parameter in the specified *position* was omitted. The return value is zero (false) if the parameter was passed. Any *position* past the last parameter passed is considered omitted.

A parameter may only be omitted if its data type is enclosed in angle brackets (< >) in the PROCEDURE or FUNCTION prototype in the MAP structure.

Return Data Type: LONG

Example:

```
  PROGRAM
  MAP
   SomeProc(STRING,<LONG>,STRING)        !Procedure prototype
   SomeFunction(STRING,<LONG>),STRING     !Function prototype
  END
  CODE
  SomeProc(Field1,,Field3)
    !For this statement:
    ! OMITTED(1) returns 0
    ! OMITTED(2) returns 1
    ! OMITTED(3) returns 0
    ! OMITTED(4) returns 1

SomeProc PROCEDURE(Field1,Date,Field3)
  CODE
  IF OMITTED(2)        !If date parameter was omitted
    Date = TODAY()     ! substitute the system date
  END
```

See Also:

FUNCTION and PROCEDURE Prototypes

DDE Library Reference

Dynamic Data Exchange

Dynamic Data Exchange (DDE) is a very powerful Windows tool that allows a user to access data from another separately executing Windows application. This allows the user to work with the data in its native format (in the originating application), while ensuring that the application in which the data is used always has the most current values.

DDE is based upon establishing "conversations" (links) between two concurrently executing Windows applications. One of the applications acts as the DDE server to provide the data, and the other is the DDE client that receives the data. A single application may be both a DDE client and server, getting data from other applications and providing data to other applications. Multiple DDE "conversations" can occur concurrently between any given DDE server and client.

To be a DDE server, a Clarion application must:

Open at least one window, since all DDE servers must be associated with a window.

Register with Windows as a DDE server, using the DDESERVER function.

Provide the requested data to the client, using the DDEWRITE statement.

When DDE is no longer required, terminate the link by using the DDECLOSE statement. You can also allow it to terminate automatically when the user closes the server application or the window that started the link.

To be a DDE client, a Clarion application must:

Open a link to a DDE server as its client, using the DDECLIENT function.

Ask the server for data, using the DDEREAD stateemnt, or ask the server for a service using the DDEEXECUTE statement.

When DDE is no longer required, terminate the link by using the DDECLOSE statement. You can also allow it to terminate automatically when the user closes the client window or application.

The DDE process posts DDE-specific field-independent events to the ACCEPT loop of the window that opened the link between applications, either as a server or client.

DDE Events

The DDE process is governed by several field-independent events specific to DDE. These events are posted to the ACCEPT loop of the window that opened the link between applications, either as a server or client.

The following events are posted only to a Clarion server application:

EVENT:DDErequest
A client has requested a data item.

EVENT:DDEadvise
A client has requested continuous updates of a data item.

EVENT:DDEexecute
A client has executed a DDEEXECUTE statement.

The following events are posted only to a Clarion client application:

EVENT:DDEdata	A server has supplied an updated data item.
EVENT:DDEclose	A server has terminated the DDE link.

When one of these DDE events occur there are several functions that tell you what posted the event:

DDECHANNEL() returns the handle of the DDE server or client.

DDEITEM() returns the item or command string passed to the server by the DDEREAD or DDEEXECUTE statements.

DDEAPP() returns the name of the application.

DDETOPIC() returns the name of the topic.

When a Clarion program creates a DDE server, external clients can link to this server and request data. Each data request is accompanied by a string (in some specific format which the server program knows) indicating the required data item. If the Clarion server already knows the value for a given item, it supplies it to the client automatically without generating any events. If it doesn´t, an EVENT:DDErequest or EVENT:DDEadvise event is posted to the server window´s ACCEPT loop.

When a Clarion program creates a DDE client, it can link to external servers which can provide data. When the server first provides the value for a given item, it supplies it to the client automatically without generating any events. If the client has established a "hot" link with the server, an EVENT:DDEdata event is posted to the client window´s ACCEPT loop whenever the server posts a new value for the data item.

DDE Functions

DDESERVER (return DDE server channel)

DDESERVER([*application*] [, *topic*] **)**

DDESERVER Returns a new DDE server channel number.

application A string constant or variable containing the name of the application. Usually, this is the name of the application. If omitted, the filename of the application (without extension) is used.

topic A string constant or variable containing the name of the application-specific topic. If omitted, the *application* will respond to any data request.

The **DDESERVER** function returns a new DDE server channel number for the *application* and *topic*. The channel number specifies a *topic* for which the *application* will provide data. This allows a single Clarion *application* to register as a DDE server for multiple *topics*.

Return Data Type: LONG

Example:

```
DDERetVal     STRING(20)
WinOne     WINDOW,AT(0,0,160,400)
             ENTRY(@s20),USE(DDERetVal)
           END
MyServer    LONG
  CODE
  OPEN(WinOne)
  MyServer = DDESERVER(´MyApp´,´DataEntered´)  !Open as server
  ACCEPT
    CASE EVENT()
    OF EVENT:DDErequest                   !As server for data requested once
      DDEWRITE(MyServer,DDE:manual,´DataEntered´,DDERetVal)  !Provide data once
    OF EVENT:DDEadvise                  !As server for constant update request
      DDEWRITE(MyServer,15,´DataEntered´,DDERetVal)
                                       !Check for change every 15 seconds
                                       ! and provide data whenever changed
    END
  END
```

See Also: DDECLIENT, DDEWRITE

DDECLIENT (return DDE client channel)

DDECLIENT([*application*] [, *topic*] **)**

DDECLIENT Returns a new DDE client channel number.

application A string constant or variable containing the name of the server application to link to. Usually, this is the name of the application. If omitted, the first DDE server application available is used.

topic A string constant or variable containing the name of the application-specific topic. If omitted, the first topic available in the *application* is used.

The **DDECLIENT** function returns a new DDE client channel number for the *application* and *topic*. If the *application* is not currently executing, DDECLIENT returns zero (0).

Typically, when opening a DDE channel as the client, the *application* is the name of the server application. The *topic* is a string that the *application* has either registered with Windows as a valid *topic* for the *application*, or represents some value that tells the *application* what data to provide. You can use the DDEQUERY function to determine the *applications* and *topics* currently registered with Windows.

Return Data Type: LONG

Example:

```
DDEReadVal  REAL
WinOne     WINDOW,AT(0,0,160,400)
             ENTRY(@s20),USE(DDEReadVal)
           END
ExcelServer LONG
  CODE
  OPEN(WinOne)
  ExcelServer = DDECLIENT(´Excel´,´MySheet.XLS´)
                                      !Open as client to Excel spreadsheet
  IF NOT ExcelServer                    !If the server is not running
    MESSAGE(´Please start Excel´)           !alert the user to start it
    RETURN                              ! and try again
  END
  DDEREAD(ExcelServer,DDE:auto,´R5C5´,DDEReadVal)
  ACCEPT
    CASE EVENT()
    OF EVENT:DDEdata                      !As changed data comes from Excel
      PassedData(DDEReadVal)                ! process the new data
    END
  END
```

See Also: DDEQUERY, DDEWRITE, DDESERVER

DDEQUERY (return registered DDE servers)

DDEQUERY([*application*] [, *topic*] **)**

DDEQUERY Returns currently executing DDE servers.

application A string constant or variable containing the name of the application to query. For most applications, this is the name of the application. If omitted, all registered *applications* registered with the specified *topic* are returned.

topic A string constant or variable containing the name of the application-specific topic to query. If omitted, all *topics* for the *application* are returned.

The **DDEQUERY** function returns a string containing the names of the currently available DDE server *applications* and their *topics*.

If the *topic* parameter is omitted, all *topics* for the specified *application* are returned. If the *application* parameter is omitted, all registered *applications* registered with the specified *topic* are returned. If both parameters are omitted, DDEQUERY returns all currently available DDE servers.

The format of the data in the return string is *application:topic* and it can contain multiple *application* and *topic* pairs delimited by commas (for example, ´Excel:MySheet.XLS,ClarionApp:DataFile.DAT´).

Return Data Type: STRING

Example:

```
  !This example code does not handle DDEADVISE
WinOne WINDOW,AT(0,0,160,400)
       END
SomeServer   LONG
ServerString STRING(200)
  CODE
  OPEN(WinOne)
  LOOP
    ServerString = DDEQUERY()                    !Return all registered servers
    IF NOT INSTRING(´SomeApp:MyTopic´,ServerString,1,1)
      MESSAGE(´Open SomeApp, Please´)
    ELSE
      BREAK
    END
  END
  SomeServer = DDECLIENT(´SomeApp´,´MyTopic´)      !Open as client
  ACCEPT
  END
  DDECLOSE(SomeServer)
```

DDECHANNEL (return DDE channel number)

DDECHANNEL( )

The **DDECHANNEL** function returns a LONG integer containing the channel number of the DDE client or server application that has just posted a DDE event. This is the same value returned by the DDESERVER or DDECLIENT function when the DDE channel is established.

Return Data Type: LONG

Example:

```
WinOne WINDOW,AT(0,0,160,400)
       END
TimeServer LONG
DateServer LONG
FormatTime STRING(5)
FormatDate STRING(8)
  CODE
  OPEN(WinOne)
  TimeServer = DDESERVER(´SomeApp´,´Time´)      !Open as server
  DateServer = DDESERVER(´SomeApp´,´Date´)      !Open as server
  ACCEPT
    CASE EVENT()
    OF EVENT:DDErequest
      CASE DDECHANNEL()                        !Check which channel
      OF TimeServer
        FormatTime = FORMAT(CLOCK(),@T1)
        DDEWRITE(TimeServer,DDE:manual,´Time´,FormatTime)
      OF DateServer
        FormatDate = FORMAT(TODAY(),@D1)
        DDEWRITE(DateServer,DDE:manual,´Date´,FormatDate)
      END
    END
  END
```

DDEAPP (return server application)

DDEAPP( )

The **DDEAPP** function returns a string containing the application name in the DDE channel that has just posted a DDE event. This is usually the same as the first parameter to the DDESERVER or DDECLIENT function when the DDE channel is established.

Return Data Type: STRING

Example:

```
ClientApp STRING(20)
WinOne  WINDOW,AT(0,0,160,400)
          STRING(@S20),AT(5,5,90,20),USE(ClientApp)
        END

TimeServer LONG
DateServer LONG
FormatTime STRING(5)
FormatDate STRING(8)

  CODE
  OPEN(WinOne)
  TimeServer = DDESERVER(´SomeApp´,´Time´)      !Open as server
  DateServer = DDESERVER(´SomeApp´,´Date´)      !Open as server
  ACCEPT
    CASE EVENT()
    OF EVENT:DDErequest
      CASE DDECHANNEL()
      OF TimeServer
        ClientApp = DDEAPP()                  !Get client´s name
        DISPLAY                              ! and display on screen
        FormatTime = FORMAT(CLOCK(),@T1)
        DDEWRITE(TimeServer,DDE:manual,´Time´,FormatTime)
      OF DateServer
        ClientApp = DDEAPP() !Get client´s name
        DISPLAY                              ! and display on screen
        FormatDate = FORMAT(TODAY(),@D1)
        DDEWRITE(DateServer,DDE:manual,´Date´,FormatDate)
      END
    END
  END
```

DDEITEM (return server item)

DDEITEM( )

The **DDEITEM** function returns a string containing the name of the item for the current DDE event. This is the item requested by a DDEREAD, the data item supplied by DDEPOKE, or the command to execute from a DDEEXECUTE statement.

Return Data Type: STRING

Example:

```
WinOne   WINDOW,AT(0,0,160,400)
         END

Server      LONG
FormatTime  STRING(5)
FormatDate  STRING(8)

  CODE
  OPEN(WinOne)
  Server = DDESERVER(´SomeApp´,´Clock´)   !Open as server for my topic
  ACCEPT
    CASE EVENT()
    OF EVENT:DDErequest
      CASE DDEITEM()
      OF ´Time´
        FormatTime = FORMAT(CLOCK(),@T1)
        DDEWRITE(Server,DDE:manual,´Time´,FormatTime)
      OF ´Date´
        FormatDate = FORMAT(TODAY(),@D1)
        DDEWRITE(Server,DDE:manual,´Date´,FormatDate)
      END
    OF EVENT:DDEadvise
      CASE DDEITEM()
      OF ´Time´
        FormatTime = FORMAT(CLOCK(),@T1)
        DDEWRITE(Server,1,´Time´,FormatTime)
      OF ´Date´
        FormatDate = FORMAT(TODAY(),@D1)
        DDEWRITE(Server,60,´Date´,FormatDate)
      END
    END
  END
```

See Also: DDEREAD, DDEEXECUTE

DDETOPIC (return server topic)

DDETOPIC( )

The **DDETOPIC** function returns a string containing the topic name for the DDE channel that has just posted a DDE event.

Return Data Type: STRING

Example:

```
WinOne WINDOW,AT(0,0,160,400)
       END

TimeServer LONG
DateServer LONG
FormatTime STRING(5)
FormatDate STRING(8)

  CODE
  OPEN(WinOne)
  TimeServer = DDESERVER(´SomeApp´)       !Open as server
  ACCEPT
    CASE EVENT()
    OF EVENT:DDErequest
      CASE DDETOPIC()                  !Get requested topic
      OF ´Time´
        FormatTime = FORMAT(CLOCK(),@T1)
        DDEWRITE(TimeServer,DDE:manual,´Time´,FormatTime)
      OF ´Date´
        FormatDate = FORMAT(TODAY(),@D1)
        DDEWRITE(DateServer,DDE:manual,´Date´,FormatDate)
      END
    END
  END
```

See Also: DDEREAD

DDEVALUE (return data value sent to server)

DDEVALUE( )

The **DDEVALUE** function returns a string containing the data sent to a Clarion DDE server by the DDEPOKE statement.

Return Data Type: STRING

Example:

```
WinOne    WINDOW,AT(0,0,160,400)
          END
TimeServer  LONG

TimeStamp   FILE,DRIVER(ASCII),PRE(Tim)
Record      RECORD
FormatTime      STRING(5)
FormatDate      STRING(8)
Message         STRING(50)
          . .

  CODE
  OPEN(WinOne)
  TimeServer = DDESERVER(´TimeStamp´)            !Open as server
  ACCEPT
    CASE EVENT()
    OF EVENT:DDEpoke
      OPEN(TimeStamp)
      Tim:FormatTime = FORMAT(CLOCK(),@T1)
      Tim:FormatDate = FORMAT(TODAY(),@D1)
      Tim:Message   = DDEVALUE()               !Get data
      ADD(TimeStamp)
      CLOSE(TimeStamp)
      CYCLE                                  !Ensure acknowledgement
    END
  END
```

See Also: DDEPOKE

DDE Procedures

DDEREAD (get data from DDE server)

DDEREAD(*channel, mode, item* [, *variable*] **)**

DDEREAD	Gets data from a previously opened DDE client channel.
channel	A LONG integer constant or variable containing the client channel--the value returned by the DDECLIENT function.
mode	An EQUATE defining the type of data link: DDE:auto, DDE:manual, or DDE:remove (defined in EQUATES.CLW).
item	A string constant or variable containing the application-specific name of the data item to retrieve.
variable	The name of the variable to receive the retrieved data. If omitted and *mode* is DDE:remove, all links to the *item* are canceled.

The **DDEREAD** procedure allows a DDE client program to read data from the *channel* into the *variable*. The type of update is determined by the *mode* parameter. The *item* parameter supplies some string value to the server application that tells it what specific data item is being requested. The format and structure of the *item* string is dependent upon the server application.

If the *mode* is DDE:auto, the *variable* is continually updated by the server (a "hot" link). If the *mode* is DDE:manual, the *variable* is updated once and another DDEREAD request must be sent to the server to check for any changed value (a "cold" link). If the *mode* is DDE:remove, a previous "hot" link to the *variable* is terminated. If the *mode* is DDE:remove and *variable* is omitted, all previous "hot" links to the *item* are terminated, no matter what *variables* were linked. This means the client must send another DDEREAD request to the server to check for any changed value.

Errors Posted: 601 Invalid DDE Channel
602 DDE Channel Not Open
605 Time Out

Events Generated:

These events are posted to the client application:

EVENT:DDEdata	A server has supplied an updated data item for a hot link.
EVENT:DDEclose	A server has terminated the DDE link.

Example:

```
WinOne   WINDOW,AT(0,0,160,400)
         END

ExcelServer LONG(0)
DDEReadVal  REAL

  CODE
  OPEN(WinOne)
  ExcelServer = DDECLIENT(´Excel´,´MySheet.XLS´)
                                !Open as client to Excel spreadsheet
  IF NOT ExcelServer             !If the server is not running
    MESSAGE(´Please start Excel´)     ! alert the user to start it
    CLOSE(WinOne)
    RETURN
  END
```

```
END
DDEREAD(ExcelServer,DDE:auto,´R5C5´,DDEReadVal)
                            !Request continual update from server
ACCEPT
  CASE EVENT()
  OF EVENT:DDEdata             !As changed data comes from Excel
    PassedData(DDEReadVal)       ! call proc to process the new data
  END
END
```

See Also: DDEQUERY, DDEWRITE

DDEWRITE (provide data to DDE client)

DDEWRITE(*channel, mode, item* [, *variable*] **)**

DDEWRITE	Provide data to an open DDE server channel.
channel	A LONG integer constant or variable containing the server channel--the value returned by the DDESERVER function.
mode	An integer constant or variable containing the interval (in seconds) to poll for changes to the *variable*, or an EQUATE defining the type of data link: DDE:auto, DDE:manual, or DDE:remove (defined in EQUATES.CLW).
item	A string constant or variable containing the application-specific name of the data item to provide.
variable	The name of the variable providing the data. If omitted and *mode* is DDE:remove, all links to the *item* are canceled.

The **DDEWRITE** procedure allows a DDE server program to provide the *variable´s* data to the client. The *item* parameter supplies a string value that identifies the specific data item being provided. The format and structure of the *item* string is dependent upon the *server* application. The type of update performed is determined by the *mode* parameter.

If the *mode* is DDE:auto, the client program receives the current value of the variable and the internal libraries continue to provide that value whenever the client (or any other client) asks for it again. If the client requested a "hot" link, any changes to the *variable* should be tracked by the Clarion program so it can issue a new DDEWRITE statement to update the client with the new value.

If the *mode* is DDE:manual, the *variable* is updated only once. If the client requested a "hot" link, any changes to the *variable* should be tracked by the Clarion program so it can issue a new DDEWRITE statement to update the client with the new value.

If the *mode* is a positive integer, the internal libraries check the value of the *variable* whenever the specified number of seconds has passed. If the value has changed, the client is automatically updated with the new value by the internal libraries (without the need for any further Clarion code). This can incur significant overhead, depending upon the data, and so should be used only when necessary.

If the *mode* is DDE:remove, any previous "hot" link to the *variable* is terminated. If the *mode* is DDE:remove and *variable* is omitted, all previous "hot" links to the *item* are terminated, no matter what *variables* were linked. This means the client must send another DDEREAD request to the server to check for any changed value.

Errors Posted: 601 Invalid DDE Channel
602 DDE Channel Not Open
605 Time Out

Events Generated:

EVENT:DDErequest
A client has requested a data item (a "cold" link).

EVENT:DDEadvise
A client has requested continuous updates of a data item (a "hot" link).

Example:

```
DDERetVal     STRING(20)
```

```
WinOne     WINDOW,AT(0,0,160,400)
             ENTRY(@s20),USE(DDERetVal)
           END
MyServer     LONG
  CODE
  OPEN(WinOne)
  MyServer = DDESERVER(´MyApp´,´DataEntered´)    !Open as server
  ACCEPT
    CASE EVENT()
    OF EVENT:DDErequest                   !As server for data requested once
      DDEWRITE(MyServer,DDE:manual,´DataEntered´,DDERetVal)
                                        !Provide data once
    OF EVENT:DDEadvise                    !As server for constant update request
      DDEWRITE(MyServer,15,´DataEntered´,DDERetVal)
                                        !Check for change every 15 seconds
                                        ! and provide data whenever changed
    END
  END
```

See Also: DDEQUERY, DDEREAD

DDEEXECUTE (send command to DDE server)

DDEEXECUTE(*channel, command* **)**

DDEEXECUTE Sends a command string to an open DDE client channel.

channel A LONG integer constant or variable containing the client channel--the value returned by the DDECLIENT function.

command A string constant or variable containing the application-specific command for the server to execute.

The **DDEEXECUTE** procedure allows a DDE client program to communicate a *command* to the server. The *command* must be in a format the server application can recognize and act on. The server does not need to be a Clarion program. By convention, the entire *command* string is normally contained within square brackets ([ ]).

A Clarion DDE server can use the DDEITEM() function to determine what *command* the client has sent.The CYCLE statement after an EVENT:DDEexecute signals positive acknowledgement to the client that sent the *command*.

Errors Posted: 601 Invalid DDE Channel
602 DDE Channel Not Open
603 DDEEXECUTE Failed
605 Time Out

Events Generated:

EVENT:DDEcommand
A client has sent a command.

EVENT:DDEexecute
A client has sent a command.

Example:

```
     !The client application´s code contains:
WinOne  WINDOW,AT(0,0,160,400)
        END
SomeServer LONG
DDEChannel LONG
  CODE
  OPEN(WinOne)
  DDEChannel = DDECLIENT(´PROGMAN´,´PROGMAN´)
                                  !Open a channel to Windows Program Manager
  DDEEXECUTE(DDEChannel,´[CreateGroup(Clarion Applications)]´)
                                    !Create a new program group
  DDEEXECUTE(DDEChannel,´[ShowGroup(1)]´)   !Display it
  DDEEXECUTE(DDEChannel,´[AddItem(MYAPP.EXE,My Program,PROGMAN.EXE,2)]´)
                                    !Create new item in the group
                                    ! using second icon in progman.exe
```

DDEPOKE (send unsolicited data to DDE server)

DDEPOKE(*channel, item, value* **)**

DDEPOKE Sends unsolicited data through an open DDE client channel to a DDE server.

channel A LONG integer constant or variable containing the client channel--the value returned by the DDECLIENT function.

item A string constant or variable containing the application-specific item to receive the unsolicited data.

value A string constant or variable containing the data to place in the *item*.

The **DDEPOKE** procedure allows a DDE client program to communicate unsolicited data to the server. The *item* and *value* parameters must be in a format the server application can recognize and act on. The server does not need to be a Clarion program.

A Clarion DDE server can use the DDEITEM() and DDEVALUE() functions to determine what the client has sent. The CYCLE statement after an EVENT:DDEpoke signals positive acknowledgement to the client that sent the unsolicited data.

Errors Posted: 601 Invalid DDE Channel
602 DDE Channel Not Open
604 DDEPOKE Failed
605 Time Out

Events Generated:

EVENT:DDEpoke
A client has sent unsolicited data

Example:

```
WinOne     WINDOW,AT(0,0,160,400)
           END
DDEChannel  LONG
  CODE
  OPEN(WinOne)
  DDEChannel = DDECLIENT(´Excel´,´System´)            !Open channel to Excel
  DDEEXECUTE(DDEChannel,´[NEW(1)]´)                  !Create a new spreadsheet
  DDEEXECUTE(DDEChannel,´[Save.As("DDE_CHART.XLS")]´) !Save it as DDE_CHART.XLS
  DDECLOSE(DDEChannel)                               !Close conversation
  DDEChannel = DDECLIENT(´Excel´,´DDE_CHART.XLA´)     !Open channel to new chart
  DDEPOKE(DDEChannel,´R1C2´,´Widgets´)               !Send it data
  DDEPOKE(DDEChannel,´R1C3´,´Gadgets´)
  DDEPOKE(DDEChannel,´R2C1´,´East´)
  DDEPOKE(DDEChannel,´R3C1´,´West´)
  DDEPOKE(DDEChannel,´R2C2´,´450´)
  DDEPOKE(DDEChannel,´R3C2´,´275´)
  DDEPOKE(DDEChannel,´R2C3´,´340´)
  DDEPOKE(DDEChannel,´R3C3´,´390´)
  DDEEXECUTE(DDEChannel,´[SELECT("R1C1:R3C2")]´)      !Highlight the data
  DDEEXECUTE(DDEChannel,´[NEW(2,2)]´)                 ! and create a new chart
     !Send some more commands to format the chart and work with it
  DDECLOSE(DDEChannel)                               !Close channel when done
```

DDECLOSE (terminate DDE server link)

DDECLOSE(*channel* **)**

DDECLOSE Closes an open DDE channel.

channel The label of the LONG integer variable containing the channel number--the value returned by the DDESERVER or DDECLIENT function.

The **DDECLOSE** procedure allows a DDE client program to terminate the specified *channel*. A *channel* is automatically terminated when the window which opened the *channel* is closed.

Errors Posted: 601 Invalid DDE Channel
602 DDE Channel Not Open
605 Time Out

Example:

```
WinOne     WINDOW,AT(0,0,160,400)
           END
SomeServer  LONG
  CODE
  OPEN(WinOne)
  SomeServer = DDECLIENT(´SomeApp´,´MyTopic´) !Open as client
  ACCEPT
  END
  DDECLOSE(SomeServer)
```

Keycodes

Windows Keycode Mapping Format

KEYCODES.CLW

Clarion Keycodes

Windows Keycode Mapping Format

KEYCODES.CLW

Windows Keycode Mapping Format

Each key on the keyboard is assigned a keycode. Keycodes are 16-bit values where the low-order 8 bits (values from 0 to 255) represent the key that was pressed, and the high-order 8 bits indicate the state of the Shift, Ctrl, and Alt keys. Keycodes are returned by the KEYCODE() and KEYBOARD() functions, and use the following format:

```
          | A | C | S |  CODE   |

 Bits: 10   9   8   7        0

CODE   - The Key pressed
A      - Alt key bit
C      - Ctrl key bit
S      - Shift key bit
```

Calculating a keycode´s numeric value is generally unnecessary, since most of the possible key combinations are listed as EQUATES in KEYCODES.CLW (INCLUDE this file and use the equates instead of the numbers). The contents of KEYCODES.CLW are listed in Appendix A.

KEYCODES.CLW

Keycode equate labels assign mnemonic labels to Clarion keycodes. The keycode equates file (KEYCODES.CLW) is a Clarion source file which contains an EQUATE statement for each keycode. This file is located in the directory in which you installed Clarion Database Developer. It may be merged into a source PROGRAM with the statement: INCLUDE(´KEYCODES.CLW´).

This file contains EQUATE statements for all the keycodes:

```
Key0            EQUATE(0030H)           !0 Key
Key1            EQUATE(0031H)           !1 Key
Key2            EQUATE(0032H)           !2 Key
Key3            EQUATE(0033H)           !3 Key
Key4            EQUATE(0034H)           !4 Key
Key5            EQUATE(0035H)           !5 Key
Key6            EQUATE(0036H)           !6 Key
Key7            EQUATE(0037H)           !7 Key
Key8            EQUATE(0038H)           !8 Key
Key9            EQUATE(0039H)           !9 Key
AKey            EQUATE(0041H)           !A Key
BKey            EQUATE(0042H)           !B Key
CKey            EQUATE(0043H)           !C Key
DKey            EQUATE(0044H)           !D Key
EKey            EQUATE(0045H)           !E Key
FKey            EQUATE(0046H)           !F Key
GKey            EQUATE(0047H)           !G Key
HKey            EQUATE(0048H)           !H Key
IKey            EQUATE(0049H)           !I Key
JKey            EQUATE(004AH)           !J Key
KKey            EQUATE(004BH)           !K Key
LKey            EQUATE(004CH)           !L Key
MKey            EQUATE(004DH)           !M Key
NKey            EQUATE(004EH)           !N Key
OKey            EQUATE(004FH)           !O Key
PKey            EQUATE(0050H)           !P Key
QKey            EQUATE(0051H)           !Q Key
RKey            EQUATE(0052H)           !R Key
SKey            EQUATE(0053H)           !S Key
TKey            EQUATE(0054H)           !T Key
UKey            EQUATE(0055H)           !U Key
VKey            EQUATE(0056H)           !V Key
WKey            EQUATE(0057H)           !W Key
XKey            EQUATE(0058H)           !X Key
YKey            EQUATE(0059H)           !Y Key
ZKey            EQUATE(005AH)           !Z Key
F1Key           EQUATE(0070H)           !F1 Key
F2Key           EQUATE(0071H)           !F2 Key
F3Key           EQUATE(0072H)           !F3 Key
F4Key           EQUATE(0073H)           !F4 Key
F5Key           EQUATE(0074H)           !F5 Key
F6Key           EQUATE(0075H)           !F6 Key
F7Key           EQUATE(0076H)           !F7 Key
F8Key           EQUATE(0077H)           !F8 Key
F9Key           EQUATE(0078H)           !F9 Key
F10Key          EQUATE(0079H)          !F10 Key
F11Key          EQUATE(007AH)          !F11 Key
F12Key          EQUATE(007BH)          !F12 Key
AstKey          EQUATE(006AH)          !Asterisk Key
BSKey             EQUATE(0008H)           !Backspace Key
```

```
CapsLockKey     EQUATE(0014H)               !CapsLock Key
DecimalKey      EQUATE(006EH)               !Decimal Key
DeleteKey         EQUATE(002EH)               !Delete Key
DivideKey         EQUATE(006FH)               !Divide Key
DownKey             EQUATE(0028H)               !Cursor Down Key
EndKey              EQUATE(0023H)               !End Key
EnterKey          EQUATE(000DH)               !Enter Key
EscKey              EQUATE(001BH)               !Esc Key
HomeKey             EQUATE(0024H)               !Home Key
InsertKey         EQUATE(002DH)               !Insert Key
LeftKey             EQUATE(0025H)               !Cursor Left Key
MinusKey          EQUATE(006DH)               !Minus Key
PauseKey          EQUATE(0013H)               !Pause Key
PgDnKey             EQUATE(0022H)               !PgDn Key
PgUpKey             EQUATE(0021H)               !PgUp Key
PlusKey             EQUATE(006BH)               !Plus Key
PrintKey          EQUATE(002CH)               !PrintScreen Key
RightKey          EQUATE(0027H)               !Cursor Right Key
SlashKey          EQUATE(006FH)               !Slash Key
SpaceKey          EQUATE(0020H)               !Spacebar
TabKey              EQUATE(0009H)               !Tab Key
UpKey                 EQUATE(0026H)               !Cursor Up Key
KeyPad0           EQUATE(0060H)               !0 on numeric keypad
KeyPad1           EQUATE(0061H)               !1 on numeric keypad
KeyPad2           EQUATE(0062H)               !2 on numeric keypad
KeyPad3           EQUATE(0063H)               !3 on numeric keypad
KeyPad4           EQUATE(0064H)               !4 on numeric keypad
KeyPad5           EQUATE(0065H)               !5 on numeric keypad
KeyPad6           EQUATE(0066H)               !6 on numeric keypad
KeyPad7           EQUATE(0067H)               !7 on numeric keypad
KeyPad8           EQUATE(0068H)               !8 on numeric keypad
KeyPad9           EQUATE(0069H)               !9 on numeric keypad
MouseLeft       EQUATE(0001H)               !Left mouse button
MouseRight      EQUATE(0002H)               !Right mouse button
MouseCenter     EQUATE(0004H)               !Middle mouse button
Alt0                EQUATE(0430H)               !Alt-0 Key
Alt1                EQUATE(0431H)               !Alt-1 Key
Alt2                EQUATE(0432H)               !Alt-2 Key
Alt3                EQUATE(0433H)               !Alt-3 Key
Alt4                EQUATE(0434H)               !Alt-4 Key
Alt5                EQUATE(0435H)               !Alt-5 Key
Alt6                EQUATE(0436H)               !Alt-6 Key
Alt7                EQUATE(0437H)               !Alt-7 Key
Alt8                EQUATE(0438H)               !Alt-8 Key
Alt9                EQUATE(0439H)               !Alt-9 Key
AltA                EQUATE(0441H)               !Alt-A Key
AltB                EQUATE(0442H)               !Alt-B Key
AltC                EQUATE(0443H)               !Alt-C Key
AltD                EQUATE(0444H)               !Alt-D Key
AltE                EQUATE(0445H)               !Alt-E Key
AltF                EQUATE(0446H)               !Alt-F Key
AltG                EQUATE(0447H)               !Alt-G Key
AltH                EQUATE(0448H)               !Alt-H Key
AltI                EQUATE(0449H)               !Alt-I Key
AltJ                EQUATE(044AH)               !Alt-J Key
AltK                EQUATE(044BH)               !Alt-K Key
AltL                EQUATE(044CH)               !Alt-L Key
AltM                EQUATE(044DH)               !Alt-M Key
AltN                EQUATE(044EH)               !Alt-N Key
AltO                EQUATE(044FH)               !Alt-O Key
```

```
AltP                EQUATE(0450H)               !Alt-P Key
AltQ                EQUATE(0451H)               !Alt-Q Key
AltR                EQUATE(0452H)               !Alt-R Key
AltS                EQUATE(0453H)               !Alt-S Key
AltT                EQUATE(0454H)               !Alt-T Key
AltU                EQUATE(0455H)               !Alt-U Key
AltV                EQUATE(0456H)               !Alt-V Key
AltW                EQUATE(0457H)               !Alt-W Key
AltX                EQUATE(0458H)               !Alt-X Key
AltY                EQUATE(0459H)               !Alt-Y Key
AltZ                EQUATE(045AH)               !Alt-Z Key
AltF1               EQUATE(0470H)               !Alt-F1 Key
AltF2               EQUATE(0471H)               !Alt-F2 Key
AltF3               EQUATE(0472H)               !Alt-F3 Key
AltF4               EQUATE(0473H)               !Alt-F4 Key
AltF5               EQUATE(0474H)               !Alt-F5 Key
AltF6               EQUATE(0475H)               !Alt-F6 Key
AltF7               EQUATE(0476H)               !Alt-F7 Key
AltF8               EQUATE(0477H)               !Alt-F8 Key
AltF9               EQUATE(0478H)               !Alt-F9 Key
AltF10              EQUATE(0479H)              !Alt-F10 Key
AltF11              EQUATE(047AH)              !Alt-F11 Key
AltF12              EQUATE(047BH)              !Alt-F12 Key
AltAst              EQUATE(046AH)              !Alt-Asterisk Key
AltBS                 EQUATE(0408H)                !Alt-Backspace Key
AltDecimal      EQUATE(046EH)              !Alt-Decimal Key
AltDelete         EQUATE(042EH)               !Alt-Delete Key
AltDivide         EQUATE(046FH)               !Alt-Divide Key
AltDown             EQUATE(0428H)                !Alt-Cursor Down Key
AltEnd              EQUATE(0423H)                !Alt-End Key
AltEnter          EQUATE(040DH)               !Alt-Enter Key
AltEsc              EQUATE(041BH)                !Alt-Esc Key
AltHome             EQUATE(0424H)                !Alt-Home Key
AltInsert         EQUATE(042DH)               !Alt-Insert Key
AltLeft             EQUATE(0425H)                !Alt-Cursor Left Key
AltMinus          EQUATE(046DH)               !Alt-Minus Key
AltPause          EQUATE(0413H)               !Alt-Pause Key
AltPgDn             EQUATE(0422H)                !Alt-PgDn Key
AltPgUp             EQUATE(0421H)                !Alt-PgUp Key
AltPlus             EQUATE(046BH)                !Alt-Plus Key
AltPrint          EQUATE(042CH)               !Alt-PrintScreen Key
AltRight          EQUATE(0427H)               !Alt-Cursor Right Key
AltSlash          EQUATE(046FH)               !Alt-Slash Key
AltSpace          EQUATE(0420H)               !Alt-Spacebar
AltTab              EQUATE(0409H)                !Alt-Tab Key
AltUp                 EQUATE(0426H)                  !Alt-Cursor Up Key
AltPad0             EQUATE(0460H)              !Alt-0 on numeric keypad
AltPad1             EQUATE(0461H)              !Alt-1 on numeric keypad
AltPad2             EQUATE(0462H)              !Alt-2 on numeric keypad
AltPad3             EQUATE(0463H)              !Alt-3 on numeric keypad
AltPad4             EQUATE(0464H)              !Alt-4 on numeric keypad
AltPad5             EQUATE(0465H)              !Alt-5 on numeric keypad
AltPad6             EQUATE(0466H)              !Alt-6 on numeric keypad
AltPad7             EQUATE(0467H)              !Alt-7 on numeric keypad
AltPad8             EQUATE(0468H)              !Alt-8 on numeric keypad
AltPad9             EQUATE(0469H)              !Alt-9 on numeric keypad
AltMouseLeft    EQUATE(0401H)             !Alt-Left mouse button
AltMouseRight   EQUATE(0402H)             !Alt-Right mouse button
AltMouseCenter  EQUATE(0404H)             !Alt-Middle mouse button
Ctrl0               EQUATE(0230H)               !Ctrl-0 Key
```

```
Ctrl1               EQUATE(0231H)             !Ctrl-1 Key
Ctrl2               EQUATE(0232H)             !Ctrl-2 Key
Ctrl3              EQUATE(0233H)             !Ctrl-3 Key
Ctrl4              EQUATE(0234H)             !Ctrl-4 Key
Ctrl5              EQUATE(0235H)             !Ctrl-5 Key
Ctrl6              EQUATE(0236H)             !Ctrl-6 Key
Ctrl7              EQUATE(0237H)             !Ctrl-7 Key
Ctrl8              EQUATE(0238H)             !Ctrl-8 Key
Ctrl9              EQUATE(0239H)             !Ctrl-9 Key
CtrlA              EQUATE(0241H)             !Ctrl-A Key
CtrlB              EQUATE(0242H)             !Ctrl-B Key
CtrlC              EQUATE(0243H)             !Ctrl-C Key
CtrlD              EQUATE(0244H)             !Ctrl-D Key
CtrlE              EQUATE(0245H)             !Ctrl-E Key
CtrlF              EQUATE(0246H)             !Ctrl-F Key
CtrlG              EQUATE(0247H)             !Ctrl-G Key
CtrlH              EQUATE(0248H)             !Ctrl-H Key
CtrlI              EQUATE(0249H)             !Ctrl-I Key
CtrlJ              EQUATE(024AH)             !Ctrl-J Key
CtrlK              EQUATE(024BH)             !Ctrl-K Key
CtrlL              EQUATE(024CH)             !Ctrl-L Key
CtrlM              EQUATE(024DH)             !Ctrl-M Key
CtrlN              EQUATE(024EH)             !Ctrl-N Key
CtrlO              EQUATE(024FH)             !Ctrl-O Key
CtrlP              EQUATE(0250H)             !Ctrl-P Key
CtrlQ              EQUATE(0251H)             !Ctrl-Q Key
CtrlR              EQUATE(0252H)             !Ctrl-R Key
CtrlS              EQUATE(0253H)             !Ctrl-S Key
CtrlT              EQUATE(0254H)             !Ctrl-T Key
CtrlU              EQUATE(0255H)             !Ctrl-U Key
CtrlV              EQUATE(0256H)             !Ctrl-V Key
CtrlW              EQUATE(0257H)             !Ctrl-W Key
CtrlX              EQUATE(0258H)             !Ctrl-X Key
CtrlY              EQUATE(0259H)             !Ctrl-Y Key
CtrlZ              EQUATE(025AH)             !Ctrl-Z Key
CtrlF1            EQUATE(0270H)            !Ctrl-F1 Key
CtrlF2            EQUATE(0271H)            !Ctrl-F2 Key
CtrlF3            EQUATE(0272H)            !Ctrl-F3 Key
CtrlF4            EQUATE(0273H)            !Ctrl-F4 Key
CtrlF5            EQUATE(0274H)            !Ctrl-F5 Key
CtrlF6            EQUATE(0275H)            !Ctrl-F6 Key
CtrlF7            EQUATE(0276H)            !Ctrl-F7 Key
CtrlF8            EQUATE(0277H)            !Ctrl-F8 Key
CtrlF9            EQUATE(0278H)            !Ctrl-F9 Key
CtrlF10           EQUATE(0279H)            !Ctrl-F10 Key
CtrlF11           EQUATE(027AH)            !Ctrl-F11 Key
CtrlF12           EQUATE(027BH)            !Ctrl-F12 Key
CtrlAst           EQUATE(026AH)            !Ctrl-Asterisk Key
CtrlBS            EQUATE(0208H)            !Ctrl-Backspace Key
CtrlDecimal    EQUATE(026EH)            !Ctrl-Decimal Key
CtrlDelete     EQUATE(022EH)            !Ctrl-Delete Key
CtrlDivide     EQUATE(026FH)            !Ctrl-Divide Key
CtrlDown         EQUATE(0228H)              !Ctrl-Cursor Down Key
CtrlEnd            EQUATE(0223H)              !Ctrl-End Key
CtrlEnter        EQUATE(020DH)              !Ctrl-Enter Key
CtrlEsc            EQUATE(021BH)              !Ctrl-Esc Key
CtrlHome         EQUATE(0224H)              !Ctrl-Home Key
CtrlInsert     EQUATE(022DH)            !Ctrl-Insert Key
CtrlLeft         EQUATE(0225H)              !Ctrl-Cursor Left Key
CtrlMinus        EQUATE(026DH)              !Ctrl-Minus Key
```

```
CtrlPause        EQUATE(0213H)              !Ctrl-Pause Key
CtrlPgDn         EQUATE(0222H)              !Ctrl-PgDn Key
CtrlPgUp         EQUATE(0221H)              !Ctrl-PgUp Key
CtrlPlus         EQUATE(026BH)              !Ctrl-Plus Key
CtrlPrint        EQUATE(022CH)              !Ctrl-PrintScreen Key
CtrlRight        EQUATE(0227H)              !Ctrl-Cursor Right Key
CtrlSlash        EQUATE(026FH)              !Ctrl-Slash Key
CtrlSpace        EQUATE(0220H)              !Ctrl-Spacebar
CtrlTab            EQUATE(0209H)              !Ctrl-Tab Key
CtrlUp             EQUATE(0226H)              !Ctrl-Cursor Up Key
CtrlPad0         EQUATE(0260H)            !Ctrl-0 on numeric keypad
CtrlPad1         EQUATE(0261H)            !Ctrl-1 on numeric keypad
CtrlPad2         EQUATE(0262H)            !Ctrl-2 on numeric keypad
CtrlPad3         EQUATE(0263H)            !Ctrl-3 on numeric keypad
CtrlPad4         EQUATE(0264H)            !Ctrl-4 on numeric keypad
CtrlPad5         EQUATE(0265H)            !Ctrl-5 on numeric keypad
CtrlPad6         EQUATE(0266H)            !Ctrl-6 on numeric keypad
CtrlPad7         EQUATE(0267H)            !Ctrl-7 on numeric keypad
CtrlPad8         EQUATE(0268H)            !Ctrl-8 on numeric keypad
CtrlPad9         EQUATE(0269H)            !Ctrl-9 on numeric keypad
CtrlMouseLeft  EQUATE(0201H)            !Ctrl-Left mouse button
CtrlMouseRight EQUATE(0202H)            !Ctrl-Right mouse button
CtrlMouseCenter  EQUATE(0204H)            !Ctrl-Middle mouse button
Shift0             EQUATE(0130H)              !Shift-0 Key
Shift1             EQUATE(0131H)              !Shift-1 Key
Shift2             EQUATE(0132H)              !Shift-2 Key
Shift3             EQUATE(0133H)              !Shift-3 Key
Shift4             EQUATE(0134H)              !Shift-4 Key
Shift5             EQUATE(0135H)              !Shift-5 Key
Shift6             EQUATE(0136H)              !Shift-6 Key
Shift7             EQUATE(0137H)              !Shift-7 Key
Shift8             EQUATE(0138H)              !Shift-8 Key
Shift9             EQUATE(0139H)              !Shift-9 Key
ShiftA             EQUATE(0141H)              !Shift-A Key
ShiftB             EQUATE(0142H)              !Shift-B Key
ShiftC             EQUATE(0143H)              !Shift-C Key
ShiftD             EQUATE(0144H)              !Shift-D Key
ShiftE             EQUATE(0145H)              !Shift-E Key
ShiftF             EQUATE(0146H)              !Shift-F Key
ShiftG             EQUATE(0147H)              !Shift-G Key
ShiftH             EQUATE(0148H)              !Shift-H Key
ShiftI             EQUATE(0149H)              !Shift-I Key
ShiftJ             EQUATE(014AH)              !Shift-J Key
ShiftK             EQUATE(014BH)              !Shift-K Key
ShiftL             EQUATE(014CH)              !Shift-L Key
ShiftM             EQUATE(014DH)              !Shift-M Key
ShiftN             EQUATE(014EH)              !Shift-N Key
ShiftO             EQUATE(014FH)              !Shift-O Key
ShiftP             EQUATE(0150H)              !Shift-P Key
ShiftQ             EQUATE(0151H)              !Shift-Q Key
ShiftR             EQUATE(0152H)              !Shift-R Key
ShiftS             EQUATE(0153H)              !Shift-S Key
ShiftT             EQUATE(0154H)              !Shift-T Key
ShiftU             EQUATE(0155H)              !Shift-U Key
ShiftV             EQUATE(0156H)              !Shift-V Key
ShiftW             EQUATE(0157H)              !Shift-W Key
ShiftX             EQUATE(0158H)              !Shift-X Key
ShiftY             EQUATE(0159H)              !Shift-Y Key
ShiftZ             EQUATE(015AH)              !Shift-Z Key
ShiftF1            EQUATE(0170H)              !Shift-F1 Key
```

```
ShiftF2            EQUATE(0171H)              !Shift-F2 Key
ShiftF3            EQUATE(0172H)              !Shift-F3 Key
ShiftF4            EQUATE(0173H)              !Shift-F4 Key
ShiftF5            EQUATE(0174H)              !Shift-F5 Key
ShiftF6            EQUATE(0175H)              !Shift-F6 Key
ShiftF7            EQUATE(0176H)              !Shift-F7 Key
ShiftF8            EQUATE(0177H)              !Shift-F8 Key
ShiftF9            EQUATE(0178H)              !Shift-F9 Key
ShiftF10         EQUATE(0179H)              !Shift-F10 Key
ShiftF11         EQUATE(017AH)              !Shift-F11 Key
ShiftF12         EQUATE(017BH)              !Shift-F12 Key
ShiftAst         EQUATE(016AH)              !Shift-Asterisk Key
ShiftBS            EQUATE(0108H)              !Shift-Backspace Key
ShiftDecimal   EQUATE(016EH)            !Shift-Decimal Key
ShiftDelete    EQUATE(012EH)            !Shift-Delete Key
ShiftDivide    EQUATE(016FH)            !Shift-Divide Key
ShiftDown        EQUATE(0128H)              !Shift-Cursor Down Key
ShiftEnd         EQUATE(0123H)              !Shift-End Key
ShiftEnter     EQUATE(010DH)            !Shift-Enter Key
ShiftEsc         EQUATE(011BH)              !Shift-Esc Key
ShiftHome        EQUATE(0124H)              !Shift-Home Key
ShiftInsert    EQUATE(012DH)            !Shift-Insert Key
ShiftLeft        EQUATE(0125H)              !Shift-Cursor Left Key
ShiftMinus     EQUATE(016DH)            !Shift-Minus Key
ShiftPause     EQUATE(0113H)            !Shift-Pause Key
ShiftPgDn        EQUATE(0122H)              !Shift-PgDn Key
ShiftPgUp        EQUATE(0121H)              !Shift-PgUp Key
ShiftPlus        EQUATE(016BH)              !Shift-Plus Key
ShiftPrint     EQUATE(012CH)            !Shift-PrintScreen Key
ShiftRight     EQUATE(0127H)            !Shift-Cursor Right Key
ShiftSlash     EQUATE(016FH)            !Shift-Slash Key
ShiftSpace     EQUATE(0120H)            !Shift-Spacebar
ShiftTab         EQUATE(0109H)              !Shift-Tab Key
ShiftUp            EQUATE(0126H)              !Shift-Cursor Up Key
ShiftPad0        EQUATE(0160H)            !Shift-0 on numeric keypad
ShiftPad1        EQUATE(0161H)            !Shift-1 on numeric keypad
ShiftPad2        EQUATE(0162H)            !Shift-2 on numeric keypad
ShiftPad3        EQUATE(0163H)            !Shift-3 on numeric keypad
ShiftPad4        EQUATE(0164H)            !Shift-4 on numeric keypad
ShiftPad5        EQUATE(0165H)            !Shift-5 on numeric keypad
ShiftPad6        EQUATE(0166H)            !Shift-6 on numeric keypad
ShiftPad7        EQUATE(0167H)            !Shift-7 on numeric keypad
ShiftPad8        EQUATE(0168H)            !Shift-8 on numeric keypad
ShiftPad9        EQUATE(0169H)            !Shift-9 on numeric keypad
ShiftMouseLeft  EQUATE(0101H)          !Shift-Left mouse button
ShiftMouseRight   EQUATE(0102H)          !Shift-Right mouse button
ShiftMouseCenter EQUATE(0104H)           !Shift-Middle mouse button
AltShift0        EQUATE(0530H)               !Alt-Shift-0 Key
AltShift1        EQUATE(0531H)               !Alt-Shift-1 Key
AltShift2        EQUATE(0532H)               !Alt-Shift-2 Key
AltShift3        EQUATE(0533H)               !Alt-Shift-3 Key
AltShift4        EQUATE(0534H)               !Alt-Shift-4 Key
AltShift5        EQUATE(0535H)               !Alt-Shift-5 Key
AltShift6        EQUATE(0536H)               !Alt-Shift-6 Key
AltShift7        EQUATE(0537H)               !Alt-Shift-7 Key
AltShift8        EQUATE(0538H)               !Alt-Shift-8 Key
AltShift9        EQUATE(0539H)               !Alt-Shift-9 Key
AltShiftA        EQUATE(0541H)               !Alt-Shift-A Key
AltShiftB        EQUATE(0542H)               !Alt-Shift-B Key
AltShiftC        EQUATE(0543H)               !Alt-Shift-C Key
```

```
AltShiftD        EQUATE(0544H)              !Alt-Shift-D Key
AltShiftE        EQUATE(0545H)              !Alt-Shift-E Key
AltShiftF        EQUATE(0546H)              !Alt-Shift-F Key
AltShiftG        EQUATE(0547H)              !Alt-Shift-G Key
AltShiftH        EQUATE(0548H)              !Alt-Shift-H Key
AltShiftI        EQUATE(0549H)              !Alt-Shift-I Key
AltShiftJ        EQUATE(054AH)              !Alt-Shift-J Key
AltShiftK        EQUATE(054BH)              !Alt-Shift-K Key
AltShiftL        EQUATE(054CH)              !Alt-Shift-L Key
AltShiftM        EQUATE(054DH)              !Alt-Shift-M Key
AltShiftN        EQUATE(054EH)              !Alt-Shift-N Key
AltShiftO        EQUATE(054FH)              !Alt-Shift-O Key
AltShiftP        EQUATE(0550H)              !Alt-Shift-P Key
AltShiftQ        EQUATE(0551H)              !Alt-Shift-Q Key
AltShiftR        EQUATE(0552H)              !Alt-Shift-R Key
AltShiftS        EQUATE(0553H)              !Alt-Shift-S Key
AltShiftT        EQUATE(0554H)              !Alt-Shift-T Key
AltShiftU        EQUATE(0555H)              !Alt-Shift-U Key
AltShiftV        EQUATE(0556H)              !Alt-Shift-V Key
AltShiftW        EQUATE(0557H)              !Alt-Shift-W Key
AltShiftX        EQUATE(0558H)              !Alt-Shift-X Key
AltShiftY        EQUATE(0559H)              !Alt-Shift-Y Key
AltShiftZ        EQUATE(055AH)              !Alt-Shift-Z Key
AltShiftF1     EQUATE(0570H)              !Alt-Shift-F1 Key
AltShiftF2     EQUATE(0571H)              !Alt-Shift-F2 Key
AltShiftF3     EQUATE(0572H)              !Alt-Shift-F3 Key
AltShiftF4     EQUATE(0573H)              !Alt-Shift-F4 Key
AltShiftF5     EQUATE(0574H)              !Alt-Shift-F5 Key
AltShiftF6     EQUATE(0575H)              !Alt-Shift-F6 Key
AltShiftF7     EQUATE(0576H)              !Alt-Shift-F7 Key
AltShiftF8     EQUATE(0577H)              !Alt-Shift-F8 Key
AltShiftF9     EQUATE(0578H)              !Alt-Shift-F9 Key
AltShiftF10    EQUATE(0579H)              !Alt-Shift-F10 Key
AltShiftF11    EQUATE(057AH)              !Alt-Shift-F11 Key
AltShiftF12    EQUATE(057BH)              !Alt-Shift-F12 Key
AltShiftAst    EQUATE(056AH)              !Alt-Shift-Asterisk Key
AltShiftBS     EQUATE(0508H)              !Alt-Shift-Backspace
AltShiftDecimal  EQUATE(056EH)              !Alt-Shift-Decimal Key
AltShiftDelete EQUATE(052EH)              !Alt-Shift-Delete Key
AltShiftDivide EQUATE(056FH)              !Alt-Shift-Divide Key
AltShiftDown   EQUATE(0528H)              !Alt-Shift-Cursor Down
AltShiftEnd    EQUATE(0523H)              !Alt-Shift-End Key
AltShiftEnter  EQUATE(050DH)              !Alt-Shift-Enter Key
AltShiftEsc    EQUATE(051BH)              !Alt-Shift-Esc Key
AltShiftHome   EQUATE(0524H)              !Alt-Shift-Home Key
AltShiftInsert EQUATE(052DH)              !Alt-Shift-Insert Key
AltShiftLeft   EQUATE(0525H)              !Alt-Shift-Cursor Left Key
AltShiftMinus  EQUATE(056DH)              !Alt-Shift-Minus Key
AltShiftPause  EQUATE(0513H)              !Alt-Shift-Pause Key
AltShiftPgDn   EQUATE(0522H)              !Alt-Shift-PgDn Key
AltShiftPgUp   EQUATE(0521H)              !Alt-Shift-PgUp Key
AltShiftPlus   EQUATE(056BH)              !Alt-Shift-Plus Key
AltShiftPrint  EQUATE(052CH)              !Alt-Shift-PrintScreen
AltShiftRight  EQUATE(0527H)              !Alt-Shift-Cursor Right
AltShiftSlash  EQUATE(056FH)              !Alt-Shift-Slash Key
AltShiftSpace  EQUATE(0520H)              !Alt-Shift-Spacebar
AltShiftTab    EQUATE(0509H)              !Alt-Shift-Tab Key
AltShiftUp       EQUATE(0526H)                 !Alt-Shift-Cursor Up
AltShiftPad0     EQUATE(0560H)           !Alt-Shift-0 on numeric keypad
AltShiftPad1     EQUATE(0561H)           !Alt-Shift-1 on numeric keypad
```

```
AltShiftPad2      EQUATE(0562H)          Alt-Shift-2 on numeric keypad
AltShiftPad3      EQUATE(0563H)          !Alt-Shift-3 on numeric keypad
AltShiftPad4      EQUATE(0564H)          !Alt-Shift-4 on numeric keypad
AltShiftPad5      EQUATE(0565H)          !Alt-Shift-5 on numeric keypad
AltShiftPad6      EQUATE(0566H)          !Alt-Shift-6 on numeric keypad
AltShiftPad7      EQUATE(0567H)          !Alt-Shift-7 on numeric keypad
AltShiftPad8      EQUATE(0568H)          !Alt-Shift-8 on numeric keypad
AltShiftPad9      EQUATE(0569H)          !Alt-Shift-9 on numeric keypad
AltShiftMouseLeft  EQUATE(0501H)       !Alt-Shift-Left mouse button
AltShiftMouseRight EQUATE(0502H)      !Alt-Shift-Right mouse button
AltShiftMouseCenter EQUATE(0504H)      !Alt-Shift-Middle mouse button
CtrlShift0      EQUATE(0330H)              !Ctrl-Shift-0 Key
CtrlShift1      EQUATE(0331H)              !Ctrl-Shift-1 Key
CtrlShift2      EQUATE(0332H)              !Ctrl-Shift-2 Key
CtrlShift3      EQUATE(0333H)              !Ctrl-Shift-3 Key
CtrlShift4      EQUATE(0334H)              !Ctrl-Shift-4 Key
CtrlShift5      EQUATE(0335H)              !Ctrl-Shift-5 Key
CtrlShift6      EQUATE(0336H)              !Ctrl-Shift-6 Key
CtrlShift7      EQUATE(0337H)              !Ctrl-Shift-7 Key
CtrlShift8      EQUATE(0338H)              !Ctrl-Shift-8 Key
CtrlShift9      EQUATE(0339H)              !Ctrl-Shift-9 Key
CtrlShiftA      EQUATE(0341H)              !Ctrl-Shift-A Key
CtrlShiftB      EQUATE(0342H)              !Ctrl-Shift-B Key
CtrlShiftC      EQUATE(0343H)              !Ctrl-Shift-C Key
CtrlShiftD      EQUATE(0344H)              !Ctrl-Shift-D Key
CtrlShiftE      EQUATE(0345H)              !Ctrl-Shift-E Key
CtrlShiftF      EQUATE(0346H)              !Ctrl-Shift-F Key
CtrlShiftG      EQUATE(0347H)              !Ctrl-Shift-G Key
CtrlShiftH      EQUATE(0348H)              !Ctrl-Shift-H Key
CtrlShiftI      EQUATE(0349H)              !Ctrl-Shift-I Key
CtrlShiftJ      EQUATE(034AH)              !Ctrl-Shift-J Key
CtrlShiftK      EQUATE(034BH)              !Ctrl-Shift-K Key
CtrlShiftL      EQUATE(034CH)              !Ctrl-Shift-L Key
CtrlShiftM      EQUATE(034DH)              !Ctrl-Shift-M Key
CtrlShiftN      EQUATE(034EH)              !Ctrl-Shift-N Key
CtrlShiftO      EQUATE(034FH)              !Ctrl-Shift-O Key
CtrlShiftP      EQUATE(0350H)              !Ctrl-Shift-P Key
CtrlShiftQ      EQUATE(0351H)              !Ctrl-Shift-Q Key
CtrlShiftR      EQUATE(0352H)              !Ctrl-Shift-R Key
CtrlShiftS      EQUATE(0353H)              !Ctrl-Shift-S Key
CtrlShiftT      EQUATE(0354H)              !Ctrl-Shift-T Key
CtrlShiftU      EQUATE(0355H)              !Ctrl-Shift-U Key
CtrlShiftV      EQUATE(0356H)              !Ctrl-Shift-V Key
CtrlShiftW      EQUATE(0357H)              !Ctrl-Shift-W Key
CtrlShiftX      EQUATE(0358H)              !Ctrl-Shift-X Key
CtrlShiftY      EQUATE(0359H)              !Ctrl-Shift-Y Key
CtrlShiftZ      EQUATE(035AH)              !Ctrl-Shift-Z Key
CtrlShiftF1     EQUATE(0370H)              !Ctrl-Shift-F1 Key
CtrlShiftF2     EQUATE(0371H)              !Ctrl-Shift-F2 Key
CtrlShiftF3     EQUATE(0372H)              !Ctrl-Shift-F3 Key
CtrlShiftF4     EQUATE(0373H)              !Ctrl-Shift-F4 Key
CtrlShiftF5     EQUATE(0374H)              !Ctrl-Shift-F5 Key
CtrlShiftF6     EQUATE(0375H)              !Ctrl-Shift-F6 Key
CtrlShiftF7     EQUATE(0376H)              !Ctrl-Shift-F7 Key
CtrlShiftF8     EQUATE(0377H)              !Ctrl-Shift-F8 Key
CtrlShiftF9     EQUATE(0378H)              !Ctrl-Shift-F9 Key
CtrlShiftF10    EQUATE(0379H)              !Ctrl-Shift-F10 Key
CtrlShiftF11    EQUATE(037AH)              !Ctrl-Shift-F11 Key
CtrlShiftF12    EQUATE(037BH)              !Ctrl-Shift-F12 Key
CtrlShiftAst    EQUATE(036AH)              !Ctrl-Shift-Asterisk
```

```
CtrlShiftBS    EQUATE(0308H)              !Ctrl-Shift-Backspace
CtrlShiftDecimal EQUATE(036EH)            !Ctrl-Shift-Decimal
CtrlShiftDelete   EQUATE(032EH)             !Ctrl-Shift-Delete
CtrlShiftDivide  EQUATE(036FH)              !Ctrl-Shift-Divide Key
CtrlShiftDown  EQUATE(0328H)              !Ctrl-Shift-Cursor Down
CtrlShiftEnd   EQUATE(0323H)              !Ctrl-Shift-End Key
CtrlShiftEnter EQUATE(030DH)              !Ctrl-Shift-Enter Key
CtrlShiftEsc   EQUATE(031BH)              !Ctrl-Shift-Esc Key
CtrlShiftHome  EQUATE(0324H)              !Ctrl-Shift-Home Key
CtrlShiftInsert  EQUATE(032DH)              !Ctrl-Shift-Insert Key
CtrlShiftLeft  EQUATE(0325H)              !Ctrl-Shift-Cursor Left
CtrlShiftMinus EQUATE(036DH)              !Ctrl-Shift-Minus Key
CtrlShiftPause EQUATE(0313H)              !Ctrl-Shift-Pause Key
CtrlShiftPgDn  EQUATE(0322H)              !Ctrl-Shift-PgDn Key
CtrlShiftPgUp  EQUATE(0321H)              !Ctrl-Shift-PgUp Key
CtrlShiftPlus  EQUATE(036BH)              !Ctrl-Shift-Plus Key
CtrlShiftPrint EQUATE(032CH)              !Ctrl-Shift-PrintScreen
CtrlShiftRight EQUATE(0327H)              !Ctrl-Shift-Cursor Right
CtrlShiftSlash EQUATE(036FH)              !Ctrl-Shift-Slash Key
CtrlShiftSpace EQUATE(0320H)              !Ctrl-Shift-Spacebar
CtrlShiftTab   EQUATE(0309H)              !Ctrl-Shift-Tab Key
CtrlShiftUp    EQUATE(0326H)              !Ctrl-Shift-Cursor Up
CtrlShiftPad0  EQUATE(0360H)          !Ctrl-Shift-0 on numeric keypad
CtrlShiftPad1  EQUATE(0361H)          !Ctrl-Shift-1 on numeric keypad
CtrlShiftPad2  EQUATE(0362H)          !Ctrl-Shift-2 on numeric keypad
CtrlShiftPad3  EQUATE(0363H)          !Ctrl-Shift-3 on numeric keypad
CtrlShiftPad4  EQUATE(0364H)          !Ctrl-Shift-4 on numeric keypad
CtrlShiftPad5  EQUATE(0365H)          !Ctrl-Shift-5 on numeric keypad
CtrlShiftPad6  EQUATE(0366H)          !Ctrl-Shift-6 on numeric keypad
CtrlShiftPad7  EQUATE(0367H)          !Ctrl-Shift-7 on numeric keypad
CtrlShiftPad8  EQUATE(0368H)          !Ctrl-Shift-8 on numeric keypad
CtrlShiftPad9  EQUATE(0369H)          !Ctrl-Shift-9 on numeric keypad
CtrlShiftMouseLeft  EQUATE(0301H)     !Ctrl-Shift-Left mouse button
CtrlShiftMouseRight  EQUATE(0302H)      !Ctrl-Shift-Right mouse button
CtrlShiftMouseCenter  EQUATE(0304H)     !Ctrl-Shift-Middle mouse button
CtrlAlt0          EQUATE(0630H)               !Ctrl-Alt-0 Key
CtrlAlt1          EQUATE(0631H)               !Ctrl-Alt-1 Key
CtrlAlt2          EQUATE(0632H)               !Ctrl-Alt-2 Key
CtrlAlt3          EQUATE(0633H)               !Ctrl-Alt-3 Key
CtrlAlt4          EQUATE(0634H)               !Ctrl-Alt-4 Key
CtrlAlt5          EQUATE(0635H)               !Ctrl-Alt-5 Key
CtrlAlt6          EQUATE(0636H)               !Ctrl-Alt-6 Key
CtrlAlt7          EQUATE(0637H)               !Ctrl-Alt-7 Key
CtrlAlt8          EQUATE(0638H)               !Ctrl-Alt-8 Key
CtrlAlt9          EQUATE(0639H)               !Ctrl-Alt-9 Key
CtrlAltA          EQUATE(0641H)               !Ctrl-Alt-A Key
CtrlAltB          EQUATE(0642H)               !Ctrl-Alt-B Key
CtrlAltC          EQUATE(0643H)               !Ctrl-Alt-C Key
CtrlAltD          EQUATE(0644H)               !Ctrl-Alt-D Key
CtrlAltE          EQUATE(0645H)               !Ctrl-Alt-E Key
CtrlAltF          EQUATE(0646H)               !Ctrl-Alt-F Key
CtrlAltG          EQUATE(0647H)               !Ctrl-Alt-G Key
CtrlAltH          EQUATE(0648H)               !Ctrl-Alt-H Key
CtrlAltI          EQUATE(0649H)               !Ctrl-Alt-I Key
CtrlAltJ          EQUATE(064AH)               !Ctrl-Alt-J Key
CtrlAltK          EQUATE(064BH)               !Ctrl-Alt-K Key
CtrlAltL          EQUATE(064CH)               !Ctrl-Alt-L Key
CtrlAltM          EQUATE(064DH)               !Ctrl-Alt-M Key
CtrlAltN          EQUATE(064EH)               !Ctrl-Alt-N Key
CtrlAltO          EQUATE(064FH)               !Ctrl-Alt-O Key
```

```
CtrlAltP          EQUATE(0650H)               !Ctrl-Alt-P Key
CtrlAltQ          EQUATE(0651H)               !Ctrl-Alt-Q Key
CtrlAltR          EQUATE(0652H)               !Ctrl-Alt-R Key
CtrlAltS          EQUATE(0653H)               !Ctrl-Alt-S Key
CtrlAltT          EQUATE(0654H)               !Ctrl-Alt-T Key
CtrlAltU          EQUATE(0655H)               !Ctrl-Alt-U Key
CtrlAltV          EQUATE(0656H)               !Ctrl-Alt-V Key
CtrlAltW          EQUATE(0657H)               !Ctrl-Alt-W Key
CtrlAltX          EQUATE(0658H)               !Ctrl-Alt-X Key
CtrlAltY          EQUATE(0659H)               !Ctrl-Alt-Y Key
CtrlAltZ          EQUATE(065AH)               !Ctrl-Alt-Z Key
CtrlAltF1         EQUATE(0670H)               !Ctrl-Alt-F1 Key
CtrlAltF2        EQUATE(0671H)              !Ctrl-Alt-F2 Key
CtrlAltF3        EQUATE(0672H)              !Ctrl-Alt-F3 Key
CtrlAltF4        EQUATE(0673H)              !Ctrl-Alt-F4 Key
CtrlAltF5        EQUATE(0674H)              !Ctrl-Alt-F5 Key
CtrlAltF6        EQUATE(0675H)              !Ctrl-Alt-F6 Key
CtrlAltF7        EQUATE(0676H)              !Ctrl-Alt-F7 Key
CtrlAltF8        EQUATE(0677H)              !Ctrl-Alt-F8 Key
CtrlAltF9        EQUATE(0678H)              !Ctrl-Alt-F9 Key
CtrlAltF10     EQUATE(0679H)             !Ctrl-Alt-F10 Key
CtrlAltF11     EQUATE(067AH)             !Ctrl-Alt-F11 Key
CtrlAltF12     EQUATE(067BH)             !Ctrl-Alt-F12 Key
CtrlAltAst     EQUATE(066AH)             !Ctrl-Alt-Asterisk Key
CtrlAltBS        EQUATE(0608H)               !Ctrl-Alt-Backspace Key
CtrlAltDecimal EQUATE(066EH)             !Ctrl-Alt-Decimal Key
CtrlAltDelete  EQUATE(062EH)             !Ctrl-Alt-Delete Key
CtrlAltDivide  EQUATE(066FH)             !Ctrl-Alt-Divide Key
CtrlAltDown    EQUATE(0628H)             !Ctrl-Alt-Cursor Down
CtrlAltEnd     EQUATE(0623H)             !Ctrl-Alt-End Key
CtrlAltEnter   EQUATE(060DH)             !Ctrl-Alt-Enter Key
CtrlAltEsc     EQUATE(061BH)             !Ctrl-Alt-Esc Key
CtrlAltHome    EQUATE(0624H)             !Ctrl-Alt-Home Key
CtrlAltInsert  EQUATE(062DH)             !Ctrl-Alt-Insert Key
CtrlAltLeft    EQUATE(0625H)             !Ctrl-Alt-Cursor Left
CtrlAltMinus   EQUATE(066DH)             !Ctrl-Alt-Minus Key
CtrlAltPause   EQUATE(0613H)             !Ctrl-Alt-Pause Key
CtrlAltPgDn    EQUATE(0622H)             !Ctrl-Alt-PgDn Key
CtrlAltPgUp    EQUATE(0621H)             !Ctrl-Alt-PgUp Key
CtrlAltPlus    EQUATE(066BH)             !Ctrl-Alt-Plus Key
CtrlAltPrint   EQUATE(062CH)             !Ctrl-Alt-PrintScreen
CtrlAltRight   EQUATE(0627H)             !Ctrl-Alt-Cursor Right
CtrlAltSlash   EQUATE(066FH)             !Ctrl-Alt-Slash Key
CtrlAltSpace   EQUATE(0620H)             !Ctrl-Alt-Spacebar
CtrlAltTab     EQUATE(0609H)             !Ctrl-Alt-Tab Key
CtrlAltUp        EQUATE(0626H)               !Ctrl-Alt-Cursor Up Key
CtrlAltPad0    EQUATE(0660H)           !Ctrl-Alt-0 on numeric keypad
CtrlAltPad1    EQUATE(0661H)           !Ctrl-Alt-1 on numeric keypad
CtrlAltPad2    EQUATE(0662H)           !Ctrl-Alt-2 on numeric keypad
CtrlAltPad3    EQUATE(0663H)           !Ctrl-Alt-3 on numeric keypad
CtrlAltPad4    EQUATE(0664H)           !Ctrl-Alt-4 on numeric keypad
CtrlAltPad5    EQUATE(0665H)           !Ctrl-Alt-5 on numeric keypad
CtrlAltPad6    EQUATE(0666H)           !Ctrl-Alt-6 on numeric keypad
CtrlAltPad7    EQUATE(0667H)           !Ctrl-Alt-7 on numeric keypad
CtrlAltPad8    EQUATE(0668H)           !Ctrl-Alt-8 on numeric keypad
CtrlAltPad9    EQUATE(0669H)           !Ctrl-Alt-9 on numeric keypad
CtrlAltMouseLeft   EQUATE(0601H)       !Ctrl-Alt-Left mouse button
CtrlAltMouseRight   EQUATE(0602H)        !Ctrl-Alt-Right mouse button
CtrlAltMouseCenter EQUATE(0604H)       !Ctrl-Alt-Middle mouse button
```

Error Codes

```
!                   ERRORS.EQU  -- ERRORCODE EQUATES

!                   Return Value                Return Value
!                   From ERRORCODE()            From ERROR()

NoError             EQUATE(0)                   ! ''
NoFileErr           EQUATE(02)                  ! 'File Not Found'
NoPathErr           EQUATE(03)                  ! 'Path Not Found'
TooManyErr          EQUATE(04)                  ! 'Too Many Open Files'
NoAccessErr         EQUATE(05)                  ! 'Access Denied'
BadMemErr           EQUATE(07)                  ! 'Memory Corrupted'
NoMemErr            EQUATE(08)                  ! 'Insufficient Memory'
BadDriveErr         EQUATE(15)                  ! 'Invalid Drive'

NoEntryErr          EQUATE(30)                  ! 'Entry Not Found'
IsLockedErr         EQUATE(32)                  ! 'File Is Already Locked'
BadRecErr           EQUATE(33)                  ! 'Record Not Available'
NoRecErr            EQUATE(35)                  ! 'Record Not Found'
BadFileErr          EQUATE(36)                  ! 'Invalid Data File'
NotOpenErr          EQUATE(37)                  ! 'File Not Open'
DupKeyErr           EQUATE(40)                  ! 'Creates Duplicate Key'
IsHeldErr           EQUATE(43)                  ! 'Record Is Already Held'
BadNameErr          EQUATE(45)                  ! 'Invalid Filename'
BadKeyErr           EQUATE(46)                  ! 'Key Files must be Rebuilt'
InvalidFileErr      EQUATE(47)                  ! 'Invalid File Declaration'
BadTranErr          EQUATE(48)                  ! 'Unable to log transaction'
IsOpenErr           EQUATE(52)                  ! 'File Already Open'
NoCreateErr         EQUATE(54)                  ! 'No Create Attribute'
NoShareErr          EQUATE(55)                  ! 'File Must Be Shared'
BadMemoErr          EQUATE(57)                  ! 'Invalid Memo File'
ExclReqErr          EQUATE(63)                  ! 'Exclusive Access Required'
ShareVioErr         EQUATE(64)                  ! 'Sharing Violation'
CantRollErr         EQUATE(65)                  ! 'Unable to rollback transaction'
MemoMissing         EQUATE(73)                  ! 'Memo File is Missing'
TypeDescErr         EQUATE(75)                  ! 'Invalid Field Type Descriptor'
BadIndexErr         EQUATE(76)                  ! 'Invalid Index String'
IndexAccessErr      EQUATE(77)                  ! 'Unable To Access Index'
BadParmErr          EQUATE(78)                  ! 'Invalid Number Of Parameters'
NoDriverSupport     EQUATE(80)                  ! 'Function not supported'
```

Property Assignments

Data Structure Properties

The attributes (properties) of many of the APPLICATION, WINDOW, and REPORT data structures, and their component controls, are designed take constant values (not variables) as their parameters in the data structure declaration. The same is true of FILE, VIEW, and QUEUE data structures. This may seem to be a restriction, however, the values of these constant properties may be easily changed or determined using simple assignment statements containing property expressions.

Property expressions represent the attributes (properties) and the parameters of attributes declared in APPLICATION, WINDOW, REPORT, FILE, VIEW, and QUEUE structures, and their components. Most attributes have corresponding property expressions. However, some attributes (such as PRE, OVER, and THREAD) are actually compiler directives which have no associated property expression. In addition, there are some property expressions which are not associated with declared attributes (undeclared properties).

A property expression can be used as the destination of an assignment statement. This changes the value of the attribute (or attribute parameter) associated with the property. A property expression can also be used in any string expression to determine the current value of the attribute (or attribute parameter).

Built-in Variables

There are three built-in variables in the Clarion for Windows runtime library: TARGET, PRINTER, and SYSTEM. These are only used with the property assignment syntax to identify the target of a property assignment.

TARGET normally references the window that currently has focus. It can also be set to reference a window in another execution thread or the currently printing REPORT, enabling you to affect the properties of controls and windows in other execution threads and dynamically change report control properties while printing. The SETTARGET procedure is used to change the TARGET variable's reference.

PRINTER references the printer properties used by the next REPORT opened (and all subsequest reports). This is used only with the Printer Properties.

SYSTEM is a built-in variable that specifies global properties used by the the entire application. There are several specific undeclared properties that may use the SYSTEM variable to set or query global application-wide properties.

Property Expressions

[*target*] [**$**] [*control*] { *property* [,*element*] }

target	The label of an APPLICATION, WINDOW, REPORT, VIEW, or FILE structure, the label of a BLOB, or one of the built-in variables: TARGET, PRINTER, or SYSTEM. If omitted, TARGET is assumed.
$	Required separator when both *target* and *control* are specified. May be omitted if either *target* or *control* is omitted.
control	A field number or field equate label for the control in the *target* structure (APPLICATION, WINDOW, or REPORT) to affect. If omitted, the *target* must be specified. The *control* must be omitted if the *target* is a FILE structure, the label of a BLOB, or the PRINTER or SYSTEM built-in variables.
property	An integer constant, EQUATE, or variable that specifies the property (attribute) to change. It can also be a string when referencing a .VBX property.
element	An integer constant or variable that specifies which element to change (for those attributes that are arrays with multiple values).

This property expression syntax allows you access to all the attributes (properties) of APPLICATION, WINDOW, or REPORT structures, or any control within these structures. To specify an attribute of an APPLICATION, WINDOW, REPORT, VIEW, or FILE structure (not a component control), omit the *control* portion of the property expression. To specify a control in the current window, omit the *target* portion of the property expression.

REPORT data structures are never the default *target*. Therefore, either SETTARGET must be used to change the default *target*, or the structure's label must be explicitly specified as the *target* before you can change any property of the structure, or any control it contains.

Property expressions may be used in Clarion language statements anywhere a string expression is allowed, or as the destination of a simple assignment statement. Therefore, assigning a new value to a property is an assignment with the property as the destination and the new value as the source. Determining the current value of a property is an assignment where the property is the source and the variable to recieve its value is the destination.

All properties are treated as string data at runtime; the compiler automatically performs any necessary data type conversion. Any property without parameters is binary. Binary properties are either "present" or "missing" and returns a '1' if it is present, and '' (null) if it is missing. Changing the value of a binary property to '' (null), '0' (zero), or any non-numeric string sets it to missing. Changing it to any other value sets it to "present."

Most properties can be both examined (read) and changed (written). However, some properties are "read-only" and cannot be changed. Assigning a value to a "read-only" property has no effect at all. Other properties are "write-only" properties that are meaningless if read.

Some properties are arrays that contain multiple values. The syntax for addresssing a particular property array *element* uses a comma (not square brackets) as the delimiter between the *property* and the *element* number.

Example:

```
MainWin  APPLICATION('My Application'),SYSTEM,MAX,ICON('MyIcon.ICO'),STATUS,RESIZE
         MENUBAR
           MENU('File'),USE(?FileMenu)
             ITEM('Open...'),USE(?OpenFile)
             ITEM('Close'),USE(?CloseFile),DISABLE
             ITEM('E&xit'),USE(?MainExit)
```

```
          END
          MENU('Help'),USE(?HelpMenu)
            ITEM('Contents'),USE(?HelpContents),STD(STD:HelpIndex)
            ITEM('Search for Help On...'),USE(?HelpSearch),STD(STD:HelpSearch)
            ITEM('How to Use Help'),USE(?HelpOnHelp),STD(STD:HelpOnHelp)
            ITEM('About MyApp...'),USE(?HelpAbout)
          END
        END
        TOOLBAR
          BUTTON('Open'),USE(?OpenButton),ICON(ICON:Open)
        END
      END
CODE
OPEN(MainWin)
MainWin{PROP:text} = 'A New Title'        !Change window title
?OpenButton{PROP:icon} = ICON:Asterisk    !Change button icon
?OpenButton{PROP:at,1} = 5             !Change button x position
?OpenButton{PROP:at,2} = 5             !Change button y position
IF MainWin$?HelpContents{PROP:std} <> STD:HelpIndex
  MainWin$?HelpContents{PROP:std} = STD:HelpIndex
END
MainWin{PROP:maximize} = 1             !Expand to full screen
ACCEPT
  CASE ACCEPTED()                      !Which control was chosen?
  OF ?OpenFile                        !Open... menu selection
  OROF ?OpenButton                    !Open button on toolbar
    START(OpenFileProc)                 !Start new execution thread
  OF ?MainExit                        !Exit menu selection
  OROF ?MainExitButton                  !Exit button on toolbar
    BREAK                             !Break ACCEPT loop
  OF ?HelpAbout                       !About... menu selection
    HelpAboutProc                     !Call application information procedure
  END
END
CLOSE(MainWin)                        !Close APPLICATION
RETURN
```

See Also: SETTARGET

Attribute Property Equates

Equates for all properties are contained in the PROPERTY.CLW file. This file also contains equates for the standard values used by some of these properties. Some properties are "read-only" and their value may not be changed, and others are "write-only" properties whose value cannot be determined. These restrictions are noted for each control affected.

Each of the following properties references an attribute (or one of its parameters) of a window, report, or control. The referenced attribute is listed in the explanation and you should look up the attribute itself for further explanation of its effect on the window or control it modifies.

Some property descriptions state: (" if missing, else present), which means the attribute is either active for the window,report, or control, or it is not. Querying the property returns a blank string when the attribute is not active for the window, report, or control. Assigning a blank string (") to such an attribute turns it off, and assigning any other value turns it on.

PROP:Text	The *text* parameter of an APPLICATION(*text*), WINDOW(*text*), or control(*text*). This could contain any value that is valid as the parameter to a control declaration. For example, ?Image{PROP:Text} = 'My.BMP' displays a new bitmap in the referenced IMAGE control.
PROP:Type	Contains the type of control. Values are the CREATE:xxxx equates (listed in EQUATES.CLW). (READ-ONLY)

AT attribute properties:

PROP:At	AT attribute. An array (4 values).
PROP:Xpos	AT(x) parameter, equivalent to {PROP:At,1}
PROP:Ypos	AT(,y) parameter, equivalent to {PROP:At,2}
PROP:Width	AT(,,width) parameter, equivalent to {PROP:At,3}
PROP:Height	AT(,,,height) parameter, equivalent to {PROP:At,4}

FONT attribute properties:

PROP:Font	FONT attribute. An array (4 values).
PROP:FontName	FONT(fontname) parameter, equivalent to {PROP:Font,1}.
PROP:FontSize	FONT(,fontsize) parameter, equivalent to {PROP:Font,2}.
PROP:FontColor	FONT(,,fontcolor) parameter, equivalent to {PROP:Font,3}.
PROP:FontStyle	FONT(,,,fontstyle) parameter, equivalent to {PROP:Font,4}.

CLASS attribute properties:

PROP:Class	CLASS attribute. An array (2 values).
PROP:VbxFile	CLASS(vbxfile) parameter, equivalent to {PROP:Class,1}.
PROP:VbxName	CLASS(,vbxname) parameter, equivalent to {PROP:Class,2}.

All other attribute properties (in alphabetical order):

PROP:Absolute	ABSOLUTE attribute (" if missing, else present).
PROP:Alone	ALONE attribute (" if missing, else present).
PROP:Alrt	ALRT attribute. An array.
PROP:Auto	AUTO attribute (" if missing, else present).

PROP:Ave AVE attribute (" if missing, else present).

PROP:Boxed ABSOLUTE attribute (" if missing, else present).

PROP:Cap ABSOLUTE attribute (" if missing, else present).

PROP:Center CENTER attribute (" if missing, else present).

PROP:CenterOffset
CENTER(offset) parameter, equivalent to {PROP:Center,2}.

PROP:Check CHECK attribute, (" if missing, else present).

PROP:Cnt CNT attribute (" if missing, else present).

PROP:Color COLOR attribute (COLOR:none if none).

PROP:Column COLUMN attribute (0 = off, else currently highlighted column number).

PROP:Cursor CURSOR attribute (" if missing, else present).

PROP:Decimal DECIMAL attribute (" if missing, else present).

PROP:DecimalOffset
DECIMAL(offset) parameter, equivalent to {PROP:Decimal,2}.

PROP:Default DEFAULT attribute (" if missing, else present).

PROP:Disable DISABLE attribute (" if missing, else present).

PROP:Double DOUBLE attribute (" if missing, else present).

PROP:Dragid DRAGID attribute. An array.

PROP:Drop DROP attribute (0 if none). You may not change this to or from zero (0).

PROP:Dropid DROPID attribute. An array.

PROP:Fill FILL attribute (COLOR:none if none).

PROP:First FIRST attribute (" if missing, else present).

PROP:Format FORMAT attribute (" if missing, else present). This property is updated whenever the user changes the format of the LIST at runtime.

PROP:From FROM attribute (queue, queue field, or string). (WRITE-ONLY)

PROP:Full FULL attribute (" if missing, else present).

PROP:Gray GRAY attribute (" if missing, else present).

PROP:Hide HIDE attribute (" if missing, else present).

PROP:Hlp HLP attribute (blank if none).

PROP:Hscroll HSCROLL attribute (" if missing, else present).

PROP:Icon ICON attribute (blank if none).

PROP:Iconize ICONIZE attribute (" if missing, else present).

PROP:Imm IMM attribute (" if missing, else present).

PROP:Ins INS attribute (" if missing, else present).

PROP:Key KEY attribute (blank if none).

PROP:Landscape LANDSCAPE attribute, (" if missing, else present).

PROP:Last LAST attribute (" if missing, else present).

PROP:Left LEFT attribute (" if missing, else present).

PROP:LeftOffset LEFT(offset) parameter, equivalent to {PROP:Left,2}.

PROP:Mark MARK attribute (queue or queue field). (WRITE-ONLY)

PROP:Mask MASK attribute (" if missing, else present).

PROP:Max MAX attribute (" if missing, else present).

PROP:Maximize MAXIMIZE attribute (" if missing, else present).

PROP:Mdi MDI attribute (" if missing, else present). (READ-ONLY)

PROP:Meta META attribute (" if missing, else present).

PROP:Min MIN attribute (" if missing, else present).

PROP:Mm MM attribute (" if missing, else present).

PROP:Modal MODAL attribute (" if missing, else present). (READ-ONLY)

PROP:Msg MSG attribute (" if missing, else present).

PROP:NoBar NOBAR attribute (" if missing, else present).

PROP:NoFrame NOFRAME attribute (" if missing, else present).

PROP:NoMerge NOMERGE attribute (" if missing, else present).

PROP:Ovr OVR attribute (" if missing, else present).

PROP:Page PAGE attribute (" if missing, else present).

PROP:PageAfter PAGEAFTER attribute (" if missing, else present).

PROP:PageAfterNum
PAGEAFTER(pageafternum) parameter, equivalent to {PROP:PageAfter,2}.

PROP:PageBefore PAGEBEFORE attribute (" if missing, else present).

PROP:PageBeforeNum
PAGEBEFORE(pagebeforenum) parameter, equivalent to {PROP:PageBefore,2}.

PROP:Pageno PAGENO attribute (" if missing, else present).

PROP:Palette PALETTE attribute. Single value.

PROP:Password PASSWORD attribute (" if missing, else present).

PROP:Points POINTS attribute (" if missing, else present).

PROP:Preview PREVIEW attribute (queue or queue field). (WRITE-ONLY)

PROP:Range RANGE attribute. An array (2 values).

PROP:RangeHigh RANGE(,rangehigh) parameter, equivalent to {PROP:Range,2}.

PROP:RangeLow RANGE(rangelow) parameter, equivalent to {PROP:Range,1}.

PROP:ReadOnly READONLY attribute (" if missing, else present).

PROP:Req REQ attribute (" if missing, else present).

PROP:Reset RESET attribute (0 = off, else breaklevel nesting depth).

PROP:Resize RESIZE attribute (" if missing, else present).

PROP:Right RIGHT attribute (" if missing, else present).

PROP:RightOffset RIGHT(offset) parameter, equivalent to {PROP:Right,2}.

PROP:Round ROUND attribute (" if missing, else present).

PROP:Scroll SCROLL attribute (" if missing, else present).

PROP:Separate SEPARATE attribute (" if missing, else present).

PROP:Skip SKIP attribute (" if missing, else present).

PROP:Spread SPREAD attribute (" if missing, else present).

PROP:Status STATUS attribute. An array (0 terminates).

PROP:StatusText STATUS bar text. An array (0 terminates).

PROP:Std STD attribute (" if missing, else present).

PROP:Step STEP attribute (" if missing, else present).

PROP:Sum SUM attribute (" if missing, else present).

PROP:System SYSTEM attribute (" if missing, else present).

PROP:Thous THOUS attribute (" if missing, else present).

PROP:Timer TIMER attribute (0 if none).

PROP:Toolbox TOOLBOX attribute (" if missing, else present).

PROP:ToolTip TIP attribute (" if missing, else present).

PROP:Trn TRN attribute, (" if missing, else present).

PROP:Upr UPR attribute (" if missing, else present).

PROP:Use USE attribute (variable name). Writing to it changes the USE variable. Reading it returns the contents of the USE variable.

PROP:Value VALUE attribute (" if missing, else present).

PROP:Vcr VCR attribute (" if missing, else present).

PROP:VcrFeq VCR(vcrfeq) parameter, equivalent to {PROP:Vcr,2}.

PROP:Vscroll VSCROLL attribute (" if missing, else present).

PROP:WithNext WITHNEXT attribute (0 if none).

PROP:WithPrior WITHPRIOR attribute (0 if none).

PROP:Wizard WIZARD attribute (" if missing, else present).

Example:

```
Screen WINDOW,PRE(Scr)
         ENTRY(@N3),USE(Ctl:Code)
         ENTRY(@S30),USE(Ctl:Name),REQ
         IMAGE('SomePic.BMP'),USE(?Image)
         BUTTON('OK'),USE(?OkButton),KEY(EnterKey)
         BUTTON('Cancel'),USE(?CanxButton),KEY(EscKey)
       END
  CODE
  OPEN(Screen)
  Screen{PROP:At,1} = 0             !Position window to top left corner
  Screen{PROP:At,2} = 0
```

```
Screen{PROP:Gray} = 1             !Give window 3D look
Screen{PROP:Status,1} = -1        !Create status bar with two sections
Screen{PROP:Status,2} = 180
Screen{PROP:Status,3} = 0          !Terminate staus bar array
Screen{PROP:StatusText,2} = FORMAT(TODAY(),@D2)
                             !Put date in status bar section 2
?CtlCode{PROP:Alrt,1} = F10Key     !Alert F10 on Ctl:Code entry control
?CtlCode{PROP:Text} = '@N4'        !Change entry picture token
?Image{PROP:Text} = 'MyPic.BMP'    !Change image control filename
?OkButton{PROP:Default} = '1'      !Put DEFAULT attribute on OK button
ACCEPT
END
```

List Box Format String Properties

The properties of individual fields in a multi-column LIST or COMBO control can also be set using property equates.Each of these properties relates to one element of the FORMAT attribute's string parameter. These properties eliminate the need to create a complete FORMAT attribute string just to change a single property of a single field in the LIST.

These are all property arrays that require an explicit array element number following the property equate (separated by a comma) to specify which field in the LIST or COMBO is affected.

PROPLIST:Center
: The **C** that indicates center justification, (blank if missing, 1 if present).

PROPLIST:CenterOffset
: An integer that specifes the indent, (blank if missing, 1 if present).

PROPLIST:Color
: The ***** (asterisk) that indicates color information for the field is contained in four LONG fields that immediately follow the data field in the QUEUE (or FROM attribute string), (blank if missing, 1 if present).

PROPLIST:Decimal
: The **D** that indicates decimal justification, (blank if missing, 1 if present).

PROPLIST:DecimalOffset
: An integer that specifes the indent, (blank if missing, 1 if present).

PROPLIST:Fixed
: The **F** that specifies the field remains fixed at left edge of the list, (blank if missing, 1 if present).

PROPLIST:Header
: The ~*header*~ text for the field or group, (blank if missing, 1 if present).

PROPLIST:HeaderCenter
: The **C** that indicates center header justification, (blank if missing, 1 if present).

PROPLIST:HeaderCenterOffset
: An integer that specifes the indent, (blank if missing, 1 if present).

PROPLIST:HeaderDecimal
: The **D** that indicates decimal header justification, (blank if missing, 1 if present).

PROPLIST:HeaderDecimalOffset
: An integer that specifes the indent, (blank if missing, 1 if present).

PROPLIST:HeaderLeft
: The **L** that indicates left header justification, (blank if missing, 1 if present).

PROPLIST:HeaderLeftOffset
: An integer that specifes the indent, (blank if missing, 1 if present).

PROPLIST:HeaderRight
: The **R** that indicates right header justification, (blank if missing, 1 if present).

PROPLIST:HeaderRightOffset
: An integer that specifes the indent, (blank if missing, 1 if present).

PROPLIST:Icon
: The **I** that indicates an icon number for the field is contained in a LONG field that immediately follows the data field in the QUEUE (or FROM attribute string), (blank if missing, 1 if present).

PROPLIST:LastOnLine
The / (slash) that indicates the next field in the group appears on the next line, (blank if missing, 1 if present).

PROPLIST:Left The **L** that indicates left justification, (blank if missing, 1 if present).

PROPLIST:LeftOffset
An integer that specifes the indent, (blank if missing, 1 if present).

PROPLIST:Locator
The **?** (question mark) that specifies the field for a locator, (blank if missing, 1 if present).

PROPLIST:Picture
The *@picture@* display format for the field, (blank if missing, 1 if present).

PROPLIST:Resize
The **M** that allows the user to resize the field or group, (blank if missing, 1 if present).

PROPLIST:RightBorder
The | (vertical bar) that places a right border on the field or group, (blank if missing, 1 if present).

PROPLIST:Right The **R** that indicates right justification, (blank if missing, 1 if present).

PROPLIST:RightOffset
An integer that specifes the indent, (blank if missing, 1 if present).

PROPLIST:Scroll The **S(***integer***)** that puts a scroll bar on the field or group. Specifies the *integer* portion, (blank if missing, 1 if present).

PROPLIST:Tree The **T** that indicates the LIST is a tree control, (blank if missing, 1 if present).

PROPLIST:TreeLines
The **T(L)** that indicates the tree control suppresses the conecting lines between levels, (blank if missing, 1 if present).

PROPLIST:TreeBoxes
The **T(B)** that indicates the tree control suppresses the expansion boxes, (blank if missing, 1 if present).

PROPLIST:TreeIndent
The **T(I)** that indicates the tree control suppresses level indentation (which also implicitly suppresses both lines and boxes), (blank if missing, 1 if present).

PROPLIST:Underline
The _ (underscore) that underlines the field or group, (blank if missing, 1 if present).

PROPLIST:Width The integer that specifies the width of the field or group.

Any of these properties can also apply to a field group by adding PROPLIST:Group to the property.

PROPLIST:Group Add this property to the PROPLIST field property to affect field group properties.

Example:

```
?List{PROPLIST:Header,1} = 'First Field'         !Change first field's header text
?List{PROPLIST:Header + PROPLIST:Group,1} = 'First Group'
                                                !Change first group's header text
```

See Also: FORMAT

Other Properties

List Box Mouse Click Properties

The following properties return the mouse position within the LIST or COMBO control when pressed or released.They can also be written to, which has no effect except to temporarily change the value that the property returns when next read (within the same ACCEPT loop iteration). This may make coding easier in some circumstances.

PROPLIST:MouseDownField
Returns the field number when the mouse is pressed.

PROPLIST:MouseDownRow
Returns the row number when the mouse is pressed.

PROPLIST:MouseDownZone
Returns the zone number when the mouse is pressed.

PROPLIST:MouseMoveField
Returns the field number when the mouse is moved.

PROPLIST:MouseMoveRow
Returns the row number when the mouse is moved.

PROPLIST:MouseMoveZone
Returns the zone number when the mouse is moved.

PROPLIST:MouseUpField
Returns the field number when the mouse is released.

PROPLIST:MouseUpRow
Returns the row number when the mouse is released.

PROPLIST:MouseUpZone
Returns the zone number when the mouse is released.

The three "Row" properties all return -1 for header text and -2 if below the last displayed item. Equates for the following Zones are listed on EQUATES.CLW:

LISTZONE:Field	On a field in the LIST
LISTZONE:Right	On the field's right border resize zone
LISTZONE:Header	On a field or group header
LISTZONE:ExpandBox	On an expand box in a Tree
LISTZONE:Tree	On the connecting lines of a Tree
LISTZONE:Icon	On an icon (Tree or not)
LISTZONE:Nowhere	Anywhere else

Example:

```
Que        QUEUE
F1           STRING(50)
F2           STRING(50)
F3           STRING(50)
           END
WinView      WINDOW('View'),AT(,,340,200),SYSTEM,CENTER,ALRT(MouseLeft)
             LIST,AT(20,0,300,200),USE(?List),FROM(Que),IMM,HVSCROLL  |
                 FORMAT('80L~F1~80L~F2~80L~F3~'),IMM
           END
  CODE
  OPEN(WinView)
  DO BuildListQue
  X# = 0
  ACCEPT
```

```
  CASE EVENT()
  OF EVENT:AlertKey
    IF ?List{PROPLIST:MouseUpRow} = -1           !Check for click in header
      CASE ?List{PROPLIST:MouseDownField} + X#     !Check which header
      OF 1
        SORT(Que,Que:F1)
        ?List{PROP:Format} = '80L~F1~#1#80L~F2~#2#80L~F3~#3#'
        X# = 0
      OF 2
        SORT(Que,Que:F2)
        ?List{PROP:Format} = '80L~F2~#2#80L~F3~#3#80L~F1~#1#'
        X# = 1
      OF 3
        SORT(Que,Que:F3)
        ?List{PROP:Format} = '80L~F3~#3#80L~F1~#1#80L~F2~#2#'
        X# = 2
      END
      DISPLAY
. . .
FREE(Que)
```

Undeclared Properties

The following properties can only be accessed at runtime, and do not relate directly to data structure and control field attributes:

PROP:AcceptAll Returns one (1) if AcceptAll mode is active and zero (0) if it is not, and may also be used to toggle AcceptAll (non-stop) mode. SELECT with no parameters usually initiates AcceptAll mode. This is a field edit mode in which each control in the window is processed in TAB key sequence by generating EVENT:Accepted for each. This allows data entry validation code to execute for all controls, including those that the user has not touched.

AcceptAll mode immediately terminates when any of the following conditions is met:

SELECT(?)
Window{PROP:AcceptAll} = 0
A REQ control is left blank or zero.

The SELECT(?) statement selects the same control for the user to edit. This code usually indicates the value it contains is invalid and the user must re-enter data. The Window{PROP:AcceptAll} = 0 statement toggles AcceptAll mode off. Assigning values to this property can be used to initiate and terminate AcceptAll mode. When a control with the REQ attribute is left blank or zero, AcceptAll mode terminates with the control highlighted for user entry, without processing any more fields in the TAB key sequence.

When all controls have been successfully processed, EVENT:Completed is posted to the window.

Example:

```
Screen WINDOW,PRE(Scr)
         ENTRY(@N3),USE(Ctl:Code)
         ENTRY(@S30),USE(Ctl:Name),REQ
         BUTTON('OK'),USE(?OkButton),KEY(EnterKey)
         BUTTON('Cancel'),USE(?CanxButton),KEY(EscKey)
       END
  CODE
  OPEN(Screen)
  ACCEPT
    IF EVENT() = EVENT:Completed THEN BREAK.    !AcceptAll mode terminated
    CASE ACCEPTED()
    OF ?Ctl:Code
      IF Ctl:Code > 150                   !If data entered is invalid
        BEEP                            ! alert the user and
        SELECT(?)                         ! make them re-enter the data
      END
    OF ?OkButton
      Screen{PROP:AcceptAll} = 1          !Initiate AcceptAll mode
  . .                                   !Terminate ACCEPT and CASE ACCEPTED
```

PROP:Active Returns 1 if the window is the active window, blank if not. Set to 1 to make the top window of a thread the active window.

Example:

```
  CODE
  OPEN(ThisWindow)
  X# = START(AnotherThread)             !Start another thread
  ACCEPT
    CASE EVENT()
```

```
   OF EVENT:LoseFocus                !When this window is losing focus
     IF Y# <> X#                     ! check for the first focus change
       ThisWindpw{PROP:Active} = 1       ! and return focus to this thread
       Y# = X#                       ! then flag first focus change completed
  . . .
```

PROP:AppInstance

Returns the instance handle (HInstance) of the .EXE file for use in low-level API calls which require it. This is only used with the SYSTEM built-in variable. (READ-ONLY)

Example:

```
  PROGRAM
HInstance   LONG
  CODE
  OPEN(AppFrame)
  HInstance = SYSTEM{PROP:AppInstance}   !Get .EXE instance handle for later use
  ACCEPT
  END
```

PROP:ChoiceFeq Returns or sets the field number of the currently selected TAB in a SHEET, or RADIO in an OPTION structure.

Example:

```
WinView  WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
         OPTION('Option 1'),USE(OptVar1),KEY(F10Key),HLP('Option1Help')
           RADIO('Radio 1'),AT(0,0,20,20),USE(?R1)
           RADIO('Radio 2'),AT(20,0,20,20),USE(?R2)
         END
        END

  CODE
  OPEN(WinView)
  ?OptVar1{PROP:ChoiceFeq} = ?R1    !Select radio one
  ACCEPT
  END
```

PROP:ClientHandle

Returns the client window handle (the area of the window that contains the controls) for use with low-level Windows API calls that require it. (READ-ONLY)

Example:

```
WinView     WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL
           END
MessageText    CSTRING('You cannot exit the program from this window ')
MessageCaption CSTRING('No EVENT:CloseDown Allowed ')
TextAddr      LONG
CaptionAddr     LONG
RetVal        SHORT
  CODE
  OPEN(WinView)
  ACCEPT
    CASE EVENT()
    OF EVENT:CloseDown
      TextAddress = ADDRESS(MessageText)
      CaptionAddress = ADDRESS(MessageCaption)
      RetVal = MessageBox(WinView{PROP:ClientHandle},TextAddr,CaptionAddr,MB_OK)
                             !Windows API call using a window handle
```

```
    CYCLE                       !Disallow program closedown from this window
   END
  END
```

PROP:ClientWndProc

Sets or gets the client window messaging procedure for use with low-level Windows API calls that require it. Generally used with sub-classing to track all Windows messages.

Example:

```
   PROGRAM
   MAP
     main
     SubClassFunc(USHORT,SHORT,SHORT,LONG),LONG,PASCAL
     MODULE('Windows')
   !TopSpeed Win31 Library
      CallWindowProc(LONG,USHORT,SHORT,SHORT,LONG),LONG,PASCAL
     END
   END
SavedProc   LONG
PT         GROUP,PRE(PT)
X            SHORT
Y            SHORT
           END
       CODE
  Main

Main             PROCEDURE
WinView         WINDOW('View'),AT(0,0,320,200),HVSCROLL,MAX,TIMER(1)
            STRING('X Pos'),AT(1,1,,),USE(?String1)
            STRING(@n3),AT(24,1,,),USE(PT:X)
            STRING('Y Pos'),AT(44,1,,),USE(?String2)
            STRING(@n3),AT(68,1,,),USE(PT:Y)
            BUTTON('Close'),AT(240,180,60,20),USE(?Close)
           END
  CODE
  OPEN(WinView)
  SavedProc = WinView{PROP:ClientWndProc}           !Save this procedure
  WinView{PROP:ClientWndProc} = ADDRESS(SubClassFunc)
                                                    !Change to subclass procedure
  ACCEPT
    CASE ACCEPTED()
    OF ?Close
      BREAK
    END
  END

SubClassFunc    FUNCTION(hWnd,wMsg,wParam,lParam)     !Sub class procedure
WM_MOUSEMOVE    EQUATE(0200H)                       ! to track mouse movement in
  CODE                                              ! client area of window
  CASE wMsg
  OF  WM_MOUSEMOVE
    PT:X = MOUSEX()
    PT:Y = MOUSEY()
  END
  RETURN(CallWindowProc(SavedProc,hWnd,wMsg,wParam,lParam))
                                                    !Pass control back to
                                                    ! saved procedure
```

PROP:ClipBits Property of an IMAGE control that allows bitmap images to be moved into (but not out of) the Windows clipboard when set to one (1). Any displayable image type can be stored as a bitmap (.BMP) image in the Clipboard.

Example:

```
WinView     WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL
            IMAGE(),AT(0,0,,),USE(?Image)
            BUTTON('Save Picture'),AT(80,180,60,20),USE(?SavePic)
            BUTTON('New Picture'),AT(160,180,60,20),USE(?NewPic)
          END

FileName  STRING(64)                    !Filename variable

  CODE
  OPEN(WinView)
  DISABLE(?LastPic)
  IF NOT FILEDIALOG('Choose File to View',FileName,'BitMap|*.BMP|PCX|*.PCX',0)
    RETURN                                !Return if no file chosen
  END
  ?Image{PROP:Text} = FileName
  ACCEPT
    CASE ACCEPTED()
    OF ?NewPic
      IF NOT FILEDIALOG('Choose File to View',FileName,'BitMap|*.BMP|PCX|*.PCX',0)
        BREAK                             !Return if no file chosen
      END
      ?Image{PROP:Text} = FileName
    OF ?SavePic
      ?Image{PROP:ClipBits} = 1              !Put image into Clipboard
      ENABLE(?LastPic)                      ! activate Last Picture button
    END
  END
```

PROP:DeferMove A property of the SYSTEM built-in variable that defers the resizing and/or movement of controls until the end of the ACCEPT loop or SYSTEM{PROP:DeferMove} is reset to zero (0). This disables the immediate effect of all assignments to position and size properties, and enables the library to perform all the moves at once (eliminating possible temporarily overlapping controls).

The absolute value of the number assigned to SYSTEM{PROP:DeferMove} defines the number of deferred moves for which space is pre-allocated (automatically expanded when necessary, but less efficient and may fail). Assigning a positive number automatically resets it to zero at the next ACCEPT, while a negative number leaves it set until explicitly reset to zero (0).

Example:

```
WinView     WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL
            IMAGE(),AT(0,0,,),USE(?Image)
            BUTTON('New Picture'),AT(160,180,60,20),USE(?NewPic)
            BUTTON('Close'),AT(80,180,60,20),USE(?Close)
          END
FileName  STRING(64)                    !Filename variable
ImageWidth  SHORT
ImageHeight SHORT
  CODE
  OPEN(WinView)
  DISABLE(?LastPic)
```

```
  IF NOT FILEDIALOG('Choose File to View',FileName,'BitMap|*.BMP|PCX|*.PCX',0)
    RETURN                                  !Return if no file chosen
  END
  ?Image{PROP:Text} = FileName
  ACCEPT
    CASE ACCEPTED()
    OF ?NewPic
      IF NOT FILEDIALOG('Choose File to View',FileName,'BitMap|*.BMP|PCX|*.PCX',0)
        BREAK                                          !Return if no file chosen
      END
      ?Image{PROP:Text} = FileName
      SYSTEM{PROP:DeferMove} = 4                        !Defer move and resize
      ImageWidth = ?Image{PROP:Width}
      ImageHeight = ?Image{PROP:Height}
      IF ImageWidth > 320
        ?Image{PROP:Width} = 320
        ?Image{PROP:XPos} = 0
      ELSE
        ?Image{PROP:XPos} = (320 - ImageWidth) / 2     !Center horizontally
      END
      IF ImageHeight > 180
        ?Image{PROP:Height} = 180
        ?Image{PROP:YPos} = 0
      ELSE
        ?Image{PROP:YPos} = (180 - ImageHeight) / 2    !Center vertically
      END
    OF ?Close
      BREAK
  . .                       !Moves and resizing happen at end of ACCEPT loop
```

PROP:Edit Specifies the field equate label of the control to perform edit-in-place for a LIST box column. This is an array whose element number indicates the column number to edit. When non-zero, the control is unhidden and moved/resized over the current row in the column indicated to allow the user to input data. Assign zero to re-hide the data entry control.

Example:

```
Q    QUEUE
f1     STRING(15)
f2     STRING(15)
         END
Win1 WINDOW('List Edit In Place'),AT(0,1,308,172),SYSTEM
       LIST,AT(6,6,120,90),USE(?List),COLUMN,FORMAT('60L@s15@60L@s15@'),FROM(Q),IMM
         END
?EditEntry  EQUATE(100)
  CODE
  OPEN(Win1)
  CREATE(?EditEntry,CREATE:Entry)
  ACCEPT
    CASE FIELD()
    OF ?List
      CASE EVENT()
      OF EVENT:NewSelection
        IF ?List{PROP:edit,?List{PROP:column}}
          GET(Q,CHOICE())
        END
      OF EVENT:Accepted
        IF KEYCODE() = MouseLeft2
          GET(Q,CHOICE())
```

```
        ?EditEntry{PROP:text} = ?List{PROPLIST:picture,?List{PROP:column}}
        CASE ?List{PROP:column}
        OF 1
          ?EditEntry{PROP:use} = F1
        OF 2
          ?EditEntry{PROP:use} = F2
        END
        ?List{PROP:edit,?List{PROP:column}} = ?EditEntry
    . .
  OF ?EditEntry
    CASE EVENT()
    OF EVENT:Selected
      ?EditEntry{PROP:Touched} = 1
    OF EVENT:Accepted
      PUT(Q)
      ?List{PROP:edit,?List{PROP:column}} = 0
. . .
```

PROP:Enabled Returns an empty string if the control is not enabled either because it itself has been disabled, or because it is a member of a "parent" control (OPTION, GROUP, MENU, SHEET, or TAB) that has been disabled. (READ-ONLY)

Example:

```
MDIChild  WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
          SHEET,AT(0,0,320,175),USE(SelectedTab)
            TAB('Tab One'),USE(?TabOne)
              PROMPT('Enter Data:'),AT(100,100,20,20),USE(?P1)
              ENTRY(@S8),AT(100,140,32,20),USE(E1)
              PROMPT('Enter More Data:'),AT(100,200,20,20),USE(?P2)
              ENTRY(@S8),AT(100,240,32,20),USE(E2)
            END
            TAB('Tab Two'),USE(?TabTwo)
              PROMPT('Enter Data:'),AT(100,100,20,20),USE(?P3)
              ENTRY(@S8),AT(100,140,32,20),USE(E3)
              PROMPT('Enter More Data:'),AT(100,200,20,20),USE(?P4)
              ENTRY(@S8),AT(100,240,32,20),USE(E4)
            END
          END
          BUTTON('Ok'),AT(100,180,20,20),USE(?Ok)
          BUTTON('Cancel'),AT(200,180,20,20),USE(?Cancel)
        END
  CODE
  OPEN(MDIChild)
  ACCEPT
    CASE EVENT()
    OF EVENT:Completed
      BREAK
    END
    CASE FIELD()
    OF ?Ok
      CASE EVENT()
      OF EVENT:Accepted
        SELECT
      END
    OF ?E3
      CASE EVENT()
      OF EVENT:Accepted
        IF ?E3{PROP:Enabled} AND MDIChild{PROP:AcceptAll}
                                  !Check for visibility during AcceptAll mode
```

```
        E3 = UPPER(E3)              !Convert the data entered to Upper case
        DISPLAY(?E3)                ! and display the upper cased data
      END
    END
  OF ?Cancel
    CASE EVENT()
    OF EVENT:Accepted
      BREAK
    END
  END
END
```

PROP:Filter Sets the FILTER attribute of a VIEW structure.

Example:

```
BRW1::View:Browse  VIEW(Members)
                PROJECT(Mem:MemberCode,Mem:LastName,Mem:FirstName)
              END
KeyValue STRING(20)

  CODE
  KeyValue = 'Smith'
  BIND('KeyValue',KeyValue)
  BIND('Mem:LastName',Mem:LastName)
  Mem:LastName = KeyValue
  SET(Mem:LastNameKey,Mem:LastNameKey)
  BRW1::View:Browse{PROP:Filter} = 'Mem:LastName = KeyValue'
  OPEN(BRW1::View:Browse)
```

PROP:FlushPreview

Flushes the REPORT structure's PREVIEW attribute metafiles to the printer (0 = off, else on, always 0 at report open).

Example:

```
SomeReport  PROCEDURE

WMFQue     QUEUE                  !Queue to contain .WMF filenames
             STRING(64)
           END

NextEntry  BYTE(1)                !Queue entry counter variable

Report   REPORT,PREVIEW(WMFQue)    !Report with PREVIEW attribute
DetailOne   DETAIL
             !Report controls
           END
         END

ViewReport  WINDOW('View Report'),AT(0,0,320,200),MDI,MAX,HVSCROLL
             IMAGE(),AT(0,0,320,180),USE(?ImageField)
             BUTTON('View Next Page'),AT(0,180,60,20),USE(?NextPage),DEFAULT
             BUTTON('Print Report'),AT(80,180,60,20),USE(?PrintReport)
             BUTTON('Exit Without Printing'),AT(160,180,60,20),USE(?ExitReport)
           END

  CODE
  OPEN(Report)
  SET(SomeFile)                          !Code to generate the report
```

```
  LOOP
    NEXT(SomeFile)
      IF ERRORCODE() THEN BREAK.
    PRINT(DetailOne)
  END
  ENDPAGE(Report)
  OPEN(ViewReport)                        !Open report preview window
  GET(WMFQue,NextEntry)                    !Get first queue entry
  ?ImageField{PROP:text} = WMFQue           !Load first report page
  ACCEPT
    CASE ACCEPTED()
    OF ?NextPage
      NextEntry += 1                      !Increment entry counter
      IF NextEntry > RECORDS(WMFQue) THEN CYCLE. !Check for end of report
      GET(WMFQue,NextEntry)                !Get next queue entry
      ?ImageField{PROP:text} = WMFQue         !Load next report page
      DISPLAY                             ! and display it
    OF ?PrintReport
      Report{PROP:FlushPreview} = 1          !Flush files to printer
      BREAK                               ! and exit procedure
    OF ?ExitReport
      BREAK                               !Exit procedure
    END
  END
  RETURN                                 !Return to caller, automatically
                                         ! closing the window and report
                                         ! freeing the queue and automatically
                                         ! deleting all the temporary .WMF files
```

PROP:Follows Changes the tab order to specify the position within the parent that the control will occupy. The control follows the control number you specify in the tab order. This must specify an existing control within the parent (window, option, group). (WRITE-ONLY)

Example:

```
WinView     WINDOW('View Report'),AT(0,0,320,200),MDI,MAX,HVSCROLL
            BUTTON('View Next Page'),AT(0,180,60,20),USE(?NextPage),DEFAULT
            BUTTON('Print Report'),AT(80,180,60,20),USE(?PrintReport)
            BUTTON('Exit Without Printing'),AT(160,180,60,20),USE(?ExitReport)
          END
  CODE
  OPEN(WinView)
                   !Print Report button normally follows View button
  ?PrintReport{PROP:Follows} = ?ExitReport
                   !Now Print Report button follows Exit button in the tab order
  ACCEPT
  END
```

PROP:Handle Returns the window or control handle for use with low-level Windows API calls that require it.

Example:

```
WinView     WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL
          END
MessageText    CSTRING('You cannot exit the program from this window ')
MessageCaption CSTRING('No EVENT:CloseDown Allowed ')
TextAddress    LONG
CaptionAddress LONG
```

```
RetVal        SHORT
  CODE
  OPEN(WinView)
  ACCEPT
    CASE EVENT()
    OF EVENT:CloseDown
      TextAddress = ADDRESS(MessageText)
      CaptionAddress = ADDRESS(MessageCaption)
      RetVal = MessageBox(WinView{PROP:Handle},TextAddress,CaptionAddress,MB_OK)
                              !Windows API call using a window handle
      CYCLE                    !Disallow program closedown from this window
    END
  END
```

PROP:HscrollPos Returns the position of the horizontal scroll bar's "thumb" (from 0 to 255) on a window, IMAGE, TEXT, LIST or COMBO with the HSCROLL attribute. Setting this property causes the control or window's contents to scroll horizontally.

Example:

```
Que        QUEUE
F1           STRING(50)
F2           STRING(50)
F3           STRING(50)
           END
WinView      WINDOW('View'),AT(,,340,200),SYSTEM,CENTER
             LIST,AT(20,0,300,200),USE(?List),FROM(Que),IMM,HVSCROLL  |
                FORMAT('80L#1#80L#2#80L#3#')
           END
  CODE
  OPEN(WinView)
  DO BuildListQue
  ACCEPT
    CASE FIELD()
    OF ?List
      CASE EVENT()
      OF EVENT:ScrollDrag
        CASE ?List{PROP:HscrollPos} % 200) + 1
        OF 1
          ?List{PROP:Format} = '80L#1#80L#2#80L#3#'
        OF 2
          ?List{PROP:Format} = '80L#2#80L#3#80L#1#'
        OF 3
          ?List{PROP:Format} = '80L#3#80L#1#80L#2#'
        END
        DISPLAY
  . . .
  FREE(Que)

BuildListQue  ROUTINE
  LOOP 15 TIMES
    Que:F1 = 'F1F1F1F1'
    Que:F2 = 'F2F2F2F2'
    Que:F3 = 'F3F3F3F3'
    ADD(Que)
  END
```

PROP:IconList An array that sets the icons displayed in a LIST formatted to display icons

(usually a tree control).

Example:

```
     PROGRAM
     MAP
       RandomAlphaData(*STRING)
     END

TreeDemo     QUEUE,PRE()          !Data list box FROM queue
FName           STRING(20)
ColorNFG       LONG            !Normal Foreground color for FName
ColorNBG       LONG            !Normal Background color for FName
ColorSFG       LONG            !Selected Foreground color for FName
ColorSBG       LONG            !Selected Background color for FName
IconField        LONG            !Icon number for FName
TreeLeve       LONG            !Tree Level
LName            STRING(20)
Init           STRING(4)
               END

Win WINDOW('List Boxes'),AT(0,0,366,181),SYSTEM,DOUBLE
     LIST,AT(0,34,366,146),FROM(TreeDemo),USE(?Show),HVSCROLL,   |
          FORMAT('80L*IT~First Name~*80L~Last Name~16C~Initials~')
   END

  CODE
  LOOP 20 TIMES
    RandomAlphaData(FName)
    ColorNFG = COLOR:White        !Assign FNAME's colors
    ColorNBG = COLOR:Maroon
    ColorSFG = COLOR:Yellow
    ColorSBG = COLOR:Blue
    IconField = ((x#-1) % 4) + 1  !Assign icon number
    TreeLevel = ((x#-1) % 4) + 1  !Assign tree level
    RandomAlphaData(LName)
    RandomAlphaData(Init)
    ADD(TD)
  END
  OPEN(Win)
  ?Show{PROP:iconlist,1} = ICON:VCRback          !Icon 1 = <
  ?Show{PROP:iconlist,2} = ICON:VCRrewind        !Icon 2 = <<
  ?Show{PROP:iconlist,3} = ICON:VCRplay          !Icon 3 = >
  ?Show{PROP:iconlist,4} = ICON:VCRfastforward   !Icon 4 = >>
  ACCEPT
  END

RandomAlphaData PROCEDURE(Field)       !MAP Prototype is: RandomAlphaData(*STRING)
  CODE
  y# = RANDOM(1,SIZE(Field))         !Random fill size
  LOOP x# = 1 to y#                  !Fill each character with
    Field[x#] = CHR(RANDOM(97,122))    ! a random lower case letter
  END
```

PROP:ImageBits Property of an IMAGE control that allows bitmap images displayed in the control to be moved into and out of memo fields. Any image displayed in the control can be stored.

Example:

```
WinView     WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL
```

```
            IMAGE(),AT(0,0,,),USE(?Image)
            BUTTON('Save Picture'),AT(80,180,60,20),USE(?SavePic)
            BUTTON('New Picture'),AT(160,180,60,20),USE(?NewPic)
            BUTTON('Last Picture'),AT(240,180,60,20),USE(?LastPic)
          END

SomeFile  FILE,DRIVER('Clarion'),PRE(Fil)   !A file with a memo field
MyMemo      MEMO(65520),BINARY
Rec         RECORD
F1            LONG
          . .

FileName  STRING(64)                    !Filename variable

  CODE
  OPEN(SomeFile)
  OPEN(WinView)
  DISABLE(?LastPic)
  IF NOT FILEDIALOG('Choose File to View',FileName,'BitMap|*.BMP|PCX|*.PCX',0)
    RETURN                                !Return if no file chosen
  END
  ?Image{PROP:Text} = FileName
  ACCEPT
    CASE ACCEPTED()
    OF ?NewPic
      IF NOT FILEDIALOG('Choose File to View',FileName,'BitMap|*.BMP|PCX|*.PCX',0)
        BREAK                             !Return if no file chosen
      END
      ?Image{PROP:Text} = FileName
    OF ?SavePic
      Fil:MyMemo = ?Image{PROP:ImageBits}        !Put image into memo
      ADD(SomeFile)                       ! and save it to the file on disk
      ENABLE(?LastPic)                    ! activate Last Picture button
    OF ?LastPic
      ?Image{PROP:ImageBits} = Fil:MyMemo        !Put last saved memo into image
    END
  END
```

PROP:ImageBlob Property of an IMAGE control that allows bitmap images displayed in the control to be moved into and out of BLOB fields. Any image displayed in the control can be stored.

Example:

```
WinView     WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL
            IMAGE(),AT(0,0,,),USE(?Image)
            BUTTON('Save Picture'),AT(80,180,60,20),USE(?SavePic)
            BUTTON('New Picture'),AT(160,180,60,20),USE(?NewPic)
            BUTTON('Last Picture'),AT(240,180,60,20),USE(?LastPic)
          END

SomeFile  FILE,DRIVER('TopSpeed'),PRE(Fil)   !A file with a memo field
MyBlob      BLOB,BINARY
Rec         RECORD
F1            LONG
          . .

FileName  STRING(64)                    !Filename variable

  CODE
  OPEN(SomeFile)
```

```
  OPEN(WinView)
  DISABLE(?LastPic)
  IF NOT FILEDIALOG('Choose File to View',FileName,'BitMap|*.BMP|PCX|*.PCX',0)
    RETURN                                  !Return if no file chosen
  END
  ?Image{PROP:Text} = FileName
  ACCEPT
    CASE ACCEPTED()
    OF ?NewPic
      IF NOT FILEDIALOG('Choose File to View',FileName,'BitMap|*.BMP|PCX|*.PCX',0)
        BREAK                                !Return if no file chosen
      END
      ?Image{PROP:Text} = FileName
    OF ?SavePic
      Fil:MyBlob{PROP:Handle} = ?Image{PROP:ImageBlob}       !Put image into BLOB
      ADD(SomeFile)                          ! and save it to the file on disk
      ENABLE(?LastPic)                       ! activate Last Picture button
    OF ?LastPic
      ?Image{PROP:ImageBlob} = Fil:MyBlob{PROP:Handle}
                                            !Put last saved BLOB into image
    END
  END
```

PROP:Items Returns the number of entries visible in a LIST or COMBO control. (READ-ONLY)

Example:

```
Que          QUEUE
             STRING(30)
               END

WinView      WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL,SYSTEM
            LIST,AT(6,6,120,90),USE(?List),FORMAT('120L'),FROM(Q),IMM
           END

  CODE
  OPEN(WinView)
  SET(SomeFile)
  LOOP ?List{PROP:Items} TIMES  !Fill display queue to limit of displayable items
    NEXT(SomeFile)
    Que = Fil:Record
    ADD(Que)
  END
  ACCEPT
  END
```

PROP:LazyDisplay

Disables (when set to 1) or enables (when set to 0, the default) the feature where all window re-painting is completely done before processing continues with the next statement following a DISPLAY. Setting PROP:LazyDisplay = 1 creates seemingly faster video processing, since the re-paints occur at the end of the ACCEPT loop if there are no other messages pending. This can improve the performance of some applications, but can also have a negative impact on appearance.

Example:

```
WinView      APPLICATION('MyApp'),AT(0,0,320,200),MAX,HVSCROLL,SYSTEM
           END
```

```
  CODE
  OPEN(WinView)
  SYSTEM{PROP:LazyDisplay} = 1  !Disable extra paint message display
                               ! throughout entire application
  ACCEPT
  END
```

PROP:Line An array whose elements each contain one line of the text in a TEXT control. (READ ONLY)

PROP:LineCount Returns the number of lines of text in a TEXT control. (READ ONLY)

Example:

```
LineCount SHORT
MemoLine STRING(80)

CustRpt   REPORT,AT(1000,1000,6500,9000),THOUS
Detail1     DETAIL,AT(0,0,6500,6000)
            TEXT,AT(0,0,6500,6000),USE(Fil:MemoField)
          END
Detail2     DETAIL,AT(0,0,6500,125)
            STRING(@s80),AT(0,0,6500,125),USE(MemoLine)
          END
        END

  CODE
  OPEN(File)
  SET(File)
  OPEN(CustRpt)
  LOOP
    NEXT(File)
    LIneCount = ?Fil:MemoField{PROP:LineCount}
    LOOP X# = 1 TO LIneCount
      MemoLine = ?Fil:MemoField{PROP:Line,X#}
      PRINT(Detail2)
    END
  END
```

PROP:MaxHeight Sets or returns the maximum height of a resizable window.

PROP:MaxWidth Sets or returns the maximum width of a resizable window.

PROP:MinHeight Sets or returns the minimum height of a resizable window.

PROP:MinHeight Sets or returns the minimum width of a resizable window.

Example:

```
WinView     WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL,SYSTEM,RESIZE
            LIST,AT(6,6,120,90),USE(?List),FORMAT('120L'),FROM(Q),IMM
          END

  CODE
  OPEN(WinView)
  WinView{PROPMaxHeight} = 200 !Set boundaries beyond which the user cannot
  WinView{PROPMaxWidth} = 320   ! resize the window
  WinView{PROPMinHeight} = 90
  WinView{PROPMinWidth} = 120
```

```
  ACCEPT
  END
```

PROP:NoTips Disables (when set to 1) or re-enables (when set to 0) tooltip display (TIP attribute) for the SYSTEM, window, or control.

Example:

```
WinView     APPLICATION('MyApp'),AT(0,0,320,200),MAX,HVSCROLL,SYSTEM
          END

  CODE
  OPEN(WinView)
  SYSTEM{PROP:NoTips} = 1  !Disable TIP display throughout entire application
  ACCEPT
  END
```

PROP:Progress You can directly update the display of a PROGRESS cointrol by assigning a value (which must be within the range defined by the RANGE attribute) to the control's PROP:progress property.

Example:

```
BackgroundProcess   PROCEDURE        !Background processing batch process

Win  WINDOW('Batch Processing...'),AT(,,400,400),TIMER(1),MDI,CENTER
       PROGRESS,AT(100,140,200,20),USE(?ProgressBar),RANGE(0,200)
       BUTTON('Cancel'),AT(190,300,20,20),STD(STD:Close)
     END

  CODE
  OPEN(Win)
  OPEN(File)
  ?ProgressBar{PROP:rangehigh} = RECORDS(File)
  SET(File)                        !Set up a batch process
  ACCEPT
    CASE EVENT()
    OF EVENT:CloseWindow
      BREAK
    OF EVENT:Timer                      !Process records when timer allows it
      ProgressVariable += 3               !Auto-updates 1st progress bar
      LOOP 3 TIMES
        NEXT(File)
        IF ERRORCODE() THEN BREAK.
        ?ProgressBar{PROP:progress} += 1   !Manually update progress bar
        !Perform some batch processing code
  . . .
  CLOSE(File)
```

PROP:ScreenText Returns the text displayed on screen in the specified ENTRY or entry-like (SPIN/COMBO) control. Also mention the use in connection with REJECTED events (perhaps in the example).

Example:

```
WinView     WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL
            SPIN(@n3),AT(0,0,320,180),USE(Fil:Field),RANGE(0,255)
          END
  CODE
```

```
  OPEN(WinView)
  ACCEPT
    CASE FIELD()
    OF ?Fil:Field
      CASE EVENT()
      OF EVENT:Rejected
        MESSAGE(?Fil:Field{PROP:ScreenText} & ' is not in the range 0-255')
        SELECT(?)
        CYCLE
      END
    END
  END
```

PROP:SelStart Sets or retrieves the beginning (inclusive) character to mark as a block in an ENTRY or TEXT control. It positions the data entry cursor left of the character, and sets PROP:SelEnd to zero (0) to indicate no block is marked.

PROP:SelEnd Sets or retrieves the ending (inclusive) character to mark as a block in an ENTRY or TEXT control.

Example:

```
WinView     WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL
            ENTRY(@S30),AT(0,0,320,180),USE(Fil:Field),ALRT(F10Key)
          END
  CODE
  OPEN(WinView)
  ACCEPT
    CASE ACCEPTED()
    OF ?Fil:Field
      SETCLIPBOARD(Fil:Field[?Fil:Field{PROP:SelStart}:?Fil:Field{PROP:SelEnd}])
                  !Place highlighted string slice in Windows' clipboard
    END
  END
```

PROP:Size Returns (or sets) the size of a BLOB field.

Example:

```
Names      FILE,DRIVER('TopSpeed')
NbrKey     KEY(Names:Number)
Notes        BLOB           !Can be larger than 64K
Rec        RECORD
Name         STRING(20)
Number       SHORT
         . .

BlobSize   LONG
BlobBuffer1 STRING(65520),STATIC    !Maximum size string
BlobBuffer2 STRING(65520),STATIC    !Maximum size string

WinView     WINDOW('View BLOB Contents'),AT(0,0,320,200),SYSTEM
            TEXT,AT(0,0,320,180),USE(BlobBuffer1),VSCROLL
            TEXT,AT(0,190,320,180),USE(BlobBuffer2),VSCROLL,HIDE
          END
  CODE
  OPEN(Names)
  SET(Names)
  NEXT(Names)
  OPEN(WinView)
```

```
  BlobSize = Names:Notes{PROP:Size}  !Get size of BLOB contents
  IF BlobSize > 65520
    BlobBuffer1 = Names:Notes[1:65520]
    BlobBuffer2 = Names:Notes[65521:BlobSize]
    WinView{PROP:Height} = 400
    UNHIDE(?BlobBuffer2)
  ELSE
    BlobBuffer1 = Names:Notes[1:BlobSize]
  END
  ACCEPT
  END
```

PROP:Thread Returns the thread number of a window. This is not necessarily the currently executing thread, if you've used SETTARGET to set the TARGET built-in variable. (READ-ONLY)

Example:

```
WinView     WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL,SYSTEM
           END
ToolboxThreadBYTE
  CODE
  OPEN(WinView)
  ToolboxThread = ToolboxWin{PROP:Thread}     !Get window thread number
  ACCEPT
  END
```

PROP:TipDelay Sets the time delay before tooltip display (TIP attribute) for the SYSTEM (16-bit only).

PROP:TipDisplay Sets the duration of tooltip display (TIP attribute) for the SYSTEM (16-bit only).

Example:

```
WinView     APPLICATION('MyApp'),AT(0,0,320,200),MAX,HVSCROLL,SYSTEM
           END

  CODE
  OPEN(WinView)
  SYSTEM{PROP:TipDelay} = 50    !Delay TIP display for 1/2 second
  SYSTEM{PROP:TipDisplay} = 500   !TIP display for 5 seconds
  ACCEPT
  END
```

PROP:Touched When non-zero, indicates the data in the ENTRY, TEXT, SPIN, or COMBO control with input focus has been changed by the user since the last EVENT:Accepted. Automatically reset to zero each time the control generates an EVENT:Accepted.

Example:

```
WinView     WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL
            ENTRY(@S30),AT(0,0,320,180),USE(Fil:Field)
           END
  CODE
  OPEN(WinView)
  ACCEPT
    CASE EVENT()
    OF EVENT:Selected
      ?Fil:Field{PROP:Touched} = 1    !Force an EVENT:Accepted to generate
    OF EVENT:Accepted
      !Process the data, whether entered by the user or in the field at the start
    END
```

```
  END
```

PROP:TrueValue Sets the value received by the USE variable of a CHECK box when the user checks it on. This overrides the default assigned value of one (1).

PROP:FalseValue Sets the value received by the USE variable of a CHECK box when the user checks it off. This overrides the default assigned value of zero (0).

Example:

```
CheckField  STRING(1)

WinView     WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL
            CHECK('True or False'),AT(0,0,,),USE(CheckField)
          END
  CODE
  OPEN(WinView)
  ?CheckField{PROP:TrueValue} = 'T'
  ?CheckField{PROP:FalseValue} = 'F'
  ACCEPT
  END
```

PROP:VBXEvent Returns the name of a VBX event. (READ-ONLY)

PROP:VBXEventArg
VBX event parameters. An array.

Example:

```
WinView     WINDOW('View'),AT(0,0,320,200),MDI,MAX,HVSCROLL
            CUSTOM,USE(?Graph),CLASS('graph.vbx','graph'),'graphstyle'('2')
          END

  CODE
  OPEN(WinView)
  ACCEPT
    CASE EVENT()
    OF EVENT:VBXEvent
      IF ?Graph{PROP:VBXEvent} = 'FooEvent'                    !Check event name
        ProcessFoo(?Graph{PROP:VBXEventArg,1},?Graph{PROP:VBXEventArg,2})
                  !Get 1st and 2nd event parameters and pass to process procedure
      END
    END
  END
```

PROP:Visible Returns an empty string if the control is not visible because either because it has been hidden, or it is a member of a "parent" control (OPTION, GROUP, MENU, SHEET, or TAB) that is hidden, or is on a TAB control page that is not currently selected. (READ-ONLY)

Example:

```
MDIChild  WINDOW('Child One'),AT(0,0,320,200),MDI,MAX,HVSCROLL
          SHEET,AT(0,0,320,175),USE(SelectedTab)
            TAB('Tab One'),USE(?TabOne)
              PROMPT('Enter Data:'),AT(100,100,20,20),USE(?P1)
              ENTRY(@S8),AT(100,140,32,20),USE(E1)
              PROMPT('Enter More Data:'),AT(100,200,20,20),USE(?P2)
              ENTRY(@S8),AT(100,240,32,20),USE(E2)
            END
            TAB('Tab Two'),USE(?TabTwo)
```

```
            PROMPT('Enter Data:'),AT(100,100,20,20),USE(?P3)
            ENTRY(@S8),AT(100,140,32,20),USE(E3)
            PROMPT('Enter More Data:'),AT(100,200,20,20),USE(?P4)
            ENTRY(@S8),AT(100,240,32,20),USE(E4)
          END
        END
        BUTTON('Ok'),AT(100,180,20,20),USE(?Ok)
        BUTTON('Cancel'),AT(200,180,20,20),USE(?Cancel)
      END
  CODE
  OPEN(MDIChild)
  ACCEPT
    CASE EVENT()
    OF EVENT:Completed
      BREAK
    END
    CASE FIELD()
    OF ?Ok
      CASE EVENT()
      OF EVENT:Accepted
        SELECT
      END
    OF ?E3
      CASE EVENT()
      OF EVENT:Accepted
        E3 = UPPER(E3)               !Convert the data entered to Upper case
        IF ?E3{PROP:Visible} AND MDIChild{PROP:AcceptAll}
                                    !Check for visibility during AcceptAll mode
          DISPLAY(?E3)               ! and display the upper cased data
        END
      END
    OF ?Cancel
      CASE EVENT()
      OF EVENT:Accepted
        BREAK
      END
    END
  END
```

PROP:VscrollPos Returns the position of the vertical scroll bar's "thumb" (from 0 to 255) on a window, IMAGE, TEXT, LIST, or COMBO control with the VSCROLL attribute. Setting this property causes the control or window's contents to be scrolled vertically (unless the IMM attribute is on the LIST or COMBO, then only the "thumb" moves).

Example:

```
Que         QUEUE
              STRING(50)
            END
WinView       WINDOW('View'),AT(0,0,320,200),MDI,SYSTEM
              LIST,AT(0,0,320,200),USE(?List),FROM(Que),IMM,VSCROLL
            END
  CODE
  OPEN(WinView)
  Fil:KeyField = 'A' ;  DO BuildListQue
  ACCEPT
    CASE FIELD()
    OF ?List
      CASE EVENT()
      OF EVENT:ScrollDrag
```

```
      EXECUTE INT(?List{PROP:VscrollPos}/10) + 1
        Fil:KeyField = 'A'
        Fil:KeyField = 'B'
        Fil:KeyField = 'C'
        Fil:KeyField = 'D'
        Fil:KeyField = 'E'
        Fil:KeyField = 'F'
        Fil:KeyField = 'G'
        Fil:KeyField = 'H'
        Fil:KeyField = 'I'
        Fil:KeyField = 'J'
        Fil:KeyField = 'K'
        Fil:KeyField = 'L'
        Fil:KeyField = 'M'
        Fil:KeyField = 'N'
        Fil:KeyField = 'O'
        Fil:KeyField = 'P'
        Fil:KeyField = 'Q'
        Fil:KeyField = 'R'
        Fil:KeyField = 'S'
        Fil:KeyField = 'T'
        Fil:KeyField = 'U'
        Fil:KeyField = 'V'
        Fil:KeyField = 'W'
        Fil:KeyField = 'X'
        Fil:KeyField = 'Y'
        Fil:KeyField = 'Z'
      END
      DO BuildListQue
  . . .
  FREE(Que)
BuildListQue  ROUTINE
  FREE(Queue)
  SET(Fil:SomeKey,Fil:SomeKey)  !Set to selected key field
  LOOP ?List{PROP:Items} TIMES  !Process number of records visible in list
    NEXT(SomeFile) ;  IF ERRORCODE() THEN BREAK.  !Break at end of file
    Que = Fil:KeyField         !Assign field to display to QUEUE
    ADD(Que)                  ! and add it to the QUEUE
  END
```

PROP:WndProc Sets or gets the window (not the client area) messaging procedure for use with low-level Windows API calls that require it. Generally used with sub-classing to track all Windows messages. (READ-ONLY)

Example:

```
   PROGRAM
   MAP
     main
     SubClassFunc1(USHORT,SHORT,SHORT,LONG),LONG,PASCAL
     SubClassFunc2(USHORT,SHORT,SHORT,LONG),LONG,PASCAL
     MODULE('Windows')
   !TopSpeed Win31 Library
      CallWindowProc(LONG,USHORT,SHORT,SHORT,LONG),LONG,PASCAL
     END
   END
SavedProc1  LONG
SavedProc2  LONG
WM_MOUSEMOVE    EQUATE(0200H)
PT         GROUP,PRE(PT)
X            SHORT
```

```
Y           SHORT
          END
       CODE
  Main

Main             PROCEDURE
WinView        WINDOW('View'),AT(0,0,320,200),HVSCROLL,MAX,TIMER(1),STATUS
            STRING('X Pos'),AT(1,1,,),USE(?String1)
            STRING(@n3),AT(24,1,,),USE(PT:X)
            STRING('Y Pos'),AT(44,1,,),USE(?String2)
            STRING(@n3),AT(68,1,,),USE(PT:Y)
            BUTTON('Close'),AT(240,180,60,20),USE(?Close)
          END
  CODE
  OPEN(WinView)
  SavedProc1 = WinView{PROP:WndProc}               !Save this procedure
  WinView{PROP:WndProc} = ADDRESS(SubClassFunc1)    !Change to subclass procedure
  SavedProc2 = WinView{PROP:ClientWndProc}          !Save this procedure
  WinView{PROP:ClientWndProc} = ADDRESS(SubClassFunc2)      !Change to subclass proc
  ACCEPT
    CASE ACCEPTED()
    OF ?Close
      BREAK
  . .
SubClassFunc1    FUNCTION(hWnd,wMsg,wParam,lParam)    !Sub class procedure
  CODE                                          ! to track mouse movement in
  IF  wMsg = WM_MOUSEMOVE                          ! window's status bar (only)
    PT:X = MOUSEX()
    PT:Y = MOUSEY()
  END
  RETURN(CallWindowProc(SavedProc1,hWnd,wMsg,wParam,lParam))

SubClassFunc2    FUNCTION(hWnd,wMsg,wParam,lParam)    !Sub class procedure
  CODE                                          ! to track mouse movement in
  IF  wMsg = WM_MOUSEMOVE                          ! window's client area
    PT:X = MOUSEX()
    PT:Y = MOUSEY()
  END
  RETURN(CallWindowProc(SavedProc2,hWnd,wMsg,wParam,lParam))
                                                !Pass control back to
                                                ! saved procedure
```

Printer Control Properties

These properties control report and printer behavior. All of these properties can be used with either the PRINTER built-in variable or the label of the report as the *target*, however they may not all make sense with both.

PROPPRINT:Collate
Specify the printer should collate the output: 0=off, 1=on (not supported by all printers).

PROPPRINT:Color Color or monochrome print flag:1=mono, 2=color (not supported by all printers).

PROPPRINT:Context
Returns the handle to the printer's device context after the first PRINT statement for the report, or an information context before the first PRINT statement. This may not be set for the built-in Global PRINTER variable and is normally only read (not set).

PROPPRINT:Copies
The number of copies to print (not supported by all printers).

PROPPRINT:Device
The name of the Printer as it appears in the Windows Printer Dialog. If multiple printer names start with the same characters, the first encountered is used (not case sensitive). May be set for the PRINTER built-in variable only before the report is open.

PROPPRINT:DevMode
The entire device mode (devmode) structure as defined in the Windows Software Development Kit. This provides direct API access to all printer properties. Consult a Windows API manual before using this.

```
DevMode       GROUP
DeviceName      STRING(32)    !PROPPRINT:Device
SpecVersion     USHORT
DriverVersion   USHORT
Size          USHORT
DriverExtra     USHORT
Fields        ULONG
Orientation     SHORT
PaperSize       SHORT       !PROPPRINT:Paper
PaperLength     SHORT       !PROPPRINT:PaperHeight
PaperWidth      SHORT       !PROPPRINT:PaperWidth
Scale           SHORT       !PROPPRINT:Percent
Copies        SHORT       !PROPPRINT:Copies
DefaultSource   SHORT       !PROPPRINT:PaperBin
PrintQuality    SHORT       !PROPPRINT:Resolution
Color           SHORT       !PROPPRINT:Color
Duplex        SHORT       !PROPPRINT:Duplex
            END
```

PROPPRINT:Driver The printer driver's filename (without the .DLL extension).

PROPPRINT:Duplex
The duplex printing mode (not supported by all printers). Equates (DUPLEX::xxx) for the standard choices are listed in the PRNPROP.CLW file.

PROPPRINT:FontMode
The TrueType font mode. Equates (FONTMODE:xxx) for the modes are listed in the PRNPROP.CLW file.

PROPPRINT:FromMin

When set for the built-in PRINTER variable, this forces the value into the "From:" page number in the PRINTERDIALOG. Specify -1 to disable ranges

PROPPRINT:FromPage
The page number on which to start printing. Specify -1 to print from the start.

PROPPRINT:Paper Standard paper size. Equates (PAPER:xxx) for the standard sizes are listed in the PRNPROP.CLW file. This defines the dimensions of the .WMF files that are created by the Clarion runtime library's "print engine."

PROPPRINT:PaperBin
The paper source. Equates (PAPERBIN:xxx) for the standard locations are listed in the PRNPROP.CLW file.

PROPPRINT:PaperHeight
The paper height in tenths of millimeters (mm/10). There are 25.4 mm per inch. Used when setting PROPPRINT:Paper to PAPER:Custom (not normally used for laser printers).

PROPPRINT:PaperWidth
The paper width in tenths of millimeters (mm/10). There are 25.4 mm per inch. Used when setting PROPPRINT:Paper to PAPER:Custom (not normally used for laser printers).

PROPPRINT:Percent
The scaling factor used to enlarge or reduce the printed output, in percent (not supported by all printers). This defaults to 100 percent. Set this value to print at the desired percentage (if your printer and driver support scaling). For example, set to 200 to print at double size, or 50 to print at half size.

PROPPRINT:Port Output port name (LPT1, COM1, etc.).

PROPPRINT:PrintToFile
The Print to File flag: 0=off, 1=on.

PROPPRINT:PrintToName
The output filename when printing to a file.

PROPPRINT:Resolution
The print resolution in Dots Per Inch (DPI). Equates (RESOLUTION:xxx) for the standard resolutions are listed in the PRNPROP.CLW file.

PROPPRINT:ToMax
When set for the built-in PRINTER variable, this forces the value into the "To:" page number in the PRINTERDIALOG. Specify -1 to disable ranges

PROPPRINT:ToPage
The page number on which to end printing. Specify -1 to print to end.

PROPPRINT:Yresolution
Vertical print resolution in Dots Per Inch (DPI). Equates (RESOLUTION:xxx) for the standard resolutions are listed in the PRNPROP.CLW file.

Example:

```
SomeReport  REPORT
          END

  CODE
  PRINTER{PROPPRINT:Device} = 'Epson'            !Pick 1st Epson in the list
```

```
PRINTER{PROPPRINT:Port} = 'LPT2:'            !Send report to LPT2

SomeReport{PROPPRINT:Paper} = PAPER:User       !Custom paper size
SomeReport{PROPPRINT:PAPERHeight} = 6 * 254    !6" form height
SomeReport{PROPPRINT:PAPERWidth} = 3.5 * 254     !3.5" form width

PRINTER{PROPPRINT:Percent} = 250                 !page printed 2.5 times normal

PRINTER{PROPPRINT:Copies} = 3                       !print 3 copies of each page
PRINTER{PROPPRINT:Collate} = False                  !print 1,1,1,2,2,2,3,3,3,...
PRINTER{PROPPRINT:Collate} = True                   !print 1,2,3..., 1,2,3...,

PRINTER{PROPPRINT:PrintToFile} = True          !print to a file
PRINTER{PROPPRINT:PrintToName} = 'OUTPUT.RPT'    !filename to print to

OPEN(SomeReport)                        !Open report after setting properties
```

Embedded SQL

Clarion's property syntax can be used to embed SQL statements in your program code by using PROP:SQL naming the file as the *target*. This is only appropriate when using an SQL file driver (such as the ODBC, AS/400, or Oracle drivers).

You may embed any SQL statements supported by the back-end SQL server. If you issue an SQL statement that causes a result set to be returned (such as an SQL SELECT statement), you use NEXT(file) to retrieve the result set (one row at a time) into the file's record buffer. The FILEERRORCODE() and FILEERROR() functions will return any error code and error string set by the back-end SQL server.

You may also query the contents of PROP:SQL to get the last SQL SELECT statement issued by the file driver.

Example:

```
SQLFile{PROP:SQL} = 'SELECT field1,field2 FROM table1'                          |
                  & 'WHERE field1 > (SELECT max(field1)'    |
                  & 'FROM table2'
                                    !Returns a result set that you
                                    ! get one row at a time using
                                    ! NEXT(SQLFile)

SQLFile{PROP:SQL} = 'CALL GetRowsBetween(2,8)'    !Call a stored procedure

SQLFile{PROP:SQL} = 'CREATE INDEX ON table1 (field1, field2 DESC)"
                                    !No result set

SelectString = SQLFile{PROP:SQL}              !Get last SELECT issued by driver
```

Event Equates

Events

In Clarion Windows programs, most of the messages from Windows are automatically handled internally by the ACCEPT event processor. These are the common events handled by the runtime library (screen re-draws, etc.). Only those events that actually may require program action are passed on by ACCEPT to your Clarion code. The net effect of this is to make your programming job easier by removing the low-level "drudgery" code from your program, allowing you to concentrate on the high-level aspects of programming, instead. Of course, it is also possible to handle these low-level messages yourself by "sub-classing" the window, but that is a low-level technique that should only be used if absolutely necessary. Consult Charles Petzold's book *Programming Windows* published by Microsoft Press if you need more information on sub-classing.

There are two types of events passed on to the program by ACCEPT: **Field-specific** and **Field-independent** events. The following lists are the event EQUATEs that are contained in EQUATES.CLW.

Field-Independent Events

A **Field-independent** event does not relate to any one control but requires some program action (for example, to close a window, quit the program, or change execution threads). Most of these events cause the system to become modal for the period during which they are processing, since they require a response before the program may continue.

EVENT:PreAlertKey
: The user pressed an ALRT attribute hot key for an ALRT attribute on the window. If a CYCLE statement is encountered in the code to process this event, the EVENT:AlertKey is not generated and the action is aborted.

EVENT:AlertKey
: The user pressed an ALRT attribute hot key for an ALRT attribute on the window. This is the event on which you perform the action the user has requested by pressing the alert key.

EVENT:CloseWindow
: The window is closing. POSTing this event closes the window. This is the event on which you perform any window cleanup code.

EVENT:CloseDown
: The application is closing. POSTing this event closes the application. This is the event on which you perform any application cleanup code.

EVENT:OpenWindow
: The window is opening. This is the event on which you perform any window initialization code.

EVENT:LoseFocus
: The window is losing input focus to another thread. This is the event on which you save any data that could be at risk of being changed by another thread.

EVENT:GainFocus
: The window is gaining input focus from another thread. This is the event on which you restore any data you saved in EVENT:LoseFocus.

EVENT:Suspend
: The window still has input focus but is giving control to another thread to process timer events.

EVENT:Resume
: The window still has input focus and is regaining control from an EVENT:Suspend.

EVENT:Timer
: The TIMER attribute has triggered. This is the event on which you perform any timed actions, such as clock display, or background record processing for reports or batch processes.

EVENT:Move
: The user is moving the window. If a CYCLE statement is encountered in the code to process this event, the EVENT:Moved is not generated and the action is aborted. This is the event on which you can prevent users from moving a window.

EVENT:Moved
: The user has moved the window. This is the event on which you readjust anything that is screen-position-dependent.

EVENT:Size
: The user is resizing the window. If a CYCLE statement is encountered in the code to process this event, the EVENT:Sized is not generated and the action is aborted. This is the event on which you can prevent users from resizing a window.

EVENT:Sized
: The user has resized the window. This is the event on which you readjust anything that is screen-size-dependent.

EVENT:Restore
: The user is restoring the window's previous size. If a CYCLE statement is

encountered in the code to process this event, the EVENT:Restored is not generated and the action is aborted. This is the event on which you can prevent users from restorng a window.

EVENT:Restored The user has restored the window's previous size. This is the event on which you readjust anything that is screen-size-dependent.

EVENT:Maximize The user is maximizing the window. If a CYCLE statement is encountered in the code to process this event, the EVENT:Maximized is not generated and the action is aborted. This is the event on which you can prevent users from maximizing a window.

EVENT:Maximized The user has maximized the window. This is the event on which you readjust anything that is screen-size-dependent.

EVENT:Iconize The user is minimizing the window. If a CYCLE statement is encountered in the code to process this event, the EVENT:Iconized is not generated and the action is aborted. This is the event on which you can prevent users from minimizing a window.

EVENT:Iconized The user has minimized the window. This is the event on which you readjust anything that is screen-size-dependent.

EVENT:Completed AcceptAll (non-stop) mode has finished processing all the window's controls. This is the event on which you have executed all data entry validation code for the controls in the window and can safely write to disk.

EVENT:DDErequest
A client has requested a data item from this Clarion DDE server application. This is the event on which you execute DDEWRITE to provide the data to the client once.

EVENT:DDEadvise
A client has requested continuous updates of a data item from this Clarion DDE server application. This is the event on which you execute DDEWRITE to provide the data to the client every time it changes.

EVENT:DDEexecute
A client has sent a command to this Clarion DDE server application (if the client is another Clarion application, it has executed a DDEEXECUTE statement). This is the event on which you determine the action the client has requested and perform it, then execute a CYCLE statement to signal positive acknowledgement to the client that sent the command.

EVENT:DDEpoke A client has sent unsolicited data to this Clarion DDE server application. This is the event on which you determine what the client has sent and where to place it, then execute a CYCLE statement to signal positive acknowledgement to the client that sent the data.

EVENT:DDEdata A DDE server has supplied an updated data item to this Clarion client application.

EVENT:DDEclose A DDE server has terminated the DDE link to this Clarion client application.

Field-Specific Events

A **Field-specific** event occurs when the user presses a key that may require the program to perform a specific action related to that control.

EVENT:Selected The control has received input focus. This is the event on which you should perform any data initialization code.

EVENT:Accepted The user has entered data or made a selection then pressed TAB or CLICKED the mouse to move on to another control. This is the event on which you should perform any data input validation code.

EVENT:Rejected The user has entered an invalid value for the entry picture, or an out-of-range number on a SPIN control. The REJECTCODE function returns the reason the user's input has been rejected and you can use the PROP:ScreenText property to get the user's input from the screen. This is the event on which you alert the user to the exact problem with their input.

EVENT:PreAlertKey
The user pressed an ALRT attribute hot key for an ALRT attribute on the control. If a CYCLE statement is encountered in the code to process this event, the EVENT:AlertKey is not generated and the action is aborted.

EVENT:AlertKey The user pressed an ALRT attribute hot key for an ALRT attribute on the control. This is the event on which you perform the action the user has requested by pressing the alert key.

EVENT:Dragging The user is dragging the mouse from a control with the DRAGID attribute and the mouse cursor is over a valid potential drop target. This event is posted to the control from which the user is dragging. This is the event on which you can change the mouse cursor to indicate a valid drop target.

EVENT:Drag The user released the mouse button over a valid drop target. This event is posted to the control from which the user is dragging. This is the event on which you set the program to pass the dragged data to the drop target.

EVENT:Drop The user released the mouse button over a valid drop target. This event is posted to the drop target control. This is the event on which you receive the dragged data.

EVENT:NewSelection
The current selection in the LIST or COMBO control has changed (the highlight bar has moved up or down). This is the event on which you perform any "housekeeping" to synchronize other controls with the currently highlighted record in the list.

EVENT:ScrollUp On a LIST or COMBO control with the IMM attribute, the user has attempted to move the highlight bar off the top of the LIST. This is the event on which you get a previous record when "page-loading" the list.

EVENT:ScrollDown
On a LIST or COMBO control with the IMM attribute, the user has attempted to move the highlight bar off the bottom of the LIST. This is the event on which you get the next record when "page-loading" the list.

EVENT:PageUp On a LIST or COMBO control with the IMM attribute, the user pressed PGUP. This is the event on which you get the previous page of records when "page-loading" the list.

EVENT:PageDown

On a LIST or COMBO control with the IMM attribute, the user pressed PGDN. This is the event on which you get the next page of records when "page-loading" the list.

EVENT:ScrollTop On a LIST or COMBO control with the IMM attribute, the user pressed CTRL+PGUP. This is the event on which you get the first page of records when "page-loading" the list.

EVENT:ScrollBottom
On a LIST or COMBO control with the IMM attribute, the user pressed CTRL+PGDN. This is the event on which you get the last page of records when "page-loading" the list.

EVENT:Locate On a LIST control with the VCR attribute, the user pressed the locator (?) VCR button. This is the event on which you can unhide the locator entry control, if it is kept hidden.

EVENT:DroppingDown
On a LIST or COMBO control with the DROP attribute, the user pressed the down arrow button. This is the event on which you get the records when "demand-loading" the list.

EVENT:DroppedDown
On a LIST or COMBO control with the DROP attribute, the list has dropped. This is the event on which you can hide other controls that the droplist covers to prevent "screen clutter" from distracting the user.

EVENT:VBXevent On a CUSTOM control, a VBX-specific event occurred. This is the event on which you query the PROP:VBXEvent and PROP:VBXEventArg properties to determine what event occurred and its parameters.

EVENT:Expanding On a LIST control with T in the FORMAT attribute string, the user has clicked on a tree expansion box. If a CYCLE statement is encountered in the code to process this event, the EVENT:Expanded is not generated and the expansion is aborted.

EVENT:Expanded On a LIST control with T in the FORMAT attribute string, the user has clicked on a tree expansion box.

EVENT:Contracting
On a LIST control with T in the FORMAT attribute string, the user has clicked on a tree contraction box. If a CYCLE statement is encountered in the code to process this event, the EVENT:Contracted is not generated and the contraction is aborted.

EVENT:Contracted On a LIST control with T in the FORMAT attribute string, the user has clicked on a tree expansion box.

EVENT:MouseIn On a REGION with the IMM attribute, the mouse cursor has entered the region.

EVENT:MouseOut On a REGION with the IMM attribute, the mouse cursor has left the region.

EVENT:MouseMove
On a REGION with the IMM attribute, the mouse cursor has moved within the region.

EVENT:TabChanging
On a SHEET control, focus is passing to another tab. This is the event on which you perform any necessary "housekeeping" code.

String Control Properties

Click on a TAB to see its help

The String control allows you to place a string constant in a window or report. It optionally allows you to substitute a variable.

General

Parameter — Specify a string constant by typing it in the **Parameter** box. If the control is to display a variable, type a picture token in this box.

Use — Type a field equate label to reference the control in executable code, or the name of a variable. Press the ellipsis button to select of define a variable.

Justification — Specify left, center, right, decimal, or default justification. Default justification matches that specified in the data dictionary, if applicable. If you use decimal justification, you set the Offset to allow display of digits to the right of the decimal point.

Offset — Specify an indentation value for the list text, in dialog units for a window, or the default measurement unit for a report.

Variable String — Optionally check the **Variable String** box. This specifies that you want to display the contents of a variable in the string control. If so, place a picture in the Parameter box, such as @s24.

Transparent — Specify whether you wish the control background to be **Transparent**. This instructs Windows to suppress the rectangular region around the text--the background. Normally, Windows will paint this the same uniform color as the window below the control.

Total Type — This drop down list is available only from the **Report Formatter**, and only when the **Variable String** box is checked. Choose from this drop down list to implement one of Clarion's built in report totaling functions. Choose from ***Sum, Average, Maximum, Minimum, Count,*** and ***Page No***.

To create page totals or group totals, use the **Reset On** drop down list in conjunction with your **Total Type** selection.

For total strings in a DETAIL, built in totals are calculated each time the DETAIL containing the string control is PRINTed. For total strings in a *Group* FOOTER, the totals are calculated each time any DETAIL within the Group BREAK is PRINTed. For total strings in a *Page* FOOTER, the totals are calculated each time *any* DETAIL is PRINTed.

As a result of the foregoing, the built in totals are most useful for calculations made at the lowest level of detail. Calculations on higher levels should be hand coded. See the Getting Started manual for examples.

Reset On — This drop down list is available only from the **Report Formatter**, and only when the **Variable String** box is checked. To create page totals or group totals, use the **Reset On** drop down list in conjunction with your **Total Type** selection. Reset your totals at the beginning of each **Page**, or at the beginning of any BREAK group within the report.

Mode

Hide Makes the control invisible at the time Windows would initially display it. Windows actually creates the control--it just doesn't display it on screen. The **Window Formatter** places the HIDE attribute on the control. Use the UNHIDE statement to display the control.

Disable Disables or 'grays-out' the control when your program initially displays it. The **Window Formatter** places the DISABLE attribute on the control. Use the ENABLE statement to allow the user access to the control.

Scroll Specifies whether the control should move with the window when the user scrolls the window. By default, (unchecked), the control does not move with the window. Leave the **Scroll** box unchecked to create a control that stays fixed when the user scrolls the window. The **Window Formatter** places the SCROLL attribute on the control when checked.

Extra

Drop ID To specify the type of Drag operations this control will accept, type up to 16 *signatures*, separated by commas. The **Window Formatter** adds the **DROPID** attribute to the control, which indicates the control is a valid target for the drag and drop operations identified by the signatures.

Help

Cursor The *Cursor* field (the CURSOR attribute) allows you to specify an alternate shape for the cursor when the user passes the cursor over the control. The drop-down list provides standard cursor choices such as I-Beam and Crosshair. To select an external cursor file (whose extension must be .CUR), choose **Select File** from the drop-down list, then pick the file using the standard file dialog.

Position

Allows you to set the location and size of a control.

The **Position** dialog allows you to specify the AT (set control position and size in window) attribute. Filling in the attribute manually is optional--you may set position visually, dragging with the mouse in the **Window Formatter**.

To set a *precise* screen location or size for your control, specify fixed coordinates via this dialog. To set the location and size, choose from the following options for the **Top Left Corner**, the **Width** of the control, and its **Height**.

Default This instructs Windows to set a value which depends on the end user's system. Default size for a control depends on the size of the system font.

Full Sizes a control to expand to the full width or height of the window. The control automatically expands or contracts as the end user resizes the window.

Fixed To set a specific position and size, mark the **Fixed** choices.

The measurement units for these boxes are dialog units. These provide a relative screen measure. Dialog units are a relative measure based on the default system font character size. Windows automatically repositions the control proportionally at different screen resolutions.

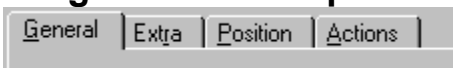
Font

Calls the **Select Font** dialog which allows you to change the font (typeface), size, style (such as bold or italic), color, and font effects (underline and strikeout) for the selected control or window. As you choose options, the dialog box displays a sample of the selected font.

Actions

There are no prompts for the Actions Tab for this control.

Image Control Properties



Click on a TAB to see its help

The Image control allows you to place bitmapped and vector images in a window or report. The bitmap file formats supported are .BMP, .PCX, .GIF, .ICO and .JPG. The vector file format supported is .WMF. Clarion for Windows can support up to 16.7 million color resolution.

You must add the PALETTE attribute to a WINDOW to support custom palettes and color depths beyond the resolution of the end user's machine at runtime. For example, to support a 16.7 million color palette, the proper attribute is PALETTE(16777215).

The Image control cannot receive focus, nor can it generate events.

General

File — Select a graphics file. Type in a file name, or press the ellipsis (**...**) button to the right of the **File** field to select a graphics file using the standard **File Open** dialog.

Use — Type a field equate label to reference the control in executable code.

Mode

Hide — Makes the control invisible at the time Windows would initially display it. Windows actually creates the control--it just doesn't display it on screen. The **Window Formatter** places the HIDE attribute on the control. Use the UNHIDE statement to display the control.

Disable — Disables or 'grays-out' the control when your program initially displays it. The **Window Formatter** places the DISABLE attribute on the control. Use the ENABLE statement to allow the user access to the control.

Scroll — Specifies whether the control should move with the window when the user scrolls the window. By default, (unchecked), the control does not move with the window. Leave the **Scroll** box unchecked to create a control that stays fixed when the user scrolls the window. The **Window Formatter** places the SCROLL attribute on the control when checked.

Extra

Scroll Bars — To add a horizontal scroll bar to the control, mark the **Horizontal** check box. Scroll bars only appear when the contents of the text box are bigger than the window. To add a vertical scroll bar, check the **Vertical** check box. These options add the HSCROLL, VSCROLL, and HVSCROLL attributes to the control.

Position

Allows you to set the location and size of a control.

The **Position** dialog allows you to specify the AT (set control position and size in window) attribute. Filling in the attribute manually is optional--you may set position visually, dragging with the mouse in the **Window Formatter**.

To set a *precise* screen location or size for your control, specify fixed coordinates via this dialog. To set the location and size, choose from the following options for the **Top Left Corner**, the **Width** of the control, and its **Height**.

Default	This instructs Windows to set a value which depends on the end user's system. Default size for a control depends on the size of the system font.
Full	Sizes a control to expand to the full width or height of the window. The control automatically expands or contracts as the end user resizes the window.
Fixed	To set a specific position and size, mark the **Fixed** choices. The measurement units for these boxes are dialog units. These provide a relative screen measure. Dialog units are a relative measure based on the default system font character size. Windows automatically repositions the control proportionally at different screen resolutions.

Actions

There are no prompts for the Actions tab for this control.

Region Control Properties

Click on a TAB to see its help

General

Use Type a field equate label to reference the control in executable code.

Mode

Hide Makes the control invisible at the time Windows would initially display it. Windows actually creates the control--it just doesn't display it on screen. The **Window Formatter** places the HIDE attribute on the control. Use the UNHIDE statement to display the control.

Disable Disables or 'grays-out' the control when your program initially displays it. The **Window Formatter** places the DISABLE attribute on the control. Use the ENABLE statement to allow the user access to the control.

Scroll Specifies whether the control should move with the window when the user scrolls the window. By default, (unchecked), the control does not move with the window. Leave the **Scroll** box unchecked to create a control that stays fixed when the user scrolls the window. The **Window Formatter** places the SCROLL attribute on the control when checked.

Extra

Fill Allows you to specify the **Fill** color--the inside color. Check the **Fill** check box if you do *not* wish the region to be transparent. Then press the **Fill Color...** button. The standard **Color** dialog appears. Select a color by clicking on the color sample square, or add a custom color.

Border Allows you to specify the **Border** color--the outside line color. Check the **Border** check box if you *do* wish the region to have a border. Then press the **Border Color...** button. The standard **Color** dialog appears. Select a color by clicking on the color sample square, or add a custom color.

Immediate To generate a message event each time the mouse moves over the region, check the **Immediate** box. This adds the IMM attribute to the control. You are responsible for the code that executes upon notification of the event.

Drag ID To specify the type of Drag operations this control will generate, type up to 16 *signatures*, separated by commas. The **DRAGID** attribute specifies the REGION control can serve as a drag-and-drop host. DRAGID works in conjunction with the **DROPID** attribute.

Drop ID To specify the type of Drag operations this control will accept, type up to 16 *signatures*, separated by commas. The **Window Formatter** adds the **DROPID** attribute to the control, which indicates the control is a valid target for the drag and drop operations identified by the signatures.

Help

Cursor The CURSOR attribute allows you to specify an alternate shape for the cursor when the user passes the cursor over the control. The drop-down list provides standard cursor choices such as I-Beam and Crosshair. To select an external cursor file (whose extension must be .CUR), choose **Select File** from the drop-down list, then pick the file using the standard file dialog.

Position

Allows you to set the location and size of a control.

The **Position** dialog allows you to specify the AT (set control position and size in window) window attribute. Filling in the attribute manually is optional--you may set position visually, dragging with the mouse in the **Window Formatter**.

To set a *precise* screen location or size for your control, specify fixed coordinates via this dialog. To set the location and size, choose from the following options for the **Top Left Corner**, the **Width** of the control, and its **Height**.

Default This instructs Windows to set a value which depends on the end user's system. Default size for a control depends on the size of the system font.

Full Sizes a control to expand to the full width or height of the window. The control automatically expands or contracts as the end user resizes the window.

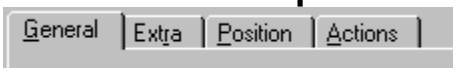
Fixed To set a specific position and size, mark the **Fixed** choices.

The measurement units for these boxes are dialog units. These provide a relative screen measure. Dialog units are a relative measure based on the default system font character size. Windows automatically repositions the control proportionally at different screen resolutions.

Actions

There are no prompts for the Actions Tab for this control.

Line Control Properties



Click on a TAB to see its help

The line control allows you to place a straight line in your window or report. You specify a color. The line control cannot receive focus, nor can it generate events.

General

Use Type a field equate label to reference the control in executable code.

Mode

Hide Makes the control invisible at the time Windows would initially display it. Windows actually creates the control--it just doesn't display it on screen. The **Window Formatter** places the HIDE attribute on the control. Use the UNHIDE statement to display the control.

Disable Disables or 'grays-out' the control when your program initially displays it. The **Window Formatter** places the DISABLE attribute on the control. Use the ENABLE statement to allow the user access to the control.

Scroll Specifies whether the control should move with the window when the user scrolls the window. By default, (unchecked), the control does not move with the window. Leave the **Scroll** box unchecked to create a control that stays fixed when the user scrolls the window. The **Window Formatter** places the SCROLL attribute on the control when checked.

Extra

Line Color Allows you to specify the **Line** color. The standard **Color** dialog appears. Select a color by clicking on the color sample square, or add a custom color.

Position

Allows you to set the location and size of a control.

The **Position** dialog allows you to specify the AT (set control position and size in window) attribute. Filling in the attribute manually is optional--you may set position visually, dragging with the mouse in the **Window Formatter**.

To set a *precise* screen location or size for your control, specify fixed coordinates via this dialog. To set the location and size, choose from the following options for the **Top Left Corner**, the **Width** of the control, and its **Height**.

Default This instructs Windows to set a value which depends on the end user's system. Default size for a control depends on the size of the system font.

Full Sizes a control to expand to the full width or height of the window. The control automatically expands or contracts as the end user resizes the window.

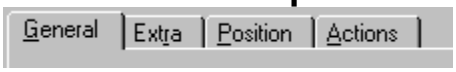
Fixed To set a specific position and size, mark the **Fixed** choices.

The measurement units for these boxes are dialog units. These provide a relative screen measure. Dialog units are a relative measure based on the default system font character size. Windows automatically repositions the control proportionally at different screen resolutions.

Actions

There are no prompts for the Actions Tab for this control.

Box Control Properties



Click on a TAB to see its help

The box control allows you to place a square or rectangle in your window or report. You may fill it with a color, specify a border color, and specify that the borders be rounded. The box control cannot receive focus, nor can it generate events.

General

Use Type a field equate label to reference the control in executable code.

Mode

Hide Makes the control invisible at the time Windows would initially display it. Windows actually creates the control--it just doesn't display it on screen. The **Window Formatter** places the HIDE attribute on the control. Use the UNHIDE statement to display the control.

Disable Disables or 'grays-out' the control when your program initially displays it. The **Window Formatter** places the DISABLE attribute on the control. Use the ENABLE statement to allow the user access to the control.

Scroll Specifies whether the control should move with the window when the user scrolls the window. By default, (unchecked), the control does not move with the window. Leave the **Scroll** box unchecked to create a control that stays fixed when the user scrolls the window. The **Window Formatter** places the SCROLL attribute on the control when checked.

Extra

Fill Allows you to specify the **Fill** color--the inside color. Check the **Fill** check box, unless you wish the box to be transparent. Then press the **Fill Color...** button. The standard **Color** dialog appears. Select a color by clicking on the color sample square, or add a custom color.

Border Allows you to specify the **Border** color--the outside line color. Check the **Border** check box, unless you wish the box to have no border. Then press the **Border Color...** button. The standard **Color** dialog appears. Select a color by clicking on the color sample square, or add a custom color.

Round Allows you to specify that the box control should have rounded edges.

Position

Allows you to set the location and size of a control.

The **Position** dialog allows you to specify the AT (set control position and size in window) attribute. Filling in the attribute manually is optional--you may set position visually, dragging with the mouse in the **Window Formatter**.

To set a *precise* screen location or size for your control, specify fixed coordinates via this dialog. To set

the location and size, choose from the following options for the **Top Left Corner**, the **Width** of the control, and its **Height**.

Default — This instructs Windows to set a value which depends on the end user's system. Default size for a control depends on the size of the system font.

Full — Sizes a control to expand to the full width or height of the window. The control automatically expands or contracts as the end user resizes the window.

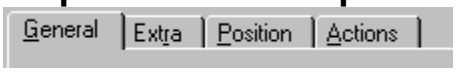
Fixed — To set a specific position and size, mark the **Fixed** choices.

The measurement units for these boxes are dialog units. These provide a relative screen measure. Dialog units are a relative measure based on the default system font character size. Windows automatically repositions the control proportionally at different screen resolutions.

Actions

There are no prompts for the Actions Tab for this control.

Ellipse Control Properties



Click on a TAB to see its help

The ellipse control allows you to place a circle or ellipse in your window or report. You may fill the ellipse with a color, and specify a border color. The ellipse control cannot receive focus, nor can it generate events.

General

Use Type a field equate label to reference the control in executable code.

Mode

Hide Makes the control invisible at the time Windows would initially display it. Windows actually creates the control--it just doesn't display it on screen. The **Window Formatter** places the HIDE attribute on the control. Use the UNHIDE statement to display the control.

Disable Disables or 'grays-out' the control when your program initially displays it. The **Window Formatter** places the DISABLE attribute on the control. Use the ENABLE statement to allow the user access to the control.

Scroll Specifies whether the control should move with the window when the user scrolls the window. By default, (unchecked), the control does not move with the window. Leave the **Scroll** box unchecked to create a control that stays fixed when the user scrolls the window. The **Window Formatter** places the SCROLL attribute on the control when checked.

Extra

Fill Allows you to specify the **Fill** color--the inside color. Check the **Fill** check box, unless you wish the ellipse to be transparent. Then press the **Fill Color...** button. The standard **Color** dialog appears. Select a color by clicking on the color sample square, or add a custom color.

Border Allows you to specify the **Border** color--the outside line color. Check the **Border** check box, unless you wish the ellipse to have no border. Then press the **Border Color...** button. The standard **Color** dialog appears. Select a color by clicking on the color sample square, or add a custom color.

Position

Allows you to set the location and size of a control.

The **Position** dialog allows you to specify the AT (set control position and size in window) attribute. Filling in the attribute manually is optional--you may set position visually, dragging with the mouse in the **Window Formatter**.

To set a *precise* screen location or size for your control, specify fixed coordinates via this dialog. To set the location and size, choose from the following options for the **Top Left Corner**, the **Width** of the control, and its **Height**.

Default	This instructs Windows to set a value which depends on the end user's system. Default size for a control depends on the size of the system font.
Full	Sizes a control to expand to the full width or height of the window. The control automatically expands or contracts as the end user resizes the window.
Fixed	To set a specific position and size, mark the **Fixed** choices. The measurement units for these boxes are dialog units. These provide a relative screen measure. Dialog units are a relative measure based on the default system font character size. Windows automatically repositions the control proportionally at different screen resolutions.

Actions

There are no prompts for the Actions Tab for this control.

Entry Box Control Properties

Click on a TAB to see its help

An entry box allows you to process data input from the user. The data entry control is a specialized form of Windows edit box. It can help you automatically validate data as the user enters it in a dialog box.

General

Picture

The Picture field takes a display picture token that specifies input format. You may press the ellipsis (...) button next to the field to pick a display picture from the **Edit Picture String** dialog.

You may check the user entry against the picture at two points: as the user types the data in, or when the user closes the dialog box. Checking the data as the user types it incurs a slight performance penalty. To do so, check the **Entry Patterns** box in the **Window Properties** dialog for the window in which the entry box resides. This turns the MASK attribute on for *all* controls in the window.

If the MASK attribute is off, entry checking takes place when the user moves the focus to another control (for example, by TABBING to another field).

If the user types in data in a format different from the picture, the program will attempt to determine the format, then convert it to match the picture (if no MASK was specified). For example, if the user types 'January 1, 1995' and the picture is @D1, the program formats the input to "1/1/95. If the program cannot determine the entry format, it will *not* update the USE variable. The user will receive an audible prompt (beep), and the focus will return to the entry control, ready for additional input.

Use

Place a variable or field equate label in the **Use** field. You may specify a variable which receives the value that the user types. Or, a field equate label which references the entry box in program statements.

Justification

Specify left, center, right, decimal, or default justification. Default justification matches that specified in the data dictionary, if applicable. If you use decimal justification, you set the Offset to allow display of digits to the right of the decimal point.

Offset

Specify an indentation value for the text, in dialog units. The indention is in the opposite direction from the justification.

Mode

Hide

Makes the control invisible at the time Windows would initially display it. Windows actually creates the control--it just doesn't display it on screen. The **Window Formatter** places the HIDE attribute on the control. Use the UNHIDE statement to display the control.

Disable

Disables or 'grays-out' the control when your program initially displays it. The **Window Formatter** places the DISABLE attribute on the control. Use the ENABLE statement to allow the user access to the control.

Scroll

Specifies whether the control should move with the window when the user scrolls the window. By default, (unchecked), the control does not move with the window.

Leave the **Scroll** box unchecked to create a control that stays fixed when the user scrolls the window. The **Window Formatter** places the SCROLL attribute on the control when checked.

Skip

Instructs the **Window Formatter** to omit the control from the Tab Order. When the user TABS from field to field in the dialog box, Windows will not give the control focus. This is useful for seldom-used data fields. The **Window Formatter** will place the SKIP attribute on the control.

Extra

Case

Specify case attributes for the entry field. The entry box can automatically convert characters from one case to another. **Uppercase** automatically converts to all caps. **Capitalize** converts to proper case. **Default** (no attribute) accepts input in the case the user types it.

Entry Mode

Optionally specify an **Entry Mode**. Choose either **Insert , Overwrite** or **As Is**. The **Entry Mode** applies only for windows with the MASK attribute set

Options

Set the Entry flags. There are three entry flags you may toggle on or off independently.

Required - (the REQ attribute) specifies that the control may not be left blank or zero.

Immediate - (the IMM attribute) specifies immediate event generation whenever the user presses any key. See Also: How to complete an entry field when the last character is entered.

Read Only - (the READONLY attribute) prevents data entry in this control. Use this to declare display-only data.

Password - (the PASSWORD attribute) specifies non-display of data entered in this control. When the user types in data, asterisks are displayed for each character entered.

Drop ID

To specify the type of Drag operations this control will accept, type up to 16 *signatures*, separated by commas. The **Window Formatter** adds the **DROPID** attribute to the control, which indicates the control is a valid target for the drag and drop operations identified by the signatures.

Help

Cursor

The *Cursor* field (the CURSOR attribute) allows you to specify an alternate shape for the cursor when the user passes the cursor over the control. The drop-down list provides standard cursor choices such as I-Beam and Crosshair. To select an external cursor file (whose extension must be .CUR), choose **Select File** from the drop-down list, then pick the file using the standard file dialog.

Help ID

The **Help ID** field (the HLP attribute) takes a string constant specifying the key for accessing a specific topic in the Help document. This may be either a Help keyword or a context string.

A Help keyword is a word or phrase indexed so that the user may search for it in the *Help* **Search** dialog. When you fill in the HLP attribute for a button, if the entry

box has focus, when the user presses F1, the help file opens to the referenced topic. If more than one topic matches a keyword, the search dialog appears.

When referencing a context string in the **Help ID** field, you must identify it with a leading tilde (~).

Message The **Message** field (the MSG attribute) allows you to specify text to display in the first zone of the status bar when the control has focus.

Tip The **TIP** attribute on a control specifies the text to display in a "balloon help" box when the mouse cursor pauses over the control. Although there is no specific limit on the number of characters, the *string* should not be longer than can be displayed on the screen.

Position

Allows you to set the location and size of a control.

The **Position** dialog allows you to specify the AT (set control position and size in window) attribute. Filling in the attribute manually is optional--you may set position visually, dragging with the mouse in the **Window Formatter**.

To set a *precise* screen location or size for your control, specify fixed coordinates via this dialog. To set the location and size, choose from the following options for the **Top Left Corner**, the **Width** of the control, and its **Height**.

Default This instructs Windows to set a value which depends on the end user's system. Default size for a control depends on the size of the system font.

Full Sizes a control to expand to the full width or height of the window. The control automatically expands or contracts as the end user resizes the window.

Fixed To set a specific position and size, mark the **Fixed** choices.

The measurement units for these boxes are dialog units. These provide a relative screen measure. Dialog units are a relative measure based on the default system font character size. Windows automatically repositions the control proportionally at different screen resolutions.

Actions

The **Actions** tab prompts are all from the templates, in other words, the prompts you see here vary with the template used to create the control. Following are the standard action prompts for all entry controls.

The standard **Actions** prompts are designed to provide data validation support for your entry controls. The tab is divided into two parallel sections. The **When the Control is Selected** section provides validation when the control *receives* focus (when the user TABS onto, or mouse CLICKS the control). The **When the Control is Accepted** section provides data validation when the control *loses* focus after data have been entered in it. The control loses focus when the user TABS off the control, mouse CLICKS to a different control or window, or closes the window without canceling. The two sections are not mutually exclusive, so you can provide validation at both points.

Lookup Key Type a key label from the *lookup* file, or press the ellipsis (...) button to select a key from the **Select Key** dialog.

A lookup file is a file which contains all the valid values for the entry field, which are directly accessible through a unique.

For example, a file containing all of the customer numbers for your application could be a lookup file. The key label could be CUS:KeyCustNumber.

Tip: This lookup validation works best with a single component unique key.

Lookup Field Type the label of a component field of the lookup key, or press the ellipsis (...) button to select a field from the **Select component from key** dialog.

This is the field within the key that contains the same value being validated. Ideally, this field is the only component of a unique key.

Lookup Procedure Type a procedure name, or choose an existing procedure from the drop down list.

This is the procedure that is called when the user enters an invalid value, and the lookup specified above *fails*. The usual purpose of this procedure is to allow the user to choose a valid value from the lookup file.

Select procedures (or Browse procedures) generated by Clarion's Wizards) are appropriate for this purpose. Alternatively, you may hand-code a procedure.

Advanced Calls the **Embedded Source** dialog. The only embed point shown is after the code generated to call the lookup procedure specified above. For more embed points, and further customization, press the **Embeds** button.

Perform lookup during non-stop select
Checking this box tells Clarion to perform the validation when the *window* is accepted, even if the *entry control* never received focus. From a practical viewpoint, checking this box prevents the user from entering blanks by virtue of having pressed the window's "**OK** button" without ever TABBING or CLICKING onto the entry field.

This option is only applicable to the **When the Control is Accepted** section.

Force Window Refresh when Accepted
Checking this box ensures that everything (including formulas and other entry fields) on the window is current and up-to-date when the user TABS off this entry control.

Files Accesses the **File Schematic Definition** dialog for this procedure.

Embeds Accesses the **Embedded Source** dialog for points surrounding the event handling for this control only.

Font

Calls the **Select Font** dialog which allows you to select the font (typeface), size, style (such as bold or italic), color, and font effects (underline and strikeout) for the selected control or window. As you choose options, the dialog box displays a sample of the selected font.

Button Control Properties

Click on a TAB to see its help

A push button (BUTTON) is a rectangular area containing text and/or a picture. When the user presses the button, it should execute a command described by the text or picture.

General

Parameter — The text to display on the button. Place an ampersand (&) before the character to act as the accelerator key for the button--this underlines the character as it appears on the button.

Tip: **Microsoft recommends you do *not* place an accelerator key on buttons labeled 'OK,' or 'Cancel.'**

Use — The field equate label references the button in program statements.

Mode

Hide — Makes the control invisible at the time Windows would initially display it. Windows actually creates the control--it just doesn't display it on screen. The **Window Formatter** places the HIDE attribute on the control. Use the UNHIDE statement to display the control.

Disable — Disables or 'grays-out' the control when your program initially displays it. The **Window Formatter** places the DISABLE attribute on the control. Use the ENABLE statement to allow the user access to the control.

Scroll — Specifies whether the control should move with the window when the user scrolls the window. By default, (unchecked), the control does not move with the window. Leave the **Scroll** box unchecked to create a control that stays fixed when the user scrolls the window. The **Window Formatter** places the SCROLL attribute on the control when checked.

Skip — Instructs the **Window Formatter** to omit the control from the Tab Order. When the user TABS from field to field in the dialog box, Windows will not give the control focus. This is useful for seldom-used data fields. The **Window Formatter** will place the SKIP attribute on the control.

Extra

Icon — In the **Icon** field, optionally select a standard icon or icon file. This displays a small bitmap on the button face (clipping or centering the bitmap as necessary).

To select a standard icon, choose one of the named items in the drop-down list. To select an icon file (whose extension must be .ICO), choose **Select File** from the drop-down list, then pick the file using the standard file dialog.

Immediate — Allows you to create a button control which repeats the executable action continuously, for as long as the user holds the button down. Normally, buttons generate an event only after the user presses *and releases* the mouse. See also: the IMM attribute.

Required — Specifies that when pressed, your program automatically checks that all entry

controls with the REQ attribute are neither blank nor zero. A button with this attribute is a 'required fields check' button.

Specify this type of button when a window also contains an ENTRY or TEXT control field with the REQ attribute (or else use the INCOMPLETE() function to test the ENTRY controls). When the user presses a button with the REQ attribute and an ENTRY field is blank or zero, the first required control which is blank or zero receives the focus.

Default Button "Presses" the button when the user presses the ENTER key. A heavy border appears around the button at runtime, to signal the default button to the user. In general, place the DEFAULT attribute on a button if it represents the most likely action the user will wish to carry out. Place only one default button in a window.

STD ID Executes a standard action when the end user presses the button. See also: STD .

Drop ID To specify the type of Drag operations this control will accept, type up to 16 *signatures*, separated by commas. The **Window Formatter** adds the **DROPID** attribute to the control, which indicates the control is a valid target for the drag and drop operations identified by the signatures.

Help

Cursor The *Cursor* field (the CURSOR attribute) allows you to specify an alternate shape for the cursor when the user passes the mouse over the button. The drop-down list provides standard cursor choices such as I-Beam and Crosshair. To select an external cursor file (whose extension must be .CUR), choose **Select File** from the drop-down list, then pick the file using the standard file dialog.

Help ID The **Help ID** field (the HLP attribute) takes a string constant specifying the key for accessing a specific topic in the Help document. This may be either a Help keyword or a context string.

A Help keyword is a word or phrase indexed so that the user may search for it in the *Help* **Search** dialog. When you fill in the HLP attribute for a button, if the button has focus, when the user presses F1, the help file opens to the referenced topic. If more than one topic matches a keyword, the search dialog appears.

When referencing a context string in the **Help ID** field, you must identify it with a leading tilde (~).

Message The **Message** field (the MSG attribute) allows you to specify text to display in the first zone of the status bar when the control has focus.

Tip The **TIP** attribute on a control specifies the text to display in a "balloon help" box when the mouse cursor pauses over the control. Although there is no specific limit on the number of characters, the *string* should not be longer than can be displayed on the screen.

Position

Allows you to set the location and size of a control.

The **Position** dialog allows you to specify the AT (set control position and size in window) attribute. Filling in the attribute manually is optional--you may set position visually, dragging with the mouse in the **Window Formatter**.

To set a *precise* screen location or size for your control, specify fixed coordinates via this dialog. To set the location and size, choose from the following options for the **Top Left Corner**, the **Width** of the control, and its **Height**.

Default This instructs Windows to set a value which depends on the end user's system. Default size for a control depends on the size of the system font.

Full Sizes a control to expand to the full width or height of the window. The control automatically expands or contracts as the end user resizes the window.

Fixed To set a specific position and size, mark the **Fixed** choices.

The measurement units for these boxes are dialog units. These provide a relative screen measure. Dialog units are a relative measure based on the default system font character size. Windows automatically repositions the control proportionally at different screen resolutions.

Font

Calls the **Select Font** dialog which allows you to select the font (typeface), size, style (such as bold or italic), color, and font effects (underline and strikeout) for the selected control or window. As you choose options, the dialog box displays a sample of the selected font.

Actions

Use the Actions tab to provide functionality to your button. Filling in these prompts causes the button to execute an action when the user presses the button.

When Pressed From the drop down list, choose *Call a Procedure, Run a Program*, or *No Special Action*.

The procedure or program you specify executes when the user pushes the button. The choices are:

Call a Procedure You must specify the **Procedure Name**, and whether the procedure will **Initiate a Thread**.

Procedure Name From the **Procedure Name** drop down list, choose an existing procedure name, or type a new procedure name. A new procedure appears as a "ToDo" item in your Application Tree.

Initiate a Thread Optionally check the **Initiate a Thread** box. If the procedure initiates a thread, specify the Thread Stack size. Clarion uses the START function to initiate a new execution thread. If the procedure initiates a thread, you cannot specify **Parameters** or **Requested File Action**. If the procedure does not initiate a thread, you can specify **Parameters**, **Requested File Action**, or both.

Tip: A BUTTON on an application frame toolbar that calls an MDI child procedure must initiate a thread.

Thread Stack Accept the default value in the **Thread Stack** spin box unless you have extraordinary program requirements. To change the value, type in a new value or

click on the spin box arrows.

Parameters In the **Parameters** field, optionally type a list of variables or data structures passed to the procedure.

Requested File Action

From the drop down list, optionally select ***None, Insert, Change, Delete***, or ***Select***. The default selection is ***None***. The Global Request variable gets the selected value. The called procedure can then check the value of the Global Request variable and perform the requested file action.

Run a Program You must specify the **Program Name**, and optionally, any parameters.

Program Name Type the program name. The program must reside in a .DLL or .LIB defined in your application's project (.PRJ) file.

Parameters Optionally type a list of values that are passed to the program.

No Special Action Choose this option if you are providing your button's functionality with another method, such as embedded source, or an STD ID (see *Extra Tab* above).

Note: You may combine a procedure or program call with embedded source, but not with an STD ID.

Files Accesses the **File Schematic Definition** dialog for this procedure.

Embeds Allows you to embed source code at points surrounding the event handling for this button only.

Prompt Control Properties

Click on a TAB to see its help

The Prompt control allows you to place a specialized string object on screen which automatically provides an accelerator or mnemonic access key to the next active control following the prompt.

General

Parameter — Specify a string constant by typing it in the **Parameter** box. If the control is to display a variable, type a picture token in this box.

Use — Type a field equate label to reference the control in executable code, or the name of a variable

Justification — Specify left, center, right, or default justification. Default justification matches that specified in the data dictionary, if applicable.

Mode

Hide — Makes the control invisible at the time Windows would initially display it. Windows actually creates the control--it just doesn't display it on screen. The **Window Formatter** places the HIDE attribute on the control. Use the UNHIDE statement to display the control.

Disable — Disables or 'grays-out' the control when your program initially displays it. The **Window Formatter** places the DISABLE attribute on the control. Use the ENABLE statement to allow the user access to the control.

Scroll — Specifies whether the control should move with the window when the user scrolls the window. By default, (unchecked), the control does not move with the window. Leave the **Scroll** box unchecked to create a control that stays fixed when the user scrolls the window. The **Window Formatter** places the SCROLL attribute on the control when checked.

Skip — Instructs the **Window Formatter** to omit the control from the Tab Order. When the user TABS from field to field in the dialog box, Windows will not give the control focus. This is useful for seldom-used data fields. The **Window Formatter** will place the SKIP attribute on the control.

Extra

Drop ID — To specify the type of Drag operations this control will accept, type up to 16 *signatures*, separated by commas. The **Window Formatter** adds the **DROPID** attribute to the control, which indicates the control is a valid target for the drag and drop operations identified by the signatures.

Help

Cursor — The *Cursor* field (the CURSOR attribute) allows you to specify an alternate shape for the cursor when the user passes the mouse over the PROMPT. The drop-down list provides standard cursor choices such as I-Beam and Crosshair. To select an external cursor file (whose extension must be .CUR), choose **Select File** from the drop-down list, then pick the file using the standard file dialog.

Position

Allows you to set the location and size of a control.

The **Position** dialog allows you to specify the AT (set control position and size in window) attribute. Filling in the attribute manually is optional--you may set position visually, dragging with the mouse in the **Window Formatter**.

To set a *precise* screen location or size for your control, specify fixed coordinates via this dialog. To set the location and size, choose from the following options for the **Top Left Corner**, the **Width** of the control, and its **Height**.

Default — This instructs Windows to set a value which depends on the end user's system. Default size for a control depends on the size of the system font.

Full — Sizes a control to expand to the full width or height of the window. The control automatically expands or contracts as the end user resizes the window.

Fixed — To set a specific position and size, mark the **Fixed** choices.

The measurement units for these boxes are dialog units. These provide a relative screen measure. Dialog units are a relative measure based on the default system font character size. Windows automatically repositions the control proportionally at different screen resolutions.

Font

Calls the **Select Font** dialog which allows you to select the font (typeface), size, style (such as bold or italic), color, and font effects (underline and strikeout) for the selected control or window. As you choose options, the dialog box displays a sample of the selected font.

Actions

There are no prompts for the Actions Tab for this control.

Option Control Properties

Click on a TAB to see its help

The **OPTION** control declares a group of RADIO controls that offer the user a list of mutually exclusive choices. The multiple RADIO controls in the OPTION structure define the choices offered to the user.

General

Parameter Specify a string constant by typing it in the **Parameter** box. If the control is to display a variable, type a picture token in this box.

Use Type a field equate label to reference the control in executable code, or the name of a variable

Mode

Hide Makes the control invisible at the time Windows would initially display it. Windows actually creates the control--it just doesn't display it on screen. The **Window Formatter** places the HIDE attribute on the control. Use the UNHIDE statement to display the control.

Disable Disables or 'grays-out' the control when your program initially displays it. The **Window Formatter** places the DISABLE attribute on the control. Use the ENABLE statement to allow the user access to the control.

Scroll Specifies whether the control should move with the window when the user scrolls the window. By default, (unchecked), the control does not move with the window. Leave the **Scroll** box unchecked to create a control that stays fixed when the user scrolls the window. The **Window Formatter** places the SCROLL attribute on the control when checked.

Skip Instructs the **Window Formatter** to omit the control from the Tab Order. When the user TABS from field to field in the dialog box, Windows will not give the control focus. This is useful for seldom-used data fields. The **Window Formatter** will place the SKIP attribute on the control.

Extra

Boxed The **BOXED** attribute specifies a single-track border around the OPTION structure. The parameter text of the OPTION control appears in a gap at the top of the border box. If BOXED is omitted, the *text* parameter is not displayed on screen

Drop ID To specify the type of Drag operations this control will accept, type up to 16 *signatures*, separated by commas. The **Window Formatter** adds the **DROPID** attribute to the control, which indicates the control is a valid target for the drag and drop operations identified by the signatures.

Help

Cursor

The *Cursor* field (the CURSOR attribute) allows you to specify an alternate shape for the cursor when the user passes the cursor over the control. The drop-down list provides standard cursor choices such as I-Beam and Crosshair. To select an external cursor file (whose extension must be .CUR), choose **Select File** from the drop-down list, then pick the file using the standard file dialog.

Help ID

The **Help ID** field (the HLP attribute) takes a string constant specifying the key for accessing a specific topic in the Help document. This may be either a Help keyword or a context string.

A Help keyword is a word or phrase indexed so that the user may search for it in the *Help* **Search** dialog. When you fill in the HLP attribute for a button, if the entry box has focus, when the user presses F1, the help file opens to the referenced topic. If more than one topic matches a keyword, the search dialog appears.

When referencing a context string in the **Help ID** field, you must identify it with a leading tilde (~).

Message

The **Message** field (the MSG attribute) allows you to specify text to display in the first zone of the status bar when the control has focus.

Tip

The **TIP** attribute on a control specifies the text to display in a "balloon help" box when the mouse cursor pauses over the control. Although there is no specific limit on the number of characters, the *string* should not be longer than can be displayed on the screen.

Position

Allows you to set the location and size of a control.

The **Position** dialog allows you to specify the AT (set control position and size in window) attribute. Filling in the attribute manually is optional--you may set position visually, dragging with the mouse in the **Window Formatter**.

To set a *precise* screen location or size for your control, specify fixed coordinates via this dialog. To set the location and size, choose from the following options for the **Top Left Corner**, the **Width** of the control, and its **Height**.

Default

This instructs Windows to set a value which depends on the end user's system. Default size for a control depends on the size of the system font.

Full

Sizes a control to expand to the full width or height of the window. The control automatically expands or contracts as the end user resizes the window.

Fixed

To set a specific position and size, mark the **Fixed** choices.

The measurement units for these boxes are dialog units. These provide a relative screen measure. Dialog units are a relative measure based on the default system font character size. Windows automatically repositions the control proportionally at different screen resolutions.

Font

Calls the **Select Font** dialog which allows you to select the font (typeface), size, style (such as bold or italic), color, and font effects (underline and strikeout) for the selected control or window. As you choose

options, the dialog box displays a sample of the selected font.

Actions

There are no prompts for the Actions Tab for this control.

Check Box Control Properties

Click on a TAB to see its help

The check box provides an attractive way to display a yes/no choice for a record field--the alternative might be an entire column that repeats "one," "yes," or even ".T." for each record.

General

Parameter Specify a string constant to display by typing it in the **Parameter** box.

Use Specify a numeric variable. The check box places a value of 1 in the numeric variable if the end user turns on the check box, zero if off.

Justification Sets alignment for the label which appears next to the control. Choose **Left** or **Right**. If you select an icon, no label appears at run time.

Mode

Hide Makes the control invisible at the time Windows would initially display it. Windows actually creates the control--it just doesn't display it on screen. The **Window Formatter** places the HIDE attribute on the control. Use the UNHIDE statement to display the control.

Disable Disables or 'grays-out' the control when your program initially displays it. The **Window Formatter** places the DISABLE attribute on the control. Use the ENABLE statement to allow the user access to the control.

Scroll Specifies whether the control should move with the window when the user scrolls the window. By default, (unchecked), the control does not move with the window. Leave the **Scroll** box unchecked to create a control that stays fixed when the user scrolls the window. The **Window Formatter** places the SCROLL attribute on the control when checked.

Skip Instructs the **Window Formatter** to omit the control from the Tab Order. When the user TABS from field to field in the dialog box, Windows will not give the control focus. This is useful for seldom-used data fields. The **Window Formatter** will place the SKIP attribute on the control.

Extra

Icon In the **Icon** field, optionally select a standard icon or icon file. This displays a small bitmap next to the check box (clipping or centering the bitmap as necessary).

To select a standard icon, choose one of the named items in the drop-down list. To select an icon file (whose extension must be .ICO), choose **Select File** from the drop-down list, then pick the file using the standard file dialog.

Drop ID To specify the type of Drag operations this control will accept, type up to 16 *signatures*, separated by commas. The **Window Formatter** adds the **DROPID** attribute to the control, which indicates the control is a valid target for the drag and drop operations identified by the signatures.

Help

Cursor

The *Cursor* field (the CURSOR attribute) allows you to specify an alternate shape for the cursor when the user passes the cursor over the control. The drop-down list provides standard cursor choices such as I-Beam and Crosshair. To select an external cursor file (whose extension must be .CUR), choose **Select File** from the drop-down list, then pick the file using the standard file dialog.

Help ID

The **Help ID** field (the HLP attribute) takes a string constant specifying the key for accessing a specific topic in the Help document. This may be either a Help keyword or a context string.

A Help keyword is a word or phrase indexed so that the user may search for it in the *Help* **Search** dialog. When you fill in the HLP attribute for a button, if the entry box has focus, when the user presses F1, the help file opens to the referenced topic. If more than one topic matches a keyword, the search dialog appears.

When referencing a context string in the **Help ID** field, you must identify it with a leading tilde (~).

Message

The **Message** field (the MSG attribute) allows you to specify text to display in the first zone of the status bar when the control has focus.

Tip

The **TIP** attribute on a control specifies the text to display in a "balloon help" box when the mouse cursor pauses over the control. Although there is no specific limit on the number of characters, the *string* should not be longer than can be displayed on the screen.

Position

Allows you to set the location and size of a control.

The **Position** dialog allows you to specify the AT (set control position and size in window) attribute. Filling in the attribute manually is optional--you may set position visually, dragging with the mouse in the **Window Formatter**.

To set a *precise* screen location or size for your control, specify fixed coordinates via this dialog. To set the location and size, choose from the following options for the **Top Left Corner**, the **Width** of the control, and its **Height**.

Default

This instructs Windows to set a value which depends on the end user's system. Default size for a control depends on the size of the system font.

Full

Sizes a control to expand to the full width or height of the window. The control automatically expands or contracts as the end user resizes the window.

Fixed

To set a specific position and size, mark the **Fixed** choices.

The measurement units for these boxes are dialog units. These provide a relative screen measure. Dialog units are a relative measure based on the default system font character size. Windows automatically repositions the control proportionally at different screen resolutions.

Font

Calls the **Select Font** dialog which allows you to select the font (typeface), size, style (such as bold or italic), color, and font effects (underline and strikeout) for the selected control or window. As you choose options, the dialog box displays a sample of the selected font.

Actions

The Check Box **Actions** tab leads to other dialogs allowing you to name variables and change their values when the end user checks or unchecks the box. Additionally, you can HIDE or UNHIDE other controls in the window.

Two group boxes with two pairs of buttons appear on the **Actions** tab. These buttons set the behavior for **When the Check Box is Checked**, and **When the Check Box is Unchecked**.

Assign Values	Opens the **Assign Values** dialog where you can assign values to variables based on the checked or unchecked state of the check box.
Hide/Unhide Controls	Opens the **Hide/Unhide Controls** dialog where you can specify window controls to hide or unhide based on the checked or unchecked state of the check box.
Files	Accesses the **File Schematic Definition** dialog for this procedure.
Embeds	Allows you to embed source code at points surrounding the event handling for this check box only.

Assign Values Dialog

Allows you to assign values to variables based on the checked or unchecked state of a check box. You may specify multiple assignments. Press the **Insert** button to add a new assignment.

Variable to Assign In the entry box, type a variable name, or press the ellipsis (...) button to choose or create a data dictionary field or a memory variable with the **Select Field** dialog.

Value to Assign In the entry box, type the value to assign to the variable. You can then add code to your program to take appropriate action based on the run time value of the variable(s).

Hide/Unhide Controls Dialog

Allows you to specify window controls to hide or unhide based on the checked or unchecked state of a check box. You may specify multiple controls to hide/unhide. Press the **Insert** button to add a new hide/unhide action to the list.

Control to hide/unhide From the drop down list, choose the control to HIDE or UNHIDE.

Hide or unhide control From the drop down list, choose **Hide** or **Unhide**.

Group Control Properties

Click on a TAB to see its help

A Group control places a box around two or more controls. It visually associates the controls for the user, and allows you to address all the controls as one entity -- making it easy, for example, to disable all at once.

General

Parameter Specify a string constant by typing it in the **Parameter** box.

Use Type a field equate label to reference the control in executable code, or the name of a variable

Mode

Hide Makes the control invisible at the time Windows would initially display it. Windows actually creates the control--it just doesn't display it on screen. The **Window Formatter** places the HIDE attribute on the control. Use the UNHIDE statement to display the control.

Disable Disables or 'grays-out' the control when your program initially displays it. The **Window Formatter** places the DISABLE attribute on the control. Use the ENABLE statement to allow the user access to the control.

Scroll Specifies whether the control should move with the window when the user scrolls the window. By default, (unchecked), the control does not move with the window. Leave the **Scroll** box unchecked to create a control that stays fixed when the user scrolls the window. The **Window Formatter** places the SCROLL attribute on the control when checked.

Skip Instructs the **Window Formatter** to omit the control from the Tab Order. When the user TABS from field to field in the dialog box, Windows will not give the control focus. This is useful for seldom-used data fields. The **Window Formatter** will place the SKIP attribute on the control.

Extra

Boxed Specifies whether to draw a visible box, containing the caption, around the grouped controls. When not checked, the Group box, including its caption, is invisible.

Drop ID To specify the type of Drag operations this control will accept, type up to 16 *signatures*, separated by commas. The **Window Formatter** adds the **DROPID** attribute to the control, which indicates the control is a valid target for the drag and drop operations identified by the signatures.

Help

Cursor The *Cursor* field (the CURSOR attribute) allows you to specify an alternate shape for the cursor when the user passes the cursor over the control. The drop-down

list provides standard cursor choices such as I-Beam and Crosshair. To select an external cursor file (whose extension must be .CUR), choose **Select File** from the drop-down list, then pick the file using the standard file dialog.

Help ID

The **Help ID** field (the HLP attribute) takes a string constant specifying the key for accessing a specific topic in the Help document. This may be either a Help keyword or a context string.

A Help keyword is a word or phrase indexed so that the user may search for it in the *Help* **Search** dialog. When you fill in the HLP attribute for a button, if the entry box has focus, when the user presses F1, the help file opens to the referenced topic. If more than one topic matches a keyword, the search dialog appears.

When referencing a context string in the **Help ID** field, you must identify it with a leading tilde (~).

Message

The **Message** field (the MSG attribute) allows you to specify text to display in the first zone of the status bar when the control has focus.

Tip

The **TIP** attribute on a control specifies the text to display in a "balloon help" box when the mouse cursor pauses over the control. Although there is no specific limit on the number of characters, the *string* should not be longer than can be displayed on the screen.

Position

Allows you to set the location and size of a control.

The **Position** dialog allows you to specify the AT (set control position and size in window) attribute. Filling in the attribute manually is optional--you may set position visually, dragging with the mouse in the **Window Formatter**.

To set a *precise* screen location or size for your control, specify fixed coordinates via this dialog. To set the location and size, choose from the following options for the **Top Left Corner**, the **Width** of the control, and its **Height**.

Default

This instructs Windows to set a value which depends on the end user's system. Default size for a control depends on the size of the system font.

Full

Sizes a control to expand to the full width or height of the window. The control automatically expands or contracts as the end user resizes the window.

Fixed

To set a specific position and size, mark the **Fixed** choices.

The measurement units for these boxes are dialog units. These provide a relative screen measure. Dialog units are a relative measure based on the default system font character size. Windows automatically repositions the control proportionally at different screen resolutions.

Font

Calls the **Select Font** dialog which allows you to select the font (typeface), size, style (such as bold or italic), color, and font effects (underline and strikeout) for the selected control or window. As you choose options, the dialog box displays a sample of the selected font.

Actions

There are no prompts for the Actions Tab for this control.

List Control Properties

Click on a TAB to see its help

The List box is useful for presenting a great number of choices for the user. It can convey a large amount of data in a small area, which has led to its use as an all purpose data control. Using Clarion for Windows, you can create list boxes which look like spreadsheet grids, perform drag-and-drop tasks, and more.

This section only discusses *placing* the list box. After you place a list box, you must format it. See the **List Box Formatter** dialog for more information on formatting and adding additional functionality to your list boxes.

See also:

How to Create a List Box

General

Use — Place a variable or field equate label in the **Use** field. You may specify the variable which will receive the value that the user selects. Or, a field equate label to reference the list box in program statements.

From — Fill the **From** field with the origin of the list data. Generally, this is the label of a QUEUE structure.

Drop — Specifies whether this should be a regular or drop-down list box. Place a zero in the **Drop** field for a normal list box. To create a drop-down list box, type the number of drop-down elements you wish to be visible.

Justification — Specify left, center, right, decimal, or default justification. Default justification matches that specified in the data dictionary, if applicable. If you use decimal justification, you set the Offset to allow display of digits to the right of the decimal point.

Offset — Specify an indentation value for the list box item text, in dialog units.

Mode

Hide — Makes the control invisible at the time Windows would initially display it. Windows actually creates the control--it just doesn't display it on screen. The **Window Formatter** places the HIDE attribute on the control. Use the UNHIDE statement to display the control.

Disable — Disables or 'grays-out' the control when your program initially displays it. The **Window Formatter** places the DISABLE attribute on the control. Use the ENABLE statement to allow the user access to the control.

Scroll — Specifies whether the control should move with the window when the user scrolls the window. By default, (unchecked), the control does not move with the window. Leave the **Scroll** box unchecked to create a control that stays fixed when the user scrolls the window. The **Window Formatter** places the SCROLL attribute on the control when checked.

Skip Instructs the **Window Formatter** to omit the control from the Tab Order. When the user TABS from field to field in the dialog box, Windows will not give the control focus. This is useful for seldom-used data fields. The **Window Formatter** will place the SKIP attribute on the control.

Extra

Mark Optionally enables multiple item selection. Type in the name of a QUEUE, field or array in the **Mark** field if you wish to allow the user to select more than one item from the list. The QUEUE field will flag the selected items.

VCR Optionally provide VCR controls. Check the **VCR** check box to provide VCR style controls for the list box. Optionally type the field equate label of an entry field to the right of the check box. When the user presses the *Locator* (?) button, the focus shifts to that field. The user may type in data, then press TAB to scroll the list box to the closest matching entry.

Immediate To generate a message event each time the end user moves or resizes the selection bar, check the **Immediate** box. This adds the IMM attribute to the window. You are responsible for the code that executes upon notification of the event.

Select Columns Check this box if you wish to allow the user to highlight a multi-column list box field by field (rather than one row at a time). This provides for spreadsheet grid style movement of the highlight bar. See also: the COLUMN attribute.

Hide Selection Specifies the selection bar appears only when the list box has focus. See also: the NOBAR attribute.

Scroll Bars To add a horizontal scroll bar to your list box, mark the **Horizontal** check box. Scroll bars only appear when the list of items in the list is bigger than the window. To add a vertical scroll bar, check the **Vertical** check box. These options add the HSCROLL, VSCROLL, and HVSCROLL attributes to the control.

Drag ID To specify the type of Drag operations this control will generate, type up to 16 *signatures*, separated by commas. The **DRAGID** attribute specifies the LIST control can serve as a drag-and-drop host. DRAGID works in conjunction with the **DROPID** attribute.

Drop ID To specify the type of Drag operations this control will accept, type up to 16 *signatures*, separated by commas. The **Window Formatter** adds the **DROPID** attribute to the control, which indicates the control is a valid target for the drag and drop operations identified by the signatures.

Help

Cursor The *Cursor* field (the CURSOR attribute) allows you to specify an alternate shape for the cursor when the user passes the cursor over the control. The drop-down list provides standard cursor choices such as I-Beam and Crosshair. To select an external cursor file (whose extension must be .CUR), choose **Select File** from the drop-down list, then pick the file using the standard file dialog.

Help ID

The **Help ID** field (the HLP attribute) takes a string constant specifying the key for accessing a specific topic in the Help document. This may be either a Help keyword or a context string.

A Help keyword is a word or phrase indexed so that the user may search for it in the *Help* **Search** dialog. When you fill in the HLP attribute for a button, if the entry box has focus, when the user presses F1, the help file opens to the referenced topic. If more than one topic matches a keyword, the search dialog appears.

When referencing a context string in the **Help ID** field, you must identify it with a leading tilde (~).

Message

The **Message** field (the MSG attribute) allows you to specify text to display in the first zone of the status bar when the control has focus.

Tip

The **TIP** attribute on a control specifies the text to display in a "balloon help" box when the mouse cursor pauses over the control. Although there is no specific limit on the number of characters, the *string* should not be longer than can be displayed on the screen.

Position

Allows you to set the location and size of a control.

The **Position** dialog allows you to specify the AT (set control position and size in window) attribute. Filling in the attribute manually is optional--you may set position visually, dragging with the mouse in the **Window Formatter**.

To set a *precise* screen location or size for your control, specify fixed coordinates via this dialog. To set the location and size, choose from the following options for the **Top Left Corner**, the **Width** of the control, and its **Height**.

Default

This instructs Windows to set a value which depends on the end user's system. Default size for a control depends on the size of the system font.

Full

Sizes a control to expand to the full width or height of the window. The control automatically expands or contracts as the end user resizes the window.

Fixed

To set a specific position and size, mark the **Fixed** choices.

The measurement units for these boxes are dialog units. These provide a relative screen measure. Dialog units are a relative measure based on the default system font character size. Windows automatically repositions the control proportionally at different screen resolutions.

Font

Calls the **Select Font** dialog which allows you to select the font (typeface), size, style (such as bold or italic), color, and font effects (underline and strikeout) for the selected control or window. As you choose options, the dialog box displays a sample of the selected font.

Actions

There are no prompts for the Actions Tab for this control.

Combo Box Control Properties

Click on a TAB to see its help

The Combo Box combines an entry box with a list box. It is useful for when you expect string data which *usually* should be a member of the list, but which also might not be. The **Window Formatter** allows you to create either a normal combo box, or a drop-down combo box.

This section only discusses *placing* the combo box. After you place it, you must format it. See the **List Box Formatter** dialog for more information on formatting and adding additional functionality to your combo boxes.

See also:

How to Use a Combo Box

FileDropCombo Control template

General

Picture — Specify the picture token for the control. The picture token you specify appears in the format string, for example, "@S10@." Pressing the ellipsis button allows you to select the picture token from the **Edit Picture String** Dialog .

Use — Place a variable or field equate label in the **Use** field. You may specify the variable which will receive the value that the user selects. Or, a field equate label to reference the combo box in program statements.

From — Fill the **From** field with the origin of the list data. Generally, this is the label of a QUEUE structure.

Drop — Specifies whether this should be a regular or drop-down combo box. Place a zero in the **Drop** field for a normal combo box. To create a drop-down combo box, type the number of drop-down elements you wish to be visible. You must resize the combo box after specifying the drop number.

Justification — Specify left, center, right, decimal, or default justification. Default justification matches that specified in the data dictionary, if applicable. If you use decimal justification, you set the Offset to allow display of digits to the right of the decimal point.

Offset — Specify an indentation value for the list box item text, in dialog units.

Mode

Hide — Makes the control invisible at the time Windows would initially display it. Windows actually creates the control--it just doesn't display it on screen. The **Window Formatter** places the HIDE attribute on the control. Use the UNHIDE statement to display the control.

Disable — Disables or 'grays-out' the control when your program initially displays it. The **Window Formatter** places the DISABLE attribute on the control. Use the ENABLE statement to allow the user access to the control.

Scroll — Specifies whether the control should move with the window when the user scrolls

the window. By default, (unchecked), the control does not move with the window. Leave the **Scroll** box unchecked to create a control that stays fixed when the user scrolls the window. The **Window Formatter** places the SCROLL attribute on the control when checked.

Skip Instructs the **Window Formatter** to omit the control from the Tab Order. When the user TABS from field to field in the dialog box, Windows will not give the control focus. This is useful for seldom-used data fields. The **Window Formatter** will place the SKIP attribute on the control.

Extra

Mark Optionally enables multiple item selection. Type in the name of a QUEUE, field or array in the **Mark** field if you wish to allow the user to select more than one item from the list. The QUEUE field will flag the selected items.

VCR Optionally provide VCR controls. Check the **VCR** check box to provide VCR style controls for the list box. Optionally type the name of an entry field to the right of the check box. When the user presses the *Locator* (?) button, the focus shifts to that field. The user may type in data, then press TAB to scroll the list box to the closest matching entry.

Case Specify case attributes for the entry field portion of the combo box. The entry box can automatically convert characters from one case to another. **Uppercase** automatically converts to all caps. **Capitalize** converts to proper case. **Default** (no attribute) accepts input in the case the user types it.

Entry Mode Optionally specify an **Entry Mode** for the entry field portion of the combo box.. Choose either **Insert , Overwrite** or **As Is**. The **Entry Mode** applies only for windows with the MASK attribute set

Immediate To generate a message event each time the end user moves or resizes the selection bar, check the **Immediate** box. This adds the IMM attribute to the window. You are responsible for the code that executes upon notification of the event.

Select Columns Check this box if you wish to allow the user to highlight the list component of a multi-column combo box field by field (rather than one row at a time). This provides for spreadsheet grid style movement of the highlight bar. See also: the COLUMN attribute.

Required - (the REQ attribute) specifies that the control may not be left blank or zero.

Hide Selection Specifies the selection bar appears only when the list box has focus. See also: the NOBAR attribute.

Read Only - (the READONLY attribute) prevents data entry in this control. Use this to declare display-only data.

Scroll Bars To add a horizontal scroll bar to your list box, mark the **Horizontal** check box. Scroll bars only appear when the list of items in the list is bigger than the window. To add a vertical scroll bar, check the **Vertical** check box. These options add the HSCROLL, VSCROLL, and HVSCROLL attributes to the control.

Drag ID To specify the type of Drag operations this control will generate, type up to 16 *signatures*, separated by commas. The **DRAGID** attribute specifies the REGION control can serve as a drag-and-drop host. DRAGID works in conjunction with the **DROPID** attribute.

Drop ID To specify the type of Drag operations this control will accept, type up to 16 *signatures*, separated by commas. The **Window Formatter** adds the **DROPID** attribute to the control, which indicates the control is a valid target for the drag and drop operations identified by the signatures.

Help

Cursor The *Cursor* field (the CURSOR attribute) allows you to specify an alternate shape for the cursor when the user passes the cursor over the control. The drop-down list provides standard cursor choices such as I-Beam and Crosshair. To select an external cursor file (whose extension must be .CUR), choose **Select File** from the drop-down list, then pick the file using the standard file dialog.

Help ID The **Help ID** field (the HLP attribute) takes a string constant specifying the key for accessing a specific topic in the Help document. This may be either a Help keyword or a context string.

A Help keyword is a word or phrase indexed so that the user may search for it in the *Help* **Search** dialog. When you fill in the HLP attribute for a button, if the entry box has focus, when the user presses F1, the help file opens to the referenced topic. If more than one topic matches a keyword, the search dialog appears.

When referencing a context string in the **Help ID** field, you must identify it with a leading tilde (~).

Message The **Message** field (the MSG attribute) allows you to specify text to display in the first zone of the status bar when the control has focus.

Tip The **TIP** attribute on a control specifies the text to display in a "balloon help" box when the mouse cursor pauses over the control. Although there is no specific limit on the number of characters, the *string* should not be longer than can be displayed on the screen.

Position

Allows you to set the location and size of a control.

The **Position** dialog allows you to specify the AT (set control position and size in window) attribute. Filling in the attribute manually is optional--you may set position visually, dragging with the mouse in the **Window Formatter**.

To set a *precise* screen location or size for your control, specify fixed coordinates via this dialog. To set the location and size, choose from the following options for the **Top Left Corner**, the **Width** of the control, and its **Height**.

Default This instructs Windows to set a value which depends on the end user's system. Default size for a control depends on the size of the system font.

Full Sizes a control to expand to the full width or height of the window. The control automatically expands or contracts as the end user resizes the window.

Fixed

To set a specific position and size, mark the **Fixed** choices.

The measurement units for these boxes are dialog units. These provide a relative screen measure. Dialog units are a relative measure based on the default system font character size. Windows automatically repositions the control proportionally at different screen resolutions.

Font

Calls the **Select Font** dialog which allows you to select the font (typeface), size, style (such as bold or italic), color, and font effects (underline and strikeout) for the selected control or window. As you choose options, the dialog box displays a sample of the selected font.

Actions

There are no prompts for the Actions Tab for this control.

Spin Box Control Properties

Click on a TAB to see its help

Spin Boxes are specialized entry boxes that only accept values in a predefined range. They also provide the user with "increase" and "decrease" buttons, which many people like because they can use the mouse to change the value. The user can also type a value directly into the control.

General

Picture

The Picture field takes a display picture token that specifies input format. You may press the ellipsis (...) button next to the field to pick a display picture from the **Edit Picture String** dialog.

You may check the user entry against the picture at two points: as the user types the data in, or when the user closes the dialog box. Checking the data as the user types it incurs a slight performance penalty. To do so, check the **Entry Patterns** box in the **Window Properties** dialog for the window in which the entry box resides. This turns the MASK attribute on for *all* controls in the window.

If the MASK attribute is off, entry checking takes place when the user moves the focus to another control (for example, by TABBING to another field).

If the user types in data in a format different from the picture, the program will attempt to determine the format, then convert it to match the picture (if no MASK was specified). For example, if the user types 'January 1, 1995' and the picture is @D1, the program formats the input to "1/1/95. If the program cannot determine the entry format, it will *not* update the USE variable. The user will receive an audible prompt (beep), and the focus will return to the entry control, ready for additional input.

Use

Place a variable or field equate label in the **Use** field. You may specify a variable which receives the value that the user selects. Or, a field equate label which references the spin box in program statements.

From

The FROM attribute is optional, but is useful for values that progress in an irregular increment. You may also wish to provide the user with strings formatted as Spin Box choices when the choices are a natural progression such as the days of the week or the months of the year. Specify a QUEUE in the **From** field. This and **Range** are mutually exclusive.

Justification

Specify left, center, right, decimal, or default justification. Default justification matches that specified in the data dictionary, if applicable. If you use decimal justification, you set the Offset to allow display of digits to the right of the decimal point.

Offset

Specify an indentation value for the list box item text, in dialog units.

Mode

Hide

Makes the control invisible at the time Windows would initially display it. Windows actually creates the control--it just doesn't display it on screen. The **Window Formatter** places the HIDE attribute on the control. Use the UNHIDE statement to display the control.

Disable Disables or 'grays-out' the control when your program initially displays it. The **Window Formatter** places the DISABLE attribute on the control. Use the ENABLE statement to allow the user access to the control.

Scroll Specifies whether the control should move with the window when the user scrolls the window. By default, (unchecked), the control does not move with the window. Leave the **Scroll** box unchecked to create a control that stays fixed when the user scrolls the window. The **Window Formatter** places the SCROLL attribute on the control when checked.

Skip Instructs the **Window Formatter** to omit the control from the Tab Order. When the user TABS from field to field in the dialog box, Windows will not give the control focus. This is useful for seldom-used data fields. The **Window Formatter** will place the SKIP attribute on the control.

Extra

Range Specify the upper and lower **Range** limits, and the **Step** value. Place the highest value which the control should return in the **Range Upper** field. The value should match the **Picture** field. Place the lowest acceptable value in the **Lower** field. Place the Step value--the amount by which each press of the increase or decrease buttons should change the spin box value--in the **Step** field.

Case Specify case attributes for the entry field. The entry box can automatically convert characters from one case to another. **Uppercase** automatically converts to all caps. **Capitalize** converts to proper case. **Normal** (no attribute) accepts input in the case the user types it.

Entry Mode Optionally specify an **Entry Mode** for the entry field portion of the combo box.. Choose either **Insert**, **Overwrite** or **As Is**. The **Entry Mode** applies only for windows with the MASK attribute set

Options Set the Entry flags. There are three entry flags you may toggle on or off independently.

Required - (the REQ attribute) specifies that the control may not be left blank or zero.

Read Only - (the READONLY attribute) prevents data entry in this control. Use this to declare display-only data.

Immediate - (the IMM attribute) specifies immediate event generation whenever the user presses any key..

Drop ID To specify the type of Drag operations this control will accept, type up to 16 *signatures*, separated by commas. The **Window Formatter** adds the **DROPID** attribute to the control, which indicates the control is a valid target for the drag and drop operations identified by the signatures.

Help

Cursor The *Cursor* field (the CURSOR attribute) allows you to specify an alternate shape

for the cursor when the user passes the cursor over the control. The drop-down list provides standard cursor choices such as I-Beam and Crosshair. To select an external cursor file (whose extension must be .CUR), choose **Select File** from the drop-down list, then pick the file using the standard file dialog.

Help ID

The **Help ID** field (the HLP attribute) takes a string constant specifying the key for accessing a specific topic in the Help document. This may be either a Help keyword or a context string.

A Help keyword is a word or phrase indexed so that the user may search for it in the *Help* **Search** dialog. When you fill in the HLP attribute for a button, if the entry box has focus, when the user presses F1, the help file opens to the referenced topic. If more than one topic matches a keyword, the search dialog appears.

When referencing a context string in the **Help ID** field, you must identify it with a leading tilde (~).

Message

The **Message** field (the MSG attribute) allows you to specify text to display in the first zone of the status bar when the control has focus.

Tip

The **TIP** attribute on a control specifies the text to display in a "balloon help" box when the mouse cursor pauses over the control. Although there is no specific limit on the number of characters, the *string* should not be longer than can be displayed on the screen.

Position

Allows you to set the location and size of a control.

The **Position** dialog allows you to specify the AT (set control position and size in window) attribute. Filling in the attribute manually is optional--you may set position visually, dragging with the mouse in the **Window Formatter**.

To set a *precise* screen location or size for your control, specify fixed coordinates via this dialog. To set the location and size, choose from the following options for the **Top Left Corner**, the **Width** of the control, and its **Height**.

Default

This instructs Windows to set a value which depends on the end user's system. Default size for a control depends on the size of the system font.

Full

Sizes a control to expand to the full width or height of the window. The control automatically expands or contracts as the end user resizes the window.

Fixed

To set a specific position and size, mark the **Fixed** choices.

The measurement units for these boxes are dialog units. These provide a relative screen measure. Dialog units are a relative measure based on the default system font character size. Windows automatically repositions the control proportionally at different screen resolutions.

Font

Calls the **Select Font** dialog which allows you to select the font (typeface), size, style (such as bold or italic), color, and font effects (underline and strikeout) for the selected control or window. As you choose options, the dialog box displays a sample of the selected font.

Actions

There are no prompts for the Actions Tab for this control.

Text Box Control Properties

Click on a TAB to see its help

The Text control provides a multi-line data entry field. This control is especially suitable for holding a long string.

General

Use — Place a variable or field equate label in the **Use** field. You may specify a variable which receives the value that the user types. When using multi-line controls, be sure the variable is large enough to hold the amount of data you expect your users to enter in the control. Or, type a field equate label which references the entry box in program statements.

Justification — Specify left, center, right, or default justification. Default justification matches that specified in the data dictionary, if applicable.

Mode

Hide — Makes the control invisible at the time Windows would initially display it. Windows actually creates the control--it just doesn't display it on screen. The **Window Formatter** places the HIDE attribute on the control. Use the UNHIDE statement to display the control.

Disable — Disables or 'grays-out' the control when your program initially displays it. The **Window Formatter** places the DISABLE attribute on the control. Use the ENABLE statement to allow the user access to the control.

Scroll — Specifies whether the control should move with the window when the user scrolls the window. By default, (unchecked), the control does not move with the window. Leave the **Scroll** box unchecked to create a control that stays fixed when the user scrolls the window. The **Window Formatter** places the SCROLL attribute on the control when checked.

Skip — Instructs the **Window Formatter** to omit the control from the Tab Order. When the user TABS from field to field in the dialog box, Windows will not give the control focus. This is useful for seldom-used data fields. The **Window Formatter** will place the SKIP attribute on the control.

Extra

Case — Specify case attributes for the entry field. The entry box can automatically convert characters from one case to another. **Uppercase** automatically converts to all caps. **Normal** (no attribute) accepts input in the case the user types it.

Options — Set the Entry flags. There are two entry flags you may toggle on or off independently.

Required - (the REQ attribute) specifies that the control may not be left blank or zero.

Read Only - (the READONLY attribute) prevents data entry in this control. Use

this to declare display-only data.

Scroll Bars To add a horizontal scroll bar to the control, mark the **Horizontal** check box. Scroll bars only appear when the contents of the text box are bigger than the window. To add a vertical scroll bar, check the **Vertical** check box. These options add the HSCROLL, VSCROLL, and HVSCROLL attributes to the control.

Drop ID To specify the type of Drag operations this control will accept, type up to 16 *signatures*, separated by commas. The **Window Formatter** adds the **DROPID** attribute to the control, which indicates the control is a valid target for the drag and drop operations identified by the signatures.

Help

Cursor The *Cursor* field (the CURSOR attribute) allows you to specify an alternate shape for the cursor when the user passes the cursor over the control. The drop-down list provides standard cursor choices such as I-Beam and Crosshair. To select an external cursor file (whose extension must be .CUR), choose **Select File** from the drop-down list, then pick the file using the standard file dialog.

Help ID The **Help ID** field (the HLP attribute) takes a string constant specifying the key for accessing a specific topic in the Help document. This may be either a Help keyword or a context string.

A Help keyword is a word or phrase indexed so that the user may search for it in the *Help* **Search** dialog. When you fill in the HLP attribute for a button, if the entry box has focus, when the user presses F1, the help file opens to the referenced topic. If more than one topic matches a keyword, the search dialog appears.

When referencing a context string in the **Help ID** field, you must identify it with a leading tilde (~).

Message The **Message** field (the MSG attribute) allows you to specify text to display in the first zone of the status bar when the control has focus.

Tip The **TIP** attribute on a control specifies the text to display in a "balloon help" box when the mouse cursor pauses over the control. Although there is no specific limit on the number of characters, the *string* should not be longer than can be displayed on the screen.

Position

Allows you to set the location and size of a control.

The **Position** dialog allows you to specify the AT (set control position and size in window) attribute. Filling in the attribute manually is optional--you may set position visually, dragging with the mouse in the **Window Formatter**.

To set a *precise* screen location or size for your control, specify fixed coordinates via this dialog. To set the location and size, choose from the following options for the **Top Left Corner**, the **Width** of the control, and its **Height**.

Default This instructs Windows to set a value which depends on the end user's system.

Default size for a control depends on the size of the system font.

Full Sizes a control to expand to the full width or height of the window. The control automatically expands or contracts as the end user resizes the window.

Fixed To set a specific position and size, mark the **Fixed** choices.

The measurement units for these boxes are dialog units. These provide a relative screen measure. Dialog units are a relative measure based on the default system font character size. Windows automatically repositions the control proportionally at different screen resolutions.

Font

Calls the **Select Font** dialog which allows you to select the font (typeface), size, style (such as bold or italic), color, and font effects (underline and strikeout) for the selected control or window. As you choose options, the dialog box displays a sample of the selected font.

Actions

There are no prompts for the Actions Tab for this control.

Radio Button Control Properties

Click on a TAB to see its help

A Radio button, also called an option button, provides the user one of a set of mutually exclusive choices. By default, a filled-in circle represents the current selection.

A group box--an OPTION structure--must always surround the radio button choices. The **Window Formatter** automatically prompts you to create this if you try to place a radio button outside an option box. When the user selects a choice, the control fills the USE variable with the Radio text, minus the ampersand indicating the accelerator key.

When you place a radio button in a blank dialog window, it forces the creation of the OPTION STRUCTURE. After activating the Radio Button tool, or choosing **Radio Button** from the **Control** menu, CLICK in the window. An option box and one radio button appear. Select the radio button and press the Properties button, or RIGHT-CLICK and select **Properties** to open the **Radio Button Properties** dialog.

General

Parameter — Specify a string constant to display. Place an ampersand (&) before the single character to set the accelerator key or mnemonic access character for the radio button--this underlines the label that appears next to the radio button.

Use — The field equate label references the radio button in program statements.

Justification — Specify left, center, right, or default justification. Default justification matches that specified in the data dictionary, if applicable.

Mode

Hide — Makes the control invisible at the time Windows would initially display it. Windows actually creates the control--it just doesn't display it on screen. The **Window Formatter** places the HIDE attribute on the control. Use the UNHIDE statement to display the control.

Disable — Disables or 'grays-out' the control when your program initially displays it. The **Window Formatter** places the DISABLE attribute on the control. Use the ENABLE statement to allow the user access to the control.

Scroll — Specifies whether the control should move with the window when the user scrolls the window. By default, (unchecked), the control does not move with the window. Leave the **Scroll** box unchecked to create a control that stays fixed when the user scrolls the window. The **Window Formatter** places the SCROLL attribute on the control when checked.

Skip — Instructs the **Window Formatter** to omit the control from the Tab Order. When the user TABS from field to field in the dialog box, Windows will not give the control focus. This is useful for seldom-used data fields. The **Window Formatter** will place the SKIP attribute on the control.

Extra

Icon — In the **Icon** field (ICON), optionally select a standard icon or icon file. This displays a small bitmap on the button face (clipping or centering the bitmap as

necessary).

To select a standard icon, choose one of the named items in the drop-down list. To select an icon file (whose extension must be .ICO), choose **Select File** from the drop-down list, then pick the file using the standard file dialog. At run time, the radio button appears as a "latched" 3D push button.

Drop ID

To specify the type of Drag operations this control will accept, type up to 16 *signatures*, separated by commas. The **Window Formatter** adds the **DROPID** attribute to the control, which indicates the control is a valid target for the drag and drop operations identified by the signatures.

Help

Cursor

The *Cursor* field (the CURSOR attribute) allows you to specify an alternate shape for the cursor when the user passes the cursor over the control. The drop-down list provides standard cursor choices such as I-Beam and Crosshair. To select an external cursor file (whose extension must be .CUR), choose **Select File** from the drop-down list, then pick the file using the standard file dialog.

Help ID

The **Help ID** field (the HLP attribute) takes a string constant specifying the key for accessing a specific topic in the Help document. This may be either a Help keyword or a context string.

A Help keyword is a word or phrase indexed so that the user may search for it in the *Help* **Search** dialog. When you fill in the HLP attribute for a button, if the entry box has focus, when the user presses F1, the help file opens to the referenced topic. If more than one topic matches a keyword, the search dialog appears.

When referencing a context string in the **Help ID** field, you must identify it with a leading tilde (~).

Message

The **Message** field (the MSG attribute) allows you to specify text to display in the first zone of the status bar when the control has focus.

Tip

The **TIP** attribute on a control specifies the text to display in a "balloon help" box when the mouse cursor pauses over the control. Although there is no specific limit on the number of characters, the *string* should not be longer than can be displayed on the screen.

Position

Allows you to set the location and size of a control.

The **Position** dialog allows you to specify the AT (set control position and size in window) attribute. Filling in the attribute manually is optional--you may set position visually, dragging with the mouse in the **Window Formatter**.

To set a *precise* screen location or size for your control, specify fixed coordinates via this dialog. To set the location and size, choose from the following options for the **Top Left Corner**, the **Width** of the control, and its **Height**.

Default

This instructs Windows to set a value which depends on the end user's system. Default size for a control depends on the size of the system font.

Full

Sizes a control to expand to the full width or height of the window. The control

automatically expands or contracts as the end user resizes the window.

Fixed To set a specific position and size, mark the **Fixed** choices.

The measurement units for these boxes are dialog units. These provide a relative screen measure. Dialog units are a relative measure based on the default system font character size. Windows automatically repositions the control proportionally at different screen resolutions.

Font

Calls the **Select Font** dialog which allows you to select the font (typeface), size, style (such as bold or italic), color, and font effects (underline and strikeout) for the selected control or window. As you choose options, the dialog box displays a sample of the selected font.

Actions

There are no prompts for the Actions Tab for this control.

Progress Control Properties

Click on a TAB to see its help

The **PROGRESS** control declares a control that displays a progress bar. This usually displays the current percentage of completion of a batch process.

If a variable is named as the USE attribute, the progress bar is automatically updated whenever the value in that variable changes. If the USE attribute is a field equate label, you must directly update the display by assigning a value (within the range defined by the RANGE attribute) to the control's PROP:progress property

General

Use The field equate label references the control in program statements.

Mode

Hide Makes the control invisible at the time Windows would initially display it. Windows actually creates the control--it just doesn't display it on screen. The **Window Formatter** places the HIDE attribute on the control. Use the UNHIDE statement to display the control.

Disable Disables or 'grays-out' the control when your program initially displays it. The **Window Formatter** places the DISABLE attribute on the control. Use the ENABLE statement to allow the user access to the control.

Scroll Specifies whether the control should move with the window when the user scrolls the window. By default, (unchecked), the control does not move with the window. Leave the **Scroll** box unchecked to create a control that stays fixed when the user scrolls the window. The **Window Formatter** places the SCROLL attribute on the control when checked.

Extra

Range Specifies the range of values the progress bar displays. If omitted, the default range is zero (0) to one hundred (100).

Drop ID To specify the type of Drag operations this control will accept, type up to 16 *signatures*, separated by commas. The **Window Formatter** adds the **DROPID** attribute to the control, which indicates the control is a valid target for the drag and drop operations identified by the signatures.

Help

Cursor The *Cursor* field (the CURSOR attribute) allows you to specify an alternate shape for the cursor when the user passes the cursor over the control. The drop-down list provides standard cursor choices such as I-Beam and Crosshair. To select an external cursor file (whose extension must be .CUR), choose **Select File** from the drop-down list, then pick the file using the standard file dialog.

Position

Allows you to set the location and size of a control.

The **Position** dialog allows you to specify the AT (set control position and size in window) attribute. Filling in the attribute manually is optional--you may set position visually, dragging with the mouse in the **Window Formatter**.

To set a *precise* screen location or size for your control, specify fixed coordinates via this dialog. To set the location and size, choose from the following options for the **Top Left Corner**, the **Width** of the control, and its **Height**.

Default — This instructs Windows to set a value which depends on the end user's system. Default size for a control depends on the size of the system font.

Full — Sizes a control to expand to the full width or height of the window. The control automatically expands or contracts as the end user resizes the window.

Fixed — To set a specific position and size, mark the **Fixed** choices.

The measurement units for these boxes are dialog units. These provide a relative screen measure. Dialog units are a relative measure based on the default system font character size. Windows automatically repositions the control proportionally at different screen resolutions.

Font

Calls the **Select Font** dialog which allows you to select the font (typeface), size, style (such as bold or italic), color, and font effects (underline and strikeout) for the selected control or window. As you choose options, the dialog box displays a sample of the selected font.

Actions

There are no prompts for the Actions Tab for this control.

Sheet Control Properties

Click on a TAB to see its help

The **SHEET** control declares a group of TAB controls that offer the user multiple "pages" of controls for the window. The multiple TAB controls in the SHEET structure define the "pages" displayed to the user.

General

Use Type a field equate label to reference the control in executable code, or the name of a variable

Mode

Hide Makes the control invisible at the time Windows would initially display it. Windows actually creates the control--it just doesn't display it on screen. The **Window Formatter** places the HIDE attribute on the control. Use the UNHIDE statement to display the control.

Disable Disables or 'grays-out' the control when your program initially displays it. The **Window Formatter** places the DISABLE attribute on the control. Use the ENABLE statement to allow the user access to the control.

Skip Instructs the **Window Formatter** to omit the control from the Tab Order. When the user TABS from field to field in the dialog box, Windows will not give the control focus. This is useful for seldom-used data fields. The **Window Formatter** will place the SKIP attribute on the control.

Scroll Specifies whether the control should move with the window when the user scrolls the window. By default, (unchecked), the control does not move with the window. Leave the **Scroll** box unchecked to create a control that stays fixed when the user scrolls the window. The **Window Formatter** places the SCROLL attribute on the control when checked.

Extra

Wizard Hides the "tab" portion of the TAB controls.

Hiding the tabs aids in creating a wizard. A wizard is a window with a "tabless" SHEET control containing one or more TABS. You'll need to write the code to handle the "turning of the pages". See *How to Create a Wizard*. Also see the CW\EXAMPLES\APPS\WIZDEMO\WIZ.APP application.

Tip: Do not check this box until you are finished designing the window!

Spread Resizes the tabs on the TABs to fill all the available space on the SHEET.

The resizing algorithm considers the length of the text displayed on each tab, the number of tabs, and the available space on the property sheet.

Drop ID To specify the type of Drag operations this control will accept, type up to 16 *signatures*, separated by commas. The **Window Formatter** adds the **DROPID** attribute to the control, which indicates the control is a valid target for the drag

and drop operations identified by the signatures.

Help

Cursor The *Cursor* field (the CURSOR attribute) allows you to specify an alternate shape for the cursor when the user passes the cursor over the control. The drop-down list provides standard cursor choices such as I-Beam and Crosshair. To select an external cursor file (whose extension must be .CUR), choose **Select File** from the drop-down list, then pick the file using the standard file dialog.

Position

Allows you to set the location and size of a control.

The **Position** dialog allows you to specify the AT (set control position and size in window) attribute. Filling in the attribute manually is optional--you may set position visually, dragging with the mouse in the **Window Formatter**.

To set a *precise* screen location or size for your control, specify fixed coordinates via this dialog. To set the location and size, choose from the following options for the **Top Left Corner**, the **Width** of the control, and its **Height**.

Default This instructs Windows to set a value which depends on the end user's system. Default size for a control depends on the size of the system font.

Full Sizes a control to expand to the full width or height of the window. The control automatically expands or contracts as the end user resizes the window.

Fixed To set a specific position and size, mark the **Fixed** choices.

The measurement units for these boxes are dialog units. These provide a relative screen measure. Dialog units are a relative measure based on the default system font character size. Windows automatically repositions the control proportionally at different screen resolutions.

Font

Calls the **Select Font** dialog which allows you to select the font (typeface), size, style (such as bold or italic), color, and font effects (underline and strikeout) for the selected control or window. As you choose options, the dialog box displays a sample of the selected font.

Actions

There are no prompts for the Actions Tab for this control.

Tab Control Properties

Click on a TAB to see its help

The **TAB** structure declares a group of controls that constitute one of the multiple "pages" of controls contained within a SHEET structure. The multiple TAB controls in the SHEET structure define the "pages" displayed to the user. The SHEET structure's USE attribute receives the *text* of the TAB control selected by the user.

General

Parameter — Specify a string constant by typing it in the **Parameter** box. If the control is to display a variable, type a picture token in this box.

Use — Type a field equate label to reference the control in executable code, or the name of a variable

Extra

Required — Specifies that when selected, your program automatically checks that all entry controls with the REQ attribute are neither blank nor zero.

Specify this type of tab when a window also contains an ENTRY or TEXT control field with the REQ attribute (or else use the INCOMPLETE() function to test the ENTRY controls). When the user clicks on a tab with the REQ attribute and an ENTRY field is blank or zero, the first required control which is blank or zero receives the focus. See also: How to Create a Multi-Page Form .

Drop ID — To specify the type of Drag operations this control will accept, type up to 16 *signatures*, separated by commas. The **Window Formatter** adds the **DROPID** attribute to the control, which indicates the control is a valid target for the drag and drop operations identified by the signatures.

Help

Cursor — The *Cursor* field (the CURSOR attribute) allows you to specify an alternate shape for the cursor when the user passes the cursor over the control. The drop-down list provides standard cursor choices such as I-Beam and Crosshair. To select an external cursor file (whose extension must be .CUR), choose **Select File** from the drop-down list, then pick the file using the standard file dialog.

Help ID — The **Help ID** field (the HLP attribute) takes a string constant specifying the key for accessing a specific topic in the Help document. This may be either a Help keyword or a context string.

A Help keyword is a word or phrase indexed so that the user may search for it in the *Help* **Search** dialog. When you fill in the HLP attribute for a button, if the entry box has focus, when the user presses F1, the help file opens to the referenced topic. If more than one topic matches a keyword, the search dialog appears.

When referencing a context string in the **Help ID** field, you must identify it with a leading tilde (~).

Message	The **Message** field (the MSG attribute) allows you to specify text to display in the first zone of the status bar when the control has focus.
Tip	The **TIP** attribute on a control specifies the text to display in a "balloon help" box when the mouse cursor pauses over the control. Although there is no specific limit on the number of characters, the *string* should not be longer than can be displayed on the screen.

Actions

There are no prompts for the Actions Tab for this control.

Toolbar Properties

Click on a TAB to see its help

Allows you to edit settings for your TOOLBAR.

See also:

How to Add a Tool Bar

Extra

No Merge Specifies *not* to merge an MDI tool bar into an application frame tool bar at run time.

Help

Cursor The *Cursor* field (the CURSOR attribute) allows you to specify an alternate shape for the cursor when the user passes the cursor over the control. The drop-down list provides standard cursor choices such as I-Beam and Crosshair. To select an external cursor file (whose extension must be .CUR), choose **Select File** from the drop-down list, then pick the file using the standard file dialog.

Position

Allows you to set the location and size of a control.

The **Position** dialog allows you to specify the AT (set control position and size in window) attribute. Filling in the attribute manually is optional--you may set position visually, dragging with the mouse in the **Window Formatter**.

To set a *precise* screen location or size for your toolbar, specify fixed coordinates via this dialog. To set the location and size, choose from the following options for the **Top Left Corner**, the **Width** of the control or window, and its **Height**.

Default Default size for a toolbar is approximately twice the height of the caption bar, and the full width of the window.

If you want to merge toolbars, both should use the Default size.

Fixed To set a specific position and size, mark the **Fixed** choices. To set the upper left corner of the toolbar, type values in the **X** and **Y** boxes.

The measurement units for these boxes are dialog units. These provide a relative screen measure. Dialog units are a relative measure based on the default system font character size. Windows automatically repositions the window proportionally at different screen resolutions.

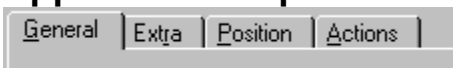
Font

Calls the **Select Font** dialog which allows you to select the font (typeface), size, style (such as bold or italic), color, and font effects (underline and strikeout) for the selected control or window. As you choose options, the dialog box displays a sample of the selected font.

Actions

There are no prompts for the Actions Tab for this control.

Application Properties Dialog



Click on a TAB to see its help

This dialog allows you to specify the appearance and functionality of your application frame window.

See also:

How to Customize Your Window

General

Title

To specify caption bar text, type a string constant in the **Title** field. The caption bar holds the name of the window.

Label

To specify the label for the application structure, type it in the **Label** field. This names the specific APPLICATION in the source code. The label may contain upper or lower case letters, numerals, the underscore character or a colon. Space characters are forbidden. The first character must be a letter or the underscore character. Clarion reserved words may not serve as labels.

Frame Type

To choose the frame for your window, pick a selection from the **Frame Type** drop-down list. The frame defines the borders of the window. The normal frame type for an application frame is the resizeable type. Choose from:

Single - a single pixel frame which the user cannot resize.

Double - a thick frame, which the user cannot resize. This adds the DOUBLE attribute to the window.

Resizeable - a thick frame, which the user can resize. This adds the RESIZE attribute to the window.

Initial Size

Sets the initial state of your window. Choose from:

Normal - displays the window at the default size which either you specifically set, or Windows sets if you don't.

Maximized - the window fills the desktop, if an application window, or the window frame, if an MDI child window. This adds the MAXIMIZE attribute to the window.

Iconized - the window appears in iconized state--as a 32 by 32 pixel window at the bottom of the desktop. This adds the ICONIZE attribute to the window.

Extra

Icon

To associate an icon with the window, specify an icon in this field. You may type in a file name or an EQUATE. You may also press the ellipsis button (...), then select an icon file name using the standard Open File dialog. The file name or equate you specify becomes the parameter for the ICON attribute.

You should always specify an icon for an application window. Specifying an icon name automatically places a minimize button on the caption bar of your application or MDI child window.

Timer To specify the window receive Timer Event messages from Windows, fill in the **Timer** field. Specify the timer interval in hundredths of seconds. The file name or equate you specify becomes the parameter for the TIMER attribute.

For example, if you specify 100 in the field, the window will automatically receive an EVENT:Timer once every second (100/100's seconds). This might be appropriate for adding a clock to a status bar.

Immediate To generate a message event each time the end user moves or resizes the window, check the **Immediate** box. This adds the IMM attribute to the window. You are responsible for the code that executes upon notification of the event.

Status Bar To provide a message bar at the bottom of your window, mark the **Status Bar** check box. This adds the STATUS attribute to the window.

Tip: A status bar in an application window is an excellent way to increase user feedback in your application. Clarion for Windows makes it simple to post messages on the status bar advising the user of what your application is doing as it does it. Increasing user feedback makes the user feel more in control. This allows the user to feel more confident and be more efficient when using your application.

System Menu To add a system menu to your window, mark the **System Menu** check box. Most windows should have a system menu. For users on a system without a mouse, the system menu provides the only means of minimizing, maximizing or re-sizing the window. This adds the SYSTEM attribute to the window.

Tip: Even if you plan that the window should NOT have a system menu when the application is complete, it's good practice to place a system menu on your application while it's under development. By DOUBLE-CLICKING the system menu, or choosing Close, you can close your application should your normal exit procedure fail.

Auto Display To add the AUTO attribute to your window, mark the **Auto Display** check box. This automatically updates the contents of all controls on screen through each pass of the ACCEPT loop.

Maximize Box To place a maximize button in your window, mark the **Maximize Box** check box. In general, you should place a maximize button only on application windows and MDI child document windows. This adds the MAX attribute to the window.

Scroll Bars To add a horizontal scroll bar to your window, mark the **Horizontal** check box. Scroll bars only appear when something inside the window--a control--is bigger than the window. To add a vertical scroll bar to your window, check the **Vertical** check box. These options add the HSCROLL, VSCROLL, and HVSCROLL attributes to the window.

Assuming your application frame will display MDI child windows, you normally check both **Horizontal** and **Vertical**.

Status Widths To set the width of the status bar zone(s), type a value or list of values in the **Status Widths** field. You must also check the **Status Bar** box in the top part of the dialog to display a status bar. The values you enter in this field fill the STATUS attribute parameters.

The zones are the areas within the status bar marked off by the 3D shaded

boxes. The first zone on the left, by default, displays MSG attribute text. This is useful for specifying short help instructions or other information to the user. If your application has only one zone for the status bar, you may omit this field. For more than one zone, enter a series of comma separated values. The default measurement unit is dialog units.

You may set a minimum value for a zone width by typing a negative number. This creates a zone with a minimum width, but is expandable by resizing the window. Use the runtime property assignment syntax to place text in any zone. To place a string in the second zone, for example:

```
{PROP:StatusText} = ?array
```

Tip: A multi-zone status bar can give your application a professional look. You may display help text in zone one, and when editing a record, the current record number in zone two, for example.

Position

Allows you to set the location and size of a control.

The **Position** dialog allows you to specify the AT (set control position and size in window) attribute. Filling in the attribute manually is optional--you may set position visually, dragging with the mouse in the **Window Formatter**.

To set a *precise* screen location or size for your control, specify fixed coordinates via this dialog. To set the location and size, choose from the following options for the **Top Left Corner**, the **Width** of the control, and its **Height**.

Default This instructs Windows to set a value which depends on the end user's system. Default size for a control depends on the size of the system font.

Full Sizes a control to expand to the full width or height of the window. The control automatically expands or contracts as the end user resizes the window.

Fixed To set a specific position and size, mark the **Fixed** choices.

The measurement units for these boxes are dialog units. These provide a relative screen measure. Dialog units are a relative measure based on the default system font character size. Windows automatically repositions the control proportionally at different screen resolutions.

Font

Calls the **Select Font** dialog which allows you to select the font (typeface), size, style (such as bold or italic), color, and font effects (underline and strikeout) for the selected control or window. As you choose options, the dialog box displays a sample of the selected font.

Actions

There are no prompts for the Actions Tab for this control.

Window Properties

Click on a TAB to see its help

This dialog allows you to specify the appearance and functionality of your window.

See also:

How to Customize Your Window

General

Title

To specify caption bar text, type a string constant in the **Title** field. The caption bar holds the name of the window.

Label

This names the specific WINDOW in the source code. The label may contain upper or lower case letters, numerals, the underscore character or a colon. Space characters are forbidden. The first character must be a letter or the underscore character. Clarion reserved words may not serve as labels.

Frame Type

To choose the frame for your window, pick a selection from the **Frame Type** drop-down list. The frame defines the borders of the window. Choose from:

Single - a single pixel frame which the user cannot resize. Most suitable for dialog boxes.

Double - a thick frame, which the user cannot resize. Use this type frame for a system modal window with no caption bar, or for a modal dialog box with a caption bar. This adds the DOUBLE attribute to the window.

Resizeable - a thick frame, which the user can resize. Choose this for application and MDI child windows. This adds the RESIZE attribute to the window.

Initial Size

Sets the initial state of your window. Choose from:

Normal: - displays the window at the default size which either you specifically set, or Windows sets if you don't.

Maximized: - the window fills the desktop, if an application window, or the window frame, if an MDI child window. This adds the MAXIMIZE attribute to the window.

Iconized: - the window appears in iconized state--as a 32 by 32 pixel window at the bottom of the desktop, for an application window, or at the inside bottom of the application frame, for an MDI child window. This adds the ICONIZE attribute to the window.

Extra

Icon

To associate an icon with the window, specify an icon in this field. You may type in a file name or an EQUATE. You may also press the ellipsis button (...), then select an icon file name using the standard **Open File** dialog. The file name or equate you specify becomes the parameter for the ICON attribute.

You should always specify an icon for an application window, and for an MDI child window. Specifying an icon name automatically places a minimize button on

the caption bar of your application or MDI child window.

Palette Use the PALETTE attribute on your window to ensure ample color support for your images. The PALETTE attribute specifies how many colors you want this window to use when it is the foreground window. This is applicable in hardware modes where a palette is in use and spare colors are available. The number you specify becomes the parameter for the PALETTE attribute. Leave this field blank to specify the default for the end user's system.

Timer To specify the window receive Timer Event messages from Windows, fill in the **Timer** field. Specify the timer interval in hundredths of seconds. The file name or equate you specify becomes the parameter for the TIMER attribute.

For example, if you specify 100 in the field, the window will automatically receive an EVENT:Timer once every second (100/100's seconds). This might be appropriate for adding a clock to a status bar.

Immediate To generate a message event each time the end user moves or resizes the window, check the **Immediate** box. This adds the IMM attribute to the window. You are responsible for the code that executes upon notification of the event.

Status Bar To provide a message bar at the bottom of your window, check the **Status Bar** box. This adds the STATUS attribute to the window.

Tip: A status bar in an application window is an excellent way to increase user feedback in your application. Clarion for Windows makes it simple to post messages on the status bar advising the user of what your application is doing as it does it. Increasing user feedback makes the user feel more in control. This allows the user to feel more confident and be more efficient when using your application.

Modal Window To specify a system modal window, check the **Modal Window** box. A system modal window prevents all other tasks--even in applications other than your own--from executing until the window is closed. This adds the MODAL attribute to the window.

Entry Patterns To enable support for an entry mask for controls in the window, check the **Entry Patterns** box. This allows you to specify key-in entry patterns for the fields you choose. This adds the MASK attribute to the window.

System Menu To place a system menu in your window, check the **System Menu** box. Most windows should have a system menu. For users on a system without a mouse, the system menu provides the only means of minimizing, maximizing or re-sizing the window. This adds the SYSTEM attribute to the window.

Tip: Even if you plan that the window should NOT have a system menu when the application is complete, it's good practice to place a system menu on your application while it's under development. By DOUBLE-CLICKING the system menu, or choosing Close, you can close your application should your normal exit procedure fail.

Auto Display To add the AUTO attribute to your window, check the **Auto Display** box. This automatically updates the contents of all controls on screen through each pass of the ACCEPT loop.

MDI Child To add the MDI attribute to your window, check the **MDI Child** box. An MDI child window cannot move outside the main application window. A typical use of an MDI window might be to present a different arrangement of the data in your

application's database.

Maximize Box To place a maximize button in your window, check the **Maximize Box** box. In general, you should place a maximize button only on application windows and MDI child document windows. This adds the MAX attribute to the window.

3D Look To provide the gray window background, chiseled control look for your application, check the **3D Look** box. This is clearly a style consideration, but will go a long way in giving your application a professional look. This adds the GRAY attribute to the window.

The gray background is not visible when you design your window with the **Window Formatter**. It is, however, visible in test mode, and when your application runs.

Toolbox To add the TOOLBOX attribute to your window, check the **Toolbox** box. This makes your window always stay on top.

Scroll Bars To add a horizontal scroll bar to your window, check the **Horizontal** box. Scroll bars only appear when something inside the window--a control--is bigger than the window. To add a vertical scroll bar to your window, check the **Vertical** box. These options add the HSCROLL, VSCROLL, and HVSCROLL attributes to the window.

Status Widths To set the width of the status bar zone(s), type a value or list of values in the **Status Widths** field. You must also check the **Status Bar** box in the top part of the dialog to display a status bar. The values you enter in this field fill the STATUS attribute parameters.

The zones are the areas within the status bar marked off by the 3D shaded boxes. The first zone on the left, by default, displays MSG attribute text. This is useful for specifying short help instructions or other information to the user. If your application has only one zone for the status bar, you may omit this field. For more than one zone, enter a series of comma separated values. The default measurement unit is dialog units.

You may set a minimum value for a zone width by typing a negative number. This creates a zone with a minimum width, but is expandable by resizing the window. Use the runtime property assignment syntax to place text in any zone. To place a string constant in the second zone, for example:

```
{PROP:StatusText,2} = 'Record will be Added'
```

Tip: A multi-zone status bar can give your application a professional look. You may display help text in zone one, and when editing a record, the current record number in zone two, for example.

Drop ID To specify the type of Drag operations this control will accept, type up to 16 *signatures*, separated by commas. The **Window Formatter** adds the **DROPID** attribute to the control, which indicates the control is a valid target for the drag and drop operations identified by the signatures.

Help

Cursor The *Cursor* field (the CURSOR attribute) allows you to specify an alternate shape for the cursor when the user passes the cursor over the control. The drop-down list provides standard cursor choices such as I-Beam and Crosshair. To select an

external cursor file (whose extension must be .CUR), choose **Select File** from the drop-down list, then pick the file using the standard file dialog.

Help ID

The **Help ID** field (the HLP attribute) takes a string constant specifying the key for accessing a specific topic in the Help document. This may be either a Help keyword or a context string.

A Help keyword is a word or phrase indexed so that the user may search for it in the *Help* **Search** dialog. When you fill in the HLP attribute for a button, if the entry box has focus, when the user presses F1, the help file opens to the referenced topic. If more than one topic matches a keyword, the search dialog appears.

When referencing a context string in the **Help ID** field, you must identify it with a leading tilde (~).

Message

The **Message** field (the MSG attribute) allows you to specify text to display in the first zone of the status bar when the control has focus.

Position

Allows you to set the location and size of a window.

The **Position** dialog allows you to specify the AT (set window position and size) attribute. Filling in the attribute manually is optional--you may set position visually, dragging with the mouse in the **Window Formatter**.

To set a *precise* screen location or size for your window, specify fixed coordinates via this dialog. To set the location and size, choose from the following options for the **Top Left Corner**, the **Width** of the control or window, and its **Height**.

Default

This instructs Windows to set a value which depends on the end user's system. Default size and location for a window depends on where the last one opened. Windows places the top left corner of a new window below and to the left of the last window, by approximately the size of the system menu box.

Tip: To give your application the "standard" look of other Windows applications, where possible specify the *Default* setting for any windows with resizeable frames.

Center

Places a window in the middle of the desktop. You may choose horizontal or vertical centering, or both.

Full

Sizes a window to expand to the full width or height of the desktop.

Fixed

To set a specific position and size, mark the **Fixed** choices.

The measurement units for these boxes are dialog units. These provide a relative screen measure. Dialog units are a relative measure based on the default system font character size. Windows automatically repositions the window proportionally at different screen resolutions.

Tip: Sizing all windows and controls in dialog units enable you to design a screen at one resolution, and expect it to look similar at another--in theory. In practice, there can be differences, especially when you display bitmaps in Image controls. Therefore, test your applications in the popular Windows resolutions. The most popular are 640 x 480, 800 x

600, and 1024 x 768 pixels.

Font

Calls the **Select Font** dialog which allows you to select the font (typeface), size, style (such as bold or italic), color, and font effects (underline and strikeout) for the selected control or window. As you choose options, the dialog box displays a sample of the selected font.

Actions

There are no prompts for the Actions Tab for this control.

Report Properties

Click on a TAB to see its help

This dialog allows you to set up the basic report options, including its page orientation, measurement units, margins, and paper size.

General

Job Name Names the print job, as listed in the Windows Print Manager application.

Label Type a valid Clarion label to name the REPORT data structure.

Prefix Specifies the label prefix for the REPORT structure.

Units Specifies the default measurement for all controls placed in the report. Choose **Dialog Units**, **thousandths of Inches**, **Millimeters** or **Points**.

Extra

Preview Specifies the name of a QUEUE which stores the filename(s) (*.WMF) for the metafile(s) generated for page preview. See the PREVIEW attribute. If you are using the Report Template, it is handled automatically if you check the Print Preview box and you should leave this entry blank.

Position

Allows you to set the location and size of the report detail print area, by filling in the AT (set detail print area) . The measurement units for these boxes are specified on the **General** tab.

To set a precise starting point for your print detail area relative to the top left corner of the paper, specify **Top Left Corner** coordinates with this dialog. In effect, this establishes the left **margin** for your report. The top margin is usually determined by the Page Header position. These settings may also be accomplished visually by dragging report sections and borders in **the Report Formatters Page Layout View**.

To set the size of the print detail area, choose from the following options for the **Width** and **Height**. When changing a report from portrait to landscape, or vice versa, you must change the width and/or height in the **Position** tab.

Default Sets a value based on the Paper Size.

Fixed To set a specific size, mark the **Fixed** choices.

See Also: Printing Labels.

Paper Size

Paper Size Choose from over 40 standard sizes, or choose ***Other*** to specify a custom size.

Width Specifies a custom paper width in units specified on the General tab.

Height Specifies a custom paper height in units specified on the General tab.

Landscape Specifies landscape paper orientation. New reports default to portrait mode.

Landscape means the report text is aligned parallel with the longest paper edges.

Font

To set the default font for all controls appearing in the report, press the **Font** button, then choose the font and style in the **Select Font** dialog. You may override the default by setting a different font in the Properties dialog for any specific control. The options you choose in the dialog become the parameters for the FONT attribute. As you choose options, the dialog box displays a sample of the formatting.

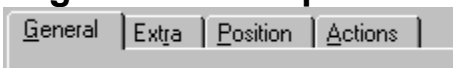
See also:

How to Use the Report Formatter -- An Overview

Actions

There are no prompts for the Actions Tab for this control.

Page Header Properties



Click on a TAB to see its help

This dialog allows you to edit the properties of the page HEADER. To specify an element to compose at the start of each report page, place it in the page HEADER.

Though they print on the page at the same time, the print engine composes the page HEADER before the page DETAIL, or FOOTER. The page HEADER is a good place for company logos, print dates, etc.

General

Use Type a field equate label to reference the page HEADER in executable code.

Extra

Alone Specifies that the HEADER section always prints alone on a page, without FORM, DETAIL, FOOTER, etc.

Absolute Specifies that the HEADER section always prints at the same fixed position on the page. This adds the ABSOLUTE attribute to the HEADER structure.

Position

Allows you to set the location and size of the page header. The position is relative to the top left corner of the paper.

The **Position** dialog allows you to specify the AT (set print structure position and size) attribute. Filling in the attribute manually is optional--you may set the position visually, by dragging with the mouse in the **Report Formatter's Page Layout View**.

To set a location or size for your report section, specify fixed coordinates with this dialog. Choose from the following options for the **Top Left Corner**, the **Width**, and the **Height**.

Default Sets a value based on the Paper Size.

Fixed To set a specific position and size, mark the **Fixed** choices.

The measurement units for these boxes are specified on the **General** tab of the **Report Properties** dialog.

Font

Calls the **Select Font** dialog which allows you to select the font (typeface), size, style (such as bold or italic), color, and font effects (underline and strikeout) for all the controls in the report section. You may override the section font by setting a different font in the Properties dialog for any specific control. As you choose options, the dialog box displays a sample of the selected font.

See also:

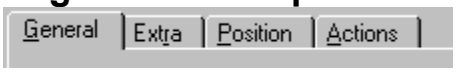
How to Control Page Breaks

Using the Report Formatter - An Overview

Actions

There are no prompts for the Actions Tab for this control.

Page Footer Properties



Click on a TAB to see its help

This dialog allows you to edit the properties of the page FOOTER. To specify an element to compose at the end of each report page, place it in the page FOOTER.

Though they print on the page at the same time, the print engine composes the page HEADER before the page DETAIL, or FOOTER. The page FOOTER is a good place for page numbers, page totals, etc.

General

Use Type a field equate label to reference the page FOOTER in executable code.

Extra

Alone Specifies that the FOOTER section always prints alone on a page, without FORM, DETAIL, HEADER, etc.

Absolute Specifies that the FOOTER section always prints at the same fixed position on the page. This adds the ABSOLUTE attribute to the FOOTER structure.

Position

Allows you to set the location and size of the page footer. The position is relative to the top left corner of the paper.

The **Position** dialog allows you to specify the AT (set print structure position and size) attribute. Filling in the attribute manually is optional--you may set the position visually, by dragging with the mouse in the **Report Formatter's Page Layout View**.

To set a location or size for your report section, specify fixed coordinates with this dialog. Choose from the following options for the **Top Left Corner**, the **Width**, and the **Height**.

Default Sets a value based on the Paper Size.

Fixed To set a specific position and size, mark the **Fixed** choices.

The measurement units for these boxes are specified on the **General** tab of the **Report Properties** dialog.

Font

Calls the **Select Font** dialog which allows you to select the font (typeface), size, style (such as bold or italic), color, and font effects (underline and strikeout) for all the controls in the report section. You may override the section font by setting a different font in the Properties dialog for any specific control. As you choose options, the dialog box displays a sample of the selected font.

See also:

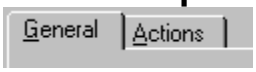
How to Control Page Breaks

Using the Report Formatter - An Overview

Actions

There are no prompts for the Actions Tab for this control.

Break Properties



This dialog allows you to add or edit the properties of a group break.

General

Label Type a valid Clarion label, naming the BREAK structure.

Variable Type a variable name, to generate a break when the value changes as you sequentially process the file.

Group breaks provide a means of breaking the data into sections and optionally providing subtotals. Each group gathers a set of data file records, all sharing the same value in the BREAK field. Within a report, you may visually separate these rows, and add a subtotal or summary information, usually below the group. Group breaks are also called group bands by some popular end user database applications.

The group break may contain the same elements as the report: a group HEADER, group DETAIL, and group FOOTER. These structures all print inside the DETAIL print area, each following the other by any offset specified in their AT attributes.

Though they print on the page at the same time, the application composes the group HEADER before the group DETAIL. The group HEADER is a good place to identify the group.

The group FOOTER, is composed after the group DETAIL. You can place a string saying "Total:" followed by a string variable which contains the field to be summed, with the SUM attribute.

To create a group break:

1. Be sure the DETAIL band is visible; if not, press the restore button.
2. Choose **Bands ▸ Surrounding Break**.
3. When the cursor changes to a crosshair, CLICK in the DETAIL.

This inserts the group BREAK.

4. In the **Break Properties** dialog, type the name of a variable or field, including the prefix, to break on.
5. Type a valid Clarion label to name the break.
6. Press the **OK** button.

When the report prints, it groups all records with the same value for the BREAK field, as well as the group HEADER and FOOTER.

Tip: If the break variable is a global or local variable, you must be sure that the executable code updates its value, so that it can generate a group BREAK.

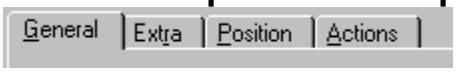
See also:

How to Set Report Group Breaks

Actions

There are no prompts for the Actions Tab for this control.

Break Group Header Properties



Click on a TAB to see its help

This dialog allows you to edit the properties of the group HEADER. To specify an element to compose at the start of each group, place it in the group HEADER.

Though they print on the page at the same time, the application composes the group HEADER before the group DETAIL. The group HEADER is a good place to identify the group, for example, with a label saying "Customer:" followed by a variable string for the customer name field.

General

Use

Type a field equate label to reference the HEADER in executable code.

Page Before

To print the HEADER structure on a new page, type a value in the **Page Before** box in the **Detail Properties** dialog. This sets the PAGEBEFORE attribute. The print engine generates a page break before printing at the top of the next page.

The page number automatically increments, unless you reset it. To reset the page number to a value you specify, type it in the **Page Before** field.

Page After

To print the HEADER, then force a new page, type a value in the **Page after** box. This sets the PAGEAFTER attribute. This prints the HEADER, then begins a new page.

The page number automatically increments, unless you reset it. To reset the page number to a value you specify, type it in the **Page after** field.

With Prior

To prevent 'orphan' elements in a printout, type a value in the **With Prior** field. This sets the WITHPRIOR attribute. An 'orphaned' print element is one which prints on a following page, separated from its related items.

The value specifies the number of preceding elements to print--a value of "1," for example, specifies that the previous element must print on the same page.

With Next

To prevent 'widow' elements in a printout, type a value in the **Keep Next**. This sets the WITHNEXT attribute. A 'widowed' print element is one which prints, but then is separated from the succeeding elements by a page break.

The value specifies the number of succeeding elements to print--a value of '1,' for examples, specifies that the next element must print on the same page, else page overflow puts them on the next.

Extra

Alone

Specifies that the HEADER section always prints alone on a page.

Absolute

Specifies that the HEADER section always prints at the same fixed position on the page. This adds the ABSOLUTE attribute to the DETAIL structure.

Position

Allows you to set the location and size of the group header. The location is relative to the top left corner of the detail print area.

The **Position** dialog allows you to specify the AT (set print structure position and size) attribute. Filling in the attribute manually is optional--you may set position visually, dragging with the mouse in the **Report Formatter's Page Layout View**.

To set the location and size, choose from the following options for the **Top Left Corner**, the **Width** of the section, and its **Height**.

Default	Sets a value based on the Paper Size and Print Detail position (see also **Report Properties** dialog).
Fixed	To set a specific position and size, mark the **Fixed** choices. The measurement units for these boxes are specified on the **General** tab of the **Report Properties** dialog.

Font

Calls the **Select Font** dialog which allows you to select the font (typeface), size, style (such as bold or italic), color, and font effects (underline and strikeout) for all the controls in the report section. You may override the section font by setting a different font in the Properties dialog for any specific control. As you choose options, the dialog box displays a sample of the selected font.

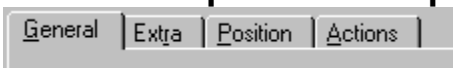
See also:

How to Set Report Group Breaks

Actions

There are no prompts for the Actions Tab for this control.

Break Group Footer Properties



Click on a TAB to see its help

This dialog allows you to edit the properties of the group FOOTER.

The group FOOTER is composed immediately after the group DETAIL, and provides the logical place for adding subtotals to your report.

General

Use — Type a field equate label to reference the FOOTER in executable code.

Page Before — To print the FOOTER structure on a new page, type a value in the **Page Before** box. This sets the PAGEBEFORE attribute. The print engine generates a page break before printing at the top of the next page.

The page number automatically increments, unless you reset it. To reset the page number to a value you specify, type it in the **Page Before** field.

Page After — To print the FOOTER, then force a new page, type a value in the **Page after** box. This sets the PAGEAFTER attribute. This prints the FOOTER, then begins a new page.

The page number automatically increments, unless you reset it. To reset the page number to a value you specify, type it in the **Page after** field.

With Prior — To prevent 'orphan' elements in a printout, type a value in the **With Prior** field. This sets the WITHPRIOR attribute. An 'orphaned' print element is one which prints on a following page, separated from its related items.

The value specifies the number of preceding elements to print--a value of "1," for example, specifies that the previous element must print on the same page.

With Next — To prevent 'widow' elements in a printout, type a value in the **Keep Next**. This sets the WITHNEXT attribute. A 'widowed' print element is one which prints, but then is separated from the succeeding elements by a page break.

The value specifies the number of succeeding elements to print--a value of '1,' for examples, specifies that the next element must print on the same page, else page overflow puts them on the next.

Extra

Alone — Specifies that the FOOTER section always prints alone on a page.

Absolute — Specifies that the FOOTER section always prints at the same fixed position on the page. This adds the ABSOLUTE attribute to the FOOTER structure.

Position

Allows you to set the location and size of the group footer. The location is relative to the top left corner of the detail print area.

The **Position** dialog allows you to specify the AT (set print structure position and size) attribute. Filling in the attribute manually is optional--you may set position visually, dragging with the mouse in the **Report**

Formatter's Page Layout View.

To set the location and size, choose from the following options for the **Top Left Corner**, the **Width** of the section, and its **Height**.

Default — Sets a value based on the Paper Size and Print Detail position (see also **Report Properties** dialog).

Fixed — To set a specific position and size, mark the **Fixed** choices.

The measurement units for these boxes are specified on the **General** tab of the **Report Properties** dialog.

Font

Calls the **Select Font** dialog which allows you to select the font (typeface), size, style (such as bold or italic), color, and font effects (underline and strikeout) for all the controls in the report section. You may override the section font by setting a different font in the Properties dialog for any specific control. As you choose options, the dialog box displays a sample of the selected font.

See also:

How to Set Report Group Breaks

Actions

There are no prompts for the Actions Tab for this control.

Page Form Properties

Click on a TAB to see its help

To specify a static page element which prints on every page, place it in the FORM. This is a free-floating section which can overlap the other sections.

Use the FORM as a layer, to 'hold' graphic frames or preprinted *forms* into which the data from the other sections 'fit.' Another use for the FORM is to hold a 'watermark,' which prints underneath the report.

The FORM size defaults to the same size as the page, less the margins.

The print engine composes the FORM at the beginning of the print job; and does not update it thereafter. Therefore, the FORM is not suitable for holding variable data fields, or even a page number. You can, however, print fields from a control file, if you wish to print the same field contents on every page of the report.

Tip: For best results when using a drawing tool to create a 'watermark,' on, for example, a 300 DPI printer, set the fill for the watermark element to 10% gray, or light gray. At higher printing resolutions, try 20% gray.

The FORM should guide the user to the data. You might use lines and boxes, for example, to divide the DETAIL into 'compartments,' grouping data and columns for the user. You may even create a 'greenbar paper' effect by alternating gray or light green color blocks.

General

Use Type a field equate label to reference the FORM in executable code.

Position

Allows you to set the location and size of the page form, by filling in the AT (set print structure position and size) attribute. The measurement units for these boxes are specified on the **General** tab of the **Report Properties** dialog.

To set a precise starting point for your page form relative to the top left corner of the paper, specify **Top Left Corner** coordinates with this dialog. In effect, this establishes the margins for your form. These settings may also be accomplished visually by dragging the form and it's borders in **the Report Formatter's Page Layout View**.

To set the size of the page form, choose from the following options for the **Width** and **Height**. When changing a report from portrait to landscape, or vice versa, you must change the width and/or height in the **Position** tab.

Default Sets a value based on the Paper Size.

Fixed To set a specific size, mark the **Fixed** choices.

Font

Calls the **Select Font** dialog which allows you to select the font (typeface), size, style (such as bold or italic), color, and font effects (underline and strikeout) for all the controls in the report section. You may override the section font by setting a different font in the Properties dialog for any specific control. As you choose options, the dialog box displays a sample of the selected font.

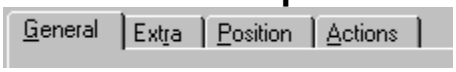
See also:

How the Print Engine Processes Report Sections at Runtime

Actions

There are no prompts for the Actions Tab for this control.

Detail Band Properties



Click on a TAB to see its help

This dialog allows you to edit the properties of the report detail.

General

Label — Type a valid Clarion label, naming the DETAIL structure.

Use — Type a field equate label to reference the Detail in executable code.

Page Before — To print the DETAIL structure on a new page, type a value in the **Page Before** box in the **Detail Properties** dialog. This sets the PAGEBEFORE attribute. The report prints the full DETAIL starting at the top of the next page. The report FOOTER, however, prints on the first page.

The page number automatically increments, unless you reset it. To reset the page number to a value you specify, type it in the **Page Before** field.

Page After — To print the DETAIL, then force a new page, type a value in the **Page after** box in the **Detail Properties** dialog. This sets the PAGEAFTER attribute. This prints the DETAIL, then prints the page FOOTER, then begins a new page.

Tip: To print a separate page for each record, place the variable strings and/or controls you wish in the DETAIL, and specify the PAGEAFTER attribute in the Detail Properties dialog.

The page number automatically increments, unless you reset it. To reset the page number to a value you specify, type it in the **Page after** field in the **Detail Properties** dialog.

With Prior — To prevent 'orphan' elements in a printout, type a value in the **With Prior** field. This sets the WITHPRIOR attribute. An 'orphaned' print element is one which prints on a following page, separated from its related items.

The value specifies the number of preceding elements to print--a value of "1," for example, specifies that the previous element must print on the same page.

Tip: When placing subtotals or totals in a DETAIL, use the WITHPRIOR attribute to insure they print with at least one member of the column above it when a page break occurs.

With Next — To prevent 'widow' elements in a printout, type a value in the **Keep Next**. This sets the WITHNEXT attribute. A 'widowed' print element is one which prints, but then is separated from the succeeding elements by a page break.

The value specifies the number of succeeding elements to print--a value of '1,' for examples, specifies that the next element must print on the same page, else page overflow puts them on the next.

Extra

Alone — Specifies the print engine should print *only* the DETAIL section, without FORM, HEADER, and FOOTER sections. This setting is most useful for report title and grand totals pages. This adds the ALONE attribute to the DETAIL structure.

Absolute Specifies that the DETAIL section always print at the same fixed position on the page. This adds the ABSOLUTE attribute to the DETAIL structure. Otherwise, the DETAIL prints at a position relative to the last section printed in the detail print area.

Position

Allows you to set the location and size of the print detail, by filling in the AT (set print structure position and size) attribute. The measurement units for these boxes are specified on the **General** tab of the **Report Properties** dialog.

To set a precise starting point for your detail relative to the last detail printed, specify **Top Left Corner** coordinates with this dialog. These settings may also be accomplished visually by dragging the report section and it's borders in **the Report Formatter's Page Layout View**.

To set the size of the detail, choose from the following options for the **Width** and **Height**.

Default Sets a value based on the Paper Size.

Fixed To set a specific size, mark the **Fixed** choices.

Font

Calls the **Select Font** dialog which allows you to select the font (typeface), size, style (such as bold or italic), color, and font effects (underline and strikeout) for all the controls in the report section. You may override the section font by setting a different font in the Properties dialog for any specific control. As you choose options, the dialog box displays a sample of the selected font.

See also:

How the Print Engine Processes Report Sections at Runtime

Actions

There are no prompts for the Actions Tab for this control.

New Project dialog

This dialog allows you to specify the method you will use to create a new project.

Quick Start	Specifies that the Quick Start will be used to create your Data Dictionary and Application.
Application Generator	Specifies that the Application Generator will be used to create your Application.
Hand Coded Project	Specifies that the Application will be hand coded and a Project file will be created.
Working Directory	Type in the full path for the directory or pick the directory using the standard file dialog.

Select Custom Control Dialog

This allows you to choose a .VBX Custom Control, from your VBX Registry, which you can then place in your window or report. When placing a .VBX control in a report, an image of the control prints on the page. To specify properties for the .VBX control that you place, see the **Custom Control Properties** dialog.

Control Name Choose a .VBX from a list of registered controls.

Sample Displays a scaled down image of the .VBX control in the dialog.

Registry Opens the .**VBX Custom Control Registry** dialog, which allows you to register a new control, or update a previously registered control.

Reminder: You must have the proper license (.LIC) file to work with the control in the Window and Report Formatters. See also .VBX Custom Control Support.

VBX Custom Control Support

Custom controls are "add-in" control components sold by many third party vendors. These perform a very wide variety of tasks, from sliders and gauge controls to TWAIN image capture add-ins. The **Window Formatter** allows you to directly place these controls in a window once you "register" the external libraries. To register a .VBX control, it must be in a directory in your path. See also: VBX Registry Dialog.

The specific custom control library Clarion supports is the Microsoft Visual Basic control format, normally given the .VBX extension. There is one important limitation:

- Clarion supports ."level one" type controls. Custom controls which *require* VB 2.0 or higher (Level two or three type controls) are incompatible.

  This is in line with other non-Visual Basic platforms, such as the Microsoft Foundation Classes v. 2.0. The biggest difference between level one and level two or higher .VBX's is that the latter contain calls to internal functions of the VB runtime .DLL which ships with Visual Basic 2.x and higher.

Tip: If the vendor description of a .VBX doesn't specifically state its level, you can immediately identify a level two or higher control if they identify it as a "data bound" control.

Custom control libraries usually require a license file (*.LIC) before you can add the control to your applications. The library vendor provides the file when you buy the library. When you distribute the application to your end users, you distribute the .VBX file only, not the license file.

Additionally, when you ship the .VBX file to your end users, follow the library vendor's instructions as to where to place the .VBX control file(s).

Custom Control Properties Dialog

Custom controls are "add-in" controls sold by many third party vendors. These perform a very wide variety of tasks, from sliders and gauge controls to TWAIN image capture add-ins. The **Window Formatter** allows you to directly place these controls once you "register" the external libraries. See also: VBX Custom Control Support ; Select Custom Control Dialog.

Before you can place a custom control in a window, you must register the .VBX file which contains it. See also: VBX Registry Dialog.

The **Custom Control Properties** dialog contains the following options.

Custom Properties — It displays the Visual Basic Control properties and their default values. If you enter a startup value in the dialog, the **Window Formatter** automatically adds it to the Clarion language statement that places the control in the window.

When you highlight a Visual Basic Control property in the list, either an edit box. or a drop-down list appears at the lower left corner of the dialog. Type a value or variable in the edit box, or choose from the drop-down.

The documentation from the .VBX library vendor will describe the Visual Basic Control properties you may set. See Setting and Retrieving VB Control Properties to find out how to change them in executable code, or how to retrieve user-entry from the custom control.

Text — Optionally type a label for the control in the **Text** field. If the control supports a label, it will appear as part of the control. In practice, most controls will require you to specify a title label as a Visual Basic Control property, explained below.

Use — Type a field equate label to reference the control in executable code.

Sample — Displays a scaled down image of the .VBX control in the dialog.

Cursor — The Cursor field (the CURSOR attribute) allows you to specify an alternate shape for the cursor when the user passes the cursor over the control. The drop-down list provides standard cursor choices such as I-Beam and Crosshair. To select an external cursor file (whose extension must be .CUR), choose **Select File** from the drop-down list, then pick the file using the standard file dialog.

Help ID — The **Help ID** field (the HLP attribute) takes a string constant specifying the key for accessing a specific topic in the Help document. This may be either a Help keyword or a context string.

A Help keyword is a word or phrase indexed so that the user may search for it in the Help **Search** dialog. When you fill in the HLP attribute for a button, if the group box has focus, when the user presses F1, the help file opens to the referenced topic. If more than one topic matches a keyword, the search dialog appears.

When referencing a context string in the **Help ID** field, you must identify it with a leading tilde (~).

Message — The **Message** field (the MSG attribute) allows you to specify text to display in the first zone of the status bar when the control has focus.

Meta — When adding a .VBX control to a report, specifies that the print engine generates

a metafile to represent the control.

Key Press to select a hot key to give immediate focus to the control.

Alert To specify a hot key active when the control has focus, press the **Alert...** button. The **Select Alert Keys** dialog appears. When the ALRT attribute is set, the control generates an EVENT:AlertKey if the user presses the Alert key while the control has the focus.

Position Press the **Position** button to display the **Position** dialog. That dialog allows you to set the location and size of the control.

Setting and Retrieving VB Control Properties

The .VBX file acts as a *mostly* self-contained external library. When the application loads it into memory, you can exchange information between the application and the custom control via the properties. The Visual Basic Control properties are a message map.

The .VBX properties are the most common means by which a non-VB application utilizes a VBX's functionality. Think of a property as a variable which both the app and the VBX can access (this is a very loose comparison).

If both the app and the VBX monitor the property, they can use it to signal each other. When the value of the property changes, it's a signal that something may need to be done. Each VBX has its own properties. You find out what properties are available by reading the VBX Vendor's documentation.

For example, a VBX has a property called 'CellColor,' which indicates the background color of a grid cell. If the app wants to know what the current color is, it retrieves the value in the property called 'CellColor.' Usually, it works the other way, too. If the app changes the value of 'CellColor' from blue to red, then the VBX updates the window control and changes the color.

Tip: The Visual Basic Control properties are usually documented with a leading dot. Drop the dot when accessing it from the Clarion application.

See the Custom Control Properties Dialog topic for notes on how to set the *startup* properties for a control with the **Window Formatter**. At other times you'll want to alter the properties in executable code, and of course, retrieve a value from a property after user entry.


To alter properties in executable code, use the property expression syntax. Access the custom control's Visual Basic Control properties by referring to the specific property in quotes:

```
?vbxVariable { 'VBProperty' } = value
```


To retrieve the current value of a Visual Basic Control property, use the property expression syntax again:

```
value = ?vbxVariable { 'VBProperty' }
```

VBX controls also generate an event. See also: Monitoring .VBX Events.

Monitoring .VBX Events

Besides properties, the other "channel" by which the .VBX "talks" to your application is via events. A .VBX might trigger an event, for example, if the end user double clicks on a particular part of it. When the event occurs, the .VBX generates a string (up to 255 characters) naming the event. The .VBX vendor's documentation lists the possible events the control may generate.

Your application can examine the event, and take appropriate action. When working with the Application Generator, you place code like the example below at the embed point labeled "After Opening the Window." For example, the following can take place in the ACCEPT loop of a dialog box containing a .VBX control.

```
CASE EVENT()
  OF EVENT:vbxevent
    SomeString = ?vbxVariable{PROP:VBXevent}
    IF SomeString = 'UserWantsToDoX'
      SomeProcedure
```

See also: Setting and Retrieving VB Control Properties.

VBX Custom Control Registry

Before you can place a custom control in a window, you must register the .VBX file which contains it. To do so:

1. Choose **Setup** **VBX Custom Control Registry**.

2. Press the **Add** button in the **VBX Custom Control Registry** dialog box.

3. Select the .VBX file within the **VBX Custom Control Open File** dialog, and press **OK**.

Some .VBX vendors install their .VBX's to the \Windows\System directory, while others prefer private directories. To register a .VBX, it *must* be in a directory in your PATH. When you install a .VBX library to your hard drive, make a note of where you put it, so that you can locate it with the Open File dialog.

4. Press **OK** to close the ***VBX Custom Control Registry*** dialog.

The VBX Custom Control Registry dialog contains these additional buttons:

Remove	Deletes the selected control from the registry. This does not delete the .VBX file from your drive.
Update	Updates the registry information for the selected control.

Database Driver Registry Dialog

The Clarion for Windows database drivers are pre-registered for you. Should you remove one and need to reinstall it, or if you obtain a new driver, you must use this dialog box to register the new driver. To do so:

1. Choose **Setup** **Database Driver Registry**.

2. Press the **Add** button in the **Database Driver Registry** dialog box.

3. Select the file driver, which normally has a DLL file extension, from the **Add Database Driver Open File** dialog, and press **OK**.

The database drivers belong in the \CW15\BIN subdirectory.

4. Press **OK** to close the **Database Driver Registry** dialog.

The dialog contains these additional buttons:

Remove	Deletes the selected driver from the registry. This does not delete the .DLL file from your drive.
Update	Updates the registry information for the selected driver.

Reminder: when distributing your application, you must ship the database driver library files used by your application. The file names are listed within this dialog.

See also:

Supported File Systems

Edit Picture String Dialog

This dialog allows you to quickly choose and customize a picture token.

Picture tokens provide a masking format for displaying and editing variables. Picture tokens may be used as parameters of STRING, ENTRY, or STRING OPTION declarations in SCREEN structures; as a parameter of STRING statements in a REPORT structure; as a parameter of some Clarion procedures and functions; or, the parameter of STRING, CSTRING and PSTRING variable declarations. There are seven types of picture tokens:

Numeric and Currency Pictures

Scientific Notation Pictures

Date Pictures

Time Pictures

Pattern Pictures

Key-in Template Pictures

String Pictures

Picture Type Choose one of the picture token types from the drop down list.

Picture The actual picture string under construction.

You can directly edit the string, or edit it by specifying options in the dynamic controls that appear in the lower part of the dialog box for individual picture token types.

The options are self explanatory, such as **Number of Characters**, **Length**, and **Decimal Places**.

List Box Formatter Dialog

The List Box Formatter dialog shows how the list box under construction looks. It fills this sample list box with placeholder characters representing the contents of each field. If any field contains a header, a header row appears over the column.

You format the fields one by one in the **List Field Properties**. The sample list box always displays a horizontal scroll bar, whether you specify one in the **List Properties** dialog or not.

The formatter does *not* display a vertical scroll bar. If the queue contains more items than rows in the list, and if you add the VSCROLL attribute by checking the box in the **List Properties** dialog, the vertical scroll bar appears at run time.

The dialog contains the following buttons:

Properties	Opens the **List Field Properties** dialog for the selected field. The **List Field Properties** dialog allows you to specify the width of the column, a picture token, heading text, plus other options such as a horizontal scroll bar for the single field. Additionally, it allows you to "group" fields, which places an extra header on top of the grouped columns, to visually indicate the fields are linked.
Insert/Populate	To add a field to the list box, press the **Populate** or **Insert** buttons. When opening the **List Box Formatter** from within the Application Generator, using a procedure template which supports it, the **Populate** button displays the **Select Field** dialog. From here, you can indicate any database field or memory variable for use as a list box column. The generated code puts the contents of the database records into the queue for use in the list box. Once you indicate the field you want, the **List Field Properties** dialog allows you to format its appearance in the list box. When opening the **List Box Formatter** from a hand coded source file, the **Insert** button replaces the **Populate**. You must build your own queue to fill the list box.
Delete	Removes the currently selected field from the list box.
(Arrow Left)	Moves the currently selected field one position to its left. If the selected field is the leftmost in a group, the field moves out of the group, without changing position. If the selected field is immediately to the right of a group, the field moves into the group, without changing position.
(Arrow Right)	Moves the currently selected field one position to its right. If the selected field is the rightmost in a group, the field moves out of the group, without changing position. If the selected field is immediately to the left of a group, the field moves into the group, without changing position.

See the following topics for more information about adding list box functionality:

How to Trap a Double Click on a List Box

How to add Drag and Drop to a List Box

List Field Properties Dialog

Click on a TAB to see its help

Format list box fields or columns in the **List Field Properties** dialog. Each choice you make in the dialog places the appropriate component in the format string (the parameter of the FORMAT attribute) which becomes part of the LIST statement.

The dialog allows you to set the following formatting options.

General

Width Specify the width in dialog units for the column. By default, the Formatter sets the value to four times the number of characters specified in the field picture in the data dictionary. For variables, the default is four times the number of characters in the picture token defined for it.

Tip: As a rough guide, allow four dialog units for an average character. For example, if you want a column 10 characters wide, type 40 in the Data Width field.

After you've placed one field, the **List Box Formatter** dialog also allows you to drag the column separators to resize a column. The cursor changes when you place it on top of the separator, to indicate you can resize it.

The data width you set appears within the format string for the field, preceding the Justification code, as in "40L."

Picture Specify the picture token for the data. The **List Box Formatter** represents the contents according to the picture you specify. For example, a string picture token displays dollar signs in the sample list box.

The picture token you specify appears in the format string, for example, "@S10@."

Justification Choose from the drop down list to specify left, right, center or decimal. If you use decimal justification, you set the Offset to allow display of digits to the right of the decimal point.

The justification appears within the format string for the field following the data width, as in "40R."

Indent Optionally specify an indent, in dialog units, for the list box data. The indent operates in the opposite direction of the justification.

The indent appears within the format string surrounded by parentheses and preceded by a letter indicating the justification, as in "L(8)."

Special

Color Cells Check this box to enable colorization of list box cells.

Icons Check this box to enable the use of Icons in the list box.

Tree Check this box to display the list box in Tree format. See Relation Tree control template.

Show Level For data in Tree format, each subordinate level is indented from its parent level.

Show Lines For data in Tree format, connecting lines are drawn between related items.

Show Boxes For data in Tree format, "expand" (+) and "contract" (-) boxes are drawn for each item. The plus (+) indicates the control is contracted and may be expanded by CLICKING on the plus (+). The minus (-) indicates the control is expanded and may be contracted by CLICKING on the minus (-).

Field

Heading Text Optionally specify header text for the column. The header appears as a gray row above the list box data items. To specify no header, leave this field blank. If any field included in the list box has a header, a header appears over each field in the list box; those fields with no header text will have a blank header.

The heading appears within the format string enclosed in tilde (~) characters, as in "~My Header~."

Justification Choose from the drop down list to specify left, right, center or decimal. If you use decimal justification, you set the Offset to allow display of digits to the right of the decimal point.

This appears within the format string following the header, as in "~My Header~L."

Indent Optionally specify an indent, in dialog units, for the heading text. The indent operates in the direction opposite to the justification.

This appears within the format string following the header, as in "~My Header~L(8)."

Scroll Bar Check the **Scroll Bar** box to specify a horizontal scroll bar for this column only. If the overall list box already has a scroll bar, the column scroll bar appears above the list box scroll bar.

Size Specify the range of the scroll bar. That is, the size of the scrollable area not currently displayed.

The scroll bar and size appear in the format string together, as in "S(4)."

Underline Check the **Underline** box to add the underline style to the list box text.

The format string includes the underscore character, immediately preceding the header text, as in "_~My Header~."

Right Border Check the **Right Border** box to specify column separators between fields in the list box at run time. The format string includes the pipe symbol (|), immediately preceding the header text, as in "|~MyHeader~."

Resizeable Check the **Resizable** box to specify that user can resize the width of the columns at run time.

The format string includes the "M" character, immediately preceding the header text as in "M~MyHeader~." See How to Restore User Resized List Box Column Widths...

Fixed Check the **Fixed** box to specify that the column cannot be scrolled. The format string includes the "F" character, immediately preceding the header text as in "F~MyHeader~."

Last on Line This specifies that the next field in the group will appear immediately below the

current field. In effect, it stacks two or more fields, causing a single record to occupy two or more rows in the list box.

The format string includes the "/" character, immediately preceding the header text as in "/~MyHeader~."

Locator To enable a column to work with a locator entry control, check the **Locator** box. When the user types a character in the locator entry control, then moves the focus, the list box scrolls to the first entry in the locator enabled column that matches the user's locator entry.

The format string includes the "?" character, immediately preceding the header text as in "?~MyHeader~."

Group

Heading Text Optionally specify header text for the column group. The header appears as a gray row above the list box data items. To specify no header, leave this field blank. If any field included in the list box has a header, a header appears over each field in the list box; those fields with no header text will have a blank header.

The heading appears within the format string, after the column group (the column group is enclosed in braces []), enclosed in tilde (~) characters, as in "~My Header~."

Justification Choose from the drop down list to specify left, right, center or decimal. If you use decimal justification, you set the Offset to allow display of digits to the right of the decimal point.

This appears within the format string, after the column group (the column group is enclosed in braces []), following the header, as in "~My Header~L."

Indent Optionally specify an indent, in dialog units, for the heading text. The indent operates in the direction opposite to the justification.

This appears within the format string, after the column group (the column group is enclosed in braces []), following the header, as in "~My Header~L(8)."

Scroll Bar Check the **Scroll Bar** box to specify a horizontal scroll bar for this column group. If the overall list box already has a scroll bar, the column group scroll bar appears above the list box scroll bar, and overrides any individual column scroll bars within the group.

Size Specify the range of the scroll bar. That is, the size of the scrollable area not currently displayed.

The scroll bar and size appear in the format string, after the column group (the column group is enclosed in braces []), together, as in "S(4)."

Underline Check the **Underline** box to add the underline style to the list box text.

The format string includes the underscore character, after the column group (the column group is enclosed in braces []), immediately preceding the group header text, as in "_~My Header~."

Right Border Check the **Right Border** box to specify column separators between fields in the list box. The format string includes the pipe symbol (|), after the column group (the column group is enclosed in braces []), immediately preceding the header text, as in "|~MyHeader~."

Resizeable	Check the **Resizable** box to specify that user can resize the width of the columns at run time. The format string includes the "M" character, after the column group (the column group is enclosed in braces []), immediately preceding the header text as in "M~MyHeader~." See How to Restore User Resized List Box Column Widths .
Fixed	Check the **Fixed** box to specify that the column cannot be scrolled. The format string includes the "F" character, after the column group (the column group is enclosed in braces []), immediately preceding the header text as in "F~MyHeader~."

See also:

How to Create a List Box

How to Create Column Groups Using the List Box Formatter

How to Make a Record Occupy Two or More Rows in a List Box.

Alert Keys Dialog

This dialog allows you to add the ALRT attribute to a window or control. When the attribute is set, the window generates an EVENT:AlertKey if the user presses the key(s) you specify in this dialog, while the window has the focus.

To specify the first Alert key:

Add	Press the **Add** button. Specify the key or key combination with the **Input Key** dialog. The key combination appears in the **List**. Repeat for any additional Alert keys.
Remove	To delete a key combination, highlight the key combination in the **List**, then press the **Remove** button.

It's up to you to add code to "do something" upon detecting the EVENT:AlertKey.

Input Key Dialog

Use this dialog to specify a key, or key combination, for a hot key (KEY attribute) or an alert key (ALRT attribute):

Key

Press the desired key or key combination (for example, CTRL+H). The keys you pressed will appear in the **Key** field, and will be supplied as parameters to the KEY or ALRT attribute for this control.

The ESC, ENTER, and TAB keys *cannot* be specified by pressing them. For these keys, press the ellipsis (...) button and type "esc," "enter," or "tab."

Modifiers

Optionally, add additional keys to your key sequence by checking the **Ctrl**, **Alt,** or **Shift** boxes, or any combination of the three.

Mouse

Mouse clicks may be used within the key sequence; however, mouse clicks *cannot* be specified by clicking the mouse. For mouse clicks, check the corresponding check box(es). For example, to act on a double-click, check the **Left Button** box *and* the **Double Click** box.

Select Font Dialog

Allows you to change the font, style (such as bold and italic), font size, color, and font effects (underline and strikeout) for the selected control or window. As you choose options, the dialog box displays a sample of the formatting.

Choose from the following:

Font	Type or select a font (typeface) name. The dialog lists the fonts available with the current printer driver and additional fonts installed in your system.
Font Style	Select a style. To use the default type style for a given font, select Regular. Depending on the fonts installed, you can choose bold, italic, or bold italic.
Size	Type or select a size. The sizes available depend on the printer and the selected font.
Effects	Select the formatting options you want. Choose: **Underline** - underlines all characters, including the spaces between words, with a single line. **Strikeout** - draws a line through selected text.
Color	Type or select one of the sixteen predefined colors. To display color, you must have a color monitor; to print color, you must have a color printer.

Standard Windows File Dialog

This dialog allows you to specify a filename and directory for a file which you wish to Create, Open, Save to, or Select. The Title Bar indicates the function for which it is intended.

The Pick List

The **Pick** dialog lists all the most recently used files in a list box categorized by **Application, Dictionary, Project, Database, Clarion Source,** and **All**. Each of these tabs displays a pick list of up to twenty of the most recently used files of that type:

The **Pick** dialog provides the following buttons:

Select	Opens the currently selected file.
Remove	Removes the currently selected file from the Pick list.
New	Allows you to create a new file.
Open	Allows you to open a file not on the Pick list.

Menu Editor

The **Menu Editor** dialog visually represents a Clarion MENUBAR data structure. The menu tree (on the left hand side of the dialog) appears as simplified Clarion language syntax, containing these Clarion keywords:

A MENUBAR keyword at the top.

A MENU statement or statements followed by a menu name, and a corresponding END statement.

An ITEM statement or statements followed by an item name.

Menu Editor command buttons allow you to add and delete MENUs and ITEMs. You may also move MENUs and ITEMs within the MENUBAR structure with the and ↓ buttons.

The right hand side of the dialog allows you to specify the text of your MENUs and ITEMs, the equate labels used to reference the MENUs and ITEMs in executable code, and the actions that occur when the user selects an ITEM.

Tip: When using the Application Generator, each ITEM you place on a MENU or MENUBAR automatically adds an embed point to the control event handling tree in the Embedded Source dialog. This allows you to easily attach functionality to your ITEMs.

Menu Editor Buttons

Menu This button adds a new MENU statement, its Menu Text, and its corresponding END statement. The MENU is added after the highlighted line. MENUs may be nested within other menus. MENUs may contain MENUs or ITEMs.

Item This button inserts an ITEM after the highlighted line. Note that ITEMs are used to execute commands or procedures, whereas MENUs are used to display a selection of other MENUs or ITEMs.

Separator To add a separator bar after the currently highlighted MENU or ITEM, press the **Separator button**.

Tip: Separator bars can provide the user with a visual cue that a group of ITEMs on the menu perform related functions.

Delete Button To delete the currently highlighted MENU, ITEM, or SEPARATOR, press the **Delete** button. If you delete a MENU statement, all ITEMs and MENUs within it, and its associated END statement are also deleted.

and ¯ **Buttons** To move the currently highlighted MENU, ITEM, or SEPARATOR up or down in the menu list, press the or ¯ button. When moving a MENU, all ITEMs and MENUs within it, and its associated END statement move also.

General

Menu Text Type the text you want displayed for this MENU or ITEM. For example, type &FILE, so the end user sees **F**ile. The ampersand within the Menu Text field signifies the character following the ampersand is the accelerator key. That is, the

character is underlined, and, when the user presses the accelerator key, the action associated with the ITEM is executed.

Note: A MENU accelerator key requires THE ALT key to take effect, whereas an ITEM accelerator key does not require the ALT key, but does require that the ITEM be currently displayed. See Hot Key below for another method of accessing your MENUs and ITEMs.

Use Variable Type a Field Equate Label.

A Field Equate Label has a leading question mark (?), and you should make it descriptive. For example ?File shows this menu is to manipulate a file. You can refer to the MENU within executable code by its Field Equate Label.

Message Type the MSG attribute contents.

This message text displays in the status bar (if enabled) when the user highlights this MENU or ITEM.

Help ID Type either a help keyword or a context string present in a .HLP file.

If you fill in the **Help ID** for a MENU or an ITEM, when the user highlights the MENU or ITEM and presses F1, the help file opens to the referenced topic.

The **Help ID** field (HLP attribute) takes a string constant specifying the key for accessing a specific topic in a Windows Help file. This may be either a Help keyword or a context string. When referencing a context string in the **Help ID** field, you must identify it with a leading tilde (~).

STD ID To specify a standard windows action for your menu ITEM, enter one of the equates listed below in the **Std ID** field. Clarion will automatically implement the command using standard windows behavior; you do not need any other support for it in your code. The standard equate labels and their associated actions are also contained in the C:\CW\LIBSRC\EQUATES.CLW file.

STD:PrintSetup Printer Options Dialog.
STD:Close Closes active window.
STD:Undo Reverses the last editing action.
STD:Cut Deletes selection, copies to clipboard.
STD:Copy Copies selection to clipboard.
STD:Paste Pastes clipboard contents at the insertion point.
STD:Clear Deletes selection.
STD:TileWindow Arranges child windows edge to edge.
STD:TileHorizontal Arranges child windows edge to edge.
STD:TileVertical Arranges child windows edge to edge.
STD:CascadeWindow Arranges child windows so all title bars are visible.
STD:ArrangeIcons Arranges iconized child windows.
STD:WindowList Adds child window names to menu.
STD:Help Opens .HLP file to the contents page.
STD:HelpIndex Opens .HLP file to the index.
STD:HelpOnHelp Opens Microsofts .HLP file for the Windows Help system.
STD:HelpSearch Opens Microsofts Help Search utility for

the .HLP file.

Position Allows you to specify MENU and ITEM order priority when Clarion merges menus. The choices are:

To allow normal ordering when merging menus, choose ***Normal*** from the **Position** drop down list. In normal merging, Global selections precede Local selections. See *Merging Menus* in the *Users Guide*.

To force the selected MENU or ITEM to the first position when merging menus, choose ***First*** from the **Position** drop down list. This adds the FIRST attribute to the MENU or ITEM statement.

To force the menu or item to the last position when merging menus, choose **Last** from the **Position** drop down list. This adds the LAST attribute to the MENU or ITEM statement. See the *Language Reference* for more information.

Hot Key Press this button to open the **Input Key** dialog. Use this dialog to add the KEY attribute to your MENU or ITEM. The KEY attribute specifies a hot key or key combination.

A hot key is very similar to an accelerator key. A hot key or hot key combination allows the end user to immediately display a MENU, or execute the action associated with an ITEM, without mouse clicking, and without displaying the menu that contains the ITEM. Customarily, hot keys take the form of CTRL + *character*, or CTRL + SHIFT + *character*.

Tip: You may want to add the hot key combination to the menu text to signal its availability to the user. See the *Windows Design* appendix in the *Users Guide* for a list of common hot keys associated with standard windows commands.

Disable Item To disable a MENU or ITEM (dim the text and make it unavailable to the user), check the **Disable Item** box. This adds the DISABLE attribute to the MENU or ITEM statement.

Tip: The Disable box is handy when you incorporate modality into a programthat is, when one type of child window does *not* support the same commands another type does. For the type that doesnt support the command, disable the ITEM rather than omitting it. This will avoid confusing the user with menu ITEMs that disappear and reappear depending on which window is active.

Toggle (on/off) Item To create an on/off toggle for a selected ITEM, check the **Toggle (on/off) Item** box. The ITEM should have a numeric variable in the **Use Variable** field. The variable should be declared using one of the data dialogs, or in embedded source. The **Menu Editor** adds the CHECK attribute to this ITEM.

With the CHECK attribute, when the user selects the ITEM for the first time, the ITEM is on, the Use Variables value is one (1), and a check mark appears beside the ITEM. When the user selects the ITEM a second time, the ITEM is off, the Use Variables value is zero (0) and no check mark is displayed. You should add source code to control the applications behavior depending on the state of the Use Variable.

Right Justify To right justify the selected MENU on the action bar, check the **Right Justify** box. This is available only for MENUs on the action bar. Nested MENUs (subMENUs) cannot be right justified. Checking this box displays the selected MENU, and all MENUs after the selected MENU, at the far right of the action bar.

Do Not Merge To tell Clarion never to merge this MENUBAR with other MENUBARs, check the **Do Not Merge** box. This is available only for the MENUBAR, not for MENUs or ITEMs. See the *Language Reference* for more information on the NOMERGE attribute.

Actions

Use the Actions tab to add functionality to your menu item. Filling in these prompts causes the menu item to execute an action when the user selectss the menu item.

When Pressed From the drop down list, choose *Call a Procedure, Run a Program*, or *No Special Action*.

The procedure or program you specify executes when the user selects the menu item. The choices are:

Call a Procedure You must specify the **Procedure Name**, and whether the procedure will **Initiate a Thread**.

Procedure Name From the **Procedure Name** drop down list, choose an existing procedure name, or type a new procedure name. A new procedure appears as a "ToDo" item in your Application Tree.

Initiate a Thread Optionally check the **Initiate a Thread** box. If the procedure initiates a thread, specify the Thread Stack size. Clarion uses the START function to initiate a new execution thread. If the procedure initiates a thread, you cannot specify **Parameters** or **Requested File Action**. If the procedure does not initiate a thread, you can specify **Parameters**, **Requested File Action**, or both.

Tip: A MENU ITEM on an application frame toolbar that calls an MDI child procedure must initiate a thread.

Thread Stack Accept the default value in the **Thread Stack** spin box unless you have extraordinary program requirements. To change the value, type in a new value or click on the spin box arrows.

Parameters In the **Parameters** field, optionally type a list of variables or data structures passed to the procedure.

Requested File Action

From the drop down list, optionally select ***None, Insert, Change, Delete***, or ***Select***. The default selection is ***None***. The Global Request variable gets the selected value. The called procedure can then check the value of the Global Request variable and perform the requested file action.

Run a Program You must specify the **Program Name**, and optionally, any parameters.

Program Name Type the program name.

Parameters Optionally type a list of values that are passed to the program.

No Special Action Choose this option if you are providing your menu item's functionality with another method, such as embedded source, or an STD ID.

Note: You may combine a procedure or program call with embedded source, but not with an STD ID.

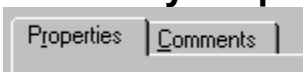
Files	Accesses the **File Schematic Definition** dialog for this procedure.
Embeds	Allows you to embed source code at points surrounding the event handling for this menu item only.

Order Controls Dialog

The **Order Controls** dialog displays all controls on the window in a hierarchical list. Reorder the controls, and their tab key order by selecting a control and pressing the and ¯ buttons to move the control up or down within the list.

Tip: This dialog is useful for moving controls among overlapping TAB controls on a SHEET or moving a control off a TAB and onto the WINDOW.

Dictionary Properties Dialog



Click on a TAB to see its help

Displays information about the current data dictionary, including creation, modification dates, and a text description. Push the **Password** button to password protect your dictionary from modification by other users.

Properties

Created The original file creation date.

Last Modified The most recent modification date. See also: **Version Checkpoint**; **Version Revert**.

Comments

Allows you to enter a text description describing the dictionary. The description is solely for your convenience, and has no effect on the application. It is useful for situations in which other programmers may pick up your code later, or for when you expect to return to the project after a long period of time since you last looked at it.

Password

Calls the **Password Validation** dialog which enables password protection for your dictionary

---

See also: How to Create a Data Dictionary

New/Edit File Properties Dialog

Click on a TAB to see its help

This dialog allows you to add a new data file to the list and choose its file driver.

Once the file appears on the list, you may declare fields, keys, set relationships, and other properties for the data file. Using the data from this dialog, the Application Generator will write the FILE structure declaration.

To modify the file properties at a later time, highlight the file name on the **Dictionary** dialog list, then either DOUBLE-CLICK or press the **Properties** button.

General

Name Type a data file name, as you wish to refer to it in your code. This serves as the label for the Clarion FILE structure. Specify a valid Clarion label--Clarion will automatically truncate the name if necessary. You may also specify a completely different name for the DOS file--see **Full Pathname**, below.

Description Enter a string description for the file. Clarion automatically displays the descriptions in certain dialogs, allowing you to quickly recognize the file contents.

Prefix As you enter the data file **Name**, Clarion automatically extracts the first three letters to use as a label prefix when referring to the file. Optionally specify up to 14 characters of your choice in this field.

The prefix allows your application to distinguish between similar variable names occurring in different data structures. A field called *Invoice* may exist in one data file called *Orders* and another called *Sales*. By establishing a unique prefix for *Orders* (ORD) and *Sales* (SAL), the application may refer to fields as ORD:INVOICE and SAL:INVOICE.

File Driver Specify the data file type. When using the Application Generator, Clarion for Windows automatically links in the correct database file driver library. See *Database Drivers* for a discussion of the relative advantages of each driver. You can specify the default driver by choosing **Setup** **Dictionary Options**.

Remember that individual file drivers may vary in their support of some of the attributes which you add to the FILE structure in this dialog box.

Driver Options Optionally type a string for an additional driver attribute. This conveys additional instructions to the file driver and corresponds to the second parameter for the DRIVER attribute, also known as a "driver string." *Database Drivers* contains additional information.

Owner Name Optionally type a string containing the password for access to the file. This is dependent on the file system. This adds the OWNER attribute to the FILE statement. You must also check the **Encrypt** box (below).

Encrypting the file means that only your application will be able to read the file. It does not mean that it automatically prompts the end user for a password. The end user, however, may not access the data with any other file viewer.

When using the ODBC driver, type the data source name, user ID, and password, separated by commas, in this field. See How to Create a File Definition for an ODBC Data Source for further information.

Full Pathname Type either the path, or a fully qualified file name for the data file. If you leave the file name out, Clarion automatically uses the first eight letters of the name entered in the **Name** field. You may also omit the file extension--Clarion will supply the correct extension depending on the file driver chosen. This supplies the parameter for the NAME attribute.

When using the TopSpeed driver, if you wish to store multiple tables in a single physical file, separate the file and table names with "`\!`," as in `TUTORIAL\!ORDERS`. This refers to the ORDERS table in the TUTORIAL.TPS file.

See TopSpeed:Storing multiple Tables (data files) in a single DOS file. for further information.

When using an ODBC driver to define a FILE such as Microsoft Access, which can store multiple tables in a single file, place the table name in this field. Typically, the name of the physical file which includes the table is listed in the ODBC.INI file; the ODBC driver manager provides this information to the driver.

Tip: To specify a variable name for the actual file name, place it in this field, and prefix the variable name with an exclamation point (!).

Enable File Creation Optionally specify that the application should create the data file if it does not exist at runtime. This adds the CREATE attribute to the FILE statement.

Reclaim Deleted Records This option is dependent upon the file driver. It specifies that the application reuse file space formerly taken up by deleted records. Otherwise, the application adds new records to the end of the file. This adds the RECLAIM attribute to the FILE statement.

Encrypt Data Records Optionally turn on file encryption. You must also specify an **Owner Name** (see above). This adds the ENCRYPT attribute to the FILE statement.

Open in Current Thread Optionally specify that each execution thread in your application that uses this file allocates memory for its own separate record buffer. This is typically for use in multiple document applications, and improves file handling. The Clarion default templates automatically add the THREAD attribute on each FILE structure.

Use OEM Collation Specifies string data is converted from OEM ASCII to ANSI when read from disk and ANSI to OEM ASCII before writing to disk. This adds the OEM attribute to the file definition.

Enable Field Binding Optionally specify that all variables in the RECORD structure are available for use in dynamic expressions at runtime. The compiler will allocate memory to hold the full Prefix:Name for each variable, instead of using its own internal reference for each variable. Therefore the BINDABLE attribute increases the amount of memory necessary for the application.

See also:

How to Design Your Dictionary and Database

How to Create a Data Dictionary

Comments

Allows you to enter a text description describing the dictionary. The description is solely for your convenience, and has no effect on the application. It is useful for situations in which other programmers may pick up your code later, or for when you expect to return to the project after a long period of time since you last looked at it.

Options

Do Not Auto-Populate This File

Directs the wizards to skip this file when creating primary Browse procedures or Report procedures.

User Options

User Options are provided to enable you to provide information to be used by a third-party template set. User Options are comma delimited, that is, each entry is separated by a comma.

Follow the instructions provided with your add-on template set.

New File Alias Dialog

Click on a TAB to see its help

An alias creates a second reference for a file without duplicating the file on disk. You can add an alias for a file only if it is already on the Dictionary list.

General

Name — Type a data file "name", as you wish to refer to it in your code. The name must be a valid Clarion label.

Description — Enter a string description for the alias. Clarion displays the descriptions in dialogs such as the **Dictionary** dialog.

Press the **>>** button to type a separate description (up to 1000 characters) in a larger text box. See also: **Edit File Description** dialog.

Prefix — By default, Clarion will use the first three letters of the Name for the prefix. Optionally specify up to 14 characters to use as a Prefix.

Alias File — Choose a file from the drop down list. This is the *original* file that the alias "references." The drop down list shows only the files previously defined using the **Add File** command in the **Dictionary Properties** dialog.

Comments

Allows you to enter a text description describing the dictionary. The description is solely for your convenience, and has no effect on the application. It is useful for situations in which other programmers may pick up your code later, or for when you expect to return to the project after a long period of time since you last looked at it.

Options

Do Not Auto-Populate This Aliased File
Directs the wizards to skip the Aliased File when creating primary Browse procedures or Report procedures.

User Options — User Options are provided to enable you to provide information to be used by a third-party template set. User Options are comma delimited, that is, each entry is separated by a comma.

Follow the instructions provided with your add-on template set.

A file alias provides several advantages, at the cost of some system overhead:

It allows you to set multiple relationships between files.
Strict relational database theoreticians state a file may only have a single relational link to another file at a time. Aliases allow you to "legally" work around this limitation. See also: How to Design Your Dictionary and Database

It allows a second file buffer for the same file.
You could use this for a second file browse, as well as entry forms and other items for each. This is particularly useful for an MDI application.


On the negative side, the second file buffer takes up additional memory and resources.

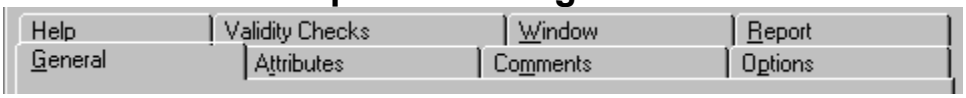
Any file driver utilizing external key files requires additional file handles for each alias. For example, a file with three external keys and three aliases requires sixteen file handles: one each for the "first" data file and its three keys, and an additional four for each of the aliases. When using aliases, we recommend choosing a file driver that stores keys internally, such as TopSpeed or Btrieve.


When using aliases, you must open the file in Share mode.

To modify the alias properties at a later time, highlight the file name on the **Dictionary** dialog list, then either double-click or press the **Properties** button.

You can edit the fields and keys for the Alias by pressing the **Fields/Keys** button. The **Field/Key Definition** dialog lists the fields and keys for the *original* file; any changes you make will update the originals.

New/Edit Field Properties Dialog



Click on a TAB to see its help

The **New Field Properties** dialog allows you to define fields and variables, and to set field or variable related options and attributes.

All the Clarion language attributes that you can place on a field also apply to memory variables. There are only a few additional attributes that can *only* be placed on global or local memory variables. These are disabled when defining a field.

The Dictionary Editor allows you to add the fields one after another, quickly. Each time you complete and close the **New Field Properties** dialog for one field, another blank **New Field Properties** dialog appears, ready for the next field. Press **Cancel** when the blank dialog appears after completing the last field, to return to the **Field/Key Definition** dialog.

General

Field Name — *To name the field*, type a valid Clarion label in the **Name** field. Valid field names may vary slightly according to the file driver.

Description — *To add a text description*, type it in the **Description** field. The description appears next to the field name in various dialogs. You can optionally assign the description to the MSG attribute by choosing **Setup** **Dictionary Options** and checking the appropriate box..

Press the **>>** button to type a separate description (up to 1000 characters) in a larger text box. See also **Edit Field Description** dialog.

Data Type *To assign a field type*, choose one from the **Type** drop down list. Clarion supports the following field types, which specify how the data will be stored on disk by the file driver, and accessed in memory by the application. These correspond to the Clarion variable types, plus memo and picture fields. The types available vary according to the selected file driver. See also: Variable Declaration Statements for a complete list of data types available.

Tip: **The Decimal type generally provides the best all around performance for mathematical calculations. The compiler optimizes the operation by multiplying values by powers of ten before processing; this greatly speeds up performance on systems without math coprocessors, at no cost in mathematical precision. See also: BCD Operations and Functions.**

Binary — *To specify that a MEMO field may hold binary data*, check the **Binary** box. This is dependent on the file driver. This adds the BINARY attribute.

Characters — *To assign a field length*, specify a number in the **Chars** field.

Places — *To assign a set number of decimal places for a real number*, specify a number in the **Places** field.

Dimensions — *To declare the variable as an array*, and to specify the array dimensions, type them in the **Dimensions** fields. You can specify up to four dimension sizes. This adds the DIM attribute.

Record Picture — *To specify the picture for a picture field*, type it in the **Record Picture** field.

Reference — *To create a reference variable*, check the **Reference** box. A reference variable

stores the memory address of another variable. This box is enabled only when defining memory variables

Record Picture — *To specify the picture for a picture field*, type it in the **Record Picture** field.

Screen Picture — *To specify a screen picture, type* it in the **Screen Picture** field.

"Lock" icon — *To lock the screen picture*, which specifies that it may not be changed if the field type is changed, press the "Lock" icon next to the **Screen Picture** field.

Default Prompt — *To specify the default prompt string*, type it in the **Default Prompt** field. The Application Generator utilizes this for controls which display an on screen prompt.

Column Heading — *To specify the default column title*, type it in the **Column Heading** field. The Application Generator utilizes this for reports.

Attributes

Case — *To specify the case attribute for controls referencing the field*, choose from the **Normal**, **Capitals** or **Uppercase** radio buttons, in the **Case** group box. The Application Generator adds the CAP or UPR attributes.

Typing Mode — *To specify the default typing mode attribute for controls referencing the field*, choose from the **Insert**, **Overwrite** or As Is radio buttons in the **Typing Mode** group box. The Application Generator adds the INS or OVR attributes.

Flags

Immediate — *To specify immediate event notification for controls referencing the field*, check the **Immediate** box. The Application Generator adds the IMM attribute.

Password — *To specify the data non-display attribute for controls referencing the field*, check the **Password** box. The Application Generator adds the PASSWORD attribute. When an end user types in an entry control referencing this field, the characters typed do not appear on screen.

Read only — *To specify the display only attribute for controls referencing the field*, check the **Read only** box. The Application Generator adds the READONLY attribute.

Justification — *To specify justification for controls referencing the field*, select from the **Alignment** drop down list. The Application Generator adds the LEFT, RIGHT, CENTER or DECIMAL attribute.

Offset — *To specify an indentation amount for controls referencing the field*, specify a number in the **Indent** field. The Application Generator uses this setting as the parameter for the LEFT or RIGHT attribute. The measurement unit depends on the default measurement unit for the window in which a control referencing the field resides.

Initial Value — *To specify a default value for the field*, type it in the **Initial Value** field. Note: you must enclose strings in single-quote marks.

Tip: You can type a function in the Initial Value field for fields in Data Files. If the field, for example, is a date field, you can add the TODAY() function to specify the initial value defaults to today's date. You can also type in a variable name, by prefacing it with an exclamation point. For Global, Local, or Module variables the initial value can only be a

constant value. To assign the value from a function, use a formula or embedded source code.

External Name — *To specify an external name for the field,* type it in the **External Name** field. This covers cases where the field label within the program is different than the name of the field in the data file; for example, you may be accessing a field through an ODBC connection to a database which allows field names longer than the maximum for a Clarion label. Place the name of the field as it exists in the data file here. See also: How to Test Your ODBC Application

Place Over — *To declare the field as an overlay*, select another field name from the drop down list. This allows the current field to redefine the other field's location in memory. The Application Generator adds the OVER attribute.

Allocation — The **Allocation** drop down list is enabled only when defining memory variables(global, module, or local). They set the EXTERNAL, STATIC, and AUTO attributes for memory variables.

Comments

Allows you to enter a text description describing the dictionary. The description is solely for your convenience, and has no effect on the application. It is useful for situations in which other programmers may pick up your code later, or for when you expect to return to the project after a long period of time since you last looked at it.

Options

Do Not Auto-Populate This Field
Directs the wizards to skip this field when creating Form, Browse or Report procedures.

Population Order — Specifies the order in which the wizards populate fields. Choose **Normal, First,** or **Last** from the drop down list. Wizards populate in this order: all Fields specified as First, then all Fields specified as Normal, and finally all Fields specified as Last.

Form Tab — Specifies the TAB onto which the wizards populate the field. Type the Caption for the TAB or select one you have previously created from the drop down list. This allows you to direct the wizard to group fields in the manner you want.

Add Extra Vertical Space Before Field Controls on Form Procedures
Check this box to direct the wizards to add vertical space between this field's control and the one populated above it.

User Options — User Options are provided to enable you to provide information to be used by a third-party template set. User Options are comma delimited, that is, each entry is separated by a comma.

Follow the instructions provided with your add-on template set.

Help

HLP — *To specify a help ID for controls referencing the field*, specify a help topic in the **HLP** field. The Application Generator adds the HLP attribute.

MSG *To specify a status bar message for controls referencing the field*, type the message in the **MSG** field. The Application Generator adds the MSG attribute. When the control referencing the field has focus, the text appears on the status bar, provided the window in which the control appears has one.

Validity Checks

Choose a validation option in this dialog. The Application Generator uses the information when creating and maintaining controls. When the user completes the field and shifts focus to another control, the application will sound a warning beep and set focus back to the control if the data is not valid..

Tip: When setting a validity check, provide the user with a helpful status bar message. For example, if you specify that a numeric field must hold a value between 1 and 50, place a message such as "Type a number between 1 and 50" in the MSG field.

No Checks To disable validity checking, choose **No Checks**.

Cannot be Zero or Blank
To require a user entry without specifying any other criteria, choose **Cannot be Zero or Blank**. The Application Generator adds the REQ attribute.

Must be in Numeric Range
To specify the entry fall between two numeric values, choose **Must be in Numeric Range**. Then enter the two values in the **Lowest** and **Highest** fields.

By entering only a lower, or only a higher value, you can specify an open ended range.

Must be True or False To specify a Yes/No entry, choose **Must be True or False**.

Must be in File To specify the value match a field in an external file, choose **Must be in File**. Choices will appear in the drop down list only if you previously related another file or files.

Must be in List To specify the value match an entry in a list, choose **Must be in List**. Then type the choices in the **Choices** field, in the format "Choice1| Choice2|Choice3." Separate the choices with a pipe character (|).

Tip: If you plan to allow the end user to choose a limited number of choices from a list box, drop down list, combo, drop down combo, or radio buttons, type the choices and separate them with a pipe symbol, or vertical bar character (|).

Window

To pre-format a window control referencing the current field, select the **Screen Controls** tab, then specify the options in this dialog.

Tip: By choosing the properties for a control at this time, you can save time later. Every application you generate from the dictionary, and every procedure in the application will automatically format the control the way you want it. If you don't format it here, and if the control requires custom formatting, you will have to custom format it for each procedure and application later.

Select either the prompt or entry field from the **Screen Controls** list, then press the **Properties** button. The prompt is the label which appears next to the control. The entry field is the actual control which accepts user input.

Window Controls — In most cases, this list box will show an ENTRY and PROMPT control for the field. Select the control to preformat.

Properties — Allows you to customize the control selected in the **Screen Controls** list by displaying its Properties dialog.

Reset Controls — Allows you to return the control type to its default, if you changed it by selecting another from the **Control Type** list.

Control Type — Allows you to select a different control type consistent with the field type and **Validity Checks** selected. For example, if you chose the **Must be in List** option in that dialog, one of the choices will be a list box.

Depending on whether the control can receive focus, (or in the case of a check box, which places the mnemonic in the label), the PROMPT in the **Screen Controls** list is deleted.

An IMAGE control cannot receive focus, and also has no PROMPT.

Report

To pre-format a report control referencing the current field, select the **Report Controls** tab, then specify the options.

Tip: **By choosing the properties for a control at this time, you can save time later. Every application you generate from the dictionary, and every procedure in the application will automatically format the control the way you want it. If you don't format it here, and if the control requires custom formatting, you will have to custom format it for each procedure and application later.**

Select the string field from the **Report Controls** list, then press the **Properties** button.

Report Controls — The STRING control for the field.

Properties — Allows you to customize the control selected in the **Report Controls** list by displaying its Properties dialog.

Reset Controls — Allows you to return the control type to its default, if you changed it by selecting another from the **Control Type** list.

Control Type — Allows you to select a different control type consistent with the field type and **Validity Checks** selected. For example, if you chose the **Must be in List** option in that dialog, one of the choices will be a list box.

Depending on whether the type of control, the first string control in the **Report Controls** list is deleted.

See also:

How to Design Your Dictionary and Database

How to Create a Data Dictionary

New/Edit Key Properties Dialog

Click on a TAB to see its help

This dialog allows you to define a key for the currently selected file. See also: How to Create a Key

The Dictionary Editor allows you to add keys and their components one after another, quickly. Each time you complete and close the Properties dialogs for one key, another blank dialog appears, ready for the next. Press **Cancel** when a blank dialog appears after completing the last key, to return to the **Field/Key Definition** dialog.

General

Key Name — *To specify a Clarion label for the key*, type a valid Clarion label in this field.

Tip: **Remember that you cannot give a key the same name as one of the fields within the RECORD. One common convention is to use the field name plus the word "key," as in *LastNameKey*.**

Description — *To place a text description for the key* in the Data Dictionary, type it in this field. The description appears in dialogs such as the *File Definition* dialog. If you anticipate using many keys for your application, we recommend filling in this field.

Type — *To specify a record key, static index file, or dynamic index file*, choose an option button in the **Type** group. The **Static Index** and **Dynamic Index** options are *disabled* when the **Unique** checkbox is marked, because indexes allow duplicates.

Attributes

External Name — *To specify a DOS filename for an external key*, type a valid DOS filename in this field.

Require Unique Value — *To disallow multiple records with duplicate values in their keys*, check this box. This option is valid only for keys, and is disabled for indexes.

Primary Key — *To establish the current key as the Primary key*, mark this checkbox. The Application Generator adds the PRIMARY attribute. This may be required for certain file drivers.

The primary key must be unique and exclude nulls.

Auto Number — *To specify the Application Generator should create code to manage record sequence numbers*, check this box.

Case Sensitive — *To sort according to case*, check this box. When creating or updating the key, all capital letters will precede the lower case letters, as per their positions in the ASCII table.

Exclude Empty Keys — *To exclude records with a null or zero value from the key*, check this box.

Comments

Allows you to enter a text description describing the dictionary. The description is solely for your convenience, and has no effect on the application. It is useful for situations in which other programmers may pick up your code later, or for when you expect to return to the project after a long period of time

since you last looked at it.

Options

Do Not Auto-Populate This Key
Directs the wizards to skip this Key when creating primary Browse procedures or Report procedures.

Population Order Specifies the order in which the wizards populate keys. Choose *Normal, First,* or *Last* from the drop down list. Wizards populate in this order: all Keys specified as First, then all Keys specified as Normal, and finally all Keys specified as Last.

User Options User Options are provided to enable you to provide information to be used by a third-party template set. User Options are comma delimited, that is, each entry is separated by a comma.

Follow the instructions provided with your add-on template set.

Fields

Specify the components of keys (the field or fields)-using the Fields tab. You may specify more than one field for a key. Each field is appended to the **Keys** list in the **Field/Keys Definition** dialog.

The **Fields** tab features a list displaying the components.

Sort Order Choose either the **Ascending** or **Descending** radio buttons to specify the order for the highlighted component.

Note: Not all file drivers support mixing ascending and descending components in the same key.

Insert Calls the Insert Key Component dialog listing the available fields. DOUBLE-CLICK on the name of a field in the list, to place it in the key.

Insert Removes the highlighted component from the key.

Move up/Move Down buttons
Moves the highlighted component up or down in the list.

See also: How to Create a Key

New/Edit Relationship Properties Dialog

Click on a TAB to see its help

Set relationships between files in this dialog. The relationships appear in the **Related Files** list on the **Dictionary** dialog, for the currently selected file. When completing this dialog, work from the top down. Start with the **Relationship for** *selected file* group box:

General

Type — Set the relationship type by choosing 1:Many or Many:1 from the drop down list.

Key — Depending on the relationship type selected, choose a primary or foreign key from the drop down list. The choices in the drop down are the keys previously defined for the currently selected file.

Depending the relationship type you choose for the selected file, the next group box will be labeled either: **Child**, **Parent** or **Link** (respective to 1:Many or Many:1):

Related File — Choose another file from the dictionary to relate to the selected file.

Key — Depending the relationship type you choose for the selected file, the label for this drop down box will be either Primary or Foreign. Select a previously defined key for the related file from the drop down box.

Field Mapping — This group box displays two lists, each showing a key, and the field in the related file which "maps" to it. If the key field names of each file match each other, then just press the **Map by Name** button (below), and Dictionary Editor will automatically link the fields. If they do not, double click on each item in the list boxes, then select a field from the related file that links to the key, in the **Select a Field** dialog.

Map By Name — Automatically defines links based on similarly named fields in each data file.

Map By Order — Automatically defines links based on the order in which fields are defined in each data file.

After choosing all other options, set the options in the **Referential Integrity Constraints** group box. The Application Generator automatically generates the code that enforces your selections.

Referential Integrity requires that a foreign key cannot contain any value which has no match in the primary key. This raises potential problems when the end user wishes to change or delete the primary key record.

The **On Update** and **On Delete** drop down boxes each offer the following choices:

No Action — Instructs the Application Generator *not* to generate any code to maintain referential integrity.

Restrict — Instructs the Application Generator to disallow the user from deleting an entry, if the value is used in a foreign key. For example, if the user attempts to change a primary key value, the generated code attempts to check for a related record with the new value, changes it back if necessary, then loops back to the entry dialog so that the user can enter another value.

Cascade Instructs the Application Generator to update or delete the foreign key record. For example, if the user changes a primary key value, the generated code changes the values in the foreign key that referenced the primary key. If the user deletes a primary key value, the code deletes the foreign key value, too.

Clear Instructs the Application Generator to change the value in the foreign key to blank or zero.

Comments

Allows you to enter a text description describing the dictionary. The description is solely for your convenience, and has no effect on the application. It is useful for situations in which other programmers may pick up your code later, or for when you expect to return to the project after a long period of time since you last looked at it.

Options

User Options User Options are provided to enable you to provide information to be used by a third-party template set. User Options are comma delimited, that is, each entry is separated by a comma.

Follow the instructions provided with your add-on template set.

See also:

How to Define File Relationships and Referential Integrity

How to Design Your Dictionary and Database

Select a Field Dialog

Allows you to select a field (from the related file) to link in the **Field Mapping** lists in the **New Relationship Properties** dialogs. Open the dialog by DOUBLE-CLICKING an item in either list.

Fields list	DOUBLE-CLICK or select a field from the related file, and press the **Select** button.
No Link	Allows you to indicate that a field is not part of the relationship.

New View Dialog

A VIEW is a virtual file constructed from selected fields in multiple files. The New View allows you to select the files and fields to include in the view. As you add files and fields, they are listed in the **View** list in the dialog.

Name — Type a view "name," as you wish to refer to it in your code. The name must be a valid Clarion label.

Description — Type a string description for the view. Clarion displays the descriptions in dialogs such as the **Dictionary** dialog.

Press the **>>** button to type a separate description (up to 1000 characters) in a larger text box. See also: **Edit View Description** dialog.

Filter — Type an expression (such as PRE:Field1 > 1) to limit the contents of the view to only those records matching the filter expression. The filter is independent of any keys defined for the files referenced by the VIEW structure. In a Client/Server environment the filter may not adversely affect performance; in any other environment, it may slow down file operations.

Add File — Allows you to add a file to the View. See also: **Select Primary File** and **Add File (View)** dialog.

Add Field — Allows you to add a field from the currently selected file to the view. See also: **Add File (View)** dialog.

Remove — Removed the currently selected file or field from the view.

To add files and fields to the VIEW structure:

1. Press the **Add File** button.

2. In the **Select Primary File** dialog, choose a file and press the **OK** button.

The file appears in the view list.

3. Back in the **New View** dialog, press the **Add Field** button.

4. In the **Add Field** dialog, click on the fields you wish to include in the view, then press the **OK** button.

The fields appear directly below the file.

5. Repeat steps ***1*** through ***4*** for any additional files and fields you wish to add to the view.

Only files already related to the current file may be added to the file list below it.

6. Press the **OK** button to close the **New View** dialog.

Creating a view provides a major advantage in a Client-Server environment because the Server has the ability to do much of the work. The Server processes the overhead of the relational "Join" and "Project" operations which would otherwise tie up the local machine. Only the data elements specified in the VIEW--not the entire RECORD structures from the files contained in the view--are updated. This means accessing a field in the RECORD structure which is *not* also defined in the VIEW structure returns an undefined value. Therefore, when working with a views, be sure to include *all* the fields you need to work with in the VIEW, and don't try to access any fields not in the VIEW.

The VIEW structure has no prefix. Access its fields by using the prefixes for the *original* RECORD structures defining the fields. This is transparent when you use the Application Generator. The **File Schematic Definition** automatically adds the proper prefix so that the generated code is correct.

To modify the view properties at a later time, highlight the file name on the **Dictionary** dialog list, then either DOUBLE-CLICK or press the **Properties** button.

Select Primary File/ Add File--View Dialog

Allows you to select the primary file for the view. Select a file from the list and press the OK button to add it to the view.

The primary file is the only file in the view which you can update. PUT or DELETE affects records *only* in the primary file. Joined files are never written to.

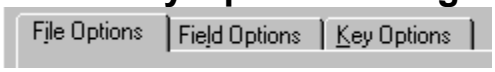
If you DELETE the view record, it deletes the record in the primary file, but not the records in the other files in the view. If you PUT the view record, it updates the fields in the primary file, but not the other files.

Edit File Description

Allows you to enter string descriptions for the file. Clarion for Windows automatically displays the short description in certain dialogs, allowing you to quickly recognize the file contents. The long text description only appears in this dialog box, and holds up to 1000 characters.

The descriptions are solely for your convenience, and have no effect on the application. They're useful for situations in which other programmers may pick up your code later, or for when you expect to return to the project after a long period of time since you last looked at it.

Dictionary Options Dialog



Click on a TAB to see its help

You can customize the default dictionary settings in this dialog. To access the dialog, choose **Setup** **Dictionary Options**.

File Options

Default Driver To select the default database driver for new files in a dictionary, choose from the **Driver** drop down list.

For detailed descriptions of the drivers available, see *Database Drivers*.

Sort Dictionary Files Alphabetically Check this box to display files in alphabetical order. If not checked, files display in the order in which they were created.

THREAD To specify new file definitions default to adding the THREAD attribute (setting aside a separate RECORD buffer for each procedure), check the **THREAD Attribute on Files** checkbox.

Display File Description Check this box to display the file description in the Files list.

Display File Driver Check this box to display the file driver in the Files list.

Field Options

Assign Description to Message
Check this box to specify that the field descriptions you specify when defining a field should automatically serve as the text for the MSG field (setting a status bar message when controls referencing the field have the focus).

Display Field Description
Check this box to display the field description in the Fields list.

Display Field Type Check this box to display the field's data type in the Fields list.

Display Field Picture Check this box to display the field's display picture in the Fields list.

Key Options

Display Key Description
Check this box to display the Key description in the Keys list.

Display Key Type
Check this box to display the Key Type in the Keys list.

Display Unique Flag
Check this box to display **Unique** if the Key is flagged as unique.

Display Primary Key Status
Check this box to display **Primary** if the Key is flagged as the Primary Key.

Display Other Key Attributes

Check this box to display the other attributes of the Key in the Keys list.

See also: How to Create a Data Dictionary

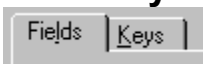
Dictionary Dialog

Allows you to manage the data files, aliases, views and relations for your dictionary.

Files List	The list of data files, aliases and views in the dictionary.
Add File	Allows you to add a file to the list. See also: **New File Properties** dialog.
Add Alias	Allows you to add an alias to the list. See also: **New File Alias** dialog.
Add View	Allows you to add a view to the list. See also: **New View dialog**.
Properties	Allows you to change the options for the selected file, view or alias. See also: **Edit Properties** dialog.
Fields/Keys	Allows you to define fields and keys for the selected file, view or alias. See also: **Field/Keys Definition** dialog.
Delete	Allows you to delete the selected file, view or alias.
Related Files List	Shows relations for the currently selected file or alias.
Add Relation	Allows you to add a relation to the selected file or alias. See also: **New Relationship Properties** dialog.
Properties	Allows you edit the selected relation. See also: **Edit Relationship Properties** dialog.
Delete	Allows you to delete the selected relation.
Dictionary Properties	Allows you to add or edit information about the current data dictionary, including creation, modification dates, and a text description. See also: **Dictionary Properties** dialog.

See also: How to Create a Data Dictionary

Field/Key Definition Dialog



Click on a TAB to see its help

The **Field/Key Definition** dialog contains two tabs--listing **Fields** and **Keys**. At the bottom of the dialog, the **Edit Fields** or **Edit Keys** group box appears, depending on which tab is selected.

To add a new field, press ALT + L if the **Fields** list does not currently have focus, then press the **Insert** button in the **Edit Fields** group box. The **New Field Properties** dialog appears.

To add a new key, press ALT + K if the **Keys** list does not currently have focus, then press the **Insert** button in the **Edit Keys** group box. The **New Key Properties** dialog appears.

Fields

To modify an existing field, select it and press the **Edit** button in the **Edit Fields** group box. The **Edit Field Properties** dialog appears.

To delete an existing field, select the field name and press the **Delete** button in the **Edit Fields** group box.

To move the selected field within the Fields list, press the and ↓ buttons in the **Edit Fields** group box. This reorders the field labels within the FILE structure.

Keys

To modify an existing key or key component, select it and press the **Properties** button. The **Edit Key Properties** dialog appears.

To delete an existing key, select it and press the **Delete** button.

To move the selected key within the Keys list, press the and ↓ buttons in the **Edit Keys** group box.

Keys and indexes specify sort orders for a single file. A key may reside within the file, or as an external file, depending on the file system. Keys are automatically updated whenever records are added, changed, or deleted. See *Database Drivers* for further information regarding how each file driver supports keys or indexes.

Indexes usually exist as external files. Remember that a separate DOS file handle is necessary for each external key or index file. Index files do *not* update automatically. The BUILD statement updates an index.

A dynamic index allows you to declare an index file without specifying the field(s) in the Data Dictionary. The application must define the field(s) at runtime, as the second parameter of the BUILD statement. The application may rebuild the same index file at a later time, specifying a different field for the index key.

See also:

How to Design Your Dictionary and Database

How to Create a Data Dictionary

Dictionary Version Control

Version CheckPoint

Increases the internal version number in the data dictionary.

The Dictionary Editor automatically places an internal version number in your dictionary file. A new dictionary begins with version 1.0. You can see the version number/revision number on the caption bar of the **Dictionary** dialog. The **Dictionary Properties** dialog also displays the original creation data and time, and the last modified date and time.

You should increase the version number, manually, whenever you make significant changes to a dictionary; for example, when you're working on version #2 of your application, choose **Version Checkpoint**. The revision number (r. #) on the caption bar increases by one.

To roll back to a previous version, choose **Version Revert**. Choose the revision to revert to by selecting it with the spin control in the **Previous Revision** dialog.

Revert Dictionary Version

Version Revert

Rolls back changes in the data dictionary to the last checkpoint.

The Dictionary Editor automatically places an internal version number in your dictionary file. A new dictionary begins with version 1.0. You can see the version number/revision number on the caption bar of the **Dictionary** dialog. The **Dictionary Properties** dialog also displays the original creation data and time, and the last modified date and time.

You should increase the version number, manually, whenever you make significant changes to a dictionary; for example, when you're working on version #2 of your application, choose **Version Checkpoint**. The revision number (r. #) on the caption bar increases by one.

To roll back to a previous version, choose **Version Revert**. Choose the revision to revert to by selecting it with the spin control in the **Previous Revision** dialog.

Global Data/Local Data/Module Data Dialogs

This dialog allows you to define or edit memory variables. It provides a list of variables for the procedure (Local), module, or the global variable list.

When you want to add a variable, you press the Insert button, then define the variable in the **New Field Properties** dialog.

Properties	Select a variable from the list, then press this button to edit it in the **Edit Field Properties** dialog.
Insert	Press this button to define a new variable.
Delete	Press this button to delete the currently selected variable.
Up	Press this button to move the currently selected variable up one position in the list. When the Application Generator generates the code defining the data, it will do so in the order they appear in this list.
Down	Press this button to move the currently selected variable down one position in the list.

File Schematic Definition Dialog

You define the files, fields, and variables a procedure--or a control inserted by a control template--can access with the **File Schematic Definition** dialog. The available data files and keys are the ones you define in the data dictionary.

You can "attach" a file to an item in the **Files** list. These represent the current procedure, module, global data, or other files. When you place a control template in a window, you can "attach" the file to the "To Do" item which appears after you place the control template.

The dialog contains the following buttons:

Insert	Allows you to pick a file, previously defined in the data dictionary, from the **Insert File** dialog.
Delete	Deletes the currently selected file from the list (not from the data dictionary, or from disk).
Key	Allows you to select a key previously defined in the data dictionary, from the **Insert Key** dialog. Normally, the file schematic will *automatically* pick the first key listed in the data dictionary for any data file you add by pressing the **Insert** button. This button allows you to pick another key.
New	Allows you to call the Dictionary Editor.

Select Field Dimension Dialog

This dialog allows you to specify the component of a dimensioned variable which you wish to "attach" to a control, via the **File Schematic Definition** dialog.

The number of dimensions you specified in the **Field Properties** dialog when you defined the variable determines how many spin boxes are enabled. Choose the specific element by setting the spin boxes.

For example, to refer to the variable DimVariable[4][5], set the first spin box to *4*, and the second to *5*.

Revert to Previous Revision

This window allows you to rollback a dictionary to a previous version.

Current Displays the current version.

New Revision Allows you to select the version to which the dictionary will revert.

Import File Dialog

The Dictionary Editor allows you to quickly add a data file to the dictionary by creating a data definition based on an existing data file.

With the Dictionary dialog active, select File ä Import File. Specify a data file and additional options in the **Import File** dialog.

Filename	Type a file name or press the ellipsis (...) button to select a name from the Import File Definition dialog. When importing a definition via an ODBC data source, do not specify a directory name; Clarion for Windows will read it from ODBC.INI.
File Driver	Choose a database driver from the drop down list.
Owner	Fill in an optional OWNER attribute. If importing from an ODBC data source, type in the datasource name, (optional) UserID, and (optional) Password.
Options	Fill in any optional driver strings.

After reminding you that the import process cannot import memo fields (you can add them manually), the data file is added to the dictionary, along with its field and key definitions. When importing from an ODBC data source, no key definitions will be imported.

See also:

ODBC

Password Validation Dialog

Allows you to password protect your dictionary, to prevent other developers from modifying it.

To add a password to the data dictionary:

1. Press the **Password** button.

2. When the **Password Validation** dialog appears, type a password in the space provided.

3. Press the **OK** button.

See the *Multi-Developer Development* appendix in the *User's Guide* for more information.

Data Sources dialog

Select the data source from which the file will be imported, then press the **Next** button to select the table to import. Select the table, then press the **Finish** button to import the file.

If the Data Source has not been defined in ODBC.INI, press the **New** button to define the data Source. This calls the ODBC Administrator, an applet which maintains file definitions in ODBC.INI.

Common Questions

Data Dictionary

How to Create a Data Dictionary

How to Design Your Dictionary and Database

How to Create a Dictionary (.DCT) File

How to Add Files to a Dictionary

How to Add Fields to Data Dictionary Files

How to Create a Key

How to Define File Relationships and Referential Integrity

Using Wizard Options

How to Import a File Definition From an Existing Data File

Application Generator

How to Create a New Application File

How to Add and Customize a Procedure

How to Add Embedded Source Code

Adding Procedure Extensions

Formula Editor

Defining Procedure Formulas

How to Create a Simple Assignment Expression

How to Create a Complex Assignment Expression

Window Formatter

How to Customize Your Window

Using the Window Formatter - Sample Window

Using the Window Formatter - Controls Toolbox

Using the Window Formatter - Fields Toolbox

Using the Window Formatter - Property Toolbox

Using the Window Formatter - Align Toolbox

How to Add a Toolbar

How to Minimize a Window

How to Create a New Menu

How to Implement Standard Windows Behavior

How to Create an MDI Menu

How to Use a Combo Box

How to Create a List Box

How to Create Column Groups Using the List Box Formatter

How to Assign an Image to Display at Runtime

How to Store and Display a Graphic Image with a Memo or Blob Data Type

How to Make a Record Occupy Two or More Rows in a List Box.

How to Restore User Resized List Box Column Widths

How to Trap a Double Click on a List Box

How to add Drag and Drop to a List Box

How to Display the Sort Field First on a Multi-Key Browse

How to Create a Wizard

What is a Dialog Unit

How to complete an entry field when the last character is entered

How to Use Spin Controls for Date or Time Fields

How to Create a Multi-Page Form

How to use Pattern Pictures on a form

How to Implement a Splash Screen

Report Formatter

How the Print Engine Processes Report Sections at Runtime

How to Use the Report Formatter - An Overview

How to Set Report Group Breaks

How to Sort Reports

How to Control Page Breaks

Using the Report Formatter - Sample Reports

Using the Report Formatter - Controls Toolbox

Using the Report Formatter - Fields Toolbox

Using the Report Formatter - Property Toolbox

Using the Report Formatter - Align Toolbox

Using Preview!

How to Print to a File

How to change the printer device without calling PRINTERDIALOG

How to Print Labels

How to Clip and Concatenate Name Fields

Project System

How to Link External Resources

How to Manage Threads

Using Windows DLLs NOT Created in Clarion for Windows

Creating a .DLL (Sub-Application)

Redirection File

Distributing Your Applications

ODBC

How to Import an ODBC File Definition

How to Create a File Definition for an ODBC Data Source

How to Choose When to Use ODBC vs. a Native Clarion Driver

How to Work With the ODBC.INI File

How to Test Your ODBC Application

Using an ODBC Connect String

Dynamic Data Exchange

How to Start a DDE Conversation

How to Send DDE Commands and Data to a DDE Server

Templates

How to Customize Procedure Templates

How to Modify Templates

How to Register a Template Set

Data File Conversion

How do I handle an Error 47

How to Convert a File--Generate Source

How to Convert a File (without generating source)

How to Make a Field Assignment

Tips for Clarion 1.0 users

Clarion Language Enhancements 1.0 to 1.5

BrowseBox Control: The Inside Story

How to Create a Data Dictionary

This section provides an overview of the **general** process of creating a data dictionary. This involves the following steps:

1. Design Your Dictionary and Database .

Planning and organizing your application's database design up front can result in a more efficient application, as well as much shorter development times.

2. Create the Dictionary (.DCT) File.

3. Add Files to the Dictionary .

Add File Aliases and Views as well. These pseudo-files provide support for conventional relational database functionality, that is "Join" and "Project."

4. Add Fields to the Files.

5. Define Keys for the Files.

6. Define File Relationships.

Including custom referential integrity constraints for related files.

How to Design Your Dictionary and Database

This topic provides a quick review of relational database theory. Planning and organizing your application's database design up front can result in a more efficient application for the end user, not to mention saving hours of redesign later.

The relational model concerns itself with three aspects of data management: **structure**, **integrity,** and **manipulation**. For our purposes, we will discuss the three practical requirements of these aspects: data normalization, keys, and relational operations.

Normalization

At its simplest, data normalization means that a data item should be stored at only one location. To avoid duplication within the database, a good design splits data into separate files.

For instance, assume a very simple order-entry system storing the following data:

Customer Number
Customer Name
Customer Address
ShipTo Address
Order Number
Order Date
Product Number
Quantity Ordered
Unit Price

You could store all the data in each record of one file, but that would be inefficient (unless the business has **no** repeat customers). A second order from a customer would repeat all the Customer data, for example. To eliminate duplication, you could split the data into three files:

Customer File:	Order File:	Item File:
Customer Number	Order Number	Product Number
Customer Name	ShipTo Address	Quantity Ordered
Customer Address	Order Date	Unit Price

This organizes the data in a logical scheme and eliminates duplication. The process of relating each record to another record in another file requires adding additional fields to at least two of the files, so that the files can share common values.

Strict relational theory specifies that:

- The database consists of one or more **tables**, which, to vastly oversimplify for a moment, correspond to the **files**.
- The table consists of column headings (which at the file level we refer to as **fields**) and zero or more rows (**records**).
- Each record contains exactly one value for each field.

Keys

In the simple order-entry system above, to **relate** the records in the customer, order and item files to one another, we could add one field each to two of the files as follows:

Customer File:	Order File:	Item File:

Customer Number	Order Number	Order Number
Customer Name	Customer Number	Product Number
Customer Address	ShipTo Address	Quantity Ordered
	Order Date	Unit Price

Relational database theory states:

- A primary key should exist for each table. A primary key is a unique field or unique combination of fields. The primary key must not accept a null or blank value, this would prevent it from being unique.
- A foreign key can match the primary key in another table. If table "A" includes a foreign key that matches table "B's" primary key, then every value in the key in table "B" must either be equal to a value in the primary key in a record in "A," or be null.

In the example above, the Customer Number is the primary key (there could be two "John Smith's," but not two customer #1001's). The Customer Number field is added to the order file, as a foreign key.

You can define two types of relationships between files:

- **One-to-Many**. One record in a file relates to many in another. In the example above, a single customer number may relate to many records in the Order file. In business database applications, this is the most common relationship. It is also referred to as a Parent-Child relationship.
- **Many-to-Many**. Multiple records in a file relate to multiple records in another. To apply it to the example, assume the Order-Entry system were made to fit a manufacturing concern which buys parts and makes products. If a part could be used in many different products, and a product could use many parts, two additional files might look like:

Parts File:	Product File:
Part Number	Product Number
Part Description	Product Description

Relational Operations

Relational database theory provides a set of operators for manipulating data. The three operations that theoreticians specify for relational database systems are **Select**, **Project**, and **Join**. A system does not have to explicitly support the statements as long as it supports their functionality. For theoretical purposes, a table simply consists of a set of column headings (or fields), plus zero or more rows (records) of data values.

- A **Select** extracts a row subset of a given table--in other words, a subset of records which satisfy a given condition.
- A **Project** extracts a column subset of a given table--in other words, a subset of specified fields, which then eliminates extraneous records (example below).
- A relational **Join** takes two tables and joins them together to form a new, wider table.

In the example, extracting a record or records (spanning all files) that meet the condition "Customer Number = 100" is an example of a relational select.

Project extracts **unique** values by field. In the example above, assuming that the Item file has many duplicates, to Project the file "Item" over the field "Product Number" yields a new table of all the products sold (not necessarily all products made). There would be only one instance of each product.

Join: Going back to the example, to work with all the combinations of parts and products possible, there must be a special relationship between these two files. The solution is to define a third file, called a "Join" file. This file creates two One-to-Many relationships. The relationships between the three files would be defined:

Parts File:

Part Number	(Primary key)
Part Description	

Parts2Prod File:

Part Number	(1st Primary key component and Foreign key)
Product Number	(2nd Primary key component and Foreign key)
Quantity Used	

Product File:

Product Number	(Primary key)
Product Description	

The Parts2Prod file has a multiple component Primary key and two foreign keys. The relationship between Parts and Parts2Prod is One-to-Many. The relationship between Product and Parts2Prod is also One-to-Many. This makes the Join file the "middleman" between two files with a Many-to-Many relationship.

Usually a Join file contains additional information. In this example, the Quantity Used logically belongs in the Parts2Prod file.

The Clarion Data Dictionary Editor

The Clarion language supports the three aspects of data management that relational database theory concerns itself with. The Dictionary Editor is a tool for planning the structure and integrity of the database. The Dictionary Editor also allows you to "preconstruct" some of the relational operations specified by database theorists; Clarion language statements handle the remaining operations.

 The Dictionary Editor allows you to easily set up the proper database structure by defining files, fields, and relations.

 The Dictionary Editor allows you to easily plan both primary and foreign keys for your database, as per the relational model's integrity rules.

 Additionally, the Dictionary Editor supports preconstruction of "Views." The View creates a "virtual" file, automatically handling any necessary "Joins" and "Projects."

How to Create a Dictionary (.DCT) File

You generally create a data dictionary as the first step in creating an application. Therefore, you will access it first from the development environment''s main menu.

To open the **Dictionary Editor** to create a new dictionary file:

1. Choose **File** **New** from the development environment menu, then select the **Dictionary** tab.
2. Specify the path (**Folders**) and **File Name** for your dictionary file, then press the **Create** button.

The **Dictionary** dialog appears.

3. Press the **Dictionary Properties** button at the bottom of the dialog.
4. On the **Comments** tab, type the description in the space provided.

The description is solely for your convenience, and has no effect on the application. It is useful when other programmers take over your project, or for when you return to the project after a long absence.

Your data dictionary file is created. At this point the dictionary is an empty shell. Use the **Dictionary** dialog to add files, fields, keys, and file relationships.

How to Add Files to a Dictionary

1. Press the **Add File** button, then, when asked if you want to use Quick Load, press the **No** button.

The **New File Properties** dialog appears.

2. On the **General** tab, type the **Name**, the **Prefix**, and choose the **File Driver** for your data file.

3. Press **OK** to close the dialog.

Your file is added to the dictionary. You may, of course, specify additional file properties if you want. You may add more files by repeating the above steps.

How to Add a File Alias to the Dictionary

An alias creates a second reference for a file without duplicating the file on disk. You can add an alias for a file only if it's already on the Dictionary list. In the Dictionary dialog, press the **Add Alias** button and fill in the **New File Alias** dialog.

A file alias provides several advantages, at the cost of some system overhead:

It allows you to set multiple relationships between files.

Strict relational database theoreticians state a file may only have a single relational link to another file at a time. Aliases allow you to "legally" work around this limitation.

It allows a second file buffer for the same file.

You could use this for a second file browse, as well as entry forms and other items for each. This is particularly useful for a Multiple Document Interface (MDI) application.

On the negative side, the second file buffer takes up additional memory and resources.

Any file driver that uses external key files requires additional file handles for each alias. For example, a file with three external keys and three aliases requires sixteen file handles: one each for the "first" data file and its three keys, and an additional four for each of the aliases. When using aliases, we recommend choosing a file driver that stores keys internally, such as TopSpeed or Btrieve.

Tip: When using aliases, you must open the file in Share mode.

You can edit the fields and keys for the Alias by pressing the **Fields/Keys** button. The **Field/Key Definition** dialog lists the fields and keys for the **original** file; any changes you make will update the originals.

How to Add a View to the Dictionary

A VIEW is a virtual file constructed from selected fields in multiple files.

Creating a view provides a (potentially) major advantage in a Client-Server environment because the Server has the ability to do much of the work. The Server processes the overhead of the relational "Join" and "Project" operations which would otherwise tie up the local machine.

When working with views, be sure to include all the fields you need to work with in the VIEW, and don't try to access any fields not in the VIEW. This is necessary because only the data elements specified in the VIEWnot the RECORD structures from the component filesare updated. This means accessing a field in the RECORD structure which is not also defined in the VIEW structure returns an undefined value.

The VIEW structure has no prefix. Access its fields by using the prefixes for the original RECORD structures defining the fields. This is transparent when you use the Application Generator. The **File Schematic Definition** automatically adds the proper prefix so that the generated code is correct.

To add a view to the files list, choose **Edit** **Add View**. Fill in the **New View** dialog.

Name Type a view name, as you wish to refer to it in your code. The name must be a valid Clarion label.

Description Type a string description for the view. Clarion displays the descriptions in dialogs such as the **Dictionary** dialog.

Press the >> button to type a separate description (up to 1000 characters) in a larger text box.

Filter Type an expression (such as PRE:Field1 > 1) to limit the contents of the view to only those records matching the filter expression. The filter is independent of any keys defined for the files referenced by the VIEW structure. In a Client/Server environment the filter may not adversely affect performance; in any other environment, it may slow down file operations.

Add File Allows you to add a file to the View.

1. Press the **Add File** button.
2. In the **Select Primary File** dialog, choose a file and press the **OK** button.

Only files already related to the primary file may be added to the file list below it.

Add Field Allows you to add a field from the currently selected file to the view.

1. Press the **Add Field** button.
2. In the **Add Field** dialog, CLICK on the fields you wish to include in the view, then press the **OK** button.

To modify the view properties at a later time, highlight the file name on the **Dictionary** dialog list, then either DOUBLE-CLICK or press the **Properties** button.

How to Add Fields to Data Dictionary Files

1. Press the **Fields/Keys** button to open the **Field/Key Definition** dialog.
2. On the **Fields** tab, press the **Insert** button.

The **New Field Properties** dialog appears.

3. On the **General** tab, type in the field **Name**, choose **Data Type**, specify length in **Characters**.
4. Select the **Validity Checks** tab, and choose a field validation option.
5. Select the **Window** tab to specify how the field and its prompt appear as controls in your application windows and dialogs.
6. Select the **Report** tab to specify how the field will appear on printed reports.

The specifications on the Window and Report tabs establish defaults for the field. You can always change the settings on a case by case basis.

7. Press **OK** to complete this field and define the next one.

The **New Field Properties** dialog appears again, ready for the next field.

8. Repeat steps 3 through 7 for additional fields within this file.

After each field is completed, the **New Field Properties** dialog appears, ready to accept the next field.

9. After adding the last field, press the **Cancel** button in the **New Field Properties** dialog to return to the **Field/Key Definition** dialog.

How to Create a Key

Add and edit keys and indexes using the **Field/Key Definition** dialog.

Keys are automatically updated whenever records are added, changed, or deleted. **Index files do not update automatically**. A BUILD statement is required to update an index.

1. Select a file from the list on the **Files** side of the **Dictionary** dialog and press the **Field/Keys** button.

2. In the **Field/Keys Definition** dialog, select the **Keys** tab.

3. Highlight a key (if one exists), then press the **Insert** button.

The **New Key Properties** dialog appears.

4. Type a valid Clarion label in the **Key Name** field.

5. Optionally type a **Description**. This displays in various dialog boxes, including the **File Definition** dialog.

6. Select the **Attributes** tab and check all boxes that are appropriate for the key.

A runtime index allows you to declare an index without specifying the key component fields in the Data Dictionary. The application must define the key component fields at runtime, as the second parameter of the BUILD statement. The application may rebuild the same index file at a later time, specifying a different key component fields for the index.

7. Optionally type a valid DOS file name in the **External Name** field, if the file system needs one.

Clarion automatically adds the proper file extension.

8. Select the **Fields** tab, then press the **Insert** button.

The **Insert Key Component** list appears.

9. DOUBLE-CLICK a field in the list;

This transfers its name to the **Fields** tab, which indicates the field will be part of the new key. Repeat steps 8 and 9 to add more fields to the key.

10. Press **OK** to close the **New Key Properties** dialog.

The **New Key Properties** dialog appears again, ready to accept additional keys.

11. Repeat steps 4 through 10 to create additional keys for this file.

12. When you are finished adding keys, press **Cancel** to close the **New Key Properties** dialog and return to the **Field/Keys Definition** dialog.

At the end of the process, your keys appear on the **Keys** tab, with their field components arranged in order, one above the other in a tree diagram.

To modify a key, select the key and press the **Properties** button in the **Field/Key Definition** dialog. The **Edit Key Properties** dialog appears. If you selected a key component, the **Fields** tab is on top. If you selected the key, the **General** tab is on top. The **Setting Key Properties** section, below, describes the options in this dialog.

How to Define File Relationships and Referential Integrity

Define relationships between files in the **New Relationship Properties** dialog. The relationships for the currently selected file appear in the **Related Files** list on the right side of the **Dictionary** dialog.

1. Select a file from the **Files** list on left side of the **Dictionary** dialog.

2. Press the **Add Relation** button.

The **New Relationship Properties** dialog appears.

3. Select the relationship **Type** from the drop down list: **1:Many** or **Many:1**.

The label for the group box immediately below will change to **Child** or **Parent**, depending on your choice.

4. Select the **Related File** from the drop down list.

5. Select **Primary Key** or **Foreign Key** for the first file from the drop down list at the top right of the dialog.

Clarion automatically changes the label for the drop down list (either **Primary Key** or **Foreign Key**) according to the relationship type.

6. Select the **Primary Key** or **Foreign Key** for the related file, if applicable, from the drop down list immediately below the first drop down list.

7. Press the **Map by Name** button to establish the link between the two keys by matching field names within the two keys.

The **Field Mapping** lists show the actual links established between the two files. Alternatively, you can map by field order, or you can map each key manually by double-clicking the key name in the **Field Mapping** list.

8. Optionally set Referential Integrity Constraints by choosing from the **On Update** and **On Delete** drop down lists in the **Referential Integrity Constraints** group box.

See the section below for further information on Referential Integrity Constraints.

9. Press the **OK** button.

Setting Referential Integrity Constraints

By setting referential integrity constraints in the data dictionary, you can instruct the Application Generator on how to set up executable code for linked field updates and deletions when working with related files.

Referential Integrity requires that a foreign key must always have a match in the primary key. This raises potential problems when the end user wishes to change or delete the primary key record.

The **New Relationship Properties** dialog allows you to specify how the executable code should handle these situations when one of several related records is updated or deleted..

No Action	Instructs the Application Generator **not** to generate any code to maintain referential integrity.
Restrict	Tells the Application Generator to prevent the user from deleting or changing an entry, if the value is used in a foreign key. For example, if the user attempts to change a primary key value, the generated code checks for a related record with the same key value. If it finds a match, it will not allow the change.
Cascade	Tells the Application Generator to update or delete the foreign key record. For example, if the user changes a primary key value, the generated code changes

any matching values in the foreign key. If the user deletes a parent record, the code deletes the children too.

Clear Instructs the Application Generator to change the value in the foreign key to null or zero.

Using Wizard Options

Wizard Options in the Data Dictionary Editor provide control over how Clarion's code generation wizard create your code. Wizards use the Options specified for a file, field, key, or alias when creating procedures.

File Options

Do Not Auto-Populate This File
Directs the wizards to skip this file when creating primary Browse procedures or Report procedures.

User Options User Options are provided to enable you to provide information to be used by a third-party template set. User Options are comma delimited, that is, each entry is separated by a comma.

Follow the instructions provided with your add-on template set.

Alias Options

Do Not Auto-Populate This Aliased File
Directs the wizards to skip the Aliased File when creating primary Browse procedures or Report procedures.

User Options User Options are provided to enable you to provide information to be used by a third-party template set. User Options are comma delimited, that is, each entry is separated by a comma.

Follow the instructions provided with your add-on template set.

Field Options

Do Not Auto-Populate This Field
Directs the wizards to skip this field when creating Form, Browse or Report procedures.

Population Order Specifies the order in which the wizards populate fields. Choose **Normal, First,** or **Last** from the drop down list. Wizards populate in this order: all Fields specified as First, then all Fields specified as Normal, and finally all Fields specified as Last.

Form Tab Specifies the TAB onto which the wizards populate the field. Type the Caption for the TAB or select one you have previously created from the drop down list. This allows you to direct the wizard to group fields in the manner you want.

Add Extra Vertical Space Before Field Controls on Form Procedures
Check this box to direct the wizards to add vertical space between this field's control and the one populated above it.

User Options User Options are provided to enable you to provide information to be used by a third-party template set. User Options are comma delimited, that is, each entry is separated by a comma.

Follow the instructions provided with your add-on template set.

Key Options

Do Not Auto-Populate This Key
Directs the wizards to skip this Key when creating primary Browse procedures or Report procedures.

Population Order Specifies the order in which the wizards populate keys. Choose **Normal, First,** or **Last** from the drop down list. Wizards populate in this order: all Keys specified as First, then all Keys specified as Normal, and finally all Keys specified as Last.

User Options User Options are provided to enable you to provide information to be used by a third-party template set. User Options are comma delimited, that is, each entry is separated by a comma.

Follow the instructions provided with your add-on template set.

Relation Options

User Options User Options are provided to enable you to provide information to be used by a third-party template set. User Options are comma delimited, that is, each entry is separated by a comma.

Follow the instructions provided with your add-on template set.

How to Import a File Definition From an Existing Data File

The Dictionary Editor allows you to quickly add a data file to the dictionary by creating a data definition based on an existing data file.

1. With the **Dictionary** dialog active, select **File** **Import File**.

The **Select File** drive dialog appears.

2. Pick a file driver from the drop down list, and press the **OK** button.

Pick the driver of the file whose definition you are creating. The **Open File** dialog appears.

3. Press the ellipsis (...) button and pick the file using the standard Open File dialog.

4. Press the **OK** button twice to close the dialogs.

The Dictionary Editor creates the file definition and the **Edit File Properties** dialog appears.

5. Make any changes to your new file definition, then press the **OK** button.

The data file is added to the dictionary, along with its field and key definitions.

How to Create a New Application File

The first step in creating a new application is to create an .APP file. The .APP file holds the procedures, data, and other properties you define for your application. Optionally create a new directory for the application; whenever you open the .APP file, Clarion for Windows will use the directory in which the .APP file resides as the working directory.

1. Use Windows 3.x, Windows 95, DOS, etc to create a subdirectory for your application, then return to Clarion for Windows.

2. Choose **File** **New** (or press the


button on the toolbar)**.**

The **New** file dialog appears.

3. Choose **Application** by CLICKING on the tab.

4. Type a name for the .APP file in the **File Name** field. If you want to use the Quick Start wizard, check the box below the file list. See Using the Quick Start Wizard.

Type a legal DOS filename. Clarion automatically adds the .APP extension.

5. Press the **Create** button.

The **Application Properties** dialog appears. This dialog allows you to define the essential files for the application.

6. Name the .DCT file the application will use in the **Dictionary File** field, or press the ellipsis (**...**) button to select the file in the **Select Dictionary** dialog.

See How to Create a Data Dictionary for information on creating your application's data dictionary. The **Select Dictionary** dialog is a standard **Open File** dialog.

The Application Generator does **not** require a data dictionary to generate an application, if you **uncheck** the **Require a dictionary** box in the **Application Options** dialog.

7. Optionally rename the first procedure from MAIN to another name of your choice.

You can do so by typing another procedure name from the **First Procedure** field. There is no practical advantage to renaming this procedure.

8. Choose the **Destination Type** from the drop down list.

This defines the type of target file for your application. Choose from **Executable** (.EXE), **Library** (.LIB), or **Dynamic Link Library** (.DLL).

9. Type a name for the application's .HLP file in the **Help File** field, or use the ellipsis (**...**) button to select the file in the Open File dialog.

The Application Generator does **not** require that the .HLP file exist at this point. You can leave the field blank for now, then fill in the field at a later time.

The Application Generator allows you to name the help topics in your application without determining that the help file exists. You are responsible for creating a .HLP file that contains the context strings and keywords that you optionally enter as HLP attributes for the various controls and dialogs.

10. Choose the **Application Template** field, type. You may accept the default Clarion template, or press the ellipsis (**...**) button to select another (third party template set) in the **Select Application Type** dialog.

The selected application template controls code generation.

11. Optionally, check the **Application Wizard** box to use the wizard to create a complete application based on the selected dictionary and a few answers you specify. See **Application Wizard** for

more information.

12. Press the **OK** button.

Clarion for Windows creates the .APP file, then displays the **Application Tree** dialog for your new application.

How to Add and Customize a Procedure

After completing the **Application Properties** dialog, your new application tree contains a single procedure, called Main. The Main procedure is a "ToDo" item, which means it is basically a placeholder with no functionality.

To add functionality you Define the Procedure Type, and Define Procedure Properties.

Part of the functionality you will add a procedure, is calls to other procedures, which will in turn appear as "ToDo" items in the Application Tree.

Adding a Procedure

1. Select the Main "ToDo" procedure in the **Application Tree** dialog, then press the **Properties** button.

The **Select Procedure Type** dialog appears.

2. Uncheck the **Use Procedure Wizard** box.

3. Select Frame, then press the **Select** button.

The **Procedure Properties** dialog appears.

4. Press the **Window** button.

The green check mark means a window has already been defined. The **Window Formatter** appears.

5. Choose Menu Menu Editor.

The **Menu Editor** dialog appears. Locate the MENU &Window item in the **Menu Editor** list.

6. Highlight the END immediately above MENU &Window, and press the **Item** button.

A new action bar item is added to the window.

7. In the **Menu Text** field, type "My Procedure."

The new item will be labelled My Procedure.

8. Select the **Actions** tab, and select **Call a Procedure** from the **When Pressed** drop down list.

Now you can fill in the blanks and specify the procedure that executes when the user selects the new action bar item.

9. In the **Procedure Name** field, type "MyProcedure."

10. Check the **Initiate Thread** box.

11. Press the **Close** button to close the **Menu Editor** dialog.

12. **Exit!** the **Window Formatter**, and press the **Yes** button to save changes.

13. Press the **OK** button to close the **Procedure Properties** dialog.

Customizing a Procedure

You have added a new procedure to your application. The procedure appears as a "ToDo" item in the Application Tree, which, means it is basically a placeholder with no functionality. Add functionality by Defining the Procedure type and Defining Procedure Properties.

Define the Procedure Type

1. Select the Main "ToDo" procedure in the **Application Tree** dialog, then press the **Properties** button, or choose **Edit** **Properties**. You can also DOUBLE-CLICK on the "ToDo" procedure.

The **Select Procedure Type** dialog appears.

2. Uncheck the **Use Procedure Wizard** box.

3. Highlight a procedure type (Frame is best for the Main procedure), then press the **Select** button.

The **Procedure Properties** dialog appears.

Note: If you select a Browse, Form, or Report procedure, **and** you check the **Use Procedure Wizard** box, a **Wizard** dialog will guide you through each step of the procedure properties definition.

Define the Procedure Properties

Use the **Procedure Properties** dialog to define the procedure's propertiesthese properties include:

- a **description** of the procedure
- the procedure **prototype**
- the **module** containing the source code
- **parameters** passed to the procedure
- **files** accessed by the procedure
- **window** displayed by the procedure, including its size, shape, appearance and functionality
- **data** items (fields and variables) used by the procedure
- **procedures** called by the procedure
- **embed**ded source code within the procedure
- **formulas** used by the procedure
- template source code that **extends** the procedure

You need not define every property for every procedure. Frequently, the default property definitions are appropriate, and need no further customization.

Defining Procedure Files

File data (data stored in your application files) are **available** to any procedure within the entire application, however, you must tell the Application Generator which files will be used so it can provide source code for reading the file.

1. Press the **Files** button in the **Procedures Properties** dialog.

The **File Schematic Definition** dialog appears.

2. Select Other Files in the **Files** list, and press the **Insert** button.

3. Choose a file from the **Insert File** dialog.

The first file you add is always the "primary" file. All others are secondary.

4. Press the **Key** button. To specify a sort key for the file.

5. Choose a key from the **Change Access Key** dialog.

Defining Procedure Windows

The **Window Formatter** allows you to visually design the size, shape, menus, controls and functionality for the window in this procedure. Access the **Window Formatter** by pressing the **Window** button in the **Procedure Properties** dialog, or from the Application Tree, select a procedure, RIGHT-CLICK and choose **Window** from the popup menu.

Defining Procedure Data

Procedures may access several classes of data. These GLOBAL data, MODULE data, LOCAL data, and

file or field data(see Defining Procedure Files, Defining Entry Control Data, Select Field dialog). GLOBAL data may be accessed by any procedure in the entire application. MODULE data may only be accessed by the procedures contained in the module where the data are defined. LOCAL data may only be accessed within the single procedure where the data are defined.

Defining LOCAL Data

1. Select a procedure in the **Application Tree** dialog.
2. Press the **Properties** button, choose **Edit** **Properties,** or RIGHT-CLICK and choose **Properties** from the popup menu to display the **Procedure Properties** dialog.
3. Press the **Data** button to display the **Local Data** dialog.

If any local variables already exist, they appear in the list.

4. Press the **Insert** button and define the variable.

The **New Field Properties** dialog appears. Type in the variable name, choose the variable type, and set any additional attributes, including screen attributes. You can also specify how memory is allocated for the variable.

5. Close the **Field Properties** and the **Local Data** dialogs.

The data variables are now included in the procedure.

Defining MODULE Data

1. From the **Application Tree** dialog, select the **Module** tab.
2. Highlight a module (folder) inside the **Application Tree** dialog.
3. Press the **Properties** button to display the **Module Properties** dialog, or RIGHT-CLICK and choose **Data** from the popup menu.
4. Press the **Data** button.

The **Module Data** dialog appears.

5. Press the **Insert** button.

The **New Field Properties** dialog appears. Type in the variable name, choose the variable type, and set any additional attributes, including screen attributes. You can also specify how memory is allocated for the variable.

6. Close the **Field Properties**, the **Module Data**, and the **Module Properties** dialogs.

Defining Entry Control Data

Entry controls are those items in a window that display or accept values; for example, check boxes, entry boxes, and list boxes are entry controls. You define the fields and variables the entry controls can access by using the **Select Field** dialog. Access the **Select Field** dialog several ways:

By placing entry boxes, check boxes, and combo boxes in a window. When you first CLICK to place the control, the **Select Field** dialog appears. You can then select the field or variable whose value the control displays.

By choosing the **Populate** **Field** or **Populate**

Multiple Fields command in the **Window Formatter**. When you choose either of these populate commands, the **Select Field** dialog appears so you can select the field or variable that will be displayed by the control.

By placing a control template in a window. The point at which you can access the **Select Field** dialog varies according to the template. For example, if the control template contains a list box, a **Populate** button appears in the **List Box Formatter**, which allows you to select fields and variables from

the **Select Field** dialog.

Defining Calls to Other Procedures

Procedures may call other procedures. From **the Procedure Properties** dialog, press the **Procedures** button to access the **Called Procedures** dialog. To add a procedure, press the **Insert** button, and type a procedure name in the next dialog. To delete a called procedure, press the **Delete** button.

Note: The purpose of the Procedures button is to embed a call to another procedure and to add the called procedure to the Application Tree.

How to Add Embedded Source Code

Use the **Embedded Source** dialog to embed source code. Embedding source code in a procedure allows you to further customize your application. The Application Generator adds your embedded code to the code it generates, at precisely the point you specify. The procedure templates contain predefined **embed points**.

Once you embed source code, the procedure containing embedded source are flagged with a blue "**S**" on the procedure's icon in the Application Tree.

The **Embedded Source** dialog can be accessed at several levels:

 the procedure level (**Embeds** button on **Procedure Properties** dialog)
select from embed points throughout the entire procedure

 the window level (DOUBLE-CLICK the (see also)**sample window** in the (see also)**Window Formatter**)
select from embed points related to the window the events it generates

 the control level (DOUBLE-CLICK the control in the **Window Formatter**)
select from embed points related to the control and the events it generates

There are three ways to create the embedded source code: (see also)hand-coding with the text editor, (see also)calling another procedure, or (see also)using a code template. You can even combine all three methods, and the Embedded Source dialog provides powerful tools for (see also)managing embedded source code, including (see also)Cut, Copy, and Paste capability.

Hand-coding Embedded Source Code with the Text Editor

1. In the **Application Tree** dialog, highlight a procedure and press the **Properties** button, or RIGHT-CLICK and choose **Embeds** from the popup menu.
2. Press the **Embeds** button in the **Procedure Properties** dialog to display the **Embedded Source** dialog.

The **Embedded Source** dialog lists points within the procedure where your custom source code may be inserted. This includes the points where the **field specific events** occur within the procedure. For example, if you place an entry box in a window, the embed points you can access include points where the user **selects** or focuses on (TABS onto or mouse CLICKS on) the field, and points where the user completes or **accepts** (TABS off, presses the ENTER key, or presses the **OK** button) the field.

3. Select a point at which to embed the code and press the **Insert** button.

The **Select Embed Type** dialog appears. There are three ways to create the embedded source code: hand-coding with the text editor, calling another procedure, or embedding a template. You can even combine all three methods.

4. Select the **SOURCE** item in the **Select Embed Type** dialog.
5. Press the **Select** button to start the Text Editor with a blank source code window.
6. Write your custom code in the source code window.

Tip: Don't forget to use the on-line help for explanations and examples of Clarion Language syntax and techniques. Copy and paste directly from the help examples!

7. Choose **Exit!**.
8. Choose **Yes** when prompted to save the embedded source.
9. Press the **Close** button to close the **Embedded Source** dialog.

Embedding a Procedure Call

1. In the **Application Tree** dialog, highlight a procedure and press the **Properties** button, or RIGHT-CLICK and choose **Embeds** from the popup menu.
2. Press the **Embeds** button in the **Procedure Properties** dialog to display the **Embedded Source** dialog.

The **Embedded Source** dialog lists points within the procedure where your custom source code may be inserted. This includes the points where the **field specific events** occur within the procedure. For example, if you place an entry box in a window, the embed points you can access include points where the user **selects** or focuses on (TABS onto or mouse CLICKS on) the field, and points where the user completes or **accepts** (TABS off, presses the ENTER key, or presses the **OK** button) the field.

3. Select a point at which to embed the code and press the **Insert** button.

The **Select Embed Type** dialog appears. There are three ways to create the embedded source code: hand-coding with the text editor, calling another procedure, or embedding a template. You can even combine all three methods.

4. Select the **Call a Procedure** item in the **Select Embed Type** dialog.
5. Type a name for the procedure or choose an existing procedure from the drop down list which appears in the next dialog. The caption of the dialog box corresponds to the embed point chosen.

Typing a new name specifies that the application calls another procedure, which automatically appears in the Application Tree as a "To Do." If another procedure with the same name already exists, the Application Generator assumes you meant to call it, and does not add a new "To Do."

You define the functionality of the other procedure using the **Procedure Properties** dialog.

6. Press the **OK** button to close the dialog.

Embedding Source Code with a Code Template

Code templates help you construct complex source code with minimal effort on your part. When you select a code template to embed, Clarion displays a dialog box containing an explanation of what the template does as well as prompts for the information required to complete the source code.

The names of the available code templates appear in the **Select Embed Type** dialog under the Class Clarion item.

1. In the **Application Tree** dialog, highlight a procedure and press the **Properties** button, or RIGHT-CLICK and choose **Embeds** from the popup menu.
2. Press the **Embeds** button in the **Procedure Properties** dialog to display the **Embedded Source** dialog.

The **Embedded Source** dialog lists points within the procedure where your custom source code may be inserted. This includes the points where the **field specific events** occur within the procedure. For example, if you place an entry box in a window, the embed points you can access include points where the user **selects** or focuses on (TABS onto or mouse CLICKS on) the field, and points where the user completes or **accepts** (TABS off, presses the ENTER key, or presses the **OK** button) the field.

3. Select a point at which to embed the code and press the **Insert** button.

The **Select Embed Type** dialog appears. There are three ways to create the embedded source code: hand-coding with the text editor, calling another procedure, or embedding a template. You can even combine all three methods.

4. Select a code template in the **Select Embed Type** dialog and press the **Select** button.

This displays a **Prompts for...** dialog box (the title includes the name of the code template).

5. Read the instructions and explanations in the dialog.

Each code template includes explanatory text on its proper use and how to fill in the necessary options.

6. Fill in or choose from the options inside the **Prompts for...** dialog.

7. Press the **OK** button to close the dialog.

Managing Embedded Source Code

There are three ways to create the embedded source code: (see also)hand-coding with the text editor, (see also)calling another procedure, or (see also)using a code template. You can even combine all three methods, and you can embed multiple blocks of source code at a single embed point. You can even embed multiple blocks of source code at many different embed points.

Execution of multiple embedded blocks occurs in the order they are listed. Use the and ¯ buttons in the **Embedded Source** dialog to change the order of multiple embedded source blocks.

Use the **Delete** button to remove unwanted embedded source code, or use OMIT to temporarily or conditionally remove embedded source.

Use the **Properties** button to edit embedded source code.

Use the (see also)Cut, Copy, and Paste buttons to rearrange, copy, or move embedded source code.

Cut, Copy and Paste Embedded Source Code

1. In the **Embedded Source** dialog, highlight a line in the tree diagram.

Highlighting an embed point line (folder icon) selects **all** the embedded source at this embed point for subsequent cut and paste operations. Highlighting a single embed source item selects only that item.

2. Press the **Cut** or **Copy** button.

The selected code is placed in the clipboard.

3. Highlight another line in the tree diagram.

4. Press the **Paste** button.

The clipboard contents are pasted below the selected line. The paste operation will only accept embedded source definitions. In other words, you must first copy or cut embedded source, before you can paste.

Defining Procedure Formulas

Creating conditional expressions with the **Formula Editor** actually creates structures in the source code. There are three structures you can create with the **Formula Editor**a simple assignment expression, or an IF or a CASE structure. You can also nest either of these structures, creating complex conditional statements.

The **Formula Editor** creates simple (unconditional) assignments, the **Conditionals** dialog creates complex conditional assignments, and the **Formulas** dialog manages these formulaic assignments for your procedure. See:

(see also)How to Create a Simple Assignment Expression

(see also)How to Create a Complex Assignment Expression

Usually, you'll want to display the result of your assignment in a string control in a window or report. To display it in a window:

1. Press the **Window** button to open the **Window Formatter**.

2. Select the string tool from the **Controls** toolbox (or choose **Control** **String**), and CLICK in the window to place the control.

3. With the new string control selected, choose **Edit** **Properties** (or RIGHT-CLICK on the string and choose **Properties** from the popup menu).

4. Check the **Variable String** check box in the **String Properties** dialog.

The **Select Field** dialog appears.

5. Highlight the variable which contains the **result** of your formula and press the **Select** button.

The **String Properties** dialog returns, note however, that the **Parameter** field has now been replaced by the **Picture** field.

6. Type a valid display picture in the **Picture** field.

7. Press the **OK** button to close the **String Properties** dialog.

8. Choose **Exit!** to close the **Window Formatter** and return to the **Procedure Properties** dialog.

Adding Procedure Extensions

Extension and control templates provide additional functionality to basic procedure templates. **Control templates** give your procedure the ability to display and manage specific controls. For example a browse box may be added using a control template.

Extension templates give your procedure additional functionality not associated with specific controls. For example, date and time displays may be added using an extension template.

From the **Procedure Properties** dialog press the **Extensions** button to display the**Extension and Control Templates** dialog. This dialog displays a list of control and extension templates and the prompts associated with each template. Selecting a template on the left side of the dialog causes the prompts associated with the selected template to be displayed on the right side of the dialog.

Add extension templates by pressing the **Insert** button. (see also How to Customize Templates)Customize existing templates by filling in the prompts on the right side of the dialog.

Tip: Only Extension templates may be added and deleted using the Extensions button. Control templates may not be added or deleted, but may be modified. Control templates may be added or deleted from the Window Formatter by adding or deleting their associated controls.

How to Assign an Image to Display at Runtime

The parameter for an IMAGE control cannot accept a variable; however, you can reassign the image to display a runtime using a property assignment statement.

Insert the following line of source code in the embed point where the assignment will take place.

?Image{PROP:Text} = FileName

Optionally, you can use the DosFileLookup control template to allow a user to select the graphic image from a standard File Dialog.

How to Create a Simple Assignment Expression

A simple assignment evaluates an expression on the right side of the equal (=) sign and assigns it to the variable on the left side of the equal sign. The **Formula Editor** helps you build assignment expressions by providing access to all your valid variable names, plus immediate syntax checking.

 Within the Application Generator, DOUBLE-CLICK a procedure, to open the **Procedure Properties** dialog:

1. Press the **Formulas** button.

If you already have formulas in the procedure, the **Formulas** dialog appears. If this is the first formula in this procedure, the **Formula Editor** dialog appears, so skip step **2.**

2. Press the **New** button.

The **Formula Editor** dialog appears.

3. In the **Name** field, type a name for the formula.

4. Press the ellipsis (...) button next to the **Class** field to choose a formula Class.

A formula's class determines **when** its calculation is performed. Each template has its own set of classes. For example, in the Form Template there is a class called "After Lookups" which tells the Application Generator to compute the formula after all lookups to secondary files are completed for the procedure.

5. Optionally, type a description of the formula in the **Description** field.

6. Press the ellipsis (...) button next to the **Result** field to choose the variable to which the result of the expression is assigned.

You can choose a local, module, or global variable, or a data dictionary field.

7. Create your formula on the **Statement** line.

You may type in the expression, you may use the Formula Editor's buttons, or you may use a combination of the two.

The first component of an expression must be an operand, left parenthesis, or a unary minus (the negative sign).

8. Optionally, press an **Operands** button for the first component of your expression.

9. Optionally, press an **Operator** button for the next component of your expression.

10. Continue adding components to your expression until it is complete.

11. Press the **Check** button to check your syntax.

If the syntax is correct, a large green check mark appears to the left of the statement. If there is any incorrect syntax, a large red X appears.

12. Press the **OK** button.

How to Create a Complex Assignment Expression

An IF structure assigns a value to the **Result** variable based on the true/false evaluation of a *single* logical expression. There are *two* possible assignments. If the condition tested is true, one assignment is made, if not true (false), then the other assignment is made. Nesting IF structures allows for even more alternative assignments.

A CASE structure selectively assigns a value to the **Result** variable based on the evaluation of multiple OF expressions against the CASE expression. The CASE structure offers a less complicated (but less flexible) method for assigning alternative values. CASE structures may also be nested, and IF and CASE structures may be nested within each other. See the *Language Reference* for more information.

Complex Expressions - IF

Use an IF structure to assign one of two values to the **Result** field depending on a condition. Nesting IF structures allows more complex alternative assignments.

To create an IF conditional formula (from the **Procedure Properties** dialog):

1. Press the **Formulas** button.

If you already have formulas in the procedure, the **Formulas** dialog appears. If this is the first formula in this procedure, the **Formula Editor** dialog appears, so skip step **2**.

2. Press the **New** button.

The **Formula Editor** dialog appears.

3. In the **Name** field, type a name for the formula.

4. Press the ellipsis (...) button next to the **Class** field to choose Formula Class.

A formula class determines *where* in the generated source code its calculation is performed. Each Clarion procedure template has its own set of formula classes. For example, in the Form Template there is a class called "After Lookups" which tells the Application Generator to compute the formula after all lookups to secondary files are completed for the procedure.

5. In the **Description** field, type a description of the formula.

6. In the **Result** field, type the variable to which the result of the expression is assigned, or press the ellipsis (...) button to choose a variable from the **Select Field** dialog.

You can choose a local, module, or global variable, or a data dictionary field.

7. Press the **Conditionals** button.

8. Press the **IF..THEN** button.

The IF structure appears in the **Structure** window.

9. On the **Statement** line, enter the IF condition to evaluate.

You can type the expression, or you can use the **Operators** and **Operands** buttons to select expression components, or you can do both.

10. Press the **Check** button to check your syntax.

11. Press the **Accept** button to insert your expression into the structure.

12. Highlight the line below the IF line in the **Structure** window.

This is where the "True" assignment expression goes.

13. On the **Statement** line, enter the "True" assignment expression.

Again, you can type the expression, or you can use the **Operators** and **Operands** buttons to select

expression components, or you can do both. If the IF condition is true, this expression is evaluated and the resulting value is assigned to the Result variable.

A "true" assignment expression is *not* required. If no assignment is entered, then no assignment is made.

14. Press the **Check** button to check your syntax.

15. Press the **Accept** button to enter your expression into the structure.

16. Highlight the line below the ELSE line in the **Structure** window

This is where the "False" assignment expression goes.

17. On the **Statement** line, insert the "False" assignment expression.

Again, you can type the expression, or you can use the **Operators** and **Operands** buttons to select expression components, or you can do both. If the IF condition is false, this expression is evaluated and the resulting value is assigned to the Result variable.

A "false" assignment expression is *not* required. If no assignment is entered, then no assignment is made.

18. Press the **Check** button to check your syntax.

19. Press the **Accept** button to insert your expression into the structure.

To add a **nested** control structure:

20. Highlight one of the assignment lines in the **Structure** window.

21. Press either the **CASE..OF** or **IF..THEN** button.

A new nested structure appears in the **Structure** window.

22. Insert expressions on the appropriate lines as described above.

Again, you can type the expression, or you can use the **Operators** and **Operands** buttons to select expression components, or you can do both

23. When your control structure is complete, press the **OK** buttons in the **Conditionals, Formula Editor, and Formulas** dialogs.

Complex Expressions - CASE

A CASE structure can be used to assign one of several values to the **Result** field depending on which OF expression is equal to the CASE expression.

To create a CASE conditional formula (from the **Procedure Properties** dialog):

1. Press the **Formulas** button.

If you already have formulas in the procedure, the **Formulas** dialog appears. If this is the first formula in this procedure, the **Formula Editor** dialog appears, so skip step 2.

2. Press the **New** button.

The **Formula Editor** dialog appears.

3. In the **Name** field, type a name for the formula.

4. Press the ellipsis (...) button next to the **Class** field to choose Formula Class.

A formula class determines *where* in the generated source code its calculation is performed. Each Clarion procedure template has its own set of formula classes. For example, in the Form Template there is a class called "After Lookups" which tells the Application Generator to compute the formula after all lookups to secondary files are completed for the procedure.

5. In the **Description** field, type a description of the formula.

6. In the **Result** field, type the variable to which the result of the expression is assigned, or press the ellipsis (...) button to choose a variable from the **Select Field** dialog.

You can choose a local, module, or global variable, or a data dictionary field. This name appears in the **Formulas** dialog list.

7. Press the **Conditionals** button.

8. Press the **CASE..OF** button.

The CASE structure appears in the **Structure** window.

9. On the **Statement** line, enter the CASE expression that is compared to the multiple OF expressions.

You can type the expression, or you can use the **Operators** and **Operands** buttons to select expression components, or you can do both.

10. Press the **Check** button to check your syntax.

11. Press the **Accept** button to insert your expression into the structure.

12. Highlight the OF line below the CASE line in the **Structure** window.

This is where the first OF *comparison* expression goes.

13. On the **Statement** line, enter the OF *comparison* expression.

Again, you can type the expression, or you can use the **Operators** and **Operands** buttons to select expression components, or you can do both. At runtime, if the CASE expression equals this OF expression, then the subsequent assignment expression is evaluated and the resulting value is assigned to the Result variable.

14. Press the **Check** button to check your syntax.

15. Press the **Accept** button to insert your expression into the structure.

16. Highlight the line below the OF line in the **Structure** window.

This is where the first OF *assignment* expression goes.

17. On the **Statement** line, insert the OF *assignment* expression.

Again, you can type the expression, or you can use the **Operators** and **Operands** buttons to select expression components, or you can do both. At runtime, if the CASE expression equals the above OF expression, then this assignment expression is evaluated and the resulting value is assigned to the **Result** variable.

18. Press the **Check** button to check your syntax.

19. Press the **Accept** button to insert your expression into the structure.

To add additional OF statements:

20. Highlight an OF line in the **Structure** window.

21. Press the **Case..OF** button

22. Insert your expressions in the same manner described above.

oTo add a nested control structure:

23. Highlight an assignment line in the **Structure** window.

24. Press either the **CASE..OF** or **IF..THEN** button

25. Insert expressions on the appropriate lines following the instructions in the previous sections.

Again, you can type the expression, or you can use the **Operators** and **Operands** buttons to select expression components, or you can do both.

26. When your control structure is complete, press the **OK** buttons in the **Conditionals, Formula Editor, and Formulas** dialogs.

How to Customize Your Window

Use the **Window Formatter** to visually design window elementswindows, menus, toolbars, list boxes, prompts, entry fields, and other controlson screen. The **Window Formatter** automatically generates the Clarion language source code that defines these elements.

The **Window Formatter** has five major components that help design your window: the (see also)Sample Window, the (see also)Controls Toolbox, the (see also)Fields Toolbox, the (see also)Property Toolbox, and the(see also) Align Toolbox.

Using the Window Formatter - A Typical Procedure

Here is the typical process for customizing a new window with the **Window Formatter**:

1. Set the size of the window by dragging the handles so that the sample window is the size you wish.

2. Set other window attributes by using the **Window Properties** dialog.

RIGHT-CLICK the window and choose **Properties** from the popup menu, or select the window and choose **Edit**  **Properties.**

Other attributes include the window caption, whether the window is resizeable, whether the window is scrollable, icons, messages, help files, and cursor types associated with the window, and many others.

3. Close the (see also)**Window Properties** dialog.

4. Place controls in the window.

See also: (see also)Controls Menu, (see also)Populate Menu, (see also)Controls Toolbox, (see also)Fields Toolbox, (see also)Align Toolbox.

5. Set the properties for each control.

See also: (see also)Controls Menu, (see also)Controls Toolbox, (see also)Property Toolbox.

6. Preview the window by choosing **Preview!** from the action bar; repeat steps 1 - 6 to make any necessary adjustments while still in the **Window Formatter**.

7. Choose **Exit!** from the action bar to return to the Application Generator or Text Editor.

Using the Window Formatter - Sample Window

The **Window Formatter** is a visual design tool. You always see a sample of the window you're working on, as you work on it. For example, place a list box in the sample window and drag its handles to the size you want.

In addition, you can see the window, exactly as it will appear to the end user by choosing **Preview!** from the action bar. See also: **Window Properties** dialog; Choosing a window type.

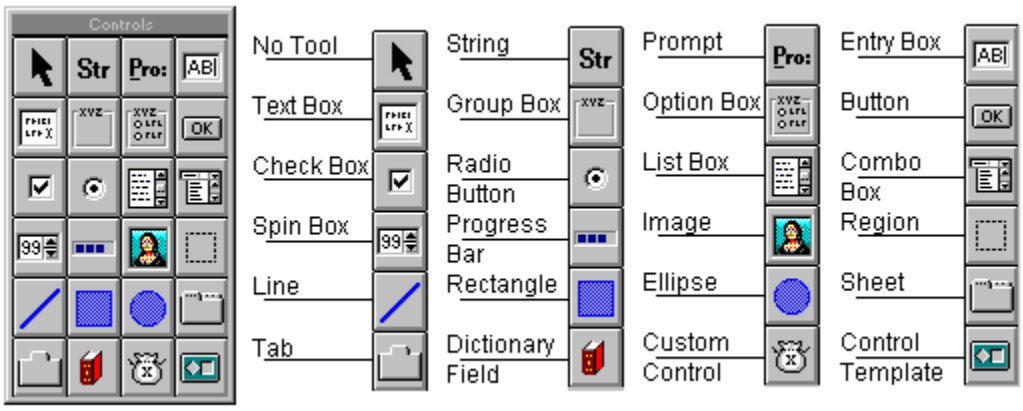
Using the Window Formatter - Controls Toolbox

The **Window Formatter** contains a floating **Controls** toolbox, similar to those found in many draw or paintbrush programs. Simply choose a control from the toolbox (CLICK on it), then CLICK in the sample window to place the control in the window.

Display or hide the **Controls** toolbox by choosing **Options** **Toolbox.** Resize the **Controls** toolbox by placing the cursor on the border of the box. When the cursor changes to a double headed arrow, CLICK and DRAG. All the controls in the toolbox are also available from the **Controls** menu.

See also: the (see also)**Controls** menu, (see also)**Fields** toolbox, (see also)**Align** toolbox, (see also)**Property** toolbox, (see also)**Window Properties** dialog; (see also)Using the Window Formatter - An Overview

Tip: Position the cursor over any tool and wait for half a second. A tool tip appears telling you the type of control that will be created by this tool.



String Allows you to place STRING control on the window under construction. See also **String Properties** dialog.

Prompt Allows you to place PROMPT control on the window under construction. See also **Prompt Properties** dialog.

Entry Box Allows you to place ENTRY control on the window under construction. See also **Entry Properties** dialog.

Text Box Allows you to place TEXT control on the window under construction. See also **Text Properties** dialog.

Group Box Allows you to place GROUP control (group box) on the window under construction. See also **Group Properties** dialog.

Option Box Allows you to place OPTION control (OPTION structure, which appears as a group box with radio buttons) on the window under construction. See also **Option Properties** dialog.

Button Allows you to place BUTTON control on the window under construction. See also **Button Properties** dialog.

Check Box Allows you to place CHECKBOX control on the window under construction. See also **Check Properties** dialog.

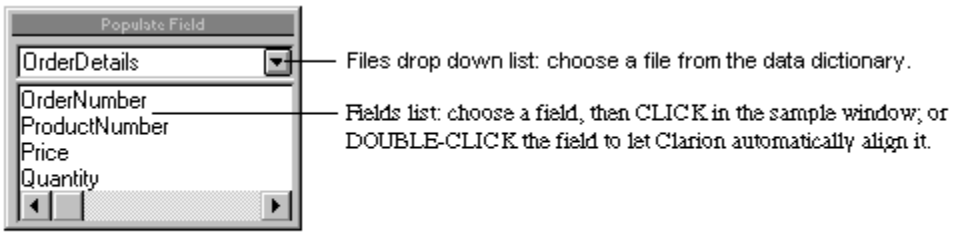
Radio Button	Allows you to place RADIO control on the window under construction. See also **Radio Properties** dialog.
List Box	Allows you to place LIST control (list box, or drop down list box) on the window under construction. See also: **List Box Formatter**. See also **List Properties** dialog.
Combo Box	Allows you to place a COMBO control (combo box, or drop combo box) on the window under construction. See also **Combo Properties** dialog.
Spin Box	Allows you to place a SPIN control on the window under construction. See also **Spin Properties** dialog.
Progress Bar	Allows you to place PROGRESS control on the window under construction. See also **Progress Properties** dialog.
Image	Allows you to place IMAGE control (graphic image) on the window under construction. See also **Image Properties** dialog.
Region	Allows you to place REGION control on the window under construction. See also **Region Properties** dialog.
Line	Allows you to place LINE control on the window under construction. See also **Line Properties** dialog.
Rectangle	Allows you to place a BOX control on the window under construction. See also **Box Properties** dialog.
Ellipse	Allows you to place ELLIPSE control on the window under construction. See also **Ellipse Properties** dialog.
Sheet	Allows you to place SHEET control on the window under construction. Sheet controls contain Tab controls. See also **Sheet Properties** dialog.
Tab	Allows you to place a TAB control on the window under construction. Tab controls may contain any other control types. See also **Tab Properties** dialog.
Dictionary Field	Allows you to select a field defined in the Data Dictionary, and place the control specified in the data dictionary, plus an associated PROMPT control, on the window under construction. See also **Select Field** dialog.
Custom Control	Allows you to place a CUSTOM control (Visual Basic custom control) on the window under construction. See also **Custom Properties** dialog.
Control Template	Allows you to place Control Template on the window under construction. See also ******* dialog.

Using the Window Formatter - Fields Toolbox

The **Window Formatter** contains a floating **Populate Field** toolbox. This toolbox allows you to quickly "populate" a window with entry controls **and** prompts for fields in your data dictionary files.

Display or hide the **Populate Field** toolbox by choosing **Options** **Fieldbox.** Resize the **Populate Field** toolbox by placing the cursor on the border of the box. When the cursor changes to a double headed arrow, CLICK and DRAG.

See also: the (see also)**Populate** menu, (see also)**Controls** toolbox.



1. Choose a **file** from the drop down list.

2. Select the **field** you want on your window.

DOUBLE-CLICK the field to let Clarion automatically align the controls.

3. CLICK in the sample window to place the control **and** its associated prompt.

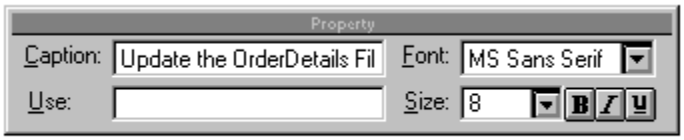
The cursor becomes a crosshair. The top left corner of the prompt is placed at the intersection of the cursor crosshair.

The **type** of control (entry box, check box, radio button, etc.) is determined by the settings for this particular field in the Data Dictionary.

Using the Window Formatter - Property Toolbox

The **Window Formatter's Property** toolbox allows you to quickly specify the appearance and content of the text on each control within the window and on the window title bar. Control the font, size, style, and content of all your text, using standard word processor buttons and drop down lists. See also **Select Font** dialog.

Display or hide the **Property** toolbox by choosing **Options** **Propertybox.** Resize the **Property** toolbox by placing the cursor on the border of the box. When the cursor changes to a double headed arrow, CLICK and DRAG.



Text Formatting: choose content, font, size, and style for the selected control or window.

Using the Window Formatter - Align Toolbox

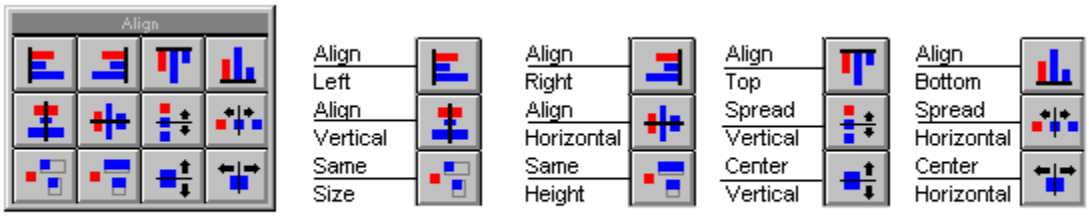
The **Window Formatter's Align** toolbox allows you to quickly, professionally, and precisely align the controls in your window.

Display or hide the **Align** toolbox by choosing **Options** **Alignbox.** Resize the **Align** toolbox by placing the cursor on the border of the box. When the cursor changes to a double headed arrow, CLICK and DRAG.

Select the controls to align (CTRL+CLICK allows you to select multiple controls, or you can "lasso" multiple controls with CTRL+DRAG), then click on the appropriate alignment tool. All the alignment actions are also available from the(see also)**Alignment** menu.

Tip: **For most alignment functions, the first controls selected (blue handles) are aligned with the last control selected (red handles). That is, the last control selected is the anchor control. It doesn't move, the others do.**

Tip: **Position the cursor over any tool and wait for half a second. A tool tip appears telling you the type of alignment this tool will accomplish.**



Align Left	Aligns the left borders of the selected controls with the left border of the last control selected (red handles).
Align Right	Aligns the right borders of the selected controls with the right border of the last control selected (red handles).
Align Top	Aligns the top borders of the selected controls with the top border of the last control selected (red handles).
Align Bottom	Aligns the bottom borders of the selected controls with the bottom border of the last control selected (red handles).
Align Vertical	Along a vertical axis, aligns the centers of the selected controls with the center of the last control selected (red handles).
Align Horizontal	Along a horizontal axis, aligns the centers of the selected controls with the center of the last control selected (red handles).
Spread Vertical	Equalizes the vertical spaces between the selected controls.
Spread Horizontal	Equalizes the horizontal spaces between the selected controls.
Same Size	Makes all selected controls the same height and width as the last control selected (red handles).
Same Height	Makes all selected controls the same height as the last control selected (red

handles).

Center Vertical	As a group (relative positions of selected controls don't change), centers the selected controls horizontally within the window.
Center Horizontal	As a group (relative positions of selected controls don't change), centers the selected controls vertically within the window.

How to Add a Toolbar

You may add a toolbar to any window with a simple command in the **Window Formatter**: choose the **Toolbar** **New Toolbar**. You may place any control on a toolbar, but the ones you will probably use the most are command buttons, check boxes, radio buttons, and drop down list boxes. As with menus, Clarion will automatically (see also) merge toolbars in certain situations.

Adding a Command Button

The following describes how to add a toolbar with a command button to a window. The starting point is the **Window Formatter**, open to an empty window:

1. From the **Toolbar** menu, choose **New Toolbar**.

A rectangular area appears at the top of the window. This is the toolbar. At runtime, it appears dark gray.

2. Optionally choose the **Options** **Grid Settings,** then check the **Snap to Grid** box.

This makes sizing and placing the controls easier.

3. Select the **Button** icon (**OK**) in the Controls toolbox, then CLICK inside the new toolbar in the sample window.

A button control appears.

4. RIGHT-CLICK on the button and select **Properties** from the popup menu, or choose **Edit** **Properties.**

The **Button Properties** dialog for the new button appears.

5. Delete the default text in the **Parameter** field.

This allows you to create a picture button without text.

6. Type a descriptive Field Equate Label in the **Use** field.

For a File/Open button, for example, you might type ?OpenButton. The Field Equate Label will appear in the **Embedded Source** dialog, making it easy to identify where to embed source.

7. From the **Extra** tab, choose an icon from the **Icon** drop down list, or type the name of an icon file (*.ICO) of your own.

The icon list contains a number of default icons for such standard actions as File/Open, or Cut, Copy, and Paste.

8. Add functionality to the button.

Select an STD ID from the drop down list, or select the **Actions** tab and embed source code, call a procedure or run a program.

9. Press the **OK** button to close the **Button Properties** dialog.

10. Resize the button to the size you want by dragging its handles.

Tip: **Clarion's . ICO files are 32 x 32 pixels . Most toolbar buttons will be smallerfor example, 16 x 18 pixels. By using these larger files, we can create the "disabled" icon from the same file, rather than requiring a separate file. When creating a custom .ICO file for a toolbar button, place the image in the center of the icon file. Clarion automatically crops the image to fit the button size.**

Adding a "Latched" Button

A latched button "stays depressed" when CLICKED, then returns to its original state when CLICKED a

second time. To place latched button:

1. Select the **Check Box** icon in the Controls toolbox, then CLICK inside the new toolbar in the sample window.

The **Select Field** dialog appears.

2. Highlight Local Data, then press the **Insert** button.

The **New Field Properties** dialog appears.

3. In the **Field Name** field, type a name, then choose **BYTE** from the data type drop down list.

The **Check Box Properties** dialog appears. A button created from a check box control has two modes: on or off. When the check box is 'on' (the button appears 'pushed in'), and the value of its USE variable is one. When the check box is 'off' (the button appears raised), and the value of its USE variable is zero.

4. From the **Extra** tab, choose an icon from the **Icon** drop down list, or type the name of an icon file (*.ICO) of your own.

5. Press the **OK** button.

The button is complete; you need only adjust its position by dragging its center, if necessary.

Adding a Button Group

A button group provides the user with **mutually exclusive choices**. For example, in a group of three buttons, only one can be "depressed." If button number two is currently "depressed," push in button number one, and button number two pops out. A button group can provide controls for left, right and center text justificationonly one option can be active at a time.

To create a button group:

1. CLICK on the Option Box icon in the Controls toolbox, then CLICK inside the toolbar.

The **Window Formatter** places an Option Box on the toolbar. You may resize it by dragging its handles. An Option Boxan OPTION structuremust always surround radio button choices, however, this Option Box will not appear on the toolbar, because you will hide it.

2. RIGHT-CLICK on the Option Box and choose **Properties** from the popup menu.

The **Option Properties** dialog appears.

3. Press the ellipsis (...) button for the **Use** field, and define a string variable.

The variable may be global, module, or local data, or it may be a data dictionary field. The variable will receive the **Value** text from the button selected by the user. If you don't specify any **Value** text, it gets the **Parameter** text from the selected button. If you define a numeric variable, it will receive an integer value corresponding to the selected button, that is, button 1, 2, or **3.**

4. From the **Extra** tab, uncheck the **Boxed** box.

This hides the Option Box from the user. It appears in the **Window Formatter** dialog, but will not appear at runtime.

5. Press the **OK** button.

6. CLICK on the Radio Button icon in the Controls toolbox, then CLICK inside the Option Box.

The Application Generator places a Radio Button where you clicked in the Option Box.

7. RIGHT-CLICK on the Radio Button and choose **Properties** from the popup menu.

The **Radio Button Properties** dialog appears.

8. Clear the **Parameter** field.

Clearing this field will remove text from the button so we can add an icon with no text.

9. In the **Value** field, type "Left."

When the user presses this button, the string "Left" is assigned to the USE variable we specified above.

10. From the **Extra** tab, choose an icon from the **Icon** drop down list, or type the name of an icon file (*.ICO) of your own.

Adding an icon causes the radio button to look like a command button.

11. Press the **OK** button.

The first button is complete; you need only adjust its position by dragging its center.

12. Repeat steps 6 through 11 for the "center" and "right" buttons.

13. Choose **Preview!** from the **Window Formatter** menu.

This displays the window, including the toolbar and menus, as it would to the user at runtime. Test the latching and radio features by pushing the buttons. Press ESC when done previewing your window.

14. Choose **Exit!** from the **Window Formatter** menu to save your window.

Toolbar Merging

Global and Local Tools

The TOOLBAR structure declares the tools displayed for a window. On an APPLICATION window, the TOOLBAR defines Global tools available to all the windows in the application. However, if the NOMERGE attribute is specified on the APPLICATION's TOOLBAR, the tools are local and are displayed only when no MDI child windows are open; there are no global tools. Global tools are active and available on all MDI child windows unless an MDI child window's TOOLBAR structure has the NOMERGE attribute.

MDI Windows

On an MDI child window, the TOOLBAR defines local tools that are automatically merged onto the Global toolbar. Both the Global and the local tools are then active while the MDI "child" window has input focus. Once the window loses focus, its specific tools are removed from the Global toolbar. If the NOMERGE attribute is specified on an MDI child window's TOOLBAR, the local toolbar replaces the Global toolbar.

Non-MDI Windows

On a non-MDI WINDOW, the TOOLBAR is **never** merged with the Global menu. A TOOLBAR on a non-MDI window always appears in the window, not on any parent window which may have been previously opened.

Merging Order

When an MDI window's local TOOLBAR is merged into an application's global TOOLBAR, the global tools appear first, followed by the local tools. The toolbars are merged so that the tools in the local toolbar begin just right of the position specified by the value of the width parameter of the global TOOLBAR's AT attribute. The height of the displayed toolbar is the maximum height of the "tallest" tool, whether global or local. If any part of a control falls below the bottom, the height is increased accordingly.

Note: To merge toolbars, the global toolbar's AT width must be less than the APPLICATION's frame width.

How to Minimize a window

A Minimize button is added to a WINDOW if you specify an ICON for the WINDOW. When the user presses the Minimize button, the window is reduced to an Icon.

In the Window Formatter:

1. Make sure the Window is selected.
2. Choose **Edit** **Properties** (or press ENTER).

   The **Window Properties** dialog appears**.**
3. Select the **Extra** tab.
4. In the **Icon** combo box, select a standard icon, type the name of the icon file, or select Select File... to locate an icon file using the standard Open File dialog.
5. Press the **Ok** button to close the **Window Properties** dialog.

How to Create a New Menu

Here are the steps for creating a menu starting from an empty window within the **Window Formatter**.

1. Choose the **Menu** **New Menu** command.

The **Menu Editor** dialog appears. Only the MENUBAR statement is present.

2. In the **New** group box, press the **Menu** button.

This adds the first MENU statement, its name, and its corresponding END statement, ready for editing.

3. In the **Menu Text** field, type the text you want displayed for this MENU.

The ampersand within the menu text signifies that the character *following* the ampersand is the *accelerator key.* For example, type &FILE, so the end user sees **File**.

4. In the **Use Variable** field, type a Field Equate Label.

A Field Equate Label has a leading question mark (?), and you should make it descriptive. You refer to the MENU within executable code by its Field Equate Label.

5. In the **New** group box, press the **Item** button.

This inserts an ITEM between the MENU statement and its END statement. Note that ITEMs are used to execute commands or procedures, whereas MENUs are used to display a selection of other MENUs or ITEMs.

Tip: When using the Application Generator, each ITEM you place on a MENU or MENUBAR automatically adds an embed point to the control event handling tree in the Embedded Source dialog. This allows you to easily attach functionality to your ITEMs.

6. In the **Menu Text** field, type the text you want to display for this menu ITEM.

For example, type &OPEN, so the end user sees **Open**. The ampersand within the ITEM name signifies the character *following* the ampersand is the *accelerator key*.

7. In the **Use Variable** field, type a Field Equate Label.

A Field Equate Label has a leading question mark (?), and you should make it descriptive. For example ?FileOpen shows at a glance the intended purpose of this ITEM: to open a file.

8. In the **Message** field, type theMSG attribute contents.

This message text displays in the status bar (if enabled) when the user highlights this MENU or ITEM.

9. In the **Help ID** field, type either a help keyword or a context string present in a .HLP file.

If you fill in the **Help ID** for a MENU or an ITEM, when the user highlights the MENU or ITEM and presses F1, the help file opens to the referenced topic. If more than one topic matches a keyword, the search dialog appears.

A Help keyword is a word or phrase indexed so that the user may search for it in the *Help* **Search** dialog. When referencing a context string in the **Help ID** field, you must identify it with a leading tilde (~).

10. From the **Actions Tab**, choose ***Call a Procedure*** from the **When Pressed** drop down list.

The procedure you specify executes when the user selects this ITEM. You may specify parameters to pass and standard file actions (insert, change, delete, or select) if applicable, or you may initiate a new thread. The *procedure* appears as a "ToDo" item in your Application Tree (unless you named a procedure that already exists).

This is one way to add functionality to your ITEM. You may also add functionality by ***Run a Program*** from the drop down list, by embedding source code, or by typing an STD ID in the (see also)**STD ID** field.

After following these steps, you have a single MENU called **File**, with a single ITEM called **Open**. To add other ITEMS to the MENU, repeat steps **5** through **10**. To add a second MENU, select the END statement

and press the **Menu** button. To add a subMENU, select a MENU or ITEM statement and press the **Menu** button.

11. To finish the menu and return to the **Window Formatter**, press the **Close** button.

How to Implement Standard Windows Behavior

There are some menus and commands that you see in almost every windows program. For example, Cut, Copy, and Paste. Clarion provides an easy method for implementing these standard actions in your application menuswith the **Std ID** field on the **Menu Editor** dialog.

Simply enter one of the equates listed below in the **Std ID** field. Clarion will automatically implement the command using standard windows behavior; you do not need any other support for it in your code. The standard equate labels and their associated actions are also contained in the C:\CW\LIBSRC\ EQUATES.CLW file.

STD:PrintSetup	Printer Options Dialog.
STD:Close	Closes active window.
STD:Undo	Reverses the last editing action.
STD:Cut	Deletes selection, copies to clipboard.
STD:Copy	Copies selection to clipboard.
STD:Paste	Pastes clipboard contents at the insertion point.
STD:Clear	Deletes selection.
STD:TileWindow	Arranges child windows edge to edge.
STD:TileHorizontal	Arranges child windows edge to edge.
STD:TileVertical	Arranges child windows edge to edge.
STD:CascadeWindow	Arranges child windows so all title bars are visible.
STD:ArrangeIcons	Arranges iconized child windows.
STD:WindowList	Adds child window names to menu.
STD:Help	Opens .HLP file to the contents page.
STD:HelpIndex	Opens .HLP file to the index.
STD:HelpOnHelp	Opens Microsoft's .HLP file for the Windows Help system.
STD:HelpSearch	Opens Microsoft's Help Search utility for the .HLP file.

How to Create an MDI Menu

Multiple Document Interface applications make special demands upon a program. Often, the program may support a variety of document windows, each of which has a slightly different set of commands from which the user may select.

Normally in this situation, the programmer writes code to monitor which window is active, then changes the menu and toolbar to reflect the options available to the user. Clarion does this automatically.

To create menus for MDI applications:

1. Create a master menu for the APPLICATION frame window.

Most likely, this will include a File menu and a Help menu, since they contain functions that are available even when no document windows are open.

Tip: Clarion's Application Frame procedure template comes with a predefined menu with many of the most common functions already provided for you.

You will use the **Window Formatter's Menu Editor** to create your menus. Be sure to choose the FIRST attribute for the File MENU, and the LAST attribute for the Help MENU from the **Position** drop down list. This ensures that when Clarion merges this global menu with local menus, File and Help will keep their correct positions.

2. *Plan* the additional menus for the child windows.

Can they all share the same menu titles? Do they share many of the same commands? Ideally, *most* of the MENUs and ITEMs can be active in *all* the child windows. If there are only a few commands specific to certain windows, plan on disabling those MENUs and ITEMs in the windows that don't support them, and enabling them in those that do.

3. Create the menu for the first child window.

Again, you will use the **Window Formatter's Menu Editor** to create the menu. Add any window-specific MENUs to the first child window. That is, the window-specific MENUs the application frame lackssuch as Edit, Insert, etc.

Optionally, add a File MENU to the first child window. This is necessary only if the child window needs an ITEM on the File MENU that is not already included on the application's File MENU. For example, adding a Close command might be appropriate. If so, add the File MENU to the first child window. Add the Close ITEM to the File MENU.

Add the Window MENU to the first child window. Window MENUs are standard for most windows programs. A typical Window MENU includes the following ITEMs: "Arrange Icons," "Tile," "Cascade," plus a document (windows) list that displays all open child windows and allows the user to switch between them. In many cases this entire MENU, including the document list, can be implemented with standard ID's (StdID's).

4. Exit the **Menu Editor** and save the menu.

5. Test the interaction of these first two menus.

Do they merge the way you planned? Are the correct selections available for the window with focus? Make any adjustments with the **Menu Editor**.

6. Repeat steps **3** through **5** for other child windows.

How to Use a Combo Box

This topic describes how to use a combo box without using the File Drop Combo Control template. For information on using the template, see File Drop Combo Control template.

There are two ways to use a combo box--for a static list and for a list of choices from a file.

For a finite list of static choices:

1. Place a COMBO control on the window.
2. In the **Select Field** dialog, select the USE variable for it to update.
3. Press the **Cancel** button to close the **List Box Formatter** (no need to use it).
4. RIGHT-CLICK the combo box, then choose **Properties**.
5. In the **From** field, type the list of choices as a string constant (in single quotes with | separating choices), eg 'One|Two|Three'
6. Enter a numeric value into the **Drop** field.

This the number of rows displayed at one time by the drop down list.

7. Enter the correct picture token in the **Picture** field.
8. Press the **OK** button.

For a list of choices from a file:

1. On the **Procedure Properties** dailog, press the **Data** button.
2. Add a QUEUE to the local data.
3. Add a single field to the QUEUE to hold the data from the file.
4. Return to the **Procedure Properties** dialog, then press the **Window** button.
5. Place a COMBO control on the window.
6. In the **Select Field** dialog, select the USE variable for it to update.
7. Press the **Cancel** button to get out of the **List Box Formatter** (no need to use it).
8. Right-click then choose **Properties**.
9. Type the name of the QUEUE into the FROM field, or press the ellipsis button to select or define a QUEUE.
10. Enter a numeric value into the **Drop** field.

This the number of rows displayed at one time by the drop down list.

11. Enter the correct picture token in the **Picture** field.
12. Press the **OK** button.
13. Return to the **Procedure Properties** dialog, then press the **Embeds** button.
14. Open the Procedure Setup embed point and add code to open the file and build the COMBO control's QUEUE.
15 Open the End of Procedure embed point and add code to close the file and FREE the QUEUE.

How to Create a List Box

When creating a list box control, you define its data source, its functionality, and its format. Clarion's development environment divides these property definitions among several dialogs:

The **List Properties** dialog specifies a drop down list versus a regular list, specifies the file or queue that supplies the data, and specifies the general scrolling capability, that is, all the properties of the list box that are **not** column-specific. This dialog is discussed in the previous chapter.

 The **List Box Formatter** dialog lets you add, delete, reorder, and resize the specific fields or columns that are displayed in the list box. This dialog is discussed in this chapter.

 The **List Field Properties** dialog defines the appearance and behavior of individual list box columns. For example, define individual column headers, widths, and scrolling. This dialog is discussed in this chapter

 The **List Field Properties** dialog also defines the appearance and behavior of **groups** of columns within the list box. For example, you can spread a header across several columns.

After you've started defining your list box with the **List Properties** dialog, these are the general steps for completing your list box.

Adding Columns to a List Box

1. From the **Window Formatter**, RIGHT-CLICK on the list box control, and choose **List Box Format** from the popup menu to display the **List Box Formatter** dialog.

The **List Box Formatter** displays a representation of the list box. Each field appears as a column in the list box, the data represented by "$" characters for strings, or "<" and "#" characters for numbers.

2. Press the **Populate** button to add a new field. (When working from the Text Editor, the **Insert** button replaces the **Populate** button).

When working from within the Application Generator, choose a field from the **Select Field** dialog. The **List Box Formatter** reappears, with the new column added. In the Text Editor, the **Select Field** dialog does not appear; you go directly to the **List Field Properties** dialog, so skip step **3.**

3. Press the **Properties** button.

The **List Field Properties** dialog appears. Use this dialog to specify column headers, widths, borders, scrolling, etc. Specifications for the first column become the default settings for subsequent columns.

4. Specify the column width in (see also)dialog units.

Provide about four dialog units for each character to be displayed.

5. Specify a picture token for the data.

The picture token determines how the data is displayed. For example, the picture token @P(###) ###-####P displays a phone number as (555) 555-5555**.**

6. Specify optional formatting.

You can choose the justification and set indentation. You can specify a column header, borders, underlining, and more.

7. Specify optional functionality.

For example, add a scroll bar for a single column. Allow column searches. You can specify resizeable borders, that allow the end user to adjust column widths with the mouse.

8. Press the **OK** button to return to the **List Box Formatter** dialog.

9. Repeat steps 2 - 8 for each additional column.

For each modification you make to the list box on screen, the **List Box Formatter** creates the appropriate FORMAT attribute for the LIST statement that defines your list box. The LIST statement, in turn, resides in the WINDOW structure. See the **Language Reference** for a complete explanation.

How to Create Column Groups Using the List Box Formatter

List box groups are composed of two or more fields which share common formatting elements, such as a header, or even the same columnar space within the list box.

1. From the **List Box Formatter**, select the a column and press the **Properties** button.

The **List Field Properties** dialog appears.

2. In the **List Field Properties** dialog, select the **Group** tab.

3. Press the **Yes** button when prompted to create a new group.

This specifies that the currently selected field will share formatting elements with the next field you add, or with any fields you move into the group.

4. Specify the group heading text.

The simplest shared element is a common header. The group header appears directly above the field headers for the two fields.

5. Optionally specify additional formatting.

You can, for example format the fields so that no separator appears between the members of the group, but a separator does appear at the end of the group. To do so, be sure the **Right Border** box is unchecked for the first field(s) in the group, and is checked only for the last.

6. Press the **OK** button to close the **List Field Properties** dialog.

7. Press the **Populate** (or **Insert**) button to add a **new** field to the group, or use the ***left/right arrow buttons to move **existing** fields into or out of the group.

How to Make a Record Occupy Two or More Rows in a List Box.

1. From the **List Box Formatter**, select the a column and press the **Properties** button.

The **List Field Properties** dialog appears.

2. In the **List Field Properties** dialog, select the **Group** tab.

3. Press the **Yes** button when prompted to create a new group.

This specifies that the currently selected field will share formatting elements with the next field you add, or with any fields you move into the group.

4. Select the **Field** tab and check the **Last on Line** box.

This option is equivalent to adding a carriage return immediately after the current field. The next field within the group appears directly below the current field, within the same column.

5. Press the **OK** button to close the **List Field Properties** dialog.

6. Press the **Populate** (or **Insert**) button to add a **new** field to the group, or use the ***left/right arrow buttons to move **existing** fields into or out of the group.

How to Restore User Resized List Box Column Widths

The resize feature on listboxes is most useful if the user specified sizing is remembered and reapplied to the list box. The following procedure uses embedded GETINI, PUTINI, and PROPERTY assignment syntax statements to save and restore the user specified formatting changes. The formatting is stored in an application specific .INI file. See the Language Reference for more information.

1. In the Application Tree dialog, DOUBLE-CLICK on your browse procedure.

The **Procedure Properties** dialog appears.

2. Press the **Embeds** button.

The **Embedded Source** dialog appears.

3. In the **Embedded Source** dialog, DOUBLE-CLICK on the "Preparing to Process the Window" embed point.

The **Select Embed Type** dialog appears.

4. DOUBLE-CLICK on SOURCE.

The Text Editor appears, ready to accept your embedded source statements.

5. Type the following statement, then **Exit**! the Text Editor and save your changes.

?Browse:1{PROP:FORMAT}=GETINI('Preferences','?Browse:1Format',?Browse:1{PROP:FORMAT},'MyApp.INI')

where ?Browse:1 is the field equate label for your list box, and MyApp is the name of your .APP file. "Preferences" is the section in the .INI file where the information is stored. "?Browse:1Format" is the entry in the .INI file where the information is stored, and "?Browse:1{PROP:FORMAT}" supplies the current format string as the default in case there is no formatting information in the .INI file.

6. DOUBLE-CLICK on the "End of Procedure, before closing window" embed point.

The **Select Embed Type** dialog appears.

7. DOUBLE-CLICK on SOURCE.

The Text Editor appears, ready to accept your embedded source statements.

8. Type the following statement, then **Exit**! the Text Editor and save your changes.

PUTINI('Preferences','?Browse:1Format',?Browse:1{PROP:FORMAT},'MyApp.INI')

where ?Browse:1 is the field equate label for your list box, and MyApp is the name of your .APP file.

How to Trap a Double Click on a List Box

Trapping a DOUBLE-CLICK on a list box is built into the Clarion Browse templates. To trap a DOUBLE-CLICK on a list control in hand-code:

1. Establish an ALRT(double-click) on the list control.
2. Trap for EVENT:AlertKey on the list control.
3. Trap for the MouseLeft2 keycode, as in the following example:

```
ACCEPT
  CASE FIELD()
  OF ?List
    CASE EVENT()
    OF EVENT:AlertKey
      IF KEYCODE() = MouseLeft2
        CurrentSel = CHOICE(?List1) ! Get current selection in list box
        GET(TheQueue, CurrentSel)     ! Get corresponding data from queue
. . . .
```

The above code finds out what item the user DOUBLE-CLICKED on using the CHOICE() function, then uses the GET() function to retrieve the item from the QUEUE.

You can add the two lines of code within the above IF structure to the Browse Double Click Handler embed point to handle DOUBLE-CLICKS for lists populated with the BrowseBox control template in the Application Generator.

How to add Drag and Drop to a List Box

Drag and Drop capability for lists means the user can select an item in a list box, hold down the left mouse button, "drag" the item to another control, release the mouse button to "drop" the item on the control, which can look at the data that was "dropped" on it, and then do something with it.

Adding Drag and Drop to a Clarion for Windows list box is a simple operation. This section provides an example of dragging an item from one list box to another, within the same application. You can also "Drag and Drop" to or from another applicationfor example, File Managersee the **Language Reference** for more details.

To implement Drag and Drop, you must add the DRAGID and DROPID attributes to the controls. You can add either or both to a control. The simplest, quickest way to do this is with Property Syntax statements. Assume for this example that the field equate labels for the two list boxes are ?FromList and ?ToList. Assume you want the end user to be able to drag **from** ?FromList **to** ?ToList.

Set up ?FromList as a **drag host**:

1. RIGHT-CLICK on the list control and choose **Properties** from the popup menu.
2. Select the **Extra** tab.
3. In the **Drag ID** field, type "FromList."
4. Press the **OK** button.

This sets the Drag ID signature which identifies "FromList" as the source of any "drag" operation from this control.

Set up ?ToList as a **drop target**:

1. RIGHT-CLICK on the list control and choose **Properties** from the popup menu.
2. Select the **Extra** tab.
3. In the **Drop ID** field, type "FromList."
4. Press the **OK** button.

This sets the Drop ID signature which specifies that the list will accept any "drop" operation with a Drag ID signature of "FromList."

Add drag **functionality** to the drag host, that is, detect a drag event and provide something to drag and drop:

1. RIGHT-CLICK on the FromList control and choose **Embeds** from the popup menu.
2. Locate the "Control Event Handling, before generated code; event:Drag" embed point and embed the following code:

```
IF DRAGID()                    ! Doesn't matter who dropped it for now
  SETDROPID('string to drag and drop')    ! Passing a simple string
END
```

This code detects a drag eventat the time the user **releases** the mouse button over a valid drop targetand places a string to drag with the SETDROPID function.

You can just as easily use the CHOICE() and GET() functions to retrieve an item from the local QUEUE for the first list box, then place the item in a global QUEUE. Then, upon detecting a drop event in the second list box, you could ADD from the global QUEUE to the local QUEUE for the second list box.

Add drop functionality to the drop target, that is, detect a drop event and retrieve whatever was dropped:

1. RIGHT-CLICK on the ToList control and choose **Embeds** from the popup menu.
2. Locate the "Control Event Handling, before generated code;" event:Drop embed point and embed the following code:

```
MyField = DROPID()     ! Retrieve the passed string
CallMyProcedure          ! Handle the rest in procedure
```

This code detects a drop eventat the time the user **releases** the mouse button over the drop targetand retrieves the "dropped" string with the DROPID function.

You can just as easily use the CHOICE() and GET() functions to retrieve an item from the local QUEUE for the first list box, then place the item in a global QUEUE. Then, upon detecting a drop event in the second list box, you could ADD from the global QUEUE to the local QUEUE for the second list box.

How to Create a Wizard

A wizard is a window with a "tabless" SHEET control containing one or more TABS. You'll need to write the code to handle the "turning of the pages".

This topic explains one method of creating a wizard using **<<Back** and **Next>>** buttons.

1. Create a procedure using the Window Template.

2. Create two Local Variables (by pressing the Data Button on the Procedure properties dialog).

Label	**Data Type**	**Initial Value**
TabNumber	Byte	1
MaxTabs	Byte	number of desired TABs

3. Place a SHEET Control on the Window.

4. Place the desired number of Tabs on the SHEET.

5. Design each TAB.

6. On the **Extra** tab, check the **Wizard** box on the **Sheet Properties** dialog.

This adds the WIZARD attribute to the SHEET control, which hides the "tab" portion of the TAB controls. Waiting to add this attribute until after designing the TABs makes TAB design easier.

7. Place two button controls under the SHEET control.

Use	**Caption**
?Back	<<Back
?Next	Next>>

8. Place a third button control under the SHEET control using either a standard button, a Save Button Control template, or a Close Button control template.

Use	**Caption**
?Finish	Finish

The type of control will depend on the task you intend the wizard to perform. If you are using a Save Button control template, you will need to either call the wizard from a browse or set GlobalRequest=InsertRecord in the Initialize Procedure embed point.

9. In the After Opening Window embed point, add this code:

```
HIDE(?Finish)
Select(?sheet1,TabNumber)
Disable(?Back)
```

This hides the **Finish** button and disables the **<<Back** button when the window opens.

10. In the Control Event Handling before generated code, ?Back, Accepted embed point, add this code:

```
TabNumber -=1
CASE TabNumber
  OF 1
    HIDE(?Finish)
    DISABLE(?Back)
    SELECT(?SHEET1,TabNumber)
  OF MaxTabs
    UNHIDE(?Finish)
    DISABLE(?Next)
    SELECT(?SHEET1,TabNumber)
```

```
  ELSE
    HIDE(?Finish)
    ENABLE(?Back)
    ENABLE(?Next)
    SELECT(?SHEET1,TabNumber)
END
```

This code decrements TabNumber, disables inappropriate buttons, and keeps the Finish button hidden until the final TAB.

11. In the Control Event Handling before generated code, ?Next, Accepted embed point, add this code:

```
TabNumber +=1
CASE TabNumber
  OF 1
    HIDE(?Finish)
    DISABLE(?Back)
    SELECT(?SHEET1,TabNumber)
  OF MaxTabs
    UNHIDE(?Finish)
    DISABLE(?Next)
    SELECT(?SHEET1,TabNumber)
  ELSE
    HIDE(?Finish)
    ENABLE(?Back)
    ENABLE(?Next)
    SELECT(?SHEET1,TabNumber)
END
```

This code increments TabNumber, disables inappropriate buttons, and keeps the Finish button hidden until the final TAB.

12. Write the code for the Finish button to accomplish the desired tasks and close the window.

What is a Dialog Unit

Dialog units are Windows' standard way of measuring distance on screen.

Dialog units are defined as one-quarter the average character width by one-eighth the average character height of the default font used on a window (creating a roughly square unit of measurement). The actual size of a dialog unit can vary from one window to the next, depending upon your selection of the font to use for the window. If you do not choose a specific font for a window, the Windows default font is used as the default font for the window and as the basis for the definition of the size of a dialog unit.

The purpose of using such a "floating definition" for sizing and positioning controls in Windows is to ensure that the relative design of your windows is kept despite any font change. The user can change their Windows default font in SYSTEM.INI, and Windows video drivers supplied by video board manufacturers can also change the Windows default font. Therefore, using dialog units ensures consistent relative window and control size and positioning from one computer to the next.

How to complete an entry field when the last character is entered

In Clarion for DOS, Clarion Database Developer and Clarion Professional Developer an entry field with the immediate attribute (IMM) was automatically completed when the last character was typed. In Clarion for Windows, the IMM attribute behaves differently. In Clarion for Windows, an entry field with the IMM attribute generates an Event:NewSelection as each character is typed.

To mimic the behavior of Clarion DOS products, a few lines of embedded source code are needed.

In the Window Formatter:

1. RIGHT-CLICK on the control, then select **Properties.**
2. On the **Extra** tab, check the **Immediate** box, then press the **Ok** button.
3. DOUBLE-CLICK on the control to access the **Embedded Source** dialog.
4. Select the "Control Event Handling, before generated code:Event:NewSelection" embed point for the control, then press the **Insert** button.

This embed point will only appear after you have checked the **Immediate** box.

5. Select SOURCE and type one of the following code segments in the Embedded Source Code Point:

Use this code if the field is a string:

```
UPDATE(?PRE:FieldName)
IF LEN(CLIP(PRE:FieldName)) = SIZE(PRE:FieldName) AND KEYCODE()<> MouseLeft          ! use size of -1
with CSTRING or PSTRING
  SELECT(?+1)
END                         !IF
```

Use this code if the field is any non-string numeric data type:

```
UPDATE(?PRE:FieldName)
Str" = PRE:FieldName
IF LEN(CLIP(Str") = 5 AND KEYCODE() <> MouseLeft
  SELECT(?+1)
END                         !IF
```

In this example the value of the field is assigned to an implicit string variable (Str") so that its length can be determined. Its length is compared to a constant number (in this case 5) instead of using the SIZE function. Since SIZE() returns the number of BYTES in the Use variable, it is not a valid comparison for numeric data types.

Note: This example does not handle leading zeros. If your data can contains leading zeros, you will have to modify the embedded source code to handle them.

How to Use Spin Controls for Date or Time Fields

Spin Controls are commonly used for date or time entry controls. Using a SPIN control gives the end user more flexibility, allowing data entry by typing or by clicking on the up or down buttons to increment or decrement the value.

In the Window Formatter:

1. Place a SPIN control on the window by clicking on the Spin icon in the Controls toolbox and then clicking on the desired position in the window.
2. RIGHT-CLICK on the spin control and choose Properties from the popup menu.
3. In the **Use** entry box on the **General** tab, type the field's label (or press the ellipsis (...) button to select the field from the **Select Field** dialog).
4. Select the **Extra** tab, and specify the **Step** value.

   This is the amount by which the value is incremented or decremented when the Spin Control's Up or Down button is pressed. For a Date, a step value of 1 increments or decrements by one day. For a Time, a step value of 6000 increments or decrements by one minute.
5. Optionally, provide an initial value for the fields.

   You can specify an initial value in the Data Dictionary or if you are using the Form procedure template or Save button control template, you can provide an Initial value by specifying it in the **Field Priming on Insert** dialog. To specify the current date, assign the TODAY() function to a date field. To specify the current time, assign the CLOCK() function to a time field.

   Tip: If you always want spin controls for these fields, specify a SPIN control as the default Window control in the Field Properties dialog in the Data Dictionary.

How to Create a Multi-Page Form

In the Window Formatter:

1. Place a SHEET control on the form window.

One TAB or page is automatically included in the SHEET. The SHEET structure declares a *group* of TAB controls that offer the user *multiple pages* of controls for a single window. The multiple TAB controls in the SHEET structure define the pages displayed to the user.

2. Place additional TAB controls on the SHEET as required.

The TAB structure declares a group of controls. This group is one of many pages of controls that may be contained within a SHEET structure. The SHEET structure's USE attribute receives the text of the TAB control selected by the user.

The Windows 95 standard to change from tab to tab is CTRL+TAB. Clarion TAB controls follow this standard, both in the development environment and in a compiled application.

3. Place controls on the tabs as required.

Required Fields on Tabbed Dialogs

The REQ attribute behaves differently for tabbed dialogs than for single page dialogs. Because the user has the option of never even selecting secondary tabs (pages), special steps are required to enforce entry of required fields that reside on secondary tabs.

Enforcing required entry fields on Tabbed Dialogs:

 Put all required fields on the first tab; add the REQ attribute to the tab and to the required entry fields.

 Make a (see also)"Wizard".

 Embed code at the beginning or end of the procedure that selects all tabs with required fields; add the REQ attribute to the required entry fields and to their parent tabs.

How to use Pattern Pictures on a form

The standard Windows behavior for an entry control is free-form entry. To override this behavior, you must add the MASK attribute to the WINDOW.

In the Window Formatter:

1. Make sure the Window is selected.
2. Choose **Edit** **Properties** (or press ENTER).

   The **Window Properties** dialog appears.
3. On the **Extra** tab, check the **Entry Patterns** box.
4. Press the **OK** button.
5. For each entry control in the window, add a Picture token to the **Picture** field of the **Entry Properties** dialog.

How to Implement a Splash Screen

A "splash screen" opens automatically when your application starts and provides an interesting, colorful, and even a musical, beginning; all of which is fun. Practically speaking, it provides the user with a certain sense of security and confidence that comes with familiarity. This can be especially true an application that is brand new to the user. A familiar picture or sound can get them off on the right foot. To implement a splash screen for your application:

1. Call the Splash Screen Procedure from the Main Application Frame.
2. Create Your Splash Screen Window.
3. Add an Image to Your Splash Screen.
4. Add an Alert Key so your user can exit with a mouse click.
5. Embed the code to end the Splash Screen on timer or mouse click.

See the example Video application shipped with this product as C:\CW15\EXAMPLES\APPS\VIDEO\VIDEO.APP

Call the Splash Screen Procedure from the Main Application Frame

1. From the Application Tree dialog, RIGHT-CLICK the **Main** procedure and choose **Embeds** from the popup menu.
2. Highlight the "After Opening the Window" embed point and press the **Insert** button.
3. From the **Select embed type** dialog, select **SOURCE**, type the following statement, then **Exit!** the Text Editor:

```
DISPLAY
```

The DISPLAY statement causes the Application Frame to be drawn before the splash screen. If want the splash screen to appear by itself, omit this step.

4. Highlight the embedded source you just added, and press the **Insert** button again.
5. This time, select **Call a procedure**, then type "SplashScreen" in the **After Opening the Window** dialog.

This creates a new "ToDo" procedure in the Application Tree entitled SplashScreen.

6. Press **OK**, then press the **Close** button to exit the **Embedded Source** dialog.

Create Your Splash Screen Window

1. From the Application Tree, highlight the SplashScreen procedure, and press the **Properties** button.

The **Select Procedure Type** dialog appears.

2. Clear the **Use Procedure Wizard** box, then select **Window - Generic Window Handler**.
3. From the **Procedure Properties** dialog, press the **Window** button.
4. From the **New Structure** dialog, highlight **Window** and press **OK**.
5. From the **Window Formatter**, choose **Edit** **Properties**.
6. From the **General** tab, clear the **Caption** field.
7. From the **Extra** tab, type 500 in the **Timer** field.

This generates an Event:Timer, every 5 seconds. We will use this event to close the splashscreen after 5 seconds.

8. From the **Position** tab, choose **Center** for the **X** and **Y** coordinates.

This will center our splash screen on the monitor.

Add an Image to Your Splash Screen

1. From the Window Formatter, choose **Control** **Image**.
2. CLICK anywhere in the Sample Window to place the IMAGE control, then drag it's handles so it fills the entire Sample Window.
3. RIGHT-CLICK the IMAGE control and choose **Properties** from the popup menu.
4. Press the ellipsis (...) button beside the File field to select a graphic (.BMP, .ICO, .JPG, .WMF, etc.) with the standard Open File dialog.

You may also add text, lines, etc. to the splash screen window.

5. Press the **OK** button to return to the **Window Formatter**.

Add an Alert Key so Your User can Exit with a Mouse Click

1. From the **Window Formatter**, select the window (press TAB) and choose **Edit** **Alert**.
2. From the **Alert Keys** dialog, press the **Add** button to add an alert key.
3. From the **Input Key** dialog, select **Left Button** and press **OK** twice.

An Alert key generates an Event:Alert whenever the specified key is pressed when the window has focus. We will use this event to close the splash screen when the user mouse clicks.

4. **Exit!** the **Window Formatter**, and save your changes.

Embed the Code to End the Splash Screen on Timer or Mouse click

1. From the **Procedure Properties** dialog, press the **Embeds** button.
2. Highlight **AlertKey** under the "**Window Event Handling, after generated code**" embed point, then press the **Insert** button.
3. From the **Select embed type** dialog, select **SOURCE**, type in the following statement, then **Exit**! the Text Editor.

```
DO ProcedureReturn
```

4. Highlight the source item you just embedded, the press the copy button (or CTRL+C).
5. Now highlight **Timer** under the "**Window Event Handling, after generated code**" embed point, then press the paste button (or ctrl+v).

This copies the DO ProdecureReturn statement from the AlertKey embed point to the Timer embed point, so the splash screen will end after the 5 second timer event, or when the user mouse clicks, whichever comes first.

How the Print Engine Processes Report Sections at Runtime

Before learning how to create a report using the Report Formatter, it's important to understand how Clarion executes a report--in other words, the division of labor between the print engine and your source code, and the order in which Clarion processes all the sections of your report. Each section of the report is a data structure, and each in turn is contained in the REPORT structure.

Smart Processing

The REPORT data structure contains all the information necessary for formatting and printing each page. It automatically handles page overflow management, including widow and orphan control. This frees you from worrying about the "mechanics."

This means that the Clarion executable code to print a report is simple and clean. The following example shows how a total of six lines of executable code can access the file and print a fully-formatted listing of all Customers. Since the Report Formatter writes the entire REPORT data structure, this is all the code the programmer has to write:

```
CODE
OPEN(CustReport)                        !Open report for processing
SET(Cus:NameKey)                          !Top of file, alpha order
LOOP                                        !Process the entire file
  NEXT(CustomerFile)                    !one record at a time
    IF ERRORCODE() THEN BREAK.            !Check for errors
  PRINT(Rpt:Detail)
                        !PRINT tells the REPORT structure to do the work.
END
```

Of course, if you're using the Application Generator, you don't even have to write that much! In the example above, the PRINT statement prints a DETAIL structure for each record in the file retrieved with the NEXT statement inside the LOOP .

The REPORT data structure contains the items that belong on the page, plus the attributes that determine how they appear there. Since you visually design these in the Report Formatter, the code example above really is all you have to do to print the report.

The PRINT statement automatically initiates the page overflow handling. This means that when the LOOP goes through enough records to fill up the page, it automatically generates any other structures on the page--the FOOTER, for example. Then it sends the entire page to the print spooler.

Order

The parts are wholly contained and managed within the REPORT data structure. The parts of the data structure are the FORM, PAGE, HEADER, DETAIL, and page FOOTER; their functions are fully described below. The REPORT data structure may also contain group BREAK structures. Each group BREAK structure can contain its own group HEADER, DETAIL, and FOOTER.

Normally, you want to design reports with only one DETAIL. When you generate reports using the Application Generator, they will probably have only one. This usually is the one inside the group BREAK structure. The Report Formatter adds a DETAIL for each BREAK, for flexibility. You can delete the ones not used.

Once you know the order in which the parts generate at print time, you can understand how to use them better. For the following example, assume a report utilizing all the parts listed above, containing one group

structure, with a DETAIL inside. Immediately upon the PRINT command:

1. The print engine composes the FORM, but does not send it to the print spooler yet. The FORM generates only once; the application repeats the FORM and does not recompose it for additional pages.
2. The print engine composes the page HEADER.
3. The print engine processes the group HEADER.
4. The application generates the DETAIL section (within the BREAK) for as many times as will fill the first page, continuously checking for group BREAKs.

If a BREAK **occurs on the page**:

5. The print engine composes the group FOOTER for the first group.
6. The print engine composes the group HEADER for the next group.
7. The application generates the DETAIL section for the next group of records, continuously checking for further group BREAKs.

At the bottom of the page:

8. The print engine checks for widows, increments the page count, and checks the next page for orphans.
9. The print engine composes the page FOOTER.
10. The print engine sends the entire first page to the print spooler.
11. For page two, since the FORM was composed already, it does not get regenerated, though it will print on the page. The application proceeds directly to the page HEADER.
12. The application repeats the procedures above for this and any additional pages.

Flexibility

The page-oriented nature of the Report Formatter is the key to its flexibility. The print engine composes each page in its entirety before sending it to the printer. This means you may arrange the parts of the report into any page layout you wish.

You can place the FORM, page HEADER, and page FOOTER structures anywhere on the page, within certain limitations. Their placement does not affect the order that the application generates the parts.

That means you can physically place a page FOOTER above a page HEADER. Since the FOOTER generates only after the report processes all the records on the page, this allows you, for example, to place a page total above the records on the page.

You set the position and size of the DETAIL structure as an offset vs. the last DETAIL printed. The print engine prints each DETAIL from page top to page bottom. If the DETAIL is narrow enough so that more than one fits across the width of the page, they print in order left to right, top to bottom.

Group BREAK structures--group HEADER, group DETAIL and group FOOTER--all print as offsets within the DETAIL area, one after the other.

You can do some fancy footwork in cooperation with the print engine. For example, because the DETAIL structure must be printed with the PRINT statement, you can utilize embedded source to place conditional statements within your executable code, to print one DETAIL upon one condition, and another upon a different condition.

As long as you remember the order in which the application generates each section, which determines

the current record and the values of the totals, tallies and other operations on the fields in each structure, you can build in a great deal of flexibility within the REPORT data structure, and let the print engine worry about fitting it all onto the page at runtime.

How to Use the Report Formatter - An Overview

Use the **Report Formatter** to visually design report elementsheaders, totals, detail lines, graphic images, preprinted forms, etc.on screen. The **Report Formatter** automatically generates the Clarion language source code that defines these elements.

The **Report Formatter** has six major components that help design your report: the (see also)Sample Report with Rulers and Grid Snap, the (see also)Controls Toolbox, the (see also)Fields Toolbox, the (see also)Property Toolbox, the (see also)Align Toolbox, and (see also)report Preview.

Using the Report Formatter - A Typical Procedure

The **Report Formatter** represents the four basic parts of the REPORT data structure by showing the page HEADER, DETAIL, FOOTER, and FORM as four "bands."

Here is the typical process for developing a report with the **Report Formatter**:

1. Specify the files and sort keys your report procedure will use.

See (see also)How to Sort Reports.

2. Establish the general layout of your report: (see also)paper size, page orientation, page margins, and band positions.

See (see also)Paper Size, (see also)How to Set Report Margins, (see also)Page Layout, (see also)Report Properties.

3. Place constants such as report titles and logos in the header band or the form band.

See (see also)Constant Strings, (see also)Graphic Images, (see also)Controls Toolbox.

4. Place variables such as page numbers and section headers in the header band.

See (see also)Constant Strings, (see also)Variable Strings, (see also)Controls Toolbox.

5. Place data dictionary fields from your data files in the detail band.

See (see also)Fields Toolbox, (see also)Controls Toolbox(see also)Align Toolbox, (see also)Report Formatter Menu Commands.

6. Set a group break or breaks, with variable headers, subtotals, etc.

See How to Set Report Group Breaks, How to Sort Reports.

7. (see also)**Preview**! your report.

8. Repeat any of the above steps as necessary, to fine tune your report.

9. **Exit**! the Report Formatter.

Using the Report Formatter - Page Layout

1. Choose **View**  **Page Layout View**.

2. Reposition the report bands by dragging their handles.

Bands may overlap, abut, or be separated by space.

How to Set Report Margins

The default margins for the detail print area are one inch from the left edge of the page, and two inches from the top. This setting leaves space for a HEADER at the top of the page. You specify the margins on the **Position** tab of the respective **Report Properties, Page/Group Header Properties, Detail Properties,** and **Page/Group Footer** dialogs. See (see also) Page Layout.

- To specify the left margin, enter a value in the **X pos** box.
- To specify the top margin, enter a value in the **Y pos** box.
- To specify the height, enter a value in the **Height** box.
- To specify the width, enter a value in the **Width** box.

These values set the AT attribute for the selected report structure.

The AT attribute on report structures performs two different functions, depending upon the structure on which it is placed.

When placed on a FORM, or **page** HEADER or FOOTER the AT attribute defines the position and size on the page at which the structure is printed. The position specified by the **x** and **y** parameters is relative to the top left corner of the page.

When placed on a DETAIL, or **group** HEADER or FOOTER the print structure is printed according to the following rules (unless the ABSOLUTE attribute is also present):

- The width and height parameters of the AT attribute specify the **minimum** print size of the structure.
- The structure is actually printed at the next available position within the detail print area (specified by the REPORT´s AT attribute).
- The position specified by the x and y parameters of the structure's AT attribute is an offset from the next available print position within the detail print area.
- The first print structure on the page is printed at the top left corner of the detail print area (at the offset specified by its AT attribute).
- Next and subsequent print structures are printed relative to the ending position of the previous print structure:

If there is room to print the next structure beside the previous structure, it is printed there.

If not, it is printed below the previous.

The values contained in the AT attribute's **x, y, width,** and **height** parameters default to dialog units unless the THOUS, MM, or POINTS attribute is also present. See the **Report Properties** dialog. Dialog units are defined as one-quarter the average character width by one-eighth the average character height. The size of a dialog unit is dependent upon the size of the default font for the report. This measurement is based on the font specified in the FONT attribute of the report, or the printer's default font.

Using the Report Formatter - Paper Size

1. Choose **Edit** **Report Properties**.

The **Report Properties** dialog appears.

2. Select the **Paper Size** tab.
3. Standard Sizes: choose from forty-one sizes in the **Paper Size** drop down list.

The list includes standard letter, legal, ledger, envelopes, and more.

4. Custom Sizes: choose **Other** from the **Paper Size** drop down list, then type your own width and height values.
5. Check the **Landscape** box to orient the report text parallel with the longest edges of the paper.

Using the Report Formatter - Constant Strings

1. Choose **Controls** **String**, or pick the **String** tool from the **Controls** toolbox.
2. CLICK in one of the report bands.
3. DOUBLE-CLICK on the string control you just placed.

The **String Properties** dialog appears.

4. Type the constant text**:** in the **Parameter** field, then press the **OK** button.

5. Resize the control so that it's wide enough to hold the text, by DRAGGING its right handle.

Using the Report Formatter - Variable Strings

Use variable strings to display data dictionary fields, Clarion total fields, such as Page No., Sum, Average, Count, etc., and your own calculated fields.

1. Choose **Controls** **String**, or pick the **String** tool from the **Controls** toolbox.
2. CLICK in one of the report bands (except the form band - forms cannot display variables).
3. DOUBLE-CLICK on the string control you just placed.
4. Check the **Variable String** box.
5. Type a (see also)picture token in the **Picture** field.

@n2 specifies a numeric picture for the control. @S10 specifies an alpha-numeric picture for the control.

6. For data dictionary fields or memory variables, press the **Use** field ellipsis (...) button to choose a field (or define a new field) from the **Select Field** dialog.

7. For Clarion total fields, simply choose a total type from the **Total Type** drop down list.

8. Press the **OK** button to close the **String Properties** dialog.

Using the Report Formatter - Graphic Images

1. Choose **Controls** **Image**, or pick the **Image** tool from the **Controls** toolbox.
2. CLICK in one of the report bands.
3. Resize the image control by DRAGGING its handles.
4. DOUBLE-CLICK on the image control you just placed.

The **Image Properties** dialog appears.

5. Press the **File** ellipsis (...) button to choose a graphic file from the **Select Image File** dialog.

Choose bitmaps (.BMP), metafiles (.WMF), icons (.ICO), jpeg (.JPG), etc.

How to Set Report Group Breaks

Group breaks provide a means of grouping report data into sections and optionally displaying subheadings, subtotals, or other information associated with the subgroup. Each group consists of a set of records, all sharing the same value in the BREAK field.

In order to generate meaningful groups, the records should be sorted in the same sequence as the BREAKs are declared. In other words, when you select a sort key for your report, the key will determine the variables on which you define your group breaks. See (see also)How to Sort Reports

You may also break on the common fields used to relate two files. File relationships are defined in the Data Dictionary's **Relationship Properties** dialog. Adding secondary files to your procedure gives you another logical field to break on: that is, the common field(s) linking the two files.

To create a group break**:**

1. Be sure the DETAIL band is visible; if not, press the restore button.

2. Choose **Bands** **Surrounding Break**.

3. When the cursor changes to a crosshair, CLICK in the DETAIL band.

The **Break Properties** dialog appears.

4. In the **Label** field, type a valid Clarion label to use as a name for the break.

5. In the **Variable** field, type the name of a variable to break on.

You can press the ellipsis (...) button to select a break field from the **Select Field** dialog.

6. Press the **OK** button.

This inserts the group BREAK. When the report prints, it groups together all records with the same value for the BREAK field, and prints any group HEADER and FOOTER defined for the break.

Tip: If the break variable is a global or local variable, you must be sure that the executable code updates its value, so that it can generate a group BREAK.

7. Choose **Bands** **Group Header** from the menu to define a group HEADER for the BREAK.

8. When the cursor changes to a crosshair, CLICK in the BREAK mini caption bar.

The **Page/Group Header Properties** dialog appears. Specify a field equate label and any special page breaking behavior. See (see also)How to Control Page Breaks.

9. Press the **OK** button.

This inserts the group HEADER band. You may place controls here just as in any other report band. Group footers are added similarly, using **Bands** **Group Footer** from the menu

How to Sort Reports

The sort sequence of a report is determined by a KEY or INDEX defined in the Data Dictionary's **Field/Key Definition** dialog. Keys or indexes are selected for use in this particular report procedure, using the **File Schematic Definition** dialog. When you select a sort key for your report, the key will determine the variables on which you define your group breaks. See(see also) How to Set Report Group Breaks.

For example, if you select the CUS:LastNameKey as your key, then your BREAK variables should be among those fields listed as components of the CUS:LastNameKey. You can see the key's component fields in the Data Dictionary's **Field/Key Definitions** dialog.

To specify the sort sequence for your report:

1. From the **Application Tree** dialog, DOUBLE-CLICK on the report procedure name.

The **Procedure Properties** dialog appears.

2. Press the **Files** button.

The **File Schematic Definition** dialog appears. Use this dialog to tell the Application Generator which files and keys your report procedure will access.

3. DOUBLE-CLICK the ToDo item for your procedure.

The **Insert File** dialog appears.

4. DOUBLE-CLICK the file you wish to report from.

The **File Schematic Definition** dialog reappears. You may specify more than one file to report on: a primary file, and secondary files. The secondary files must be related to the primary file by a common field. Adding secondary files to your procedure gives you another logical field to break on: that is, the common field(s) linking the two files.

5. Press the **Key** button, to specify which key is used for this report procedure.

The **Change Access Key** dialog appears.

6. DOUBLE-CLICK the key you want for this report.

The **File Schematic Definition** dialog reappears.

7. Press the **OK** button.

How to Control Page Breaks

One of the main considerations when laying out a report is unplanned page breaks. For example, you can't always predict how long or short a group will be. You therefore should plan on how your report will behave when it reaches the end of the page, yet there is still more data (in the group) to print.

There are several options available. Each controls what happens when a DETAIL section may be split at the end of a page.

To access the dialogs which allow you to set these options, DOUBLE-CLICK the DETAIL section. Alternatively, select either section and press the **Properties** button. This displays the **Detail Properties** dialog, containing the options below.

You may also set these options for group headers or footers. The options are available in the **Group Header Properties** and **Group Footer Properties** dialogs.

PAGEAFTER

To print the DETAIL, then force a new page, type a value in the **Page after** box in the **Detail Properties** dialog. This sets the PAGEAFTER attribute. This prints the DETAIL, then prints the page FOOTER, then begins a new page.

Tip: To print a separate page for each record, place the variable strings and/or controls you wish in the DETAIL, and specify the PAGEAFTER attribute in the Detail Properties dialog.

The page number automatically increments, unless you reset it. To reset the page number to a value you specify, type it in the **Page after** field in the **Detail Properties** dialog.

PAGEBEFORE

To print the DETAIL structure on a new page, type a value in the **Page before** box in the **Detail Properties** dialog. This sets the PAGEBEFORE attribute. The report prints the full DETAIL starting at the top of the next page. The report FOOTER, however, prints on the first page.

The page number automatically increments, unless you reset it. To reset the page number to a value you specify, type it in the **Page before** field in the **Detail Properties** dialog.

WITHNEXT

To prevent 'widow' elements in a printout, type a value in the **Keep next** field in the **Detail Properties** dialog. This sets the WITHNEXT attribute. A 'widowed' print element is one which prints, but then is separated from the succeeding elements by a page break.

The value specifies the number of succeeding elements to print--a value of '1,' for examples, specifies that the next element must print on the same page, else page overflow puts them on the next.

WITHPRIOR

To prevent 'orphan' elements in a printout, type a value in the **Keep prior** field in the **Detail Properties** dialog. This sets the WITHPRIOR attribute. An 'orphaned' print element is one which prints on a following page, separated from its related items.

The value specifies the number of preceding elements to print--a value of "1," for example, specifies that the previous element must print on the same page.

Tip: When placing subtotals or totals in a DETAIL, use the WITHPRIOR attribute to insure they print with at least one member of the column above it when a page break occurs.

Using the Report Formatter - Sample Reports

The **Report Formatter** is a visual design tool. In **Band View**, you always see a sample of the report band you're working on, as you work on it. For example, place a list box in the detail band and drag its handles to the size you want.

Switch to (see also)**Page Layout View** to resize and reposition the report bands. Drag the band handles or drag the entire band. All bands appear together on a representation of the page.

In addition, you can quickly generate filler data and see a sample report by choosing (see also)**Preview!** from the action bar, all without actually compiling or running the report.

Using the Report Formatter - Controls Toolbox

The **Report Formatter** contains a floating **Controls** toolbox, similar to the **Window Formatter**. Simply choose a control from the toolbox (CLICK on it), then CLICK in the report band to place the control in the report.

Display or hide the **Controls** toolbox by choosing **Options** **Toolbox.** Resize the **Controls** toolbox by placing the cursor on the border of the box. When the cursor changes to a double headed arrow, CLICK and DRAG. All the controls in the toolbox are also available from the **Controls** menu.

See also: the (see also)**Controls** menu, (see also)**Fields** toolbox, (see also)**Align** toolbox, (see also)**Property** toolbox, (see also)Using the Report Formatter - An Overview

Tip: Position the cursor over any tool and wait for half a second. A tool tip appears telling you the type of control that will be created by this tool.

String	Allows you to place STRING control on the report under construction. See the **String Properties** dialog.
Text Box	Allows you to place TEXT control on the report under construction. See the **Text Properties** dialog.
Group Box	Allows you to place GROUP control (group box) on the report under construction. See the **Group Properties** dialog.
Option Box	Allows you to place OPTION control (OPTION structure, which appears as a group box with radio buttons) on the report under construction. See the **Option Properties** dialog.
Check Box	Allows you to place CHECKBOX control on the report under construction. See the **Check Box Properties** dialog.
Radio Button	Allows you to place RADIO control on the report under construction. See the **Radio Properties** dialog.
List Box	Allows you to place LIST control (list box, or drop down list box) on the report under construction. See also: **List Box Formatter**. See the **List Properties** dialog.
Image	Allows you to place IMAGE control (graphic image) on the report under construction. See the **Image Properties** dialog.
Line	Allows you to place LINE control on the report under construction. See the **Line Properties** dialog.
Rectangle	Allows you to place a BOX control on the report under construction. See the **Box Properties** dialog.
Ellipse	Allows you to place ELLIPSE control on the report under construction. See the **Ellipse Properties** dialog.
Dictionary Field	Allows you to select a field defined in the Data Dictionary, and place the control specified in the data dictionary, plus an associated PROMPT control, on the report under construction. See the **Select Field** dialog.

Custom Control	Allows you to place a CUSTOM control (Visual Basic custom control) on the report under construction. . See the **Custom Control Properties** dialog.
Control Template	Allows you to place Control Template on the report under construction. See the *** dialog.

Using the Report Formatter - Fields Toolbox

The **Report Formatter** contains a floating **Populate Field** toolbox. This toolbox allows you to quickly "populate" a window with entry controls and prompts for fields in your data dictionary files.

Display or hide the **Populate Field** toolbox by choosing **Options**  **Fieldbox.** Resize the **Populate Field** toolbox by placing the cursor on the border of the box. When the cursor changes to a double headed arrow, CLICK and DRAG.

See also: the (see also)**Populate** menu, (see also)**Controls** toolbox.

1. Choose a **file** from the drop down list.
2. CLICK on the **field** you want on your report.
3. CLICK in the report band to place the control.

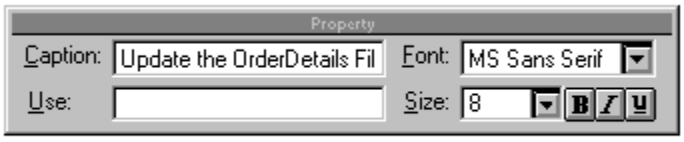
The cursor becomes a crosshair. The top left corner of the control is placed at the intersection of the cursor crosshair.

The **type** of control (text box, check box, radio button, etc.) is determined by the settings for this particular field in the Data Dictionary (see also)**Field Properties** dialog.

Using the Report Formatter - Property Toolbox

The **Report Formatter's Property** toolbox allows you to quickly specify the appearance and content of the text on each in the report. Control the font, size, style, and content of all your text, using standard word processor buttons and drop down lists. (see also)**Select Font** dialog.

Display or hide the **Property** toolbox by choosing **Options** **Propertybox.** Resize the **Property** toolbox by placing the cursor on the border of the box. When the cursor changes to a double headed arrow, CLICK and DRAG.



Text Formatting: choose content, font, size, and style for the selected control or window.

Using the Report Formatter - Align Toolbox

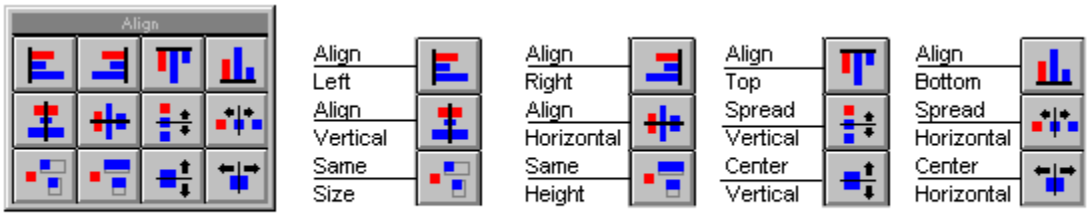
The **Report Formatter's Align** toolbox allows you to quickly, professionally, and precisely align the controls in your report.

Display or hide the **Align** toolbox by choosing **Options** **Alignbox.** Resize the **Align** toolbox by placing the cursor on the border of the box. When the cursor changes to a double headed arrow, CLICK and DRAG.

Select the controls to align (CTRL+CLICK allows you to select multiple controls, or you can "lasso" multiple controls with CTRL+DRAG), then click on the appropriate alignment tool. All the alignment actions are also available from the (see also)**Alignment** menu.

Tip: **For most alignment functions, the first controls selected (blue handles) are aligned with the last control selected (red handles). That is, the last control selected is the anchor control. It doesn't move, the others do.**

Tip: **Position the cursor over any tool and wait for half a second. A tool tip appears telling you the type of alignment this tool will accomplish.**



Align Left	Aligns the left borders of the selected controls with the left border of the last control selected (red handles).
Align Right	Aligns the right borders of the selected controls with the right border of the last control selected (red handles).
Align Top	Aligns the top borders of the selected controls with the top border of the last control selected (red handles).
Align Bottom	Aligns the bottom borders of the selected controls with the bottom border of the last control selected (red handles).
Align Vertical	Along a vertical axis, aligns the centers of the selected controls with the center of the last control selected (red handles).
Align Horizontal	Along a horizontal axis, aligns the centers of the selected controls with the center of the last control selected (red handles).
Spread Vertical	Equalizes the vertical spaces between the selected controls.
Spread Horizontal	Equalizes the horizontal spaces between the selected controls.
Same Size	Makes all selected controls the same height and width as the last control selected (red handles).
Same Height	Makes all selected controls the same height as the last control selected (red

handles).

Center Vertical	As a group (relative positions of selected controls don't change), centers the selected controls horizontally within the band.
Center Horizontal	As a group (relative positions of selected controls don't change), centers the selected controls vertically within the band.

Using Preview!

You can quickly "preview" alternative layouts for DETAILs, HEADERs, and FOOTERs. Fonts, sizes, colors, and positions of report controls are all displayed, and you can see these effects all without actually compiling or running your report.

In the Report Formatter:

1. Choose **Preview!** to "visualize" how the printed page will appear.

The **Preview Print Details** dialog appears. This dialog lets you generate "filler" data for your report. The data have no values, but serve as placeholders, so you can get a feel for the appearance of your finished report.

2. Highlight a detail (if you have more than one) in the **Details** list then press the **Add** button.

Each press of the **Add** button populates the preview with a filler record. Add one record for a one-record-per-page type report, or add lots of records to see the effects of the page breaking, group breaking, and header and footer options you have selected.

3. Press the **OK** button.

The Report Formatter generates a preview of your report including DETAILs, HEADERs, FOOTERs, BREAKs, fonts, sizes, colors, and positions of report controls.

4. When done "previewing," choose **Band View!** to continue editing your report.

How to Print to a File

You can change the windows default printer without calling the printerdialog function. This can be done by using Clarion's property syntax. The property to use is **PROPPRINT:Device**. This property definition can be found in the CW15\LIBSRC\PRNPROP.CLW. This must be included in your application before making use of any of the properties defined therein.

To include PRNPROP.CLW:

1. Press the **Global** button on the Application Tree, to open the **Global Properties** dialog.
2. Press the **Embeds** button.
3. Select the 'Inside Global Map' embed point and add the following embedded source code:

```
include('prnprop.clw')
```

To change the printer device:

1. From the Application Tree, select your report procedure and press the **Properties** button.
2. Press the **Embeds** button.
3. Select the 'Before Opening Report' embed point and add the following embedded source code:

```
sav::printer = PRINTER{PROPRINT:Device}       !   save windows default printer
PRINTER{PROPRINT:Device} = 'Generic / Text Only'     ! set to ASCII device
          PRINTER{PROPRINT:PrintToFile} = TRUE     ! print to file flag
PRINTER{PROPRINT:PrintTo Name} = 'REPORT.TXT'   ! set filename for report
```

At the end of the report, restore the original default printer:

1. From the Application Tree, select your report procedure and press the **Properties** button.
2. Press the **Embeds** button.
3. Select the 'After Closing Report' embed point and add the following embedded source code:

```
PRINTER{PROPRINT:Device} = sav::printer
PRINTER{PROPRINT:PrintToFile} = FALSE     ! print to file flag
```

How to change the printer device without calling PRINTERDIALOG

You can change the windows default printer without calling the printerdialog function. You may want to do this when a report is designed for pre-printed forms, and therefore must always be routed to a printer loaded with the forms.

This can be done by using Clarion's property syntax. The property to use is **PROPPRINT:Device**. This property definition can be found in the CW15\LIBSRC\PRNPROP.CLW. This must be included in your application before making use of any of the properties defined therein.

To include PRNPROP.CLW:

1. Press the **Global** button on the Application Tree, to open the **Global Properties** dialog.
2. Press the **Embeds** button.
3. Select the 'Inside Global Map' embed point and add the following embedded source code:

```
include('prnprop.clw')
```

To change the printer device:

1. From the Application Tree, select your report procedure and press the **Properties** button.
2. Press the **Embeds** button.
3. Select the 'Before Opening Report' embed point and add the following embedded source code:

```
sav::printer = PRINTER{PROPRINT:Device}       ! save windows default printer
PRINTER{PROPRINT:Device} = 'HP Laserjet Series II'    ! set new default printer
```

The device property takes the name of the printer device. This can be found by looking in windows print manager. The device is the actual printer name. The device property string is case insensitive.

At the end of the report, restore the original default printer:

1. From the Application Tree, select your report procedure and press the **Properties** button.
2. Press the **Embeds** button.
3. Select the 'After Closing Report' embed point and add the following embedded source code:

```
PRINTER{PROPRINT:Device} = sav::printer
```

How to Print Labels

Printing labels simply means printing a multi-column report, that is, getting the report rows and columns to match up with the commercial label forms you use.

1. With a ruler, measure the height and width of the label paper, measure the height and width of the individual labels, and measure any top or left margins on your label paper. Make your measurements in inches.

2. Create a report of your address file.

Use the Report Wizard if you want, but don't worry about formatting yet. Just make sure the report contains all the address fields you need for your labels.

3. From the Application Tree, RIGHT-CLICK your report procedure and choose Report from the popup menu.

4. Delete all report sections, except the Detail section.

5. Choose **Edit**  **Report Properties**.

6. On the **General** tab, choose 1/1000 inches in the **Units** drop down list.

7. On the **Postion** tab, set the margins you measured earlier.

In the **Top Left Corner** group, the **X** field represents the left margin in thousandths of and the **Y** field represents the top margin in thousandths of inches. So, if the left margin of your label paper is 1/2 inch, type 500 in the **X** field. If the margin is zero, type zero in the **X** field. Do the same for the top margin and the **Y** field.

8. On the **Postion** tab, set the height and width of the label paper.

In the **Width** group, click on fixed, and type the paper width in thousandths of inches. Standard letter size paper is 8 1/2 inches, so type 8500. Do the same for the paper height. Standard letter size paper is 11 inches, so type 11000.

9. Press the **OK** button to return to the **Report Formatter**.

10. Arrange your address fields in a vertical format, that is, one field below another near the left margin.

Use the (see also)Alignment tools for precise alignment.

11. RIGHT CLICK the detail band and choose Position from the popup menu.

12. On the **Postion** tab, set the height and width of the individual labels.

In the **Width** group, click on fixed, and type the label width in thousandths of inches (eg for a 2 1/2 inch label, type 2500). Do the same for the label height.

13. Press the **OK** button to return to the **Report Formatter**.

14. Readjust the position, size, and font of your address fields if necessary.

15. (see also)**Preview**! your label report.

16. **Exit**! the **Report Formatter** and save your changes.

How to Link External Resources

The Project System allows you to specify external resources to link into the executable. These include graphics, such as .BMP, .ICO and .WMF files. By linking them into the executable, you can avoid having to ship them as separate, external files.

If you directly reference a graphic file within a data structure, the compiler automatically links the graphic, so there is no need to add the graphic file to your Project Tree. For example, if you place an IMAGE control in a window, and specify a file by name in the **Image Properties** dialog, the linker automatically includes that file in your executable. But if you assign a different graphic to a control using a runtime property assignment statement, the linker will only include the new file in your executable if you add the file to your Project Tree.

To add graphic files to the executable:

1. Highlight **Library and object files** and CLICK on the **Add File** button.

Select the bitmap, icon, or metafile graphic from the standard Open File dialog.

2. Press the **OK** button to return to the **Project Editor** dialog.

3. Highlight the source code file that references the graphic, and CLICK on the **Edit** button.

The source code file is opened by the Text Editor.

4. Place a tilde (~) in front of the graphic file name in the source code assignment statement (not in data section).

For example: change ?Image{PROP:Text} = 'I.ICO' to ?Image{PROP:Text} = '~I.ICO.' The tilde indicates the program should find the item as a linked in resource, not as an external file.

Optionally, choose **Search** **Find** to locate the file name.

5. Choose **File** **Exit**, then CLICK on **Yes** when asked if you want to save.

Now, when you recompile and link, the executable will no longer require the external graphic file.

How to Manage Threads

This topic will show you how to limit an MDI Child browse procedure to a single instance using messaging.

The simplest solution is to disable the menu item when the browse procedure is active. You can't do this in the menu itself--you must send a message to the Frame procedure from the browse.

First you need to declare two new events in the Global Properties, Global Data embed point:

```
EVENT:DisableCustomerItem       EQUATE(401h)
EVENT:EnableCustomerItem        EQUATE(402h)
```

(Note that user-defined events must start after 400h.)

You also need a global variable, define this in **Global Properties, Data.** Call this **GLO:MainThreadNo**, make it a BYTE.

In the Frame procedure, Setup procedure embed, type the following:

```
  GLO:MainThreadNo = THREAD()
```

(Whenever an MDI procedure is STARTed, a thread number is allocated to it. We need the thread number in order to post a message to the frame procedure.)

Now, still in the frame procedure, you need to write code in the **Case EVENT() structure, before generated code** embed to handle the two user-defined events that the frame procedure will receive.

```
  OF EVENT:DisableCustomerItem
    DISABLE(?ShowCust)
  OF EVENT:EnableCustomerItem
    ENABLE(?ShowCust)
```

In the Browse procedure, in the **Initialize the Procedure** embed, type:

```
  POST(EVENT:DisableCustomerItem,,GLO:MainThreadNo)
```

In the **End of Procedure** embed, type:

```
  POST(EVENT:EnableCustomerItem,,GLO:MainThreadNo)
```

Now, the first time you select the Browse Procedure from your menu, ShowCust starts up, the POST() statement executes and an **EVENT:DisableCustomerItem** is sent to the Frame procedure.

If the user then clicks on the menu again, the message is processed and the item is disabled.

As the user exits ShowCust, the **EVENT:EnableCustomerItem** message is sent to the Frame. When that message is processed the menu item is enabled again.

Why store the Frame's thread number - surely it would always be number 1? Well it might NOT be the first thread in the application.

Thanks to Rob Mousley of Chariot Software for submitting this topic.

Using Windows DLLs NOT Created in Clarion for Windows

You can use Windows .DLLs which have not been created with Clarion for Windows if you know the prototypes for the .DLLs procedures and functions.

If the source language prototypes are known, then equivalent Clarion prototypes must be created and included in a CW program's MAP for all referenced DLL procedures and functions. Also, Clarion for Windows 1.0 requires a Library (.LIB) file in the Project Tree under **Library and Object files**. This Library file entry enables the linker to resolve the procedure and function references in the .DLL.

If you have a Windows DLL (not created with Clarion for Windows) that you want to use in a CW program, then the following steps are required to enable the CW program to access the DLL's procedures and functions:

 Create Equivalent Clarion for Windows Language Prototypes.

 Create a Clarion for Windows Library (.LIB) File for the DLL.

 Reference the Library (.LIB) File in the Project System.

Create Equivalent Clarion for Windows Language Prototypes.

Prototypes for any referenced .DLL procedures and functions must be in the CW program's MAP structure. Procedures and functions written in a language other than Clarion can still be referenced in a Clarion program by creating an equivalent Clarion prototype. The prototypes are placed in a MODULE structure which identifies the name of the DLL's library as the MODULE parameter. For example, if the DLL name is MY.DLL then the module structure would be:

```
MODULE('MY.LIB')
```

There are several issues to consider when creating equivalent prototypes in Clarion which are dependent upon a DLL's source code language. A primary consideration is relating equivalent data types in the other language to Clarion. Equivalent data types can be determined by considering the "underlying" machine data type represented by each language data type. For example, the CW Language Reference identifies the Clarion data type SREAL as a "four-byte signed floating point".

The following is an example of C and C++ code data type equivalents.

```
unsigned char       ==> BYTE
short               ==> SHORT
unsigned short      ==> USHORT
long                ==> LONG
unsigned long       ==> ULONG
float               ==> SREAL
double              ==> REAL
```

A Clarion GROUP is roughly equivalent to a C or C++ *struct*. For example:

```
Struct1         GROUP               ! Struct1 is defined as a GROUP
ul1               ULONG             ! containing two ULONG values
ul2               ULONG
                END
```

This form of definition reserves space for Struct1 and is equivalent to the C definition:

```
struct {
  unsigned long  ul1;
  unsigned long  ul2;
} Struct1;
```

A second important prototyping consideration is the procedure/ function calling convention utilized by

another language. Clarion provides support for three different calling conventions: PASCAL, C, and TopSpeeds Register Based. See Function and Procedure Prototypes. in the Language Reference for further information on Clarion prototypes.

Create a Clarion for Windows Library .LIB File for the DLL.

You can create a .LIB file for the DLL using the LIBMAKER.EXE utility program that comes with Clarion for Windows as one of the example programs. Simply run the program, select the .DLL and have it automatically create the TopSpeed format .LIB file for you.

Reference the .LIB File in the Project System.

You must place the Library (.LIB) file in the Project Tree under **Library and Object files** for any Project that will use the .DLL.

Creating a .DLL (Sub-Application)

This section describes the steps to create a program using one main application and several sub-applications compiled and linked as external.DLLs. It is written with Team Development in mind, so it describes some of the aspects of working in a multi-developer environment. Dividing a large project into multiple .DLLs provides many benefits:

- Each sub-application can be modified and tested independently.
- Developers can work on their portion of the project without interfering with others on the development team.
- Each sub-application can be compiled as a DLL and tested in the main program without recompiling the entire project. This reduces compile and link time.
- Dynamic Pool Limits are avoided in large projects.
- Future updates can be deployed by shipping a new .DLL, reducing shipping costs.

With this approach, each Team Member creates a separate .DLL that is called by a "master" application. This requires splitting the application into a "Main" executable and "secondary" .DLLs. The individual team members maintain separate application files for each component. The Team Leader creates a master application that calls the sub-applications and a "data" application that contains (and exports) all the File definitions and Global variables. Optionally, members can call procedures from another member's .DLL.

This method also requires extensive pre-planning of the "division of labor" between the various target files created by the application.

The following outlines a possible implementation of this strategy:

1. Create the data dictionary and set up the workstations as described above.

2. Create a "dummy" application to store and export all data declarations. All Global variables or structures and all file definitions are defined (and exported) in this application. Use the following settings:

In the Application Properties:

Dictionary File: The master dictionary residing on the network.

Target Type: DLL

In the Application's Global Properties:

Generate Global Data as External: OFF

File Control Flags

Generate All File declarations: ON

External: NONE EXTERNAL

Export All File declarations: ON

3. Team members create their own sub-application .APP files, specifying the dictionary file on the network as the data dictionary, and a directory on the local drive as the default directory for the .APP file. Each team member specifies a different target file using the following settings:

In the Application Properties:

Dictionary File: The master dictionary residing on the network.

Target Type: **EXE** during the design and testing phase
DLL when releasing to the master directory.

Note: Changing the Target Type enables procedures to be exported. Make sure that every procedure that is called by the master application or another .DLL has the Export Procedure check box in the Procedure Properties checked (the check box is only available after changing the target type).

In the Application's Global Properties:

Generate Global Data as External: ON

File Control Flags

Generate All File declarations: OFF

External: ALL EXTERNAL

All Files declared in another .App: ON

Declaring Module: Leave this blank

In the Application's Module Tree:

Choose **Application** **Insert Module,** select External DLL, then select the corresponding .LIB for the .DLL containing the data definitions.

One particular .APP creates the executable which launches or calls library functions or procedures in the others. To the end user, this is the .EXE program to start when working with the complete application.

4. Team members synchronize their local directory with an equivalent on the network at the end of each day.

5. Team Members release their compiled and linked .DLLs to the Team Leader.

Each sub-application has a "dummy" frame (not exported) that calls the sub-application's procedures so the Team Member can test the sub-application by compiling it as an .EXE. Once it passes testing, the member compiles it to a .DLL by changing the Application Properties' Target File type to .DLL and releases the file to the Team Leader.

Tip: If you edit the Redirection file to include "." at the start of the *.DLL and *.LIB search paths, Clarion will generate the *.DLL and *.LIB files into the local sub-application subdirectory instead of \CW15\BIN and \CW15\OBJ. This is a little safety precaution that prevents the *.DLL and *.LIB from getting into other Team Members' hands before it's ready. In addition, adding the Master directory to the end of these search paths enables the sub-application or main application to find the completed LIB's and DLL's belonging to other sub-apps in the master subdirectory.

6. The Team Leader copies the released .DLLs into the master directory and creates a master .APP file which calls the entry point procedures in the .DLLs.

The Master .APP is typically just a bare bones application with just a splash screen and a main frame with a menu and toolbar. The .DLLs are called at runtime so you don't need to compile a large Master .EXE. The Master .APP should have the same settings as the sub-applications except that it is always compiled as an .EXE.

The master .APP should have these settings:

In the Application Properties:

Dictionary File: The master dictionary residing on the network.

Target Type: **EXE**

In the Application's Global Properties:

Generate Global Data as External: ON

File Control Flags

Generate All File declarations: OFF

External: ALL EXTERNAL

All Files declared in another .App: ON

Declaring Module: Leave this blank

In the Application's Module Tree:

Choose **Application** **Insert Module,** Select External DLL, then select the corresponding .LIB for the .DLL containing the data definitions.

Choose **Application** **Insert Module,** Select External DLL, then select the corresponding .LIB for the sub-application .DLL. Repeat this step for each sub-application.

For each procedure the main application calls, edit the ToDo procedure as follows:

Template: External template.

Module name: Select the corresponding .LIB for the DLL drop down list.

If necessary delete any empty generated modules.

7. The Team Leader compiles the master .APP and tests the calls to the DLLs.

8. The Team leader repeats the last step on a periodic basis until all work by all developers is complete, and the entire application can be tested.

How to Import an ODBC File Definition

The Dictionary Editor allows you to quickly add a data file to the dictionary by creating a data definition based on an existing data file.

With the Dictionary dialog active:

1. Select **File** **Import File**. Select a data file from the **Import File Definition** dialog.

The **Import File** Dialog displays a list of installed database drivers.

2. Select the ODBC driver from the dropdown list, and press the **OK** button.

The **Data Sources** dialog appears. This is similar to the ODBC Administrator dialog.

3. Highlight the Data Source and press the Select button.

If the ODBC data file contains multiple tables, the **Tables for...** dialog appears.

4. Select the Table to import, then press the **Select** button.

The **File Properties** dialog appears.

5. Edit the **File Properties**, as needed, then press the **Ok** button.

Tip: **If the file system you are using supports multi-threading, and you want to use multi-threading with the file, you must check the Threaded box to add the THREAD attribute to the file definition. Consult your file system's documentation to see if multi-threading is supported.**

How to Create a File Definition for an ODBC Data Source

Adding ODBC support to your application only requires choosing Clarion's ODBC driver and providing the parameters to pass to the ODBC driver manager. You provide the parameters in the OWNER and NAME attributes of the FILE declaration.

You may also import the file definition into your data dictionary. See How to Import an ODBC File Definition for more information.

The following introduces the basics, as approached from the Data Dictionary Editor. Of course, you must also be sure that the field data types in your dictionary match the variable formats supported by the DBMS you're connecting to.

1. Create a new Dictionary file.
2. Choose **File** **Import File**.

The **Select File Driver** dialog appears.

3. Select **ODBC** from the drop down list, then press the **OK** button.

The **Data Sources** dialog appears. This is similar to the ODBC Administrator's interface. If the data source has not yet been defined, you can add it by pressing the **New** button.

4. Highlight the desired Data Source, then press the **Next** button.
5. If the Data Source has password protection, the Logon dialog appears. Provide the User ID and password, then press the **OK** button.

If the file contains multiple tables, the **Tables for ...** dialog appears.

6. Highlight the desired table, then press the **Finish** button.

The file definition is imported and the **File Properties** dialog appears allowing you to modify attributes, if you choose.

Notice the fields in the **File Properties** dialog that have been filled in during the import:

Name: This is extracted from the Table name as defined in the ODBC database. You may modify this, if desired. It is used as the Clarion label in your source code.

Prefix: This defaults to the first three characters of the table name, you may modify this if desired.

Owner Name The ODBC data source name, and optionally, the user ID, and password, separated by commas. Some databases require additional connection information. This information follows the password and is separated by semicolons, using the syntax: *keyword=value;keyword=value*.

For example, when accessing a Sybase database, this would appear as :

A Data Src,UserID,PassWord,DATABASE=DataBaseName; APP=APPName

The data source name is the section name in the ODBC.INI file which stores all the information necessary for the ODBC manager to load the driver and access the data. The Application Generator will add the information to the OWNER attribute of the file declaration:

OWNER(DataSourceName, UserID, Password)

Full Pathname: The import process places the *table name only* in this field. The ODBC driver retrieves the physical file name from ODBC.INI.

This places the file or table name in the NAME attribute of the file declaration:

NAME(DataFileName) or NAME(TableName)

The remainder of the attributes depend on your preferences and your application.

7. Repeat the last six steps for each table in the database.

How to Choose When to Use ODBC vs. a Native Clarion Driver

ODBC offers both pros and cons.

Using ODBC offers the following advantages:

- ODBC is an excellent choice in a Client-Server environment, especially if the Server is a native SQL DBMS. It allows you to add Client-Server support to your application, without having to do much more than choose a file driver. ODBC was specifically designed to create a non-vendor specific (if you exclude Microsoft) method of connecting front end applications to back end services. Via ODBC, the Server can handle much of the work, especially for SQL JOIN and PROJECT, speeding up your application.
- Existing ODBC drivers cover a great many types of databases. There are ODBC drivers available for databases for which Clarion may not have a native driver--for example, for Microsoft Excel and Lotus Notes files.
- ODBC is already widespread. Major application suites such as Microsoft Office install ODBC drivers for file formats such as dBase and Microsoft Access.
- ODBC is platform independent. One of Microsoft's prime objectives in establishing the ODBC API was to support easier access to legacy systems, or corporate environments where data resides on diverse platforms or multiple DBMS's. As long as an ODBC driver and back end are available, it doesn't matter whether you use Microsoft's NetBEUI, SPX/IPX, DECNet or others; your application can connect to the DBMS and access the data.

Given that there are many drivers available, and that the standard was developed by the company that developed Windows, you might consider using ODBC as the driver of choice for **all** your Windows applications. Yet, when choosing when to use an ODBC driver vs. a Clarion for Windows native database driver, you must also consider possible disadvantages:

- Unfortunately, ODBC adds a layer--the ODBC Driver Manager--between your application and the database. When it comes to accessing files on a local hard drive, this generally results in slower performance. The driver manager must translate the application's ODBC API call to an SQL statement before any data access.
- The information required by the ODBC database manager to connect to a data source varies from one ODBC driver to another. Unlike the selection of Clarion file drivers, where file operations are virtually transparent, you may need to do some work to gather the information required to use a particular ODBC driver. This topic provides a few tips that might make it easier, and many ODBC drivers come with a .HLP file which documents special settings (usually stored in ODBC.INI); but the burden is on **you** to solve any problems with third party ODBC drivers.
- ODBC is not included with Windows 3.1. When distributing your application, you'll need to install the ODBC drivers and the ODBC driver manager into the end user's system, if the end user doesn't have them already. This requires the ODBC SDK from Microsoft. In some cases, the back end server may have already provided a distribution kit which installs the ODBC driver on the workstation.
- The normal Microsoft setup program that installs the ODBC driver manager adds an applet to the end user's Control Panel window for managing ODBC. It's very easy for an end user to use this tool to change the settings in the ODBC.INI file. The end user can unwittingly remove the back end ODBC driver which would make it impossible for your application to connect to the data file. Additionally, since most ODBC drivers store the data directory in ODBC.INI, it's very easy for the end user to change it, again introducing a possible problem for your application.

Given the pros and cons, we recommend using native Clarion for Windows file drivers when both a native driver and an ODBC exist for the same file format.

How to Work With the ODBC.INI File

The data source listing in the ODBC.INI contains varying information according to what each driver needs to know to connect to the data. Some of the more important items:

- The file name of the actual ODBC driver; your application usually does not need to know this. The ODBC driver manager uses this information. Only this item is required.
- The data directory; or in the case of a file format that can store more than one table in a physical file, such as Microsoft Access, the name of the data file.
- A User ID.
- A Password; though it would be surprising to find a driver which stores this in an ASCII file!

You can **optionally** specify variable strings to hold the information the ODBC driver requires, then fill them in at run time. This allows you to check the ODBC.INI file to verify the information you need, protecting your application should the end user modify some of the settings, such as the data directory.

To specify variable strings for the OWNER and NAME attributes in the data dictionary, place an exclamation point (!) before the variable name as you type it in the **Owner Name** or **Full Pathname** fields in the **New File Properties** dialog; for example, !MyOwnerAttrib and !MyNameAttrib. When creating your application, the variables **must** be declared globally.

Using the GETINI function, you can then build up the proper OWNER and NAME attributes, then place them in the "Before Opening the Window" embed point.

For example, assume that the data source name is "MS Excel Databases." Assume that you've checked and found that the windows directory on the end user's machine is "C:\WINDOWS." Finally, assume you data file is named 'MyData.XLS.' Using the variables we "named" above, and two others, namely, MyDataDir, UserID, and MyPassword, in which you can temporarily store the data directory, UserID, and Password, the following would construct the attributes:

```
MyUserID = GetIni('MS Excel Databases', 'UID', '' , 'c:\windows\odbc.ini')
MyPassword = GetIni('MS Excel Databases', 'PWD', '' , 'c:\windows\odbc.ini')
MyOwnerAttrib = 'MS Excel Databases,' & CLIP(MyUserID ) & ',' & CLIP(MyPassword )
MyDataDir=GetIni('MS Excel Databases', 'DataDir', '' , 'c:\windows\odbc.ini')
MyNameAttrib = CLIP(MyDataDir ) & '\MyData.XLS'
```

- If the GETINI function fails to find entries for the UserID (UID) or Password (PWD) in the ODBC.INI file, it returns an empty string. In fact, the Excel driver included in the Microsoft SIMBA.DLL does not maintain these entries. They are in this example to illustrate how to successfully use them for driver which do supply the entries.

For the MS Access driver, which stores the physical file name in ODBC.INI, you have to know the table name, and place it in the NAME attribute. For example, assume the data source name "MS Access Databases," which contains a table named "MyTable." Since the ODBC driver manager can look up the physical file name, you don't even have to include it. Most likely, you would **not** even use a global variable to store the table name. You would type "MyTable" directly into the **Full Pathname** field in the **New File Properties** dialog. You could then use the following to fill in the OWNER attribute:

```
MyUserID = GetIni('MS Access Databases', 'UID', '' , 'c:\windows\odbc.ini')
MyPassword = GetIni('MS Access Databases', 'PWD', '' , 'c:\windows\odbc.ini')
MyOwnerAttrib = 'MS Access Databases,' & CLIP(MyUserID ) & ',' & CLIP(MyPassword )
```

ODBC.INI Section Format

The standard format for the ODBC data source entry varies from "back-end" driver to driver. It's essential to have the individual driver help file if you plan to edit the ODBC.INI file. The following is a "pseudo-entry," meant to be representative:

```
[dBase_sdk20]                   ;The data source name; uniquely
                                ;identifies the database to the
                                ;ODBC administrator.
UID=dba                         ;User ID, if required by driver
PWD=secret                      ;Password, if required by driver
Driver=c:\windows\simba.dll     ;Identifies the driver dynamic link library
Description=Sample dBase Data   ;String describing the database
FileType=dBase4                 ;Identifies the file structure to the driver
DataDirectory=c:\DBASE          ;Identifies either a directory
                                ;or file, depending on the driver.
SingleUser=False                ;Optional driver parameters
```

Additionally, the data source name must also be identified in the [ODBC Data Sources] section, which usually appears at the top of the ODBC.INI file. A single line contains the data source name as it appears in the entry described above, and a short string description:

```
dBase_sdk20=dBase Driver
```

How to Test Your ODBC Application

Here are two tips for use when developing your ODBC application.

The ODBC driver manager can create a log file documenting all ODBC calls. It includes the actual SQL statements made by the driver to the data source, and includes any errors posted. Additionally, The ODBC "back-end" driver can pass an error message back to the Clarion for Windows ODBC driver. You can access this as an external variable.

The following tells you how to take advantage of these tips.

The ODBC Log File

There are different log files you can produce. One is produced by the Clarion ODBC driver, the other through the ODBC Driver Manager.

The ODBC Driver Manager's logging writes every ODBC call and the SQL statements they generate to disk, as the calls are made. The Clarion ODBC driver only logs errors that occur. This allows you to match calls to SQLError in the ODBC manager's log to actual error messages. This slows down the process considerably, so this should only be activated during testing. Additionally, the log file can grow to large proportions very quickly, so you must turn it off and delete the file after using it.

Besides "snooping" on the actual SQL statements generated by the driver, you can zero in on any errors. If the application was unable to connect, you can open the log file using the Write or WordPad applet (the file is usually too big for Notepad). Scroll to the very bottom of the file, then work up until you find the word "SQLError."

To enable Clarion ODBC driver logging:

You can enable logging on a system-wide basis, on a per-file basis, or on demand using a SEND() command.

For system-wide logging:

1. Add the following to your WIN.INI file:

[CWODBC]

Trace=1

TraceFile=[name of trace file]

For file logging:

1. In the File Properties dialog, in the Dictionary Editor, add the following in the Driver Options entry box:

LOGFILE=filename.ext

where *filename.ext* is the name of the logfile you wish to create.

For logging on demand:

1. Use a SEND() command at the appropriate point in your code, using the following syntax:

SEND(file,'/LOGFILE=filename.ext')

where *file* is the label of the data file and *filename.ext* is the name of the logfile you wish to create.

To turn logging off:

1. Use a SEND() command at the appropriate point in your code, using the following syntax:

SEND(file,'/LOGFILE')

where *file* is the label of the data file

To enable ODBC Administrator logging:

1. Start theODBC administrator.

You can do so by either running the ODBCADM.EXE file in the \Windows\System directory, or by DOUBLE-CLICKING the ODBC icon in Control Panel.

2. Press the **Options** button in the **Data Sources** dialog.
3. Check the**Trace ODBC Calls** box.
4. Optionally uncheck the **Stop Tracing Automatically** box if you think you need to test connecting more than once.

It's common to test several times before pinning down the error.

5. Press the **Select File** button and name a file to log to.

The default is called SQL.LOG.

6. Switch to your program and begin testing.

After the errors occur, open the log file and examine it. Remember to turn off the **Trace ODBC Calls** box when done testing.

The ODBC Error Message

You can name a global variable to store any ODBC error strings returned by the driver. To do so, you must define an EXTERNAL variable of the type CSTRING, whose length is commonly 80, declaring its external NAME as _ODBC_ERROR. Don't forget the underscores! The actual variable declaration is:

MyLabel CSTRING(80),EXTERNAL,NAME('_ODBC_ERROR')

Following an error in your application, display the contents of MyLabel in a message box.

The ODBC driver can create a logfile to track error messages. This can be done on a system-wide basis or on a file-by-file basis.

To trace ODBC errors system-wide:

1. Add the following section to your WIN.INI file:

```
[CWODBC]
Trace=1
Tracefile=[name of file]
.
```

To trace ODBC errors for a single file:

1. Use either the SEND command or driver string:

```
DRIVER ('ODBC','/LOGFILE=logfile [message]')
res" = SEND (file,'/LOGFILE=logfile [message]')
res" = SEND (file,'/LOGFILE [message]')
```

/LOGFILE=logfile

Opens the named logfile for exclusive access. If the file exists, the new log data is appended to the file. You can add a message to the Start line by including the message inside square brackets ([]). There must be a space between the name of the logfile and the opening square bracket ([).

/LOGFILE

Closes the file. You can add a message to the Stop line by including the message inside square brackets ([]). There must be a space between the name of the logfile and the opening square bracket ([).

Using /LOGFILE in a DRIVER string to start logging is exactly the same as issuing it in a SEND before any call to OPEN(file) or CREATE(file).

The /LOGFILE switch must be the last switch in the DRIVER string.

Using an ODBC Connect String

Normally you need only pass the DataSource name, User ID, and Password to an ODBC Data Source. However, some drivers, such as Sybase, may require more information. With these drivers, you can supply the additional information in the OWNER attribute after the password.

Use the following syntax:

keyword=*value;* keyword=*value.*

For example, to supply the Database and App name to Sybase:

OWNER(DataSourceName,UserID,PassWord,DATABASE=DataBaseName;APP=AppName)

How to Start a DDE Conversation

DDE (Dynamic Data Exchange) is a Windows Inter-Process Communication (IPC) protocol. A DDE "conversation" consists of two applications trading messages. Within the DDE conversation, one application acts as the client, the other as the server.

The application which starts the conversation, requesting data or services from the other, is the client. The contacted application is the server. The server must "register" with Windows that it has server capability.

Clarion for Windows allows you to create both DDE clients and DDE servers. An application can be both. In fact, your application can act as both a client and server at the same time, though it requires two separate DDE conversations.

Starting a DDE conversation is as easy as using the DDECLIENT function. The only "catch" is that both applications must already be running to open the channel.

The simplest way to ensure that the conversation takes place at run time is to use an IF (conditional execution structure) structure. The DDECLIENT function returns zero if the server application isn't already running. Test its return value, and use the RUN statement to start the server app if it returns zero.

Many of the DDE procedures and functions require that you specify the DDE channel number, which is an integer that Windows returns when you open the DDE conversation. Create a local variable to hold the return value. Begin at the **Procedure Properties** dialog of the procedure you wish to contain the code for the DDE conversation.

To enable support for the DDE commands for your project or application, you must include the DDE.CLW file, located in the LIBSRC subdirectory.

Create a variable to hold the DDE channel number:

1. Press the **Data** button in the **Procedure Properties** dialog.
2. Press the **Insert** button in the **Local Data** dialog.
3. Type **Channel** in the **Name** field.
4. Choose **LONG** from the **Type** dropdown list.
5. Press the **OK** button to close the **Field Properties** dialog.
6. Press the **Close** button to close the **Local Data** dialog.

Initializing the Conversation

You must embed the code to initialize the DDE conversation, starting the server application if it's not already started. Assuming a menu choice in your application begins the conversation, embed the code at a field event associated with the Accepted menu choice.

1. Choose the appropriate field event in the **Embedded Source** list.
2. Press the **Edit** button.
3. Choose the **Source** item in the **Embedded Source** dialog.
4. Press the **Add** button.
5. Type the following code, substituting the file name (without extension) of the Server application for "Excel."

```
Channel = DDECLIENT('Excel','System')      ! Contact Excel re System topic
IF Channel < 1                                    ! If no contact made
```

```
    RUN('Excel')                                    ! Attempt to start Excel
    Channel = DDECLIENT('Excel','System')! And try again
  ELSE
    RETURN                                  ! Give up if no contact - add error msg!
  END
```

The code example is deliberately simplistic; it would be more efficient to LOOP through the attempt to contact twice, then warn the end user of the failure.

The code attempts to open a DDE conversation with Excel named as the server. The DDECLIENT function returns a value corresponding to the channel; it doesn't matter what the channel number is. If it's less than one, it failed. You must therefore start the server, and try to open the conversation again.

The second parameter of the DDECLIENT function is the DDE "Topic." It tells the server what the DDE conversation is "about." In most cases, the topic is a file name. In this case, the code names the "System" topic, which tells Excel the conversation is not regarding a particular document file.

How to Send DDE Commands and Data to a DDE Server

Once the DDE channel is open, you can use the DDE functions to send commands, data, or requests to the server.

The example code below sends a command to Excel to open a new file and save it under a specified file name. This is a common DDE task when working with commercial applications. Often, the server application allows access to "document" functions only when you specify a document name in the DDECLIENT function. The document name must be a file that already exists.

In this particular case, to execute any "document" actions, such as entering a value in a cell, Excel (and many other applications) require the DDE channel "topic" to be the name of document. Therefore, if your application is providing new data it wants the server to save in a **new** document file, your application:

- Opens a conversation about the "System" topic.
- Sends a command asking the server to save a document file under a specified name.
- Closes the conversation.
- Opens a second conversation with the server, this time specifying the newly created file's name as the topic.
- Sends the "unsolicited" (because the server didn't ask for it) data and then tells the DDE Server (Excel) to execute commands or other requests for data that apply to the file.
- Closes the conversation.

The following therefore should execute only if the example code shown in the **How to Start a DDE Conversation** topic was successful.

```
DDEEXECUTE(Channel,'[NEW(1)]')                    ! Excel's File/New command
```

The DDEEXECUTE statement takes the DDE channel number as its first parameter, and the command string as the second. Excel requires you to enclose all DDE commands in square brackets. This command creates a blank spreadsheet.

The Excel command string enclosed by the square brackets is an Excel macro statement. Excel, and many other applications allow you to send a macro statement via the DDEEXECUTE statement. In this particular case, you don't have to know the name of the open Excel file to execute the statement.

Tip: **Many commercial applications with their own macro languages allow you to both record and edit macros. Use the application to make a "dry run" of the actions you need it to execute, with its macro recorder turned on. Edit the resulting macro, and use the clipboard to copy each macro statement to your embedded source in the Text Editor. Put each macro statement in the second parameter of the DDEEXECUTE statement, and you can be assured of the correct syntax for the DDE command!**

2. In the next embedded source line, tell Excel to save the new (blank) sheet under a name that you specify.

```
DDEEXECUTE(Channel, '[SAVE.AS("DDE_TEST.XLS",1,"", FALSE,"",FALSE)]')
```

Knowing the name allows you to close this channel, then open another specifying the file name as the topic. Note that the Excel command string requires double-quote marks.

3. Terminate the channel started under the "System" topic.

```
DDECLOSE(Channel)                                 ! Close first DDE channel
```

Sending Data from Client to Server

To continue the example, to send data to Excel, you need to open another DDE conversation, this time with the newly created file name as the topic:

1. Open the DDE channel and name the file as the topic.

```
Channel = DDECLIENT('Excel','DDE_TEST.XLS')!New channel under known filename
```

 To place data in a spreadsheet cell, use the DDEPOKE statement.

```
DDEPOKE(Channel,'R1C1','999')
```

Following the successful placement of the value in the spreadsheet, you could then send further Excel macro statements using DDEEXECUTE. This would allow you to send additional spreadsheet data, highlight a range, then tell Excel to draw a chart.

How to Customize Procedure Templates

Each procedure template can have multiple default procedures. For example, the Window template can have more than one default procedure--each with its own window design, pre-populated controls and/or control templates, and pre-defined local variables.

Default procedures are customized through the **Template Registry**.

You can also create and modify templates. See How to Modify Templates

1. Choose **Setup** **Template Registry.**

The Template Registry lists all registered templates.

2. Highlight the procedure template to customize, then press the **Properties** button.

The **Template Procedure Properties** dialog appears.

3. Press the **Defaults** button.

The Edit Default Procedures dialog appears with a list of the default procedures. All of the procedure templates in the shipped with this product have one default procedure each.

4. Press the **Add** button.

5. Provide a name for the new default procedure in the **New Procedure** dialog, then press **OK.**

A standard **Procedure Properties** dialog appears.

6. Edit the procedure as desired, designing a window, adding controls and/or control templates, adding local variables, etc.

7. Exit by pressing the appropriate **OK** or **Close** buttons.

The next time you add a procedure to an application and select the procedure type, you will be prompted to select the default procedure from which to start.

How to Modify Templates

We do not recommend modifying the templates shipped with this product. Instead, you should create your own template set, and make any desired modifications to that set.

You can copy any of the template files and include them in your own set. Keep in mind that any #INSERT statements referencing groups from the Clarion Template Set must be modified to indicate its source. For example:

```
#INSERT(%StandardWindowHandling)
```

should be modified to:

```
#INSERT(%StandardWindowHandling(Clarion))
```

Creating a New Template Set

There is only one requirement for the new template set; a #TEMPLATE statement to identify the set for the template registry. Of course, it also needs to have the specific procedure, code, control, and extension templates to add to the template registry.

For example, the following code is completely valid as a template set with nothing else added:

Example:

```
#TEMPLATE(PersonalAddOns,'My personal Template set')
#CODE(ChangeProperty,'Change control property')
  #PROMPT('Control to change',CONTROL),%MyField,REQ
  #PROMPT('Property to change',@S20),%MyProperty,REQ
  #PROMPT('New Value',@S20),%MyValue,REQ
%MyField{%MyProperty} = %'MyValue            #<!Change the %Property of %MyField
#!************************************************************************
#EXTENSION(CommentBlock,'Add a comment block to the procedure'),PROCEDURE
  #PROMPT('Comment Line',@S70),%MyComment,MULTI('Programmer Comments')
#ATSTART
  #FOR(%MyComment)
  !%MyComment
  #ENDFOR
#ENDAT
```

When you (see also)register this template set, it will appear in the template registry as Class PersonalAddOns containing the ChangeProperty code template and the CommentBlock extension template.

Once a template set is registered in the template registry, all its components are available to the programmer for application development, along with all the components of all other registered template sets. This allows the programmer to "mix-and-match" template components during development.

For example, the programmer could create a procedure from a procedure template in the standard Clarion template set, populate it with a control template from a third-party vendor, insert a code template into an embed point from another third-party vendor, then add an extension template from their own personally written template set. At source generation time, all these separate components come together to create a fully functional procedure that performs all the tasks required by the programmer (and nothing else). This is the real power behind Clarion's Template-oriented programming!

For more information see the **Template Language Reference**.

How to Register a Template Set

You can register template sets shipped with this product, your own template sets, or third-party vendors' template sets.

1. Choose **Setup** **Template Registry.**

The Template Registry lists all registered templates.

2. Press the **Register** button.

Select the template (.TPL) file from the standard Open File dialog.

3. Press the **OK** button.

How do I handle an Error 47

Error 47--Invalid Record Declaration indicates that the application's file declaration does not match the data file.

To correct the problem, you should convert the file using one of these two conversion methods:

How to convert a file--Generate Source

How to Convert a File (without generating source)

How to Convert a File--Generate Source

If a file's definition needs to be changed and meaningful data exists, follow these steps to convert the file. This method creates an executable file that you can ship to end users to convert their data files. If you want to convert a file without creating a file conversion program see How to Convert a File (without generating source) .

1. Open (load) the dictionary that contains the file to be modified.

2. Copy the data file definition to a new name. To copy a file definition, highlight the file to be copied in the Files List and press CTRL+C, then press CTRL+V to paste it. You will be prompted to supply a new name and prefix. (Example - copy Customer to OldCustomer)

An alternative would be to copy the entire dictionary to a new name. You might use this method if there are multiple files to be converted in one session. Clarion for Windows allows files to be converted from one dictionary to another.

3. After the file definition has been copied, make any necessary changes (add fields, change the file driver type, etc.) to the definition with the original name. In our example above, the Customer file is the file to be modified.

4. Save the Dictionary after file modification and close it.

The Dictionary file **must** be closed in order to use it for file conversion.

5. Load the file in the Database Manager File utility (**File** **Open , Database Tab**)

6. Next, you will be prompted to pick a file to load. For this example, you would select the Customer file.

The Customer file displays.

7. Choose **File** **File Convert** (or press CTRL+V).

The File Convert dialog appears, prompting for the information below:

Source Filename	Specifies the file to convert. This defaults to the file opened by the Database Manager.
Source Dictionary	Specifies the dictionary which contains the file definition for the source data file. A Source Dictionary is not required.
Source Structure	Specifies the structure (within the dictionary) which defines the source file. A Source Structure is not required.
Target Filename	Specifies the name of the new file. This defaults to the current file name.
Target Dictionary	Specifies the dictionary which contains the file definition to which to convert. A Target Dictionary is required.
Target Structure	Specifies the structure (within the dictionary) of the target file. The Target Structure is required.

8. Specify the file name for the generated source code of accept the default of CONVERT.CLW.

9. Press the **OK** button.

This generates a source file. This file can now be compiled and linked to an executable program which will perform the file conversion.

NOTE: Prior to executing the source conversion program, the current data file loaded into the Database Manager must be closed.

10. Press the **Exit** button to close the data file in the Database Manager.

11. Load the conversion program by choosing **File** **Open** and selecting the **Source** tab.

12. Select CONVERT.CLW (or the file name you specified) in the **File Open** dialog. The conversion source code is displayed in the Text Editor.

Tip: If you changed the name of a field, edit the source code to make the field assignments. Otherwise, your data will be lost. See the Language Reference How to Make Field Assignments.

13. The project file must now be loaded. Choose **Project** **Set**. Select the project file. This defaults to CONVERT.PRJ.

14. Finally, you may now Make and/or Run the conversion program.

After the conversion program runs:

15. Check the file that has just been converted by loading the new (target) file back into the Database Manager.

After viewing the file converted, some clean up steps are all that's left to do:

16. If the file converted was located in a different directory, you may now copy it into the working program directory. If you had originally renamed the file and placed it in the same directory, you may rename it to the original file name at this time.

17. The "old" file definition may now be deleted from the active dictionary, or archived into a backup dictionary file.

The conversion process is now complete. This example creates CONVERT.EXE which may be shipped to end users to convert their files.

How to Convert a File (without generating source)

If a file's definition needs to be changed and meaningful data exists, follow these steps to convert the file. This method does not create an executable file. It converts the data file on your system to a new format. If you want to create a file conversion program see How to Convert a File--Generate Source

Tip: **It is always a good idea to make backup copies of your files before running any conversion process.**

Note: **If you change the name of a field, you must generate source code, and edit the source code to make the field assignments. Otherwise, your data will be lost.**

1. Open (load) the dictionary that contains the file to be modified.

2. Modify the data file definition as desired (add fields or keys, change the file driver type, etc.).

3. With the modified file highlighted, choose **File**  **Browse** to load the data file in the Database Manager.

A message appears, warning that the physical file structure does not match the dictionary declaration.

4. Press the **Yes** button to convert the file.

The conversion process is now complete!

How to Make a Field Assignment

The File Conversion Utility creates source code to convert a file to a different specification. The conversion is handled automatically except in two cases:

- If a field's label is changed
- If a field is split into two separate fields.

In these cases you must modify the source code to handle the field assignments. The portion of the source code you'll need to examine is the **AssignRecord ROUTINE**. This is where field assignments are made. Here is an example:.

```
AssignRecord        ROUTINE
  CLEAR(CUS:Record)
  CUS:NUMBER = IN::NUMBER
  CUS:FIRSTNAME = IN::FIRSTNAME
  CUS:LASTNAME = IN::LASTNAME
  CUS:ADDRESS = IN::ADDRESS
  CUS:CITY = IN::CITY
  CUS:STATE = IN::STATE
  CUS:ZIP = IN::ZIP
  CUS:PHONENUMBER = IN::PHONENUMBER
```

If you examine the source code, you'll see that the first line in the routine clears the record buffer. Nex,t each field in the output file is assigned the value from the matching field in the input file. However, if the field labels do not match, no assignment is made. For example, if you change the LastName field to Surname, a comment statement is generated to alert you of an assignment that may need to be made:

```
AssignRecord        ROUTINE
  CLEAR(CUS:Record)
  CUS:NUMBER = IN::NUMBER
  CUS:FIRSTNAME = IN::FIRSTNAME
  ! CUS:SURNAME = ''
  CUS:ADDRESS = IN::ADDRESS
  CUS:CITY = IN::CITY
  CUS:STATE = IN::STATE
  CUS:ZIP = IN::ZIP
  CUS:PHONENUMBER = IN::PHONENUMBER
```

To assign the values from the original file, edit the line containing the assignment to assign the value of LastName to the SurName field as shown below:

```
  CUS:SURNAME = IN::LASTNAME
```

Writing the assignment statements to split the contents of a field into two fields involves a little more work, but using string slicing minimizes the effort. For this example, let's assume that you had a single field in the original file for a phone number and area code. You now want to store the area code in one field and the phone number in a another. Assuming that these fields are numeric data types, you will need to temporarily assign the value to a string, then slice the string to assign the desired portion to each new field. In this example the original PhoneNumber field is a ten-digit number, the area code is a three-digit number, and the new PhoneNumber field is a seven-digit number. The AssignRecord ROUTINE in the generated file conversion source code will look like this:

```
AssignRecord        ROUTINE
  CLEAR(CUS:Record)
  CUS:NUMBER = IN::NUMBER
  CUS:FIRSTNAME = IN::FIRSTNAME
  CUS:LASTNAME = IN::LASTNAME
  CUS:ADDRESS = IN::ADDRESS
  CUS:CITY = IN::CITY
```

```
CUS:STATE = IN::STATE
CUS:ZIP = IN::ZIP
! CUS:AREACODE =
CUS:PHONENUMBER = IN::PHONENUMBER
```

Notice that there is an assignment from the original PhoneNumber field to the new PhoneNumber field. However, since the new field only stores seven digits, you must edit this. To handle the field assignments, you will create an implicit string variable, assign to it the value of the original PhoneNumber field, then use string slicing to assign the desired portions to the new fields, as shown below:

```
AssignRecord        ROUTINE
  CLEAR(CUS:Record)
  CUS:NUMBER = IN::NUMBER
  CUS:FIRSTNAME = IN::FIRSTNAME
  CUS:LASTNAME = IN::LASTNAME
  CUS:ADDRESS = IN::ADDRESS
  CUS:CITY = IN::CITY
  CUS:STATE = IN::STATE
  CUS:ZIP = IN::ZIP
  TempPhoneNumber" = IN::PHONENUMBER
  CUS:AREACODE = TempPhoneNumber"[1:3]
  CUS:PHONENUMBER = TempPhoneNumber"[4:10]
```

For more information on String Slicing, see **Implicit Arrays and String Slicing** in Chapter 4 of the **Language Reference**.

Redirection File

The development environment sets the working directory to the one in which the current .APP or .PRJ file resides. Additionally, you can tell the Project System to search for additional components at compile/link time by specifying the directories to search in the Redirection file.

The .RED file is a text file, which you edit with the Text Editor. File types appear on the left; paths on the right. The syntax for specifying paths is the same as for the DOS PATH command.

The default file types are as follows:

***.DBD**	Debug information files.
***.DLL**	Dynamic Link Libraries. The does *not* refer to the .DLL files used by the IDE; those are loaded from the current directory, the Windows\System directory, or the PATH. This refers to .DLL files referenced in the Project Tree.
***.LIB**	Static link libraries.
***.OBJ**	Object Files.

Tip: If you wish the Project System to place all .OBJ files in the same directory, no matter where the project is located (this allows you to "clean up" after your projects more quickly), specify ".;" as the first directory for the .OBJ entry.

***.RSC**	Compiled resource files.
***.ICO**	Icon files.
***.TPL**	Template libraries.
***.TPW**	Template source files.
***.TRF**	The template registry.
.	All other files.
QCKSTART.TXA	Quick Start script.
QCKSTART.TXD	Quick Start script.

Distributing Your Applications

Clarion for Windows 1.5 produces true 16 or 32-bit executable application files which you may distribute on a royalty-free basis. The 16-bit applications you distribute require Windows 3.10, or 3.11. The 32-bit applications you distribute require Windows NT 3.x, or WIN95.

Applications created using Clarion for Windows require that the file CWRUNxx.DLL be present on the end user's system or linked into the executable. The file may reside in the same directory as the application, in the Windows\System subdirectory, or in a directory referenced in the DOS PATH. TopSpeed recommends that you install CWRUNxx.DLL to the *application directory* when you create a setup program for distributing your applications.

Multiple Clarion for Windows applications may use the same CWRUNxx.DLL file, avoiding the need to duplicate space on the end users' hard drive. Windows loads CWRUNxx.DLL into memory once for each running Clarion for Windows executable.

Additionally, you should also distribute any file drivers required by your application. The Database Drivers section lists all the drivers and their files. Multiple Clarion for Windows applications may use the same file drivers also saving space on the end users' hard drive.

In Clarion for Windows 1.5, the runtime libraries' .DLL--CWRUN10.DLL--has been replaced with two versions. CWRUN16.DLL is used for 16-bit applications, and CWRUN32.DLL is used for 32-bit. In addition you now have the option to link in the runtime libraries and database drivers to create a "one-piece" executable. Keep in mind that when installing multiple Clarion for Windows applications, keeping them as external .DLLs saves disk space.

When creating an application using the Application Generator, the Clarion for Windows template library automatically creates a ship list in a text file. The file has the same file name as the .APP file, and its extension is .SHP. You can view the file in the Text Editor. The list includes only those files which the Application Generator knows about. If you name a file in a variable, such as, for example, a .JPG file which the app places in an IMAGE control via Property Assignment Syntax, it does not appear in the ship list. You are responsible for checking that the list is complete.

How to Clip and Concatenate Name Fields

First names and last names are often stored in separate fixed length fields. If printed directly from those fields they usually contain extra spaces and no punctuation, like this: Katie Kelton E. However, you may want the name to look like this: Kelton, Katie E. Follow these steps to display names with punctuation and without the extra spaces. This procedure assumes you already have a report procedure that displays your name fields.

1. Create a Memory Variable to Hold the Concatenated Names.
2. Creata an Expression to Clip and Concatenate the Names.
3. Place the Variable in Your Report.

Create a Memory Variable to Display the Concatenated Names

1. From the Application Tree dialog, highlight your report procedure and press the **Properties** button.
2. From the **Procedure Properties** dialog, press the **Data** button.
3. From **Local Data** dialog, press the **Insert** button to add a local variable.
4. From the **New Field Properties** dialog, define the new variable as follows.

In the **Field Name** box, type *FullName*. This is the name by which we will refer to the variable in our concatenation formula, and in our report.

The variable should be long enough to hold all the name fields to be concatenated, plus any punctuation and spaces. For example: if the first name field is 20 characters, the last name field is 20 characters, the middle name field is 1 character and you plan to use a comma, a period, and a space for punctuation, then you will need 20 + 20 + 1 + 3 (ie. 44) characters for your new variable.

In the **Characters** spin box, type 44, then press the **OK** button, then the **Close** button to return to the **Procedure Properties** dialog.

Create an Expression to Clip and Concatenate the Names

You can create the same result by typing the assignment statement into the "Before Print Detail" embed point for your procedure. However, we will use the Formula Editor to accomplish our goal.

1. From the **Procedure Properties** dialog, press the **Formulas** button.
2. In the **Name** field, type *Name Concatenation*.
3. For the **Class** field, press the ellipsis (...) button and choose ***Before Print Detail*** from the Template Class list, then press the **OK** button.

   The class (also called Formula Class) determines *when* the expression is evaluated and the assignment performed.
4. For the **Result** field, press the ellipsis (...) button, highlight ***Local Data***, highlight ***FullName*** (the variable we defined above), then press **Select** the button.
5. Press the **Functions** button, then choose CLIP from the Functions list, then press **OK**.
6. Press the **Data** button, highlight your report procedure file, highlight the last name field, then press **Select** the button.
7. At the insertion point, type the following code:

```
&', '&CLIP(FirstNameField)&' '(MiddleInitialField)&'.'
```

Where FirstNameField is the first name field in your report file, and MiddleInitialField is the initial field in

your report file. Steps 5 - 7 show how you may type your expression, or use the Formula Editor buttons to choose valid operands and operators.

8. Press the **Check** button to check your expression syntax.

A green check appears if syntax is correct, otherwise a red X appears.

9. Press **OK** to exit the Formula Editor; press **OK** again to exit the Formulas dialog.

Place the Variable in Your Report

1. From the **Procedure Properties** dialog, press the **Report** button.
2. Delete all but one of the name fields from your report.

CLICK on the field, then press DELETE.

Tip: Display the Property Toolbox (Option Show Propertybox) to help you identify your report fields.

3. CLICK on the remaining name field to select it.

If you haven't displayed the Property Toolbox as described in the **Tip**, do it now.

4. In the Property Toolbox **Picture** field, type *@S44*.
5. In the Property Toolbox **Use** field, type *FullName*.
6. Resize the field by dragging its handles.

How to Store and Display a Graphic Image with a Memo or Blob Data Type

Memo and Blob variables are capable of storing large variable length chunks of binary data. This makes them suitable for storing graphic images. MEMOs are limited to 64K or less. BLOBS have no size limit. Storing and displaying images with Memo or Blob variables requires the following:

Storing Graphic Images in BLOBs or MEMOs

To *store* the graphic image into the MEMO or BLOB variable, channel it through an IMAGE control.

That is, assume the image resides in C:\IMAGES\IMAGE.BMP. We need to transfer the .BMP file to a CW IMAGE control, then transfer from the IMAGE control to the MEMO or BLOB variable.

1. The BLOB or MEMO variable must have the BINARY attribute.

In the Data Dictionary, use the **Field Properties** dialog's General tab to set the **Data Type** to MEMO or BLOB, then check the **Binary** box.

2. Assign the image file to an IMAGE control.

In the **Image Properties** dialog, in the **File** field, use the ellipsis (...) button to name the file containing the graphic image.

or

Assign the file name with Clarion's property syntax as follows:

?Image1{PROP:Text} = *filename*

3. Transfer the image from the IMAGE control to the MEMO or BLOB variable using Clarion's property syntax:

For MEMOs: CON:TheMemo = ?Image1{PROP:ImageBits}

For BLOBs: CON:TheBlob{PROP:Handle} = ?Image1{PROP:ImageBlob}

Displaying Graphic Images from BLOBs or MEMOs

To restore (ie display) the image from a BLOB or MEMO to an IMAGE control, you must properly define the size of the IMAGE control. The IMAGE control must either be of Default size, or of a fixed size set *after* the MEMO or BLOB data is assigned to it.

1. To set the IMAGE control to default size.

In the **Image Properties** dialog, on the **Position** tab, check the **Default** boxes for **Height** and **Width**.

2. Use Clarion's property syntax to transfer the MEMO or BLOB data to the IMAGE control.

For MEMOs: ?Image2{PROP:ImageBits} = CON:TheMemo

For BLOBs: ?Image2{PROP:ImageBlob} = CON:TheBlob{PROP:Handle}

3. *After* the MEMO or BLOB has been assigned to the IMAGE with property syntax, a fixed width and height may be assigned to the IMAGE Control:

```
?Image2{PROP:Width} = 92
?Image2{PROP:Height} = 88
```

The PROPS.APP example in the \CW\EXAMPLES\APPS\PROPERTY directory demonstrates using property assignments for images.

BrowseBox Control: The Inside Story

This is both a template upgrade guide, and some insider information about some structures used internally.

Users of CW1.0 will find many new features in CW1.5. One of the greatest areas of change is the BrowseBox control template. This template has been rewritten to reduce both the amount of code generated and the amount of time taken to keep the browse active. To do this, it's been necessary to replace several routines and to change the functionality of others.

When routines and variables are discussed, I've used the value %InstancePrefix in the names of some of these. The InstancePrefix is an identifier unique to each instance of a control template. BrowseBoxes have an InstancePrefix of **BRW*n*:**, where **BRW** indicates that the control is a BrowseBox, *n* represents the order in which the control or extension template was populated. The first control template populated would be 1, the fourth, 4. Please note that this number does not represent the first or fourth BrowseBox, rather the first or fourth control or extension template. The colon is necessary to separate the prefix from the routine or variable. Therefore, if I mention a routine called %InstancePrefix:SelectSort, you should look in your code for a routine named (approximately) BRW1::SelectSort.

VIEWs, Filters, and Hot Fields

The 1.5 BrowseBox, Report, Process, FileDrop and FileDropCombo (new) now use the VIEW structure to control the records read for the BrowseBox. The printed documentation goes into greater detail on the VIEW structure, but there are a few new implications when using BrowseBox because of the VIEW structure:

The VIEW structure only retrieves those fields from the file that are built into the VIEW using the PROJECT statement. Any fields used for EMBEDs and hand-coded FILTERs need to be added to the VIEWs list of fields to PROJECT. This is done through the Hot Fields tab in the template's properties screen.

The VIEW structure uses PROP:Filter to filter records. This filter is evaluated through a mechanism similar to the EVALUATE() command. Because of this, values added to the filter through the Hot Fields list need to have the "Bind Field" checkbox set.

The VIEW structure uses PROP:Filter to perform range limit optimizations. If you have a string field in a case insensitive (NOCASE) key, the optimization needs to have both the key field and the limit value uppercase. For instance, if you have a status field called FIL:Status, with valid values of 'Active' and 'Inactive', and you want to limit the BrowseBox display to records with a value of 'Active', you would code the filter UPPER(FIL:Status) = 'ACTIVE'. Without this, the VIEW will not only NOT display all of the records, but it will do it very slowly.

Conditional Displays

One of the big changes is the multi-display capability. CW 1.0 could display a file by a single key only. 1.5 allows you to select, based on condition specified, on any combination of keys, or in record order, and with any number of filters. Refreshing the browse is based on the use of "Reset Fields", which minimize the number of redisplays, since the triggering of redisplay is much more selective than the 1.0 method of setting the ForceRefresh flag to true, then DOing the RefreshWindow routine. The "Reset Fields" can be unique for each conditional sort.

The %InstancePrefix:SelectSort routine is used to determine which conditional sort order to use, and to control refresh of the browse. This routine is called whenever the RefreshWindow routine is called. The internal decision-making code is something like:

Determine the sort criteria to use. If none of the conditions specified are met, use the default browse criteria.

If the sort order has not changed from the current sort order, check if any of the current sort order's Reset Fields have changed.

If the sort order has changed, or the reset fields have changed, or ForceRefresh is set, redisplay the browse.

Locating a Record

In 1.0, the %InstancePrefix:LocateRecord routine used the value returned by POSITION(key), followed by a combination of values in the record buffer, or records in the browse queue to determine its operation. This was bad for two reasons. First, the template had to guess what you wanted. Second, POSITION(key) can be a slow function for some file drivers In 1.5, we move to a more direct method of determining the operation of %InstancePrefix:LocateRecord

A variable has been declared for each browse, called %InstancePrefix:LocateMode. This variable will be set to values, represented by the EQUATEd values LocateOnPosition and LocateOnValue. If %InstancePrefix:LocateMode = LocateOnPosition, the browse will find and highlight the record currently active in the VIEW, otherwise, the closest record to the one with key values the same as those currently in the record buffer will be located.

Refreshing a Browse Page

In 1.0, %InstancePrefix:RefreshPage was, like %InstancePrefix:LocateRecord, based on POSITION(key). In 1.5, we again replace that messaging system with one that requires that %InstancePrefix:RefreshMode be set to one of a set of values. These values are:

RefreshOnPosition - puts the record active in the VIEW on the top of the browse

RefreshOnQueue - If there's a record in the QUEUE, refreshes the browse, keeping the item in it's location on the browse, if possible. If there's no record in the QUEUE, rebuilds the first page of the browse.

RefreshOnTop - Requires that the QUEUE be FREEd. Loads the first page of the browse.

RefreshOnBottom - Requires that the QUEUE be FREEd. Loads the first page of the browse.

RefreshOnCurrent - Used internally only.

Retrieving a record in sequential order

In 1.0, we had two routines, %InstancePrefix:FillForward and %InstancePrefix:FillBackward, which filled the queue with one record forward or backward in the key. Since most of the code in these two routines is the same, the routines been replaced with %InstancePrefix:FillRecord. To control whether a record is read with a NEXT or PREVIOUS, a variable, labelled %InstancePrefix:FillDirection has been declared. Additionally, as in 1.0, the variable %InstancePrefix:ItemsToFill contains the number of records to retrieve. If this is assigned a value of FillForward (equated to 2), the routine will read records using NEXT. If assigned a value of FillBackward, the routine reads using PREVIOUS. This value is NOT cleared when the routine is completed.

The %InstancePrefix:FillRecord routine has a few additional flags that it uses to make it more general purpose:

%InstancePrefix:AddQueue - If set to 1 (True), will not add the fetched records to the Browse queue. This is a trigger, and is reset upon exit from the routine.

%InstancePrefix:ItemsToFill - As mentioned above, this value is set to instruct the routine on how many records to retrieve. When an error during a read is experienced (which usually indicates that all records have been read), or when all of the records requested have been read, the routine is terminated. When control returns to your code that called this routine, you can check the value of %InstancePrefix:ItemsToFill, and if it has a non-zero value, you know that all of the records you requested were not available.

Making a NewSelection

In 1.0, it was common to POST(Event:NewSelection,?List) to invoke a redisplay of information. It happened so often that in some cases the Event was invoked so many times that stack faults occurred.

To prevent this, we've added a routine called %InstancePrefix:PostNewSelection. This routine maintains an internal flag, and if this flag is not set, the routine sets it, and posts a new selection event. The code to process this event clears the flag. Please use this routine to post this event, rather than the POST() function.

Scrolling the BrowseBox

In 1.0, there were six routines to control scrolling. Since most of their functionality was duplicated (up and down simply called a different routine each), they've been consolidated into three routines. These routines do their work based on the value of %InstancePrefix:CurrentEvent, which is assigned the value of the event being handled before calling the routines.

ScrollOne - Scrolls a single record

ScrollPage - Scroll a page of records

ScrollEnd - Displays the first or last page of the BrowseBox

We recommend that you post the appropriate events to the list box rather than use these routines.

How to Display the Sort Field First on a Multi-Key Browse

Clarion's Browse Wizard generates multi-key browses for files with multiple keys. To see records in a different sort order, the user simply selects a tab with a different key. However, when switching to a new sort order, the sort column does not automatically appear as the first (or leftmost) column in the list box.

To dynamically shift the sort column to the leftmost position in the list box, follow these steps:

1. Find the FORMAT string for the affected list box and copy it to the clipboard.

In the Application Tree dialog, DOUBLE-CLICK on your browse procedure.

In the Procedure Properties dialog, press the ellipsis (**...**) button next to the **Window** button.

On the LIST control declaration statement, highlight the entire FORMAT attribute parameter and choose **Edit** **Copy** to copy it to the clipboard.

Exit! without saving.

2. Paste the FORMAT string into the "Control Event Handling, before generated code; ?CurrentTab; NewSelection" embed point of the Browse procedure.

Press the **Embeds** button.

DOUBLE-CLICK on the "Control Event Handling, before generated code; ?CurrentTab; NewSelection" embed point.

Choose SOURCE from the **Select embed type** dialog.

When the Text Editor opens, choose **Edit** **Paste**.

3. "Chop up" the FORMAT string so there is only one list box column definition per line.

   Each column definition begins with the width of the column immediately followed by a justification letter (L, R, C, or D). If necessary, you can get the widths and justification for each column from the List Box Formatter.

On each line you need to add enclosing single quotes.

On each line except the last, you need to add a trailing ampersand (&) and pipe (|). The ampersand is the concatenation operator, and the pipe is the line continuation character.

Your code should look something like this:

```
'16L|M~Cust Number~@N4@'     & |
          '80L|M~Last Name~@S20@'       & |
          '80L|M~First Name~@S20@'      & |
          '12L|M~Area Code~@S3@'        & |
          '32L|M~Phone Number~@S8@'     & |
          '32L|M~Description~@S8@'
```

4. Add explicit QUEUE field numbers to each list box column definition.

Queue numbers are integer constants surrounded by pound (#) signs. The QUEUE field numbers start with 1 and continue in ascending sequence. Your code should now look something like this:

```
'16L|M~Cust Number~@N4@#1#'     & |
          '80L|M~Last Name~@S20@#2#'       & |
          '80L|M~First Name~@S20@#3#'      & |
          '12L|M~Area Code~@S3@#4#'        & |
          '32L|M~Phone Number~@S8@#5#'     & |
          '32L|M~Description~@S8@#6#'
```

5. Add a CASE structure and property assignment to the embedded source code.

Type ?BROWSE:1{PROP:Format}= before the first column definition. This creates the assignment statement that formats the list box at run time.

Add a CASE CHOICE(?CurrentTab) statement before the first column definition.

Add an OF 1 statement before the first column definition.

Your code should now look something like this:

```
CASE CHOICE(?CurrentTab)
OF 1                    !Tab1, sorted by Cust Number
  ?BROWSE:1{PROP:FORMAT} ='16L|M~Cust Number~@N4@#1#'     & |
                          '80L|M~Last Name~@S20@#2#'      & |
                          '80L|M~First Name~@S20@#3#'     & |
                          '12L|M~Area Code~@S3@#4#'       & |
                          '32L|M~Phone Number~@S8@#5#'    & |
                          '32L|M~Description~@S8@#6#'
```

6. Duplicate the FORMAT string, with the sort column first, for each OF in the CASE.

You should have a separate OF clause for each tab (sort key) in the browse. Cut and Paste the FORMAT string for each OF assignment so that the columns appear in the sequence you want them.

Your code should now look something like this:

```
CASE CHOICE(?CurrentTab)
OF 1                    !Tab1, sorted by Cust Number
  ?BROWSE:1{PROP:FORMAT} ='16L|M~Cust Number~@N4@#1#'     & |
                          '80L|M~Last Name~@S20@#2#'      & |
                          '80L|M~First Name~@S20@#3#'     & |
                          '12L|M~Area Code~@S3@#4#'       & |
                          '32L|M~Phone Number~@S8@#5#'    & |
                          '32L|M~Description~@S8@#6#'
OF 2                    !Tab2, sorted by Last Name
  ?BROWSE:1{PROP:FORMAT} ='80L|M~Last Name~@S20@#2#'      & |
                          '80L|M~First Name~@S20@#3#'     & |
                          '16L|M~Cust Number~@N4@#1#'     & |
                          '12L|M~Area Code~@S3@#4#'       & |
                          '32L|M~Phone Number~@S8@#5#'    & |
                          '32L|M~Description~@S8@#6#'
END
```

In this example, notice that when the user selects tab 2, QUEUE field number #2# becomes the first column in the FORMAT string, and will be the leftmost column in the list box!

7. **Exit!** the Text Editor and save your changes.

Clarion Language Enhancements

This portion provides an easy method to jump into the Language Reference to see the syntax changes for CW 1.5. These include new Controls, new attributes, and other changes to the Clarion Language.

Chapter 2--Program Source Code Format

Field Qualification

FUNCTION and PROCEDURE Prototypes

DLL (set procedure defined externally in .DLL)

PROC (set function called as procedure without warnings)

PRIVATE (set procedure private to a single module)

Passing GROUPs and QUEUEs as Parameters

Chapter 3--Declaring Variables

STRING (fixed-length string)

CSTRING (fixed-length null terminated string)

GROUP (compound data structure)

DIM (set array dimensions)

DLL (set variable defined externally in .DLL)

Chapter 6--Window Structures

Controls

PROGRESS (declare a progress control)

SHEET (declare a group of TAB controls)

TAB (declare a page of a SHEET control)

Attributes

SPREAD (set evenly spaced TAB controls)

TIP (set 'balloon help' text)

VALUE (set RADIO control OPTION USE variable assignment)

WIZARD (set "tabless" SHEET control)

Chapter 7--Window Commands

Window Functions

POPUP (return popup menu selection)

Chapter 9--Reports

REPORT (declare a report structure)

Attributes

PAPER (set report paper size)

Chapter 10--Data Files

BLOB (declare a variable-length memo field)

DLL (set file defined externally in .DLL)

Chapter 12--Memory Queues

QUEUE (declare a memory QUEUE structure)

EXTERNAL (set queue defined externally)

DLL (set queue defined externally in .DLL)

Chapter 13--Miscellaneous Procedures and Functions

DIRECTORY (get file directory)

REJECTCODE (return reject code number)

Appendix C--Properties

Property Assignments

Appendix D--Events

Event Equates

Editor Options Dialog

Click on a TAB to see its help

To personalize your editing environment, customize appearance and cursor behavior with the **Editor Options** dialog. To view the dialog, choose **Setup** {BMC ARROW.BMP} **Editor Options**. Select the corresponding tab to set specific Text Editor options.

Insertion

Indent New Line To automatically give a new line the same indention as the previous line, check this box. This will make your code more readable.

Insert Within Column
When the insertion point is in the middle of a line, ENTER adds a new line after the current line.

Automatic Word-wrap
To cause automatic line breaks at column 70, check this box.

Split Line at Cursor When this box is checked, ENTER will split the current line at the insertion point (cursor). The second part of the line will appear on a new line. When this box is not checked, ENTER inserts a blank line below the current line, *without* splitting the current line.

Tab Size To set the default spacing between tabs, enter a number in the **Tab Size** box.

Block

Automatic Block Delete
To delete the selected text when pasting, check this box. To insert before a selected block, uncheck the box.

Remove Block On Copy
To delete the selected text when copying, check this box.

Colors

These options allow you to set color choices for twenty-one different Clarion language elements. For example, make Clarion keywords appear in red, or make equates appear in green.

Select a language or text element in the **Color Groups** list box, then CLICK on a color selection box. The sample text shows you how the selected language element will appear in the Text Editor.

Color Groups Highlight the language or text element to receive a color assignment.

Color To assign a color to the selected language element, CLICK on a color selection box.

Default To assign the default color to the selected language element, check this box.

Custom To reset the custom color for the selected language element, check this box.

Sample Text Shows how the selected language element will appear in the Text Editor.

Enabled To apply the color syntax highlighting to the file types listed in the **Source**

Extensions box, check this box.

Source Extensions

To specify the file types that color syntax highlighting is applied to, type a list of file extensions separated by semicolons.

Restore Defaults To assign the default colors to all language and text elements, check this box.

Saving

Make Backup Files

To cause the Text Editor to make a backup file (.BAK) each time you explicitly save a source file, check this box. The .BAK file contains the source as it was previously saved.

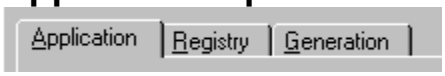
Prompt for Reload if file changed

To receive a "source.CLW has changed on disk. Do you want to reload?" message whenever the Text Editor detects such a change, CHECK THIS box.

Automatic Save time (minutes)

Clarion's Automatic Save option saves the current file according to the time interval you specify.Type the desired number of minutes in this box. The a copy of the last *explicitly* saved version of the file is stored in a temporary file and is used to restore if you cancel the current session and choose not to save.

Application Options Dialog



Click on a TAB to see its help

The **Application Options** dialog allows you to specify default settings for each new application you create.

Application

Require Dictionary This options specifies that each new application *must* have a data dictionary.

Default Dictionary Names a data dictionary file as the default which appears in each new Application Properties Dialog . You can change to another dictionary before closing the dialog.

Multi User Development Specifies file management options for multiple developer projects. See the *Multi-Programmer Development* appendix in the *User's Guide* for more information.

Display Repeated Funcs

Specifies the Application Generator displays the names of all functions, as it encounters them in the source code modules, in the progress box displayed during code generation.

Procedures per Module

The **Procedures per Module** spin control specifies the number of procedures that the Application Generator writes to each source code module. This can affect compile time when used with Conditional Generation turned on. Specifying one procedure per module, for example, means that each successive compile rebuilds only those procedures changed since the last one, and no more. The down side to this is that it requires more disk space. Generally, a smaller number is faster.

Populate Main Module This option specifies that the Application Generator writes procedures to the main source code module. When this option is off, the main module only contains the MAIN procedure, program global code, internally generated procedures and functions standard for every application. All other procedures reside in other file(s).

Import Clash Specifies how the Application Generator handles procedure names from an imported application file which clash with procedure names already resident. The drop down list choices are self-explanatory.

Disable Field Prompts Specifies that template-generated field-specific prompts will not display. This does not disable prompts created by Control Templates.

Application Wizard Check this box to specify the default when creating a new application is to use the Application Wizard. You can override this choice when creating an application by checking or unchecking the Application Wizard box in the Application Properties dialog.

Procedure Wizards Check this box to specify the default when creating a new procedure is to use the appropriate Procedure Wizard. You can override this choice when creating a procedure by checking or unchecking the **Procedure Wizard** box in the **Select Procedure Type** dialog.

Registry

Template Language code can be logically split among many files. Clarion for Windows uses the files to produce one logical template set for creating applications. The Registry Options are mainly for programmers who produce their own template files or make modifications to the default templates.

Re-register Changed To automatically re-register your templates when the Application Generator detects a change, check the **Re-register When changed** box. This defaults to "On."

Update Template Chain To automatically update the Template files when making a change in the Template Registry, check the **Update Template Chain** box.

Regenerate Deleted To specify the Application Generator should re-generate the .TPL and .TPW files from REGISTRY.TRF, should the files be deleted, check the **Regenerate Deleted Templates** box.

Generation

Conditional Generation This check box specifies that only source code modules changed since the last make should be compiled.

Debug Generation Specifies a text file for the Application Generator to log events to, and turns logging on and off. In case of a fatal error by the Application Generator, this log provides a trace to identify where the problem occurred. You can specify the file name in the **Debug Filename** box.

Generation Message Allows to to specify what displays during generation. The choices from the drop down list are self-explanatory.

Application Properties Dialog

This dialog allows you to create a new application, or edit the "essential" information for an existing .APP file.

Application File	Type a name for the .APP file. When opening the .APP file, the IDE makes the directory in which the .APP file resides the working directory.
Dictionary File	Type the name of the data dictionary file (.DCT). If your application does not require a dictionary, you can leave it blank. You must, however, *uncheck* the **Require Dictionary** box in the **Application Options** dialog. You can press the ellipsis button (...) to locate the dictionary file using the Open File dialog.
First Procedure	Type the name of the first procedure. This is usually called MAIN.
Destination Type	Select Executable, Library or Dynamic Link Library.
Help File	Optionally type the name of a Windows Help file (.HLP). You do not have to create the help file beforehand. You can press the ellipsis button (...) to locate the help file using the Open File dialog.
Application Template	The template controls code generation. You can select the default Clarion template, or choose a third party template set by pressing the ellipsis button (...), then choosing from the **Select Application Type** dialog.

See also:

How to Create a New Application File

Notes:

Create new .APP files *only* using CW. Do not copy a file (using the DOS command line, or File Manager) to a new file name, then open it in Clarion for Windows. This prevents the Application Generator from changing the internal names recorded in the file. If you need to copy and rename an .APP file, open it, then use the **File**
Save As command. (1148)

Application file names, besides being legal DOS names, must also be valid Clarion labels. The file name, 1MyApp.APP, for example, is illegal because it starts with a number instead of a character. (1304)

Module Properties Dialog

This dialog allows you to specify settings for an individual source code document file. You must first view the Application Tree in module view to access this dialog. To do so, you choose **View**  **Module** View from the IDE menu. Then press the **Properties** button to open this dialog.

Name	Allows you to specify the file name for the module.
Description	Allows you to add a short description, which appears in the Application Tree when in Module View.
Type	Allows you to choose from the **Select Module Type** dialog.
Allow Repopulate	Specifies the Application Generator may move procedures from this and other modules.
Map Include File	Allows you to specify a source code file to include in the data declarations section of the module.

New Procedure Dialog

This dialog allows you to add a new procedure to the Application Tree. Generally, you use this command to add a source code procedure, which you can then call from Embed Points accessed from other procedures.

Type a new procedure name in the dialog, then choose from the **Select Procedure Type** dialog.

Select New Dictionary Dialog

This dialog allows you to change the data dictionary for the current application.

This can be very problematical, since you must take great care to ensure that any files and fields referenced in any procedures are present in the new dictionary. Additionally, changing pre-formatting for controls, file relationships and file driver types can introduce more problems.

Therefore, the dialog box contains a warning that there are no guarantees that changing a dictionary file will work for every application.

To change the dictionary, type a new dictionary file name in the **New Dictionary** box, or press the ellipsis button (...), then select a file from the Open File dialog.

Procedure Properties

These dialogs--each is customized according to the procedure template--contain entry boxes in which you can add a text description for the procedure, or specify its source code module, plus command buttons which lead to the dialogs which allow you to customize the procedure.

Each procedure has its own custom help page which you can access by pressing the **Help** button on the **Procedure Porperties** dialog. If you are using a third party template, or a template which you wrote yourself, this help topic will appear.

Therefore, this help page only describes the essential elements of the **Procedure Properties** dialog; the controls which each procedure template builds upon.

Clarion's Template language allows the template writer to add controls to the **Procedure Properties** dialog. These controls vary from template to template. Since each template performs a different task, the template writer provides whatever controls and options are necessary to gather input from you, the developer. Most of your input is stored in template variables (Template Symbols). When generating code, the Application Generator processes the template language code, and fills in the Template Symbols with the options you specify. As it does so, it generates your application's source code.

A **Procedure Properties** dialog could have, for example, a checkbox to specify that an MDI window should save its position in the .INI file between sessions. Each template adds controls such as these to the **Procedure Properties** dialog, to gather choices from you. At code generation time, the Application Generator evaluates a symbol which stored your choice as to whether you wanted to save the MDI window position. If the checkbox was marked "yes," the Application Generator processes the template code containing the executable code to support saving the window position, and writes it to the generated source code file.

Description	A short text description for the procedure, which appears next to the procedure name in the **Application Tree** dialog. Press the ellipsis (...) button to edit a longer (up to 1000 characters) description.
Prototype	Allows you to optionally type a custom procedure prototype which the Application Generator places in the MAP section.
Module Name	The source code file to hold the code for the procedure. Select from the dropdown list. By default, the Application Generator names modules by taking the first five characters of the .APP file name, then adding a three digit number for each module.
Parameters	Allows you to specify parameter names (an optional list of variables separated by commas) for your procedure, which you can pass to it from a calling procedure. You must specify the functionality for the parameters in embedded source code.
Files	Accesses the **File Schematic Definition** dialog. You can define the procedure's access to variables or other files through the dialog.
Window	Calls the Window Formatter, to visually design the window. The ellipsis (...) button next to the **Window** button allows you to edit the WINDOW or APPLICATION structure at the source code level. Clarion for Windows allows you to easily switch back and forth between editing the window graphically, and editing the source code that describes it.

Tip: Take care when hand-editing code for any WINDOW which contains a control template. The Application Generator stores Template Language attributes which cannot be edited by

hand.

Report — Press this button to call the Report Formatter to visually design the window.

The ellipsis (...) button next to the **Report** button allows you to edit the REPORT structure at the source code level. Clarion for Windows allows you to easily switch back and forth between editing the report graphically, and editing the source code that describes it.

Data — Adds or edits local variables. Press this button and fill in the **Local Data** dialog. Any variables defined are local to the procedure. Define global variables by pressing the **Global** button in the **Application Tree** dialog.

The ellipsis (...) button next to the **Data** button allows you to view the memory variable declarations at the source code level.

Procedures — Calls procedure made in hand-coded, embedded source.

Press this button to access the **Called Procedures** dialog. To add a procedure, press the **Insert** button, and type a procedure name in the next dialog.

If procedure calls already exist, the names appear in the **Called Procedures** dialog. To add another, press the **Add** button. To delete one, press the **Delete** button. Additional buttons allow you to change the order of any procedures listed.

Tip: The purpose of the Procedure button is to add procedures called in embedded source code. The normal way to add template procedures to the Application Tree is to create a menu or toolbar command, add the procedure name via its Actions button, and let the Application Generator automatically add it to the tree.

Embeds — Displays the **Embedded Source** dialog. You can then select either a field specific event or window related action, then add executable source code to customize how the procedure will handle it.

After you choose an embed point in the **Embedded Source** dialog, you choose the code to execute. You can specify a call procedure, which is then added to the tree. You can write your own code with the text editor. Or, you can choose and customize a code template, which is a combination of pre-written code and prompts to "fill-in-the-blanks."

Embedded source gives you complete control over *all* the processing in your procedures. It's one of the most powerful tools Clarion provides you.

Formulas — Accesses the **Formula Editor**, which allows you to create computed and/or conditional fields, which you can then reference in the controls you place in your windows and reports.

Extensions — Accesses extension templates, if any are installed on your system. Extension Templates allow additional functionality through "add-ins" to the Application Generator.

Controls — If any control templates were pre-defined in the current procedure template, or were in a window or report by you, this button accesses the **Action** dialog for the control templates.

Control templates provide "off the rack" controls, such as list boxes, *and* the code

to maintain them. This allows you to start with a "bare" procedure template, such as the generic window, and add controls to create your own browse or form windows.

Edit Procedure Description

Allows you to enter string descriptions for the procedure. Clarion for Windows automatically displays the short description in certain dialogs, allowing you to quickly recognize the file contents. The long text description only appears in this dialog box, and holds up to 1000 characters.

The descriptions are solely for your convenience, and have no effect on the application. They´re useful for situations in which other programmers may pick up your code later, or for when you expect to return to the project after a long period of time since you last looked at it.

Application Tree Dialog - Procedure View

In this, its default view, the **Application Tree** dialog displays your procedures in logical call tree, nesting each procedure under its calling procedure. A procedure is a collection of instructions--Clarion language statements--which perform a task. The first procedure your application executes is called "Main" by default.

The tree controls in this dialog illustrate how the procedures branch from "Main" and from each other. This provides a schematic diagram of your program's logical structure.

The Application Tree shows the procedures you create when you add a menu item, toolbar command, or an embedded source procedure. Each new procedure is marked "To Do." When you "fill in" its functionality, the Application Tree dialog replaces the "To Do" with your description.

Global

Opens the **Global Properties** dialog, which allows you to declare, or edit the declarations of Global data.

The **Global Properties** dialog also allows you to specify your name as the program author, specify that your application store settings such as Window size and position in its own .INI file, and choose the file access modes your application will utilize when it works with its data files.

Properties

Once the procedure appears on the Application Tree, you can define its procedure type by selecting it, then pressing this button. Choose a procedure template from the **Select Procedure Type** dialog.

Once you select a procedure template, you access the other parts of the IDE to determine its functionality through the **Procedure Properties** dialog.

Application Tree Dialog - Module View

In module view, the **Application Tree** dialog displays your procedures according to the source code document which they reside in. A procedure is a collection of instructions--Clarion language statements--which perform a task. The first procedure your application executes is called "Main" by default.

The tree controls in this dialog illustrate how the procedures reside in each separate file. Within each, they branch from each other when parent and child procedures reside in the same file.

The Application Tree shows the procedures you create when you add a menu item, toolbar command, or an embedded source procedure. Each new procedure is marked "To Do." When you "fill in" its functionality, the Application Tree dialog replaces the "To Do" with your description.

Global

Opens the **Global Properties** dialog, which allows you to declare, or edit the declarations of Global data.

The **Global Properties** dialog also allows you to specify your name as the program author, specify that your application store settings such as Window size and position in its own .INI file, and choose the file access modes your application will utilize when it works with its data files.

Properties

Once the procedure appears on the Application Tree, you can define its procedure type by selecting it, then pressing this button. Choose a procedure template from the **Select Procedure Type** dialog.

Once you select a procedure template, you access the other parts of the IDE to determine its functionality through the **Procedure Properties** dialog.

Application Tree Dialog - Alphabetic View

In alphabetic view, the **Application Tree** dialog displays your procedures according to the order of the procedure names. A procedure is a collection of instructions--Clarion language statements--which perform a task. The first procedure your application executes is called "Main" by default.

Alphabetic view makes it easier to locate the precise procedure you wish to edit, when your application includes many procedures. There is also a **Find** command on the **Edit** menu, to help you locate the procedure you want when the Procedure tree is *very* long.

The Application Tree shows the procedures you create when you add a menu item, toolbar command, or an embedded source procedure. Each new procedure is marked "To Do." When you "fill in" its functionality, the Application Tree dialog replaces the "To Do" with your description.

Global

Opens the **Global Properties** dialog, which allows you to declare, or edit the declarations of Global data.

The **Global Properties** dialog also allows you to specify your name as the program author, specify that your application store settings such as Window size and position in its own .INI file, and choose the file access modes your application will utilize when it works with its data files.

Properties

Once the procedure appears on the Application Tree, you can define its procedure type by selecting it, then pressing this button. Choose a procedure template from the **Select Procedure Type** dialog.

Once you select a procedure template, you access the other parts of the IDE to determine its functionality through the **Procedure Properties** dialog.

Application Tree Dialog - Template Type View

In template type view, the **Application Tree** dialog groups all your procedures according to template type. A procedure is a collection of instructions--Clarion language statements--which perform a task. The first procedure your application executes is called "Main" by default.

The Application Tree shows the procedures you create when you add a menu item, toolbar command, or an embedded source procedure. Each new procedure is marked "To Do." When you "fill in" its functionality, the Application Tree dialog replaces the "To Do" with your description.

Global

Opens the **Global Properties** dialog, which allows you to declare, or edit the declarations of Global data.

The **Global Properties** dialog also allows you to specify your name as the program author, specify that your application store settings such as Window size and position in its own .INI file, and choose the file access modes your application will utilize when it works with its data files.

Properties

Once the procedure appears on the Application Tree, you can define its procedure type by selecting it, then pressing this button. Choose a procedure template from the **Select Procedure Type** dialog.

Once you select a procedure template, you access the other parts of the IDE to determine its functionality through the **Procedure Properties** dialog.

Select Default Dialog

If a procedure template has more than one WINDOW or REPORT structure defined, when you press the **Window** or **Report** buttons in its Procedure Properties dialog, this dialog appears.

Select the window or report you wish to use, then press the **OK** button.

Select Parent Instance Dialog

If a control template needs to attach itself to another, and there is more than one "candidate" to attach to, this dialog appears and allows you to specify which "candidate."

Select the control template you wish to associate, then press the **Select** button.

Select Destination Module Dialog

This dialog allows you to manually move a procedure from one module (source code document) to another. Select a module from the list, then press the **Select** button to move it.

Select Items to Import Dialog

This dialog allows you to choose a procedure from another .APP file to import into your current application.

You can select an item by DOUBLE-CLICKING on it. A check mark appears to indicate the item is selected. Select additional items by DOUBLE-CLICKING. De-select an item by DOUBLE-CLICKING a previously selected item.

When your selections are made, press the **Select** button to import them.

Select Items to Export as Text Dialog

This dialog allows you to choose a procedure from the current .APP file, then export it to a .TXA file for incorporation into another .APP file.

You can select an item by DOUBLE-CLICKING on it. A check mark appears to indicate the item is selected. Select additional items by DOUBLE-CLICKING. De-select an item by DOUBLE-CLICKING a previously selected item.

When your selections are made, press the **Select** button to export them.

Edit Extensions Dialog

This dialog allows you to access the properties dialog for an extension template associated with the current procedure.

In cases where "extra" prompts or controls do not default to the **Procedure Properties** dialog, you can access them through this dialog.

Select an extension template from the list, then press the **Properties** button.

Edit Control Templates Dialog

This dialog allows you to access the prompts dialog for an control template associated with the current procedure.

This is equivalent to selecting the control template in the Window Formatter, then selecting **Actions**. If, however, additional prompts from field templates, such as the actions for a button when pressed, apply, those are only available through the Window Formatter.

Select a control template from the list, then press the **Properties** button.

Embedded Source Dialog

This dialog allows you to access Embed points from the Procedure Properties dialog. You can limit the list displayed by selecting the Filled Embeds tab.

As you add procedures to your application, the Application Generator incorporates the default procedure definitions from the registry into the .APP file. It then incorporates the customizations you make to the procedures. At code generation time, it again uses the template registry, this time processing and converting the template symbols to generated executable source code, including all customizations.

The Application Generator thus handles much of the "dirty work" of building procedures from groups of executable statements. Typically, you only need to embed executable statements at key points within the procedure.

By adding embedded source code to a procedure, you gain powerful customization capability. You can specify or create code to execute before, during and after the procedure. You can write your own code, or follow the code template prompts which help write the code for you. The Application Generator adds your code to the code it generates, at precisely the point at which you specify you want to place it.

The procedure templates determine the basic points within the generated source code for these embed points. For example, any procedure with a window includes points for embedding code immediately before, and immediately after opening the window.

This dialog lists all the available embed points, as defined by the procedure. Following the button explanations, you'll find sample generated code below. Each embed point, as defined in the generic window procedure, appears as a comment in the code.

Tip: If you're unsure at what point in generated code your embedded source will execute, add a source code embed consisting only of a comment--such as the "name" of the embed point. Generate the code, then examine the source code file. It will indicate exactly where the embedded source code will execute.

These buttons appear in this dialog:

Add	Opens the **Select Embed Type** dialog, which allows you to add handwritten source code, call a procedure, and/or choose a code template.
Properties	Allows you to edit the embedded code. If it is hand written code, then the Text Editor appears. If it's a code template, the prompts dialog for the code template appears.
Delete	Allows you to delete embedded code you previously added.
Up	Moves the embedded code item up above another. Each executes in the order they appear at an embed point.
Down	Moves the embedded code item down below another. Each executes in the order they appear at an embed point.

See also:

How to Add and Customize a Procedure

How to Add Embedded Source Code

Edit Default Procedures Dialog

The template registry allows for multiple starting points for a procedure template. For example, you could have two browses, one of which you wish to use most of the time, and the other, some of the time.

This dialog allows you to set the default. It also allows you to add an alternate procedure.

Add	Allows you to name the alternate, then set its properties in the Procedure Properties dialog.
Properties	Opens the Procedure Properties dialog for the selected procedure.
Delete	Deletes the selected procedure.

Template Registry Dialog

Template files (*.TPL) drive the Application Generator. Each procedure template contains generic or "model" code. The templates are interactive--they process the information you specify when you design the application within the IDE. Clarion evaluates the template file twice:


Before creating your application, Clarion pre-processes the template file and stores the information in the *REGISTRY.TRF* file. Pre-processing occurs only when the Application Generator detects a new or changed template.

When it pre-processes the template file, the Application Generator stores a list of all the information you must provide each procedure. It also determines the points at which you can embed your own Clarion source code to customize a procedure. The registry file contains the default windows, dialogs, menus, report designs, default data, and formulas. In the design process, you customize these defaults.


At code generation time, the Application Generator evaluates the information you provide in the design process--from the data dictionary, and the .APP file--then processes it along with the template language statements and symbols in the *REGISTRY.TRF* file to generate your source code.
Each template can contain multiple types of procedure templates from which you select to create the procedures in your application. Before you can use a template it must be in the Template Registry.

The **Template Registry** dialog provides command buttons for file maintenance options for the registry:

Register — Calls the Open File dialog, which allows you to register a template (.TPL) file.

UnRegister — This button deletes the currently highlighted template class from REGISTRY.TRF.

Enable — This button enables the currently highlighted template class or procedure (if you had previously disabled it).

Disable — This button disables the currently highlighted template class or procedure, which makes it unavailable to your application.

ReGenerate — This button regenerates the .TPL file for the currently highlighted template class.

The Template Registry works two ways. Besides allowing you to add procedures to your applications, you can customize the procedures in the template. For example, you can add more default window types to the a template. See the following button description, **Properties**, for instructions on how to edit the template.

When you change the properties for the your template, the changes are stored in your *REGISTRY.TRF* file. The **ReGenerate** button reads the file, then rewrites the .TPL file which contained the original template language code. This allows you to customize a template, then give a copy of the changed .TPL file to another programmer to register.

Properties — This button accesses the **Template Procedure** dialog. Press the Defaults button to edit default global data or structures contained in the procedure template.

View Definition — This button displays the Template code (the .TPL) in a text window. You may not edit the code in this window.

If the currently highlighted item in the Template Registry tree is a module, the text window opens to the first line of the MODULE definition. If a procedure, it opens to the first line of the PROCEDURE.

Control Prompts Dialogs

The Clarion template language allows the template author to create custom dialogs. When no custom help is available for a control's Actions, this help topic appears.

To access help for a template-created dialog, press the Help button instead of F1.

The name of the dialog takes the form "Prompts for" plus the field equate label, as in "Prompts for ?MyButton." The following sections will substitute the control type for the field equate label, helping you to find the section you need more easily.

Project Tree Dialog

The **Project Tree** dialog organizes all the components, and provides access to the dialogs that choose additional options.

The Project file tracks all the components that make up the final executable file. It also sets the compiler options ranging from whether to include debug code or not, to setting a preferred optimization method.

If you use the Application Generator to create your source code, the only thing you will probably use the Project System for is to set debugging options.

Tree List

The Tree List itemizes the file level elements which comprise your project, including source code files, file drivers, other projects to compile, external libraries and resources, and other programs to execute as part of the make process.

Properties

Calls a dialog allowing you to specify compile options for the selected item, or the entire project. The particular dialog which appears depends on the currently selected item.

When you select a "folder level" item (such the project itself), the **Global Options** dialog appears. This allows you to set compile options for the project.

When you select a source code file, the Compile Options dialog appears. This provides the same options as the Global Options dialog; however, the selections apply only to the selected file.

Add File

Calls the Open File dialog, allowing you to add a file, such as a source code file, below the currently selected item.

By inserting a .PRJ file after the **Projects to Include** item, you can specify that the other project should be built in the course of building the current project.

For hand coded applications, you can insert new .CLW files after the **External Source Files** item.

By inserting a.LIB file after the **Library and Object Files** item, you specify that the Project System should link it in to the project. By inserting a .DLL file, you specify that the application dynamically calls external functions from the file at run time. See also: Using DLLs not created in Clarion for Windows

You can also add external resources, such as .CUR, or .ICO files if they were not explicitly named as attributes in your source code. For example, if you specify variable naming a bitmap file in an IMAGE control (such as !MyBMP.BMP), you can link it in by adding it to the Project Tree below this item.

By inserting an executable file (*.exe, *.com, *.bat, or *.pif) after the **Programs to execute** item, you can run another application upon completion of the compile. This can be useful, for example, in network operations where you need to remap a drive after the compile, but before running the compiled application.

Remove File

Allows you to remove the currently selected item from the Project Tree. This does not physically remove the file from disk.

Edit

For hand-coded projects, if a source code file is selected, calls the Text Editor and loads it into a source code document window.

Make Statistics Dialog

This dialog displays the creation statistics of your latest compile.

For each object file created, it lists the object file name, source code file, size of the code and data segments, and the current date for each.

New Project File Dialog

This dialog allows you to type in the basic information the Project System needs to create a new project file (.PRJ) for you.

Project Title — Allows you to type in a short text description which displays at the top of the Tree List in the Project Tree dialog.

Main File — Type in (or select with the Open File dialog after pressing the ellipsis button) the name of the main source code file.

Target File — Type in (or select with the Open File dialog after pressing the ellipsis button) the name of the target file (such as MyFile.EXE).

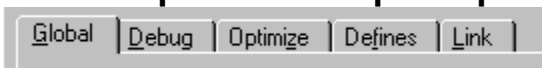
Project File — Type in (or select with the Open File dialog after pressing the ellipsis button) the name of the project file.

If you want to specify a working directory other than the one with which you started up the development environment, navigate the directory tree using the Open File dialog. When you've selected the directory you wish, type the project file name in the File Name box in the Open File dialog.

The Project File must be in the same directory as the main source file.

Target Type — Specify .EXE, .LIB or .DLL from the drop down list. The target file name will automatically add the correct extension.

Global Options /Compile Options Dialogs



Click on a TAB to see its help

These dialogs allow you to set compile options for the project, or its components. The options vary according to the item selected in the Project Tree dialog at the time you press the **Properties** button.

When you select a "folder level" item (such the project itself), the **Global Options** dialog appears. This allows you to set compile options for the project.

When you select a source code file, the **Compile Options** dialog appears. This provides the same options as the **Global Options** dialog; however, the selections apply only to the selected file. The **Compile Options** dialog also does not allow you to select the Build Mode, since that applies to the entire project.

The **Global Options** and **Compile Options** dialog contain the following options.

Global

Title — A short text description of the project. The Project System will list the description next to the Project name in the Project Tree list.

Target Type — Specify the type of executable file: choose **.EXE, .LIB**, or **.DLL** from the **Target Type** drop down list.

Target OS — Identify the type of operating system the application will run under: choose Windows 16 bit or Windows 32 bit from the **Target OS** drop down list.

Note: You can compile and link 32-bit executables with Windows 3.1 if you have Win32S installed, but you must have Windows 95 or Windows NT to run them.

Memory Model — Not implemented in this release, accept the default.

Run-Time Library — Specifies how the runtime library is called by the target file: choose ***Standalone***, ***Local***, or ***External*** from the **Run-Time Library** drop down list.

- ***Standalone*** — Uses the CWRUNxx.DLL runtime library (and database driver(s) .DLLs). In 16-Bit mode, it is called CWRUN16.DLL; in 32-bit mode it is called CWRUN32.DLL.
- ***Local*** — Links the runtime library and any database drivers into your executable using Smart Method Linking (only the necessary portions are linked in). This creates a "one-piece" executable.
- ***External*** — Specifies that another External DLL contains the runtime libraries and database drivers. The calls to this DLL must be exported.

Build Release System — To create an executable for release, check this box. To create an executable for use with the Debugger, uncheck this box.

Debug

Debug Mode Specifies the level of debug capability, choose ***Off***, ***Min***, or ***Full*** from the **Mode** drop down list.

Line Numbers Builds line numbers into the object file. This is not necessary for the Clarion debugger, but may be helpful when using other debuggers.

Stack Overflow Enables stack overflow warnings at runtime.

NIL-Pointer Allows compiler warnings when dereferencing null pointers.

Array Index Enables "array index larger than the array size" warnings at runtime.

Optimize

CPU Specify optimization by microprocessor type: choose from **286**, **386**, **486,** or **Pentium**.

Optimize for Speed

To favor program speed over creating a smaller executable file, check the **Optimize for Speed** box.

Defines

Defines To define a switch, or switches, for use with the COMPILE and OMIT compiler directives, type a list of valid Clarion labels separated by commas. Each label defines a separate switch.

For example, type 'Demo' in the **Defines** field. The Project System will create a switch called Demo and turn it "on." Now you can use the switch in conditional COMPILE and OMIT statements within your source code. For example:

```
COMPILE('END COMPILE',DEMO=ON)
  IF TODAY() > FirstRunDate + 30
    #ReturnCode = MESSAGE('Beta period expired')
    RETURN
  END
END COMPILE
```

Link

Create Map File Creates a map file, which contains information about segment sizes and public functions. The map file may be used with third party debuggers.

Pack Segments To pack the data and program segments in the .EXE file, check this box.

Stack Size To specify the stack size, type a value in Kilobytes in the **Stack Size** field.

Create New Project

This dialog allows you to create a new project file. Mark the radio button for the type of project you wish to create.

Quick Start Calls the Quick Start Wizard.

Application Generator Calls the Application Properties dialog.

Hand Coded Project Calls the New Project dialog.

Working Directory Allows you to specify the directory where the new project will be created.

Formula Dialog

This dialog lists all formulas already created for a procedure, along with their template classes. It allows you to add or edit formulas.

If any formulas already exist for the procedure, this dialog appears when you push the Formulas button in the **Procedure Properties** dialog.

Select	Loads the currently selected formula into the **Formula Editor** for editing.
New	Loads the Formula Editor, ready to create a new formula.
Delete	Deletes the currently selected formula.
Formula	Lists all existing formulas within the procedure.
Class	Lists the template class associated with the formula. A formula's class determines when its calculation is performed. Each template has its own set of classes. For example, in the Form Procedure Template there is a class called "After Lookups" which tells the Application Generator to compute the formula after all lookups to secondary files are completed for the procedure.
Description	A short text description of the formula.

See also:

How to Create a Simple Assignment Expression

How to Create a Complex Assignment Expression

Formula Editor Dialog

The **Formula Editor** dialog provides access to fields defined in the file schematic, as well as global or local variables, and facilitates creating syntactically correct expressions.
To create an expression, you press buttons to add components to the Statement line. You can also type in your expression, and check the syntax upon completion.

Name	A descriptive label for the function.
Class	A formula's class determines when its calculation is performed. Each template has its own set of classes. For example, in the Form Procedure Template there is a class called "After Lookups" which tells the Application Generator to compute the formula after all lookups to secondary files are completed for the procedure. Press the ellipsis (...) button next to the field to view the list of available template classes in the **Template Classes** dialog.
Description	A short text description for the formula.
Result	The variable to which the value of the expression is assigned at run time. Press the ellipsis (...) button next to the field to view the **File Schematic Definition** dialog, in which you can select a previously defined variable.
Statement	The actual expression under construction.
Check	Tests and validates the expression under construction. A check box appears if the expression is syntactically correct. An "X" appears if not.
Information	Describes the currently selected component in the **Statements** box.
Operators	Provides buttons for inserting logical and bitwise operators into the expression. You can also type them in directly.
Data	Accesses the **File Schematic Definition** dialog, so that you can utilize a previously defined variable or field as an operand within the expression.
Functions	Access a list of built-in Clarion functions in the **Functions** dialog.
User	Accesses user defined functions within the application under development, displaying them in the **User Function** dialog.
Conditionals	Accesses the **Conditional Dialog**, which allows you to create a conditional expression.

See also:

See also:

How to Create a Simple Assignment Expression

How to Create a Complex Assignment Expression

Conditional Dialog

A conditional field is a computed field with multiple possible expressions. There are two types of conditional fields--IF structures and CASE structures. The assignment statement executed depends on the evaluation of the IF or CASE condition. For example, an IF structure conditional field called Tax could be 0 if Taxable is FALSE, or Price times TaxRate if Taxable is TRUE.

The Formula Editor allows you to create a conditional expression whose result can then be assigned to a variable. Name your conditional formula in the **Formula Editor** dialog, then press the **Conditionals** button to open this dialog.
Each portion of the expression is edited separately. The components appear in the **Structure** list in the lower portion of the dialog box. Select a component, then edit it in the Statement box. You can add and/or nest IF and CASE structures by pressing the **IF THEN** and **CASE OF** buttons.

Statement	A currently selected component (displayed in the **Structure** list) of the actual expression under construction.
Information	Describes the currently selected component in the **Statements** box.
Check	Tests and validates currently selected component of the expression under construction. A check box appears if it is syntactically correct. An "X" appears if not.
Accept	Adds the currently selected component of the expression to the **Structure** list.
Structure	Lists the components of the expression in a hierarchical list. Each item selected can be edited separately.
Operators	Provides buttons for inserting logical and bitwise operators into the expression. You can also type them in directly.
Data	Accesses the **File Schematic Definition** dialog, so that you can utilize a previously defined variable or field as an operand within the expression.
Functions	Access a list of built-in Clarion functions in the **Functions** dialog.
User	Accesses user defined functions within the application under development, displaying them in the **User Function** dialog.
IF THEN	Adds and/or nests and IF THEN structure to the expression.
CASE OF	Adds and/or nests a CASE OF structure to the expression.

See Also:

See also:

How to Create a Simple Assignment Expression

How to Create a Complex Assignment Expression

Template Classes

Procedure Setup --Upon Entry into the Procedure

This point occurs immediately after the CODE statement, allowing you to initialize values upon entering a procedure.

Before Lookups--Refresh Window ROUTINE, before lookups

This occurs before any lookups to related records, allowing you to prime any key values needed to perform the lookups.

After Lookups--Refresh Window ROUTINE, after lookups

This occurs immediately after looking up related records, allowing you to use values retrieved from related records in your computation.

Procedure Exit--Before Leaving the Procedure

This allows you to assign values before returning to the calling procedure.

Prime Fields--Prime Fields of the Primary File record at beginning of Insert

Available when a Save Button control template is used, this allows you to pre-assign values to fields when inserting a new record.

Before Filter Check--In Validate Record ROUTINE, Before Filter Code

Available when a BrowseBox control template is used, this allows you to create a formula to be used in the filter expression.

Before Range Check--In Validate Record ROUTINE, Before Range Limit Code

Available when a BrowseBox control template is used, this allows you to assign values before range limits checks are made.

Format Browse--Format a variable in the Browse Box

Available when a BrowseBox control template is used, this allows you to compute values to display in the list box.

Before Print Detail--Before Printing Report Detail

Available only when the Report template is used, this allows you to compute values before a sending a detail structure to a report.

Quick Start Wizard and Quick Load Wizard

Using the Quick Start Wizard, you can create a data dictionary and a working application with no coding required.

Simply define a data file, and the Quick Start Wizard creates a complete Windows application--in about five minutes if you're a fast typist! Your application has a form procedure for updating the file, and as many view windows and reports as the data file has keys.

Just define the fields for a single file. For each field, you provide a name, display format picture, and key information. This creates a data dictionary. The Quick Start Wizard creates the application based on this dictionary. Once you've specified all options, the **OK** button generates the .APP file, and loads the procedures into the **Application Tree** dialog.

The Quick Load Wizard is similar to the Quick Start Wizard; the only difference is that its function is exclusively to create a data file definition as an addition to an existing data dictionary. After creating the file definition, you can use one of the Procedure Wizards to create procedures using the file.

You can call the Quick Load Wizard by pressing the **Add File** button in the **Dictionary** dialog. Once you've specified all the options, the **OK** button adds a new file definition to the **Dictionary** dialog, complete with **Field/Key Definitions**.

Application Name — Type a legal DOS file name for the .APP file. The Quick Start Wizard will use the same file name (with the .DCT extension) for the data dictionary file.

Optionally press the ellipsis button (...) to change the directory, and type a file name in the Open File dialog box. The working directory, in which all source code files will be generated, depends on where the .APP file resides.

Because the Quick Load Wizard does not create the .APP file, this control is not present in the Quick Load Wizard.

Data File Name — Type a legal DOS file name (no extension necessary) for the data file.

Prefix — This box automatically fills in with the first three letters of the name of the data file when you TAB away from the **Data File Name** box. Optionally specify up to three letters of your choice in this field.

The prefix allows your application to distinguish between similar variable names occurring in different file structures. A field called *Invoice* may exist in one data file called *Orders* and another called *Sales*. By establishing a unique prefix for *Orders* (ORD) and *Sales* (SAL), the application may distinguish the two fields as ORD:INVOICE and SAL:INVOICE.

File Driver — Specify the data file type. When using the Application Generator, Clarion for Windows automatically links in the correct database file driver library. See the *Database Drivers* topic for a discussion of the relative advantages of each driver.

Remember that individual file drivers may vary in their support of some of the attributes which you add to the FILE structure in this dialog box.

Field Name — To name each field, type a valid Clarion label in the **Name** field. Valid field names may vary slightly according to the file driver.

The Quick Start Wizard allows you to name each field, one by one, by pressing the DOWN ARROW to add a new item to the list. Before naming the next field, specify the **Picture** and **Key** options for the current key.

Picture

Specify a default picture token by typing it in the **Picture** field. The picture token, together with the selected **File Driver**, determine the data type which the Quick Start Wizard uses for the field. When the Application Generator creates window and report controls for the field, this also serves as the default picture for the control.

The Quick Start Wizard allows you to name each field, one by one, by pressing the DOWN ARROW to add a new item to the list. Before naming the next field, specify the **Key** option for the current key.

Key

This specifies whether to create a key using this field as a component, and if so, the type of key. By specifying **Unique**, your application will ensure that each record has a distinct number. **Duplicate** specifies a key that allows more than one record with the same value in the key component.

The Quick Start Wizard creates a multi-keyed browse procedure and reports for every key you specify.

The Quick Start Wizard allows you to name each field, one by one, by pressing the down arrow to add a new item to the list. Press the DOWN ARROW, or TAB, to define the next field.

Insert

This button allows you to insert a new, blank field, above the currently selected field.

Delete

This button allows you to delete the currently selected field.

Move Up

This button allows you to move the currently selected field up one position in the fields list.

Move Down

This button allows you to move the currently selected field down one position in the fields list

Pick File Dialog

The Pick dialog is a specialized *Most Recently Used Files* list. As you begin using Clarion for multiple projects, you'll appreciate this dialog because it quickly locates the files you need for any given project.

The Database Manager's **Pick** dialog lists the data files most recently opened for "browsing."

When you choose any of these options, a pick list dialog appears, listing up to twenty of the most recently used files of that type:

The **Pick** dialog provides the following buttons:

Select	Opens the currently selected file.
Remove	Removes the currently selected file from the Pick list.
New	Allows you to create a file.
Open	Allows you to open a file not on the Pick list.
Type	Allows you to change the type of files listed in the **Pick** dialog.

Field Picture Dialog

This dialog allows you to specify a new picture token for the currently selected field.

This reformats the way the data displays on screen. This does not alter the data in any way, only the manner in which it is displayed.

Justify Dialog

This dialog allows you to specify a new justification style for the currently selected field. This reformats the way the data displays on screen.

Choose the style from the drop down list. Depending upon the field selected, you may choose from Left, Center, Right and Decimal. This does not alter the data in any way, only the manner in which it is displayed.

Reformat Fields Dialog

This dialog allows you to change the field order in the window, and to hide or unhide fields from view.

The **Shown** list, on the left, lists the fields in the current view. the **Hidden** list, on the right, shows fields *not* in the view. After selecting the fields to hide, show, or move, then pressing the **OK** button, the view window displays the fields you want, in the order you want.

Up	Moves the selected field one position *up* the **Shown** list. This rearranges the view window, moving the field one column left.
Down	Moves the selected field one position *down* the **Shown** list. This rearranges the view window, moving the field one column left.
Hide All	Hides all fields, moving them from the **Shown** list to the **Hidden** list.
Show	Moves the selected field from the **Hidden** list to the **Shown** list.
Show All	Moves all fields in the **Hidden** list to the **Shown** list.

Query by Example Dialog

This dialog allows you to filter the data file, then display only the records that meet the criteria you specify by entering example values or expressions in this dialog.

Type the example value in the list box at the top of the dialog, in the column you wish to test. For example, if you want to show all the records where the value of the "Apples" field equals "1," type "1" directly below the "Apples" column header.

To create a query that has the effect of using the AND operator, type a second test value in another field in the same row as the first test value. If, for example, you type "2" directly below the "Cherries" column header, you show all records where the values of the "Apples" field equals "1," and the "Cherries" field equals "2."

To create a query that has the effect of using the OR operator, type a second test value in another row. If for example, you type "3" in the "Apples" column, one row below the "1" in the first query (with no value specified for "Cherries"), you show all records where the values of the "Apples" field equals "1" or "3."

The actual filter expression displays in the group box below the listbox as you enter values or logical expressions in the listbox. For example, to find all records with an ID number between 10 and 100, with a last name of Smith or Smythe, you create a query:

IDNumber	*FirstName*	*LastName*
>10&<100		='Smith'
>10&<100		='Smythe'

Use the ampersand character (&) to represent the AND operator and the vertical bar (|) to represent the OR operator when used in the same field. The example above can also be represented in this fashion:

IDNumber	*FirstName*	*LastName*
>10&<100		='Smith' | ='Smythe'

Both examples produce a filter expression of (IDNumber > 10 OR IDNumber < 100) AND (LastName = 'Smith' OR LastName = 'Smythe'). The expression displays in the **Filter Expression** group box.

Tip: Although the expression created in a query is not optimized, the runtime evaluator performs its own optimization.

Press the **OK** button to execute the query and display the filtered records in the view window.

Send Driver String Dialog

This allows you to execute a SEND command to the file driver. Type the driver string in the edit box, and press **OK**.

See *Database Drivers* for complete information on the SEND commands for each driver.

Edit Memo Dialog

This dialog allows you to edit a memo in ASCII Text. If the file has more than one memo field, you must first select the memo you wish to edit from a list box.

Edit the memo in the text box, then press the **OK** button to return to the view window.

Hex Edit Memo Dialog

This dialog allows you to edit a memo in Hexadecimal format. This is necessary for editing binary format memos. If the file has more than one memo field, you must first select the memo you wish to edit from a list box.

Each character value appears in Hexadecimal format in its own edit box inside the list box. CLICK on the character you wish to change, and type in a new value.

Press the **OK** button to return to the view window.

Export File Dialog

This dialog allows you to save a FILE definition for the current data file. You can copy the definition into your source code.

File Label Type a valid Clarion label for the FILE structure.

Source Filename Type a DOS file name to save the definition to.

File Convert Dialog

This dialog allows you to convert the records in an existing data file to a new file format. When you modify a data dictionary and application, you can use the conversion utility to convert your existing data to the modified format.

The method you use to call the file conversion utility affects its behavior. If you open the converter through the Dictionary Editor (with the appropriate .DCT file open) the converter uses all the information in the dictionary. If start you open a file from any other area, only the information stored in the file header is available. This offers maximum flexibility--allowing you to browse a file without the need for a .DCT.

The information stored in a file header varies according to the file driver.

There are two methods of converting a file--immediate conversion and Generate Source. Immediate conversion converts the file once. **Generate Source** creates a source code file, allowing you to make any desired modifications before compiling. Generating and compiling source also creates an executable file that you can ship to end users.

Before conversion, the utility makes backup copies of the data file and its associated index and memo files. If the conversion process is interrupted, these backup files are renamed to their original names. If you specify a target filename that differs from the original, then the original files are not renamed and are left in place.

The items you specify in this dialog's entry fields controls the conversion.

Source Filename	Specifies the file to convert. This defaults to the file opened by the Database Manager.
Source Dictionary	Specifies the dictionary which contains the file definition for the source data file. If the file conversion utility was invoked from a data dictionary, this defaults to the current dictionary. A Source Dictionary is not required.
Source Structure	Specifies the structure (within the dictionary) which defines the source file. If the file conversion utility was invoked from a data dictionary, this defaults to the current file definition. A Source Structure is not required.
Target Filename	Specifies the name of the new file. This defaults to the current filename.
Target Dictionary	Specifies the dictionary which contains the file definition to which to convert. A Target Dictionary is required.
Target Structure	Specifies the structure (within the dictionary) of the target file. The Target Structure is required.
Generated Source	The filename for the source code which will create an executable file to change the database. When converting a file, if you want to make any field assignments edit the source code before compiling and executing.

Tip: If you change the name of a field, generate source code, and edit the source code to make the field assignments. Otherwise, your data will be lost. See How to Make a Field Assignment .

For immediate file conversion, without generating source code, see How to Convert a File (without generating source) .

Select File Order Dialog

Once a file is open, you can change the sort order by specifying a different key. This dialog displays a list of available keys, and allows you to change the active key.

Select the key which matches the desired sort order (or Record Order) from the key list, then press the **OK** button.

The file is displayed in the selected sort order, and ready for any Database Manager operation.

File Statistics Dialog

This dialog allows you to examine the file statistics, but not to change them.

Filename	The DOS filename and PATH for the data file.
File Driver	The database driver the file uses.
File Attributes	CREATE, RECLAIM, and ENCRYPT attributes.
Record Length	The size of each record.
Total Number Records	The total number of records in the file (including deleted records).
Number Active Records	The total number of active records.
Deleted Records	The total number of deleted records.
Fields and Field Layout	The number of fields in the file. Pressing the ellipsis (...) button displays the field layout.
Keys and Components	The number of keys in the file. Pressing the ellipsis (...) button displays the key components.
Memos and Layout	The number of memos in the file. Pressing the ellipsis (...) button displays the memo field layout.
Indexes & Components	The number of indexes in the file. Pressing the ellipsis (...) button displays the index components.

Field List Dialog

This dialog allows you view the basic data for all fields in the data file. You can view the data, but not change it.

The information includes the field label, data type, size, digits, places, and whether the OVER attribute is specified.

Search Dialog

This dialog allows you to search for the first record containing a value you specify. You may limit the search to one field, or all fields.

Search	Type the search value.
Exact match	Searches for values that match the specified search string exactly.
Starts With	Searches for values that begin with the specified search string.
Contains	Searches for values that contain the specified search string.
Ends With	Searches for values that end with the specified search string.
Case Sensitive	Specifies case sensitive search testing.
All Fields	Specifies searching all fields in the data file. If not specified, the search is on the currently selected field.

Locate Dialog

This dialog allows you to search for the first record containing the value you specify in the key field(s). This is only possible when the data file is displayed in a keyed sequence, *not* in Record Number order.

This command only searches fields which are components of the selected key. To search other fields, use the **Search** command.

Print Dialog

This dialog allows you to print a record or records.

Current Record	Prints only the currently highlighted record.
Current Page	Prints only the records currently displayed on screen.
All Records	Prints all records in the file.
Use Filter	Prints only those records which match the filter created in the **Query-by-Example** dialog.
Columnar Mode	Prints the records in a "spreadsheet" type of format in which each field in the record is a separate column. Specify the number of records to print side by side in the **Columns** box.
Tabular Mode	Prints the records in a "form" type of format in which each field in the record is on its own separate print line. Specify the maximum width of each field in the **Table Width** box.
Print Header	Specifies whether to print column headers in the report.

Select Driver Dialog

When first loading a file, the Database Manager prompts you to name the driver used to read the file. Select a previously installed Clarion for Windows database driver from the list.

See also: Supported File Systems

Select Memo Dialog

If your data file has more than one MEMO, this dialog appears to allow you to select the MEMO to edit. Highlight the desired MEMO, then press the **Select** button.

Select Control Template

This dialog allows you to choose a control template, adding functionality to a procedure.

CLICK on a control template from the list, then press the **Select** button.

If you add third party, or your own customized templates to the Template Registry, they appear in the list. The following lists the control templates which ship with Clarion for Windows:

Accept Button	This control template provides a convenient way to close a procedure.
ASCII Box	This control template adds a list box in which you can display an ASCII (text) file.
ASCII Print Button	This control template adds a button to print an ASCII (text) file.
ASCII Search Button	This control template adds two buttons (Find and Find Next) to search an ASCII (text) file.
Browse Box	This control template places a LIST control in a window.
Browse Select button	This control template provides a quick way to return a value from a list box called to request a record.
Browse Update buttons	This control template provides a quick way to manage records in a list box.
Cancel Button	This control template provides a convenient way to close a browse procedure and cancel a record request.
Close Button	This control template adds a single button control marked **Close** which closes down the current window.
DOS File Lookup	This control template adds an ellipsis (...) button which leads the end user to a standard Open File dialog.
Field Lookup Button	This control template adds an ellipsis (...) button to call a lookup procedure specified for an entry control.
File Drop	This control template adds a drop down list showing the contents of a selected field of a file listed in the data dictionary.
File Drop Combo	This control template adds a Combo Box with drop down list showing the contents of a selected field of a file listed in the data dictionary. It also allows updates to the file.
RelationTree control template	This control template places a LIST control formatted as a tree in a window.
Relation Tree Update Buttons control template	Adds buttons to a window which call the appropriate update procedures for levels in a Relation Tree.
Save Button	This control template adds an OK button to close a window and save the action.

Select Code Template

Code templates generate executable code. The purpose is to make customizationadding embedded source code fragments that do exactly what you want it toeasier. Each Code template has one well-defined task. For example, the Initiate Thread Code template simply starts a new execution thread, and no more. Typically, the Code template provides a dialog box with options and instructions.

CLICK on a code template from the list, then press the **Select** button.

If you add third party, or your own customized templates to the Template Registry, they appear in the list. The following lists the code templates which ship with Clarion for Windows:

Initiate Thread This code template initiates an execution thread when opening an MDI window.

Call Procedure As Lookup
This code template allows you to call a procedure, usually a Browse, with a request to make a selection.

Control Value Validation
This code template validates the value of an entry control (ENTRY, LIST, COMBO, or SPIN). You can add this code template to a field event on a control; at the Accepted or Selected embed point.

Lookup Up Non-Related Record
This code template is used to perform a lookup of a value based on a relationship not defined in the Data Dictionary (ad hoc relations). You can add this code template to the Lookup Up Related Records embed point.

Close Current Window This code template simply posts an EVENT:CloseWindow, which tells the currently active window to close.

Select Utility Dialog

A Utility template allows you to produce output from your application. These templates can provide extensible supplimental utilities for such things as wizards, program documentation, or a tree diagram of procedure calls.

Highlight the desired utility template, then press the **Select** button.

Clarion for Windows provides *WIZARDS*powerful utility templates that enable you to create a Browse, Form, or Report procedure by merely answering a few quick questions. You can even use a wizard to create an entire Application from an existing dictionary!

Options you specify in advance in the Data Dictionary provide additional control over the procedures the wizards create. See **Using Wizard Options** for more information.

Application Wizard utility template Creates a complete application from an existing dictionary.

Browse Wizard utility template Creates a multi-keyed browse procedure from an existing dictionary file.

Form Wizard utility template Creates an update procedure from an existing dictionary file.

Report Wizard utility template Creates multi-keyed report procedures from an existing dictionary file.

Select Procedure Template

This dialog allows you to choose a procedure template, adding functionality to any new or "To Do" procedure in the **Application Tree** .

CLICK on a procedure template from the list, then press the **Select** button. Once you select a procedure type, you can customize it using its **Procedure Properties** dialog.

If you add third party, or your own customized templates to the Template Registry, they appear in the list. The following lists the procedure templates which ship with Clarion for Windows:

Browse	Browse fields in a page-loaded list box
Form	View/edit a record from file
Frame	Multiple document main menu
Menu	Single document menu
Process	Sequential record processor
Report	Generic reporting procedure
Source	Source procedure
Viewer	View an ASCII text file
Window	Generic window handler
External	A procedure contained in an external library (*.LIB only) or object file

Select Extension Template

Extension templates add functionality to procedures, but are not bound to a control or embed point. Each Extension template has one well-defined task. For example, the Date Time Display enables you to display the date and a running clock.

If you add third party, or your own customized templates to the Template Registry, they appear in the list.

From a **Procedure Properties** dialog, add an Extension template by pressing the **Extensions** button. CLICK on an extension template from the list, then press the **Select** button.

Clarion for Windows contains the following Extension templates:

Date Time Display	This extension adds a "live" date and/or time (updated every second) display to the procedure
Record Validation	This extension enables enforcement of dictionary-defined field value validation

Select Application Template

If you've added third party or your own templates to the template registry, and they include a new Application template, this allows you to choose which template set controls source code generation.

CLICK on an item from the list, then press the **Select** button.

Select Program Template

If you've added third party or your own templates to the template registry, and they include a new default program template, this allows you to choose which template class controls source code generation.

CLICK on an item from the list, then press the **Select** button.

Select Module Template

If you've added third party or your own templates to the template registry, and they include a new default module template, this allows you to choose a module template.

CLICK on an item from the list, then press the **Select** button.

Generated Source	Source File created in Application Generator
External LIB	External Library Module
External OBJ	External Object Module

External DLL

Control Value Validation code template

This code template gets the value of the control and matches it against the value in the key.. You can add this code template on an ENTRY, SPIN, LIST, or COMBO control; at the Accepted or Selected embed point. The code generated by this code template gets the value in the control, then matches it against the value in the key.

It can also call a lookup procedure, to let the end user select a value. You can check whether the end user has successfully completed the lookup procedure by checking the value of the LocalResponse variable.

See also: Request and Response

InitiateThread code template

When opening an MDI window from an Application Frame, you must initiate an execution thread. This Code template provides an easy way to initiate a thread (see START).

In the **Prompts for Initiate Thread** dialog, simply name the procedure that opens the MDI window.

You can optionally add a line of code to execute if the application was unable to open the thread. Type in the edit box labelled **Error Handling**. For example,

```
BEEP; MESSAGE(Could not Start Thread,Error,ICON:HAND)
```

would beep and display a message box with the halt (hand) icon, if the thread failed to start.

You can add a procedure name to call upon an error by typing the name of the procedure in the **Error Handling** box. You would then add the procedure to the **Application Tree** with the **Insert Procedure** command.

Lookup Non-Related Record code template

This Code template is used to perform a lookup of a value based on a relationship, whether it is or is not defined in the data dictionary (Ad hoc relation). You can add this Code template to the Lookup Up Related Records embed point.

Lookup Key Type in the key name or press the ellipsis (...) button to select the key from the File Schematic.

The lookup key is used to perform the lookup into the lookup file. This *must* be a unique key. If the key is a multicomponent key, the other key elements must be primed before executing this Code template.

Lookup Field Type in the field name or press the ellipsis (...) button to select the field from the Component list.

The Lookup Field must be a component of the Lookup Key. This is the unique value within the lookup file.

Related Field Type in the related field or press the ellipsis (...) button to select it from the File Schematic.

The Related Field provides the unique value used to perform the lookup.

This code template generates the following code:

```
LookUpField = RelatedField      ! Move value for lookup
GET(LookUpFile,LookUpKey)           ! Get value from file
IF ERRORCODE()                    ! IF record not found
   CLEAR(LookupfileRecord)        ! Clear the record buffer
END                               ! END (IF record not found)
```

See also:

Refresh Window routine

Call Procedure As Lookup code template

This Code template calls a procedure to select a record. It sets a variable called RequestCompleted to advise whether the lookup was successful.

Lookup Procedure
Specifies the procedure to call.

Code before Type in any executable code to execute before performing the lookup. Multiple statements can be used if separated by a semicolon.

Code After, Completed
Type in any executable code to execute after completing a lookup. Multiple statements can be used if separated by a semicolon.

Code After, Canceled
Type in any executable code to execute if the lookup is canceled. Multiple statements can be used if separated by a semicolon.

Close Current Window code template

This code template simply posts an EVENT:CloseWindow, which tells the currently active window to close. There are no prompts to fill in.

ASCII Box control template

This Control template adds a list box in which you can display an ASCII (text) file. If you wish to view the same ASCII file all the time, you can specify a file name in the **Prompts** dialog.

The **Actions** tab contains the following:

Description Allows you to add a description and to display in the progress window which displays when opening the file.

File Name to View Specifies the path and name of the file to view, or a variable preceded by an exclamation point (!).

Display Number of Bytes Read
Check this box if you want to display the files size in the progress dialog.

Warn the user if the file cannot be found?
Check this box if you want to display a message at runtime if the specified file cannot be found.

See also:

ASCII Print Button

ASCII Search Button

ASCII Print Button control template

This Control template adds a button named Print, and the underlying code necessary for printing an ASCII (text) file. Use this control template together with the ASCII Box control template.

Edit the **Actions** only if you wish to add another, separate action to take place *after* printing. All the code necessary for managing the print job itself is handled automatically.

The **Actions** tab contains the following:

When Pressed	The standard set of prompts for buttons.Normally, when using a Control template, these prompts are not used.

ASCII Search Button control template

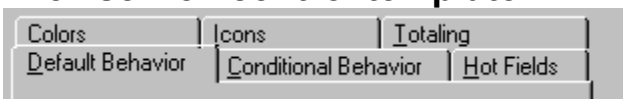
This Control template adds two buttons named Find and Find Next, and the underlying code necessary for a modal search dialog, allowing the end user to find text in an ASCII (text) file. Use this control template together with the ASCII Box control template.

Edit the **Actions** only if you wish to add another, separate action to take place *after* the search. All the code necessary for managing the search itself is handled automatically.

The **Actions** tab contains the following:

When Pressed The standard set of prompts for buttons (see *Setting Control Properties)*. Normally, when using a Control template, these prompts are not used.

Browse Box control template



Click on a TAB to see its help

This Control template places a LIST control in a window. The LIST controls popup menu takes you to the **List Box Formatter**, so that you can choose which fields or variables populate the list, and define how they appear in the list box (including enabling colorization and Icon display). The **Actions** tab on the List Properties provides the template prompts which allows you to define the browse boxs functionality, including record filters, range limits, totaling, scroll bar behavior, and locator behavior.

You can place the *BrowseBox* Control template in a window by clicking on the template control tool, then selecting ***BrowseBox - Browse List Box*** in the **Select Control template** dialog. Then CLICK in the window to place the actual list box control.

Properties

After placing the LIST control, RIGHT-CLICK on the LIST control and choose ***Properties*** from the popup menu to view the **List Properties** dialog. See the *Setting Control Properties* chapter for full information about the options available in this dialog. This section describes only the options directly affected by the BrowseBox Control template.

To enable colorization, Check the **Color Cells** box in **List Field Properties**.

To enable Icon display, Check the **Icons** box in **List Field Properties**.

The template automatically defines the FROM attribute for the LIST control, which names the source (a QUEUE) for the data in the list. The standard templates name the QUEUE as Queue:Browse. The template contains a group (a template routine) that checks to see if youve applied range limits, or are using the list as a lookup. If so, it locates the correct record. The template then loads as many records into the QUEUE as will fit in the list. The QUEUE is filled from a VIEW which gets the values from fields in data files on disk.

RIGHT-CLICK on the LIST control and choose **List Box Format** from the popup menu to access the List Box Formatter to choose the fields and variables to fill the list box, and define their appearance.

The **Populate** button allows you to add a field or variable to the list box, one field or variable at a time. The **Select Field** dialog presents the file schematic. Within the schematic, each browse control appears, with a tree control marked To Do beneath it. To add a field from a data file defined in the dictionary:

Select the To Do item.

Press the **Insert** button

Select the file from the Insert File dialog. The Browse Control item displays the name of the file.

If you want to use a Key, press the **Key** button to select the key from the **Key Access** dialog. If you do not select a Key, the list is displayed in record order, which also disables the ability to set Range Limits.

Select a field from the **Fields** list, which appears in the right side of the **Select Field** dialog.

Repeat this for each field you want to add to the list box.

To add a variable to the list box, select **Global Data** or **Local Data** from the **Select Field** dialog, select the desired variable from the **Fields** list, then press the **Select** button.

After you select the file, key and field (or variable) the **List Field Properties** dialog appears. This allows you to precisely define its appearance. The *Using the List Box Formatter* chapter fully describes the options available in this dialog.

Actions

The **Actions** tab of the **List Properties** dialog (accessed by the Actions... command on the popup menu you see when you right-click the control) displays the template prompts which allows you to specify numerous template options, as well as add custom embedded source code for standard list box events, such as when the end user moves the selection bar. The dialog contains the following options:

Default Behavior

This tab contains the prompts that control the default behavior of the Browse Box Control.

Quick-Scan Records

Specifies buffered access behavior for ODBC, ASCII, DOS, or BASIC files. These file drivers read a buffer at a time (not a record), allowing for fast access. In a multi-user environment these buffers are not 100% trustworthy for subsequent access, because another user may change the file between accesses. As a safeguard, the driver rereads the buffers before each record access. To disable the reread, enable QUICKSCAN.

Locator

A locator is a screen entry field that updates a component of the primary file access key. When the end user types a character(s) in the entry box, then presses TAB, the list box updates to show the closest matching record. This is disabled when browsing a file in Record Order (without specifying a KEY in the File Schematic).

Choose **Step** for a list box which, when the user types in a character, advances the selection to the nearest match in the key field.

Choose **Entry** for an entry box to hold the value for the locator. When the end user places a value in the entry box, TAB or reselecting the list box will move the selection to the nearest matching record.

Choose **Incremental** for a locator which accepts multiple characters and moves the selection to the nearest matching record.

Entry/Incremental Locator-Override default locator control
If you use the same field more than once as a locator, you must override the default locator. For example, if you have a multi-keyed browse which has an ascending key and a descending key on the same field. To use a separate controls (as on separate TABs) for each condition, check the override box and select the second instance from the drop-down list.

Choose **None** for no locator.

Record Filter

Type an expression to limit the contents of the browse list to only those records matching the filter expression. The filter is loops through all displayable records to select those that meet the filter.

You must BIND any file field that is used in a filter expression. The **Hot Fields** tab enables you to BIND fields.

Range Limit Field

In conjunction with the **Range Limit Type**, specifies a record or group of records

for inclusion in the list box. Choose a field by pressing the ellipsis (...) button. The range limit is key-dependent; the generated source code uses the SET statement to find the first valid record. This is enabled *only* after you specify a Key for the file associated with this control.

Range Limit Type When a field is selected for **Range Limit Field**, specifies a record or group of records for inclusion in the list box.

Current Value signifies the current value of **Range Limit Field**.

Single Value allows you to limit the list to a single value. Specify the variable containing that value in the **Range Limit Value** box which appears.

Range of Values allows you to specify upper and lower limits. Specify the variable containing the values in the **Low Limit** and **High Limit Value** boxes.

File Relationship allows you to choose a range limiting file from a 1:MANY relationship. This limits the browse to display only those child records matching the current record in the Parent file. For example, if your browse was a list of Orders, you could limit the display to only those orders for the current Customer (in the Customer file).

See also: **Using Range Limits and Filters**

Reset Fields button

Pressing this button displays a list box allowing you to add Reset Fields. If the value of any field in the Reset Fields list changes the Browse Box is refreshed.

Scroll Bar Behavior button

Pressing this button displays a dialog where you can define the way a scroll bar works.

Scroll Bar Behavior

Specifies the manner the scroll bar works. Choose ***Fixed Thumb*** or ***Movable Thumb*** from the drop down list.

Key Distribution This specifies the distribution of the points of the scroll bar. Choose one of the two predefined distributions (Alpha or Last Names), or Custom, or Runtime from the drop down list.

Alpha defines 100 evenly distributed points alphabetically.

Last Names defines 100 points distributed as last names are commonly found in the United States. If the access key is sorted on numeric data, you should a custom or runtime distribution.

Custom allows you to define your own points.

Runtime reads the first and last record and computes the values for 100 evenly distributed break points in between.

Custom Key Distribution

Allows you to specify the break points for distribution along the scroll bar (useful when you have data with a skewed distribution). Insert the values for each point in the list. String constants should be in single quotes (' ').

Runtime Distribution Parameters

Allows you to specify the type of characters considered when determining the distribution points. This is only appropriate when the Free Key Element is a STRING or CSTRING. Check the boxes for the types of characters you wish to include for consideration.

Conditional Behavior

This tab contains a list box that allows you to define specific behavior based on conditions. Add conditions to the list by pressing the Insert button. This displays a dialog where you define the Condition and the desired behavior when that condition is true.

At runtime these conditions are evaluated, and the behavior for the first true condition in the list is used.

In this dialog you can specify:

Condition Any valid expression.

Key to Use Optionally, the Key to use to determine the sort order of the browse box when this condition is true.

The remaining fields and buttons are the same as the Default behavior tab.

Hot Fields

When you select the Hot Fields tab, you can select a field (or fields) to keep live in the QUEUE. When scrolling through the file, the generated source code reads the data for these fields from the QUEUE, rather than from the disk. This speeds up list box updates.

Specifying "Hot" fields also allows you to place file field controls outside of the Browse Box that are updated whenever a different record is selected in the list box. Elements of the Primary Key and the current key are always included in the QUEUE, so they do not need to be inserted in the Hot Field list.

This dialog also enables you to BIND a field. You must BIND any file field that is used in a filter expression or as a field to total.

Totaling

This tab contains a list box that allows you to define total fields for a browse box. Press the Insert button to add total fields.

This displays a dialog where you can define total fields for a Browse Box Control.

Total Target Field The variable to store the total. This can be a local, module, or global variable. You may also use a file field; however, you must write the code to update the data file.

Total Type Choose ***Count, Sum,*** or ***Average*** from the drop down list. **Count** tallies the number of records. **Sum** adds the values of the Field to Total. **Average** determines the arithmetic mean of the Field to Total.

Field to Total The field to be summed or averaged. This box is disabled when the total field is a **Count** Type.

Total Based On Choose ***Each Record Read*** or ***Specified Condition*** from the drop down list. This specifies whether to consider every record or only those that meet a certain filter criteria.

Total Condition The condition to meet when using a Total based on a specified condition. You can use any valid expression.

Colors

This tab is only available if you check the **Color Cells** box in the List Box Formatter. It displays a list of the fields which have been specified to allow colorization.

To specify colors, highlight the desired field and press the **Properties** button.

Customize Colors

This dialog allows you to specify the default colors for Normal Foreground and Background; and for the Foreground and Background colors to display when the row is selected.

Below the default colors section, is the **Conditional Color Assignments** list. To add a condition and specify special colors to display for the field when the condition is true, press the BButton.

At runtime these conditions are evaluated, and the colors for the first true condition in the list are used.

Icons

This tab is only available if you check the **Icons** box in the List Box Formatter. It displays a list of the fields which have been specified to allow Icon display.

To specify Icons, highlight the desired field and press the Properties button.

Customize BrowseBox Icons

This dialog allows you to specify the default Icon for the field.

Default Icon The default icon to display. You may specify a standard Icon or an Icon (.ICO) file on disk.

Conditional Icon Usage
Below the default Icon section, is the Conditional Icon Usage list. To add a condition and specify special Icons to display when the condition is true, press the Insert Button. At runtime these conditions are evaluated, and the Icon for the first true condition in the list is used.

Browse Select button control template

This control template provides a quick way to process a record in a list box.

The generated source code gets the currently selected record from the list, closes down the browse, and resets the value of LocalResponse to 'RequestCompleted' (See also: Request and Response). For the end user, pressing the Select button is equivalent to doubleclicking an item in the list. You specify, on the Actions tab, what happens next. The Properties dialog for the button is identical to the normal Button Properties dialog.

The **Action** button leads to a dialog containing the following:

Hide	Specifies that the control should be hidden if the procedure is not called to request a record.
When Pressed	The standard set of prompts for buttons Normally, when using a Control template, these prompts are not used.

Browse Update buttons control template

This control templates provides a quick way to add standard functionality for managing the records in a browse list box.

The BrowseUpdateButtons control template adds three button controls for acting upon records inside a browse box. When pressed, the buttons retrieve the appropriate record and call the procedure specified in the **Update Procedure** box. Pressing the Change, Insert, or Delete button sets the variable GlobalRequest to 'ChangeRecord' , 'InsertRecord', or 'DeleteRecord', respectively. See also: Request and Response

Optionally, you can also enable a popup menu to call the update procedure when the end user RIGHT-CLICKs on the list box.

The Properties dialog for each button control is identical to the standard **Button Properties** dialog.

The **Action** buttons lead to dialogs allowing you to name the update procedure and specify special keys for implementing the button actions.

Update Procedure	Type a name or select from the drop down list. The Application Generator automatically adds the update procedure to the Procedure tree.
Allow Edit via Popup	Check this box to create a popup menu to call the update procedure when the end user RIGHT-CLICKs on the list box.
When Pressed	The standard set of prompts for buttons Normally, when using a Control template, these prompts are not used.

Cancel Button control template

This control template primarily provides a convenient control to allow the user to close the browse window, and for the developer to add code to "undo" while closing down the browse procedure.

The generated source code posts a close window event. Before closing the window, it sets the LocalResponse variable to 'RequestCancelled'. See also:Request and Response .

You can insert the executable code you need to "clean up" at an embed point. The dialog includes the following options:

When Pressed The standard set of prompts for buttons Normally, when using a Control template, these prompts are not used.

Accept Button control template

This control template primarily provides a convenient control to allow the user to close the browse window, and for the developer to add code to execute while closing down the browse procedure.

The generated source code posts a close window event. Before closing the window, it sets the LocalResponse variable to 'RequestCompleted'. See also:Request and Response .

You can insert the executable code you need to "clean up" at an embed point. The dialog includes the following options:

When Pressed The standard set of prompts for buttons Normally, when using a Control template, these prompts are not used.

Close Button control template

This control template adds a single button control marked **Close**. The generated source code closes down the current window. You specify, via the Action button, precisely what happens when the end user presses the button.

The properties dialog for is identical to the normal Button Properties dialog.

The **Action** button leads to a dialog containing the following:

When Pressed The standard set of prompts for buttons Normally, when using a Control template, these prompts are not used.

DOS File Lookup control template

This control template adds an ellipsis (...) button which leads the end user to a standard Open File dialog. You can specify a file mask, and a return variable to hold the end user's choice.

The properties dialog for is identical to the normal Button Properties dialog.

The **Action** button leads to a dialog containing the following:

File Dialog Header Type the text for the caption of the **Open File** dialog.

DOS FileName Variable
Press the ellipsis button to view the **File Schematic** dialog, and choose a variable to receive the end user's choice. You can also type the variable name directly into the entry box.

Default Directory Allows you to specify a directory name where the **Open File** dialog will start.

Mask Description Type a file type description. The string appears in the drop down list in the **Open File** dialog. You can add additional file masks by pressing the File Masks button

File Mask Type a file specification, such as "*.TXT." To use multiple patterns for this mask, separate each with a semi-colon, such as "*.BMP;*.GIF"

When Pressed The standard set of prompts for buttons Normally, when using a Control template, these prompts are not used.

Field Lookup Button control template

This control template allows you to trigger an entry control lookup. CLICK next to an entry control to place the ellipsis (...) button that enables the end user to initiate the lookup procedure.

Once you place the field, RIGHT-CLICK the button, then choose **Actions** to access the Prompts dialog.

Control with Lookup Select the field equate label of the control to perform the look up for, by choosing from the drop down list.

NOTE: The Control with Lookup must have an associated lookup procedure. Once you place the field, RIGHT-CLICK on the control, then choose Actions to access the Prompts dialog. For more details, see the Control Prompts Dialog topic.

When Pressed The standard set of prompts for buttons Normally, when using a Control template, these prompts are not used.

File Drop control template

This control template scrolls through a data file and assigns the value of the selected field to the Target Field. This allows you to perform a lookups easily.

Use this control template when you want to lookup a single field from a file of less than 100 records, where no range limit is needed. If you need a range limit, use an entry field with an associated Call Procedure As Lookup code template .

General

Field to Fill From The field from the lookup file. This value is assigned to the Target Field.

Default to First entry if Use Variable empty
Automatically assign the value of the first field in the list to the ?USE variable. The fields in the list are sorted alphabetically.

Target Field The field to which the value from the lookup file is assigned. This can be different than the ?USE variable.

Record Filter Optionally, type an expression to limit the contents of the drop down list to only those records which match the filter expression.

Sort Fields

This tab allows you to add fields by which the list is sorted. The sort order is independent of Keys. Press the **Insert** button to add fields to the list. This sorts the list dynamically at runtime.

Range Limits

This tab appears *only* after you specify a Key for the file associated with this control.

Range Limit Field In conjunction with the **Range Limit Type**, specifies a record or group of records for inclusion in the list box. Choose a field by pressing the ellipsis (...) button. The range limit is key-dependent; the generated source code uses the SET statement to find the first valid record.

Range Limit Type When a field is selected for **Range Limit Field**, specifies a record or group of records for inclusion in the list box.

Current Value signifies the current value of **Range Limit Field**.

Single Value allows you to limit the list to a single value. Specify the variable containing that value in the **Range Limit Value** box which appears.

Range of Values allows you to specify upper and lower limits. Specify the variable containing the values in the **Low Limit** and **High Limit Value** boxes.

File Relationship allows you to choose a range limiting file from a 1:MANY relationship. This limits the browse to display only those child records matching the current record in the Parent file.

Colors

This tab is only available if you check the Color Cells box in the List Field Properties in the List Box Formatter. It displays a list of the fields which have been specified to allow colorization.

To specify colors, highlight the desired field and press the **Properties** button.

Customize Colors

This dialog allows you to specify the default colors for Normal Foreground and Background; and for the Foreground and Background colors to display when the row is selected.

Below the default colors section, is the Conditional Color Assignments list. To add a condition and specify special colors to display for the field when the condition is true, press the Insert Button.

At runtime these conditions are evaluated, and the colors for the first true condition in the list are used.

Icons

This tab is only available if you check the Icons box in the List Box Formatter. It displays a list of the fields which have been specified to allow Icon display. To specify Icons, highlight the desired field and press the Properties button.

Customize BrowseBox Icons

This dialog allows you to specify the default Icon for the field.

Default Icon The default icon to display. You may specify a standard Icon or an Icon (.ICO) file on disk.

Conditional Icon Usage
Below the default Icon section, is the Conditional Icon Usage list. To add a condition and specify special Icons to display when the condition is true, press the Insert Button. At runtime these conditions are evaluated, and the Icon for the first true condition in the list is used.

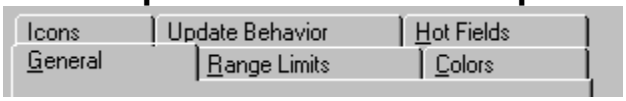
Properties

The List Properties for this control are the same as a list ; however, the **From** entry requires some explanation.

From: When placing a File Drop Control, this field is filled in with 'File|Drop'. You may modify this, but a pipe character (|) must appear in the string.

You can use list box formatter to populate this control.

File Drop Combo control template



This control template scrolls through a data file and assigns the value of the selected field to the ?Use variable. It also allows adding records by typing a new value in the entry portion of the combo box.

There are two different scenarios for which you can use this control template:
Storing and Displaying the same data and **Displaying textual data and storing a code.**

Storing and Displaying the same data

In this scenario you want to select a value from the lookup file and store it in the Primary file. For example, A Product File with a field storing a color, with a lookup file of colors.

In this case, complete the prompts as follows:

General

?Use The field to which the value is assigned from the field in the lookup file.

Field to Fill From The field from the lookup file. This value is assigned to the Target Field.

Remove Duplicates Check this box to remove duplicates from the list displayed.

Target Field The field to which the value is assigned from the field in the lookup file. In this case this is the same as the ?USE variable.

Record Filter Optionally, type an expression to limit the contents of the drop down list to only those records which match the filter expression.

Default to First entry if Use Variable empty
Automatically assign the value of the first field in the list to the ?USE variable. The fields in the list are sorted alphabetically (unless you specify Sort Fields).

Update Behavior

In this scenario, a form is NOT needed to update the lookup file. Checking the Allow Updates box enables updates directly from this control.

Sort Fields

This tab allows you to add fields by which the list is sorted. The sort order is independent of Keys. Press the **Insert** button to add fields to the list. This sorts the list dynamically at runtime.

Range Limits

This tab appears *only* after you specify a Key for the file associated with this control.

Range Limit Field In conjunction with the **Range Limit Type**, specifies a record or group of records for inclusion in the list box. Choose a field by pressing the ellipsis (...) button. The range limit is key-dependent; the generated source code uses the SET statement to find the first valid record.

Range Limit Type When a field is selected for **Range Limit Field**, specifies a record or group of

records for inclusion in the list box.

Current Value signifies the current value of **Range Limit Field**.

Single Value allows you to limit the list to a single value. Specify the variable containing that value in the **Range Limit Value** box which appears.

Range of Values allows you to specify upper and lower limits. Specify the variable containing the values in the **Low Limit** and **High Limit Value** boxes.

File Relationship allows you to choose a range limiting file from a 1:MANY relationship. This limits the browse to display only those child records matching the current record in the Parent file.

Colors

This tab is only available if you check the Color Cells box in the List Field Properties in the List Box Formatter. It displays a list of the fields which have been specified to allow colorization.

To specify colors, highlight the desired field and press the **Properties** button.

Customize Colors

This dialog allows you to specify the default colors for Normal Foreground and Background; and for the Foreground and Background colors to display when the row is selected.

Below the default colors section, is the Conditional Color Assignments list. To add a condition and specify special colors to display for the field when the condition is true, press the Insert Button.

At runtime these conditions are evaluated, and the colors for the first true condition in the list are used.

Icons

This tab is only available if you check the Icons box in the List Box Formatter. It displays a list of the fields which have been specified to allow Icon display. To specify Icons, highlight the desired field and press the Properties button.

Customize BrowseBox Icons

This dialog allows you to specify the default Icon for the field.

Default Icon The default icon to display. You may specify a standard Icon or an Icon (.ICO) file on disk.

Conditional Icon Usage

Below the default Icon section, is the Conditional Icon Usage list. To add a condition and specify special Icons to display when the condition is true, press the Insert Button. At runtime these conditions are evaluated, and the Icon for the first true condition in the list is used.

Hot Fields

When you select the Hot Fields tab, you can select a field (or fields) to keep live in the QUEUE. When scrolling through the file, the generated source code reads the data for these fields from the QUEUE, rather than from the disk. This speeds up list box updates.

Specifying "Hot" fields also allows you to place file field controls outside of the Browse Box that are

updated whenever a different record is selected in the list box. Elements of the Primary Key and the current key are always included in the QUEUE, so they do not need to be inserted in the Hot Field list.

This dialog also enables you to BIND a field. You must BIND any file field that is used in a filter expression or as a field to total.

When you select the Hot Fields tab, you can select a field (or fields) to keep live in the QUEUE. When scrolling through the file, the generated source code reads the data for these fields from the QUEUE, rather than from the disk. This speeds up list box updates.

Specifying "Hot" fields also allows you to place file field controls outside of the Browse Box that are updated whenever a different record is selected in the list box. Elements of the Primary Key and the current key are always included in the QUEUE, so they do not need to be inserted in the Hot Field list.

This dialog also enables you to BIND a field. You must BIND any file field that is used in a filter expression or as a field to total.

Displaying textual data and storing a code

In this scenario you want to select a value from a textual field in the lookup file and store its associated code in the Primary file. For example, A Product File with a field storing a Location Code, with a lookup file of Locations. You want the user to select the Location from a list of descriptions, but store the Location Number in the Product file.

In this case, complete the prompts as follows:

?Use Create a local variable that matches the textual field.

Using the List Box Formatter, populate the list with the textual field from the lookup file. It is automatically be assigned to the ?Use variable.

Field to Fill From The code field from the lookup file. This value is assigned to the Target Field.

Remove Duplicates Check this box to remove duplicates from the list displayed.

Target Field The field to which the value is assigned from the field in the lookup file.

Record Filter Optionally, type an expression to limit the contents of the drop down list to only those records which match the filter expression.

Default to First entry if Use Variable empty
Automatically assign the value of the first field in the list to the ?USE variable. The fields in the list are sorted alphabetically (unless you specify Sort Fields).

Update Behavior

In this scenario, a form is needed to update the lookup file, if you want to allow updates, specify a form procedure.

Properties

The List Properties for this control are the same as a list ; however, the **From** entry requires some explanation.

From: When placing a File Drop Combo Control, this field is filled in with Queue:FileDropCombo. You should not modify this.

You can use list box formatter to populate this control, but only the first populated is valid for the entry portion of the control.

RelationTree control template

The tree control is actually a list box formatted to display as a tree.

Using the *RelationTree* Control template, you can specify multiple file levels to display on multiple levels of a tree control. The Relation Tree control can display an unlimited number of related fileswith an associated update procedure for each level. The provides an alternative for the *Browse-Form* paradigm. A single *RelationTree* control can replace several *Browse-Form* pairs.

The *RelationTree* template employs a fully-loaded QUEUE for the root level. The child levels are demand-loaded when a branch is expanded. This template is not appropriate for databases with a very large primary file. You should use the *BrowseBox* Control template, which is page-loaded, instead.

To create a tree using the Relation Tree Control template:

1. Place a *RelationTree* Control template on a window.

The **List Box Formatter** appears. **Do *Not* use the formatter to populate your tree.**

2. If you want to enable colorization or icon display in your tree control, press the **Properties** button on the **List Box Formatter** and check the appropriate boxes.
3. Press the **OK** button on the **List Box Formatter**.
4. RIGHT-CLICK on the Relation Tree Control template and choose **Actions** from the popup menu.
5. Press the **Files** button to specify the file schematic for the control.
6. Specify the File details:

Tree Heading Text An optional text heading at the top of the tree. Tree Heading Text is required to enable the user to add a record at the root level.

Tree heading Icon An optional Icon at the top of the tree. Icons must be enabled in the List Box Formatter for this prompt to be enabled.

Primary File

Display String The field name or text to display for the primary file level.

Update Procedure The Update Procedure to call for this level.

Secondary Files

Optionally, specify display strings and Update Procedures for any secondary files by highlighting the secondary file and pressing the **Properties** button below the **Secondary Files** list box.

Calling Update Procedures

One of the most powerful features of the Relation Tree Control template is the ability to call the update procedure for the selected level of the tree (if an Update Procedure is specified for that level).

The Update Procedure is called to change a record when the user DOUBLE-CLICKS on a record.

A RIGHT-CLICK calls a popup menu to insert, change, or delete records. The menu displays the text

displayed on the associated *RelationTreeUpdateButtons*.

A third method to call a update procedures is to place a Relation Tree Update Buttons control template on the window.

Colorized Tree controls

The List Box formatter now supports colorization of cells in a list box.

Specify colors by:

1) To enable colorization, Check the **Color Cells** box in **List Field Properties**.

2) Select the Colors tab in the Actions tab for the Relation Tree control.

3) Highlight the desired field and press the Properties button.

4) Specify the default colors for Normal Foreground, Normal background, Selected Foreground, Selected background by pressing the ellipsis (...) button.

5) Optionally, specify conditional colors by pressing the Insert button below the Conditional Colors list box. Specify a valid expression and the colors to use when that expression is true.

Icons in tree controls

The List Box formatter now supports Icon display in a list box.

Specify Icons by:

1) To enable Icon display, Check the **Icons** box in **List Field Properties**.

2) Select the Icons tab in the Actions tab for the control.

3) Highlight the desired field and press the Properties button.

4) Specify the default Icon.

5) Optionally, specify conditional Icons by pressing the Insert button below the Conditional Icons list box. Specify a valid expression and the Icon to use when that expression is true.

Relation Tree Update Buttons control template

This Control template adds three buttons (**Insert**, **Change**, and **Delete**) which allow the user to call the associated update procedure for the selected level of a Relation Tree (if an update procedure has been specified) . There are no prompts for this control. The Update Procedure is specified for each level of the Relation Tree control template. .

The Change and Delete buttons correspond to the currently highlighted record. The Insert button adds a child record (the next level down the tree structure).

Save Button control template

This control template provides an OK button for your browse window. It also creates a local variable--ActionMessage-- which allows you to display an action message for the end user.

The Properties dialog for the OK button is the standard **Button Properties** dialog.

The **Actions** button leads to the **Prompts for ?OK** dialog. It allows you to specify the update procedures and action messages for the button action. It contains the following options:

Allow: Inserts, Changes, and Deletes
Checking the appropriate box enables the action. If a check box is cleared, the user will not be allowed to perform the action.

When called for Delete Allows to to select a method for deleting records. **Standard Warning** displays a standard message box prompting for confirmation of the delete. **Display Form** displays the form and sets the variable--ActionMessage to 'Record will be Deleted' or the Delete message you specify. **Automatic delete** enables deletes without a warning or prompt for confirmation.

Messages and Titles Allows you to specify the messages for Inserts, Changes, or Deletes, and the location for the message.

Insert Message Specifies the text for the ActionMessage when the procedure is called to ADD a record. The default text is "Record will be added." You must check the **Allow Inserts** box.

Change Message Specifies the text for the ActionMessage when the procedure is called to modify an existing record. The default text is "Record will be changed." You must check the **Allow Changes** box.

Delete Message Specifies the text for the ActionMessage when the procedure is called to delete a record. The default text is "Record will be deleted." You must check the **Allow Deletes** box.

Location of Message Specifies where the action message appears. You can specify a window control, the caption bar, or the status bar. If you specify the status bar, you may then specify the section in the **Status Bar Section** box.

Display Record Identifier on the Title Bar
Allows you to append a string to the caption on the Title bar.

Record Identifier Specifies the string to append to the Title bar caption, which you can use to identify the record. Type a string in the **Record Identifier** box. To use a variable name, precede it with an exclamation point (!).

Field Priming "Field Priming" allows you to provide a default data value for fields in a new record. This value supersedes any initial value specified in the data dictionary. When you press the button, you can select a field and set an initial value in the **Field Priming** dialog.

When Pressed The standard set of prompts for buttons Normally, when using a Control template, these prompts are not used.

You may also access the embed points for the controls by pressing the **Embeds** button in this dialog.

Date Time Display extension template

This extension template adds to the functionality of a procedure template, allowing you to display the time and/or date in the status bar, or a control.

The prompts for this template are accessible through the **Procedure Properties** dialog of a template which includes this extension. A **Date and Time Display** button appears in the dialog of the procedure template.

The options which appear in the **Date and Time Display** dialog are divided into two group boxes, **Date Display** and **Time Display**:

Display in Window	Check the box or boxes to add the display to your window.
Picture	Choose a date and/or time display picture from the drop down list. The list displays examples, such as "October 31, 1959," and "5:30P.M."
Other Picture	Type in a picture of your choice, if the picture type you wish does not appear in the list. See also: Date Picture Tokens.or Time Picture Tokens
Day of Week	(Date only) Optionally displays the day of week.
Location	Choose between displaying the date and/or time on the status bar, or in a control.
Status Bar Section	When specifying the Date or Time should appear on the status bar, specify the status bar section.
Display Control	When specifying the Date or Time should appear in a control, choose the control from a drop down list of field equate labels for the window.

Record Validation extension template

This Extension template adds functionality to a Procedure template by enforcing data dictionary-defined control value validation. It also allows you to specify controls to exclude from validation.

The prompts for this template are accessible through the **Procedure Properties** dialog of a template which includes this extension. A **Record Validation** group box appears in the dialog of the procedure template.

Validate when the control is Accepted
Specifies that validity checking occurs when the control generates an EVENT:Accepted, which occurs when the end user completes or moves the focus from the field.

Validate during NonStop Select
Specifies that validity checking occurs when any control value changes if the window is in AcceptAll (Non-Stop) mode and has focus.

Do Not Validate Opens the Do Not Validate dialog, which allows you to select fields from a drop down list. The fields you choose will be excluded from validity checks.

Procedure Properties : Browse 

The Browse Template consists of several control templates which add a Browse Box control template ,Browse Update buttons control template , Browse Select button control template and a Close Button control template to the default window. The control templates add the Browse Box and Update Buttons prompts to the **Procedure Properties** dialog.

Additionally, the generated code includes a ROUTINE called RefreshWindow which keeps the data displayed current.

Description A short text description for the procedure, which appears next to the procedure name in the **Application Tree** dialog.

Press the ellipsis (...) button to edit a longer (up to 1000 characters) description.

Prototype Allows you to optionally type a custom procedure prototype which the Application Generator places in the MAP section.

Module Name The source code file to hold the code for the procedure. Select from the drop down list. By default, the Application Generator names modules by taking the first five characters of the .APP file name, then adding a three digit number for each module.

Export Procedure Declares the procedure in the export file, enabling it to be called by another aplication. Note:This checkbox is only available when the target file specified in Application Properties as a Dynamic Link Library (.DLL).

Parameters Allows you to specify parameter names (an optional list of variables separated by commas, with the entire list surrounded by parentheses) for your procedure, which you can pass to it from a calling procedure. You must specify the functionality for the parameters in embedded source code. See also: Procedure and Function Calls.

Window Operation Mode

This option allow you to override the window settings specified in the **Window Properties** dialog. This allows an additional access point to modify the window's operation mode. See also: WINDOW.

Use WINDOW Setting specifies no overrides to the window settings

Normal specifies application modal operation mode. The user must respond before moving to any other window in the application.

MDI specifies that the window conforms to standard MDI child behavior.

Modal specifies system modal operation. A system modal window takes complete control until the window is closed.

INI File Settings Checking the **Save and Restore Window Location** specifies that a window's location is stored in the application .INI file, and will open in that position the next time the procedure is called. This is available only if you enable INI File settings in the **Global Properties** dialog.

Files Accesses the **File Schematic Definition** dialog. You can define the procedure's

access to variables or other files through the dialog.

Window Calls the Window Formatter, to visually design the window. See also: How to Customize Your Window

The ellipsis (...) button next to the **Window** button allows you to edit the WINDOW or APPLICATION structure at the source code level. Clarion for Windows allows you to easily switch back and forth between editing the window graphically, and editing the source code that describes it.

Tip: Take care when hand-editing code for any WINDOW which contains a control template. The Application Generator stores Template Language attributes which cannot be edited by hand.

Report The Report Button is disabled for this procedure type.

Data Adds or edits local variables. Press this button and fill in the **Local Data** dialog. Any variables defined are local to the procedure. Define global variables by pressing the **Global** button in the **Application Tree** dialog.

The ellipsis (...) button next to the **Data** button allows you to view the memory variable declarations at the source code level.

Procedures Calls procedure made in hand-coded, embedded source.

Press this button to access the **Called Procedures** dialog. To add a procedure, press the **Insert** button, and type a procedure name in the next dialog.

If procedure calls already exist, the names appear in the **Called Procedures** dialog. To add another, press the **Add** button. To delete one, press the **Delete** button. Additional buttons allow you to change the order of any procedures listed.

Tip: The purpose of the Procedure button is to add procedures called in embedded source code. The normal way to add template procedures to the Application Tree is to create a menu or toolbar command, add the procedure name via its Actions button, and let the Application Generator automatically add it to the tree.

Embeds Displays the **Embedded Source** dialog. You can then select either a field specific event or window related action, then add executable source code to customize how the procedure will handle it.

After you choose an embed point in the **Embedded Source** dialog, you choose the code to execute. You can specify a call procedure, which is then added to the tree. You can write your own code with the text editor. Or, you can choose and customize a code template, which is a combination of pre-written code and prompts to "fill-in-the-blanks."

Embedded source gives you complete control over *all* the processing in your procedures. It's one of the most powerful tools Clarion provides you. See also: Adding Embedded Source Code.

Formulas Accesses the **Formula Editor**, which allows you to create computed and/or conditional fields, which you can then reference in the controls you place in your windows and reports.

Extensions Accesses extension and Control templates. Extensions, if applicable to the

procedure, can be added or modified from this dialog. If any Control templates were placed in the window, this accesses the prompts for those Control templates. Optionally, you can specify whether or not the prompts for an extension or Control template should display on the procedure properties window.

Browse Box Behavior This button calls a dialog which allows you to define the behavior of the BrowseBox Control template..

Select Button Prompts

The **Hide Select button** check box is controlled by the Select Button control template. Check this box to hide the Select button when this procedure is not called to request a record.

Update Button Prompts

The **Update Procedure** entry control is controlled by the Update Button control template. Type in the Update Procedure name, or select it from the drop down list.

Procedure Properties -- Form

The Form Template provides a predefined window, with update buttons, plus an action message text control. The control templates add no prompts to the **Procedure Properties** dialog.

Description — A short text description for the procedure, which appears next to the procedure name in the **Application Tree** dialog.

Press the ellipsis (...) button to edit a longer (up to 1000 characters) description.

Prototype — Allows you to optionally type a custom procedure prototype which the Application Generator places in the MAP section.

Module Name — The source code file to hold the code for the procedure. Select from the drop down list. By default, the Application Generator names modules by taking the first five characters of the .APP file name, then adding a three digit number for each module.

Export Procedure — Declares the procedure in the export file, enabling it to be called by another aplication. Note:This checkbox is only available when the target file specified in Application Properties as a Dynamic Link Library (.DLL).

Parameters — Allows you to specify parameter names (an optional list of variables separated by commas, with the entire list surrounded by parentheses) for your procedure, which you can pass to it from a calling procedure. You must specify the functionality for the parameters in embedded source code. See also: Procedure and Function Calls.

Window Operation Mode

This option allow you to override the window settings specified in the **Window Properties** dialog. This allows an additional access point to modify the window's operation mode. See also: WINDOW.

Use WINDOW Setting specifies no overrides to the window settings

Normal specifies application modal operation mode. The user must respond before moving to any other window in the application.

MDI specifies that the window conforms to standard MDI child behavior.

Modal specifies system modal operation. A system modal window takes complete control until the window is closed.

INI File Settings — Checking the **Save and Restore Window Location** specifies that a window's location is stored in the application .INI file, and will open in that position the next time the procedure is called. This is available only if you enable INI File settings in the **Global Properties** dialog.

Files — Accesses the **File Schematic Definition** dialog. You can define the procedure's access to variables or other files through the dialog.

Window — Calls the Window Formatter, to visually design the window. See also: How to Customize Your Window

The ellipsis (...) button next to the **Window** button allows you to edit the WINDOW or APPLICATION structure at the source code level. Clarion for Windows allows you to easily switch back and forth between editing the window graphically, and editing the source code that describes it.

Tip: Take care when hand-editing code for any WINDOW which contains a control template. The Application Generator stores Template Language attributes which cannot be edited by hand.

Report The Report button is disabled for this procedure type.

Data Adds or edits local variables. Press this button and fill in the **Local Data** dialog. Any variables defined are local to the procedure. Define global variables by pressing the **Global** button in the **Application Tree** dialog.

The ellipsis (...) button next to the **Data** button allows you to view the memory variable declarations at the source code level.

Procedures Calls procedure made in hand-coded, embedded source.

Press this button to access the **Called Procedures** dialog. To add a procedure, press the **Insert** button, and type a procedure name in the next dialog.

If procedure calls already exist, the names appear in the **Called Procedures** dialog. To add another, press the **Add** button. To delete one, press the **Delete** button. Additional buttons allow you to change the order of any procedures listed.

Tip: The purpose of the Procedure button is to add procedures called in embedded source code. The normal way to add template procedures to the Application Tree is to create a menu or toolbar command, add the procedure name via its Actions button, and let the Application Generator automatically add it to the tree.

Embeds Displays the **Embedded Source** dialog. You can then select either a field specific event or window related action, then add executable source code to customize how the procedure will handle it.

After you choose an embed point in the **Embedded Source** dialog, you choose the code to execute. You can specify a call procedure, which is then added to the tree. You can write your own code with the text editor. Or, you can choose and customize a code template, which is a combination of pre-written code and prompts to "fill-in-the-blanks."

Embedded source gives you complete control over *all* the processing in your procedures. It's one of the most powerful tools Clarion provides you. See also: Adding Embedded Source Code.

Formulas Accesses the **Formula Editor**, which allows you to create computed and/or conditional fields, which you can then reference in the controls you place in your windows and reports.

Extensions Accesses extension and Control templates. Extensions, if applicable to the procedure, can be added or modified from this dialog. If any Control templates were placed in the window, this accesses the prompts for those Control templates. Optionally, you can specify whether or not the prompts for an extension or Control template should display on the procedure properties window.

Save Button Prompts These prompts are controlled by the Save Button control template.

Allow specifies whether Inserts, Changes, or Deletes are allowed. Check the appropriate boxes.

Field Priming on Insert allows you to assign values to fields in new records.

Messages and Titles allows you to specify messages and their locations.

Record Validation This group is controlled by the Record Validation extension template.

Validate when the control is Accepted specifies that validity checking occurs when the control generates an EVENT:ACCEPTED, which occurs when the end user completes or moves the focus from the field.

Validate during NonStop Select specifies that validity checking occurs when *any* control value changes if the window is in *AcceptAll* (Non-Stop) mode and has focus.

Do Not Validate opens the **Do Not Validate** dialog, which allows you to select fields from a drop down list. The fields you choose will be excluded from validity checks.

Procedure Properties--Window 

This procedure template functions as a blank slate, upon which you can create your own window, of any kind. Press the **Window** button in the **Procedure Properties** dialog to create your window. For the controls/control templates you place, field templates add embed points to handle the events they generate. The **Embeds** button allows you to attach appropriate code, after you place the controls.

The only "predefined" elements of the template, which you can access via the **Procedure Properties** dialog, are local variables which the executable code produced by the template uses to pass data to and from a calling procedure. These "manage" the window and procedure, keeping track of whether the window is open, and whether the procedure needs to respond to a global event. See also: **Request and Response**

The code generated by this template processes the window that you create. It contains an ACCEPT loop for the window, and a CASE structure for handling any field or window events.

Description — A short text description for the procedure, which appears next to the procedure name in the **Application Tree** dialog.

Press the ellipsis (...) button to edit a longer (up to 1000 characters) description.

Prototype — Allows you to optionally type a custom procedure prototype which the Application Generator places in the MAP section.

Module Name — The source code file to hold the code for the procedure. Select from the drop down list. By default, the Application Generator names modules by taking the first five characters of the .APP file name, then adding a three digit number for each module.

Export Procedure — Declares the procedure in the export file, enabling it to be called by another aplication. Note:This checkbox is only available when the target file specified in Application Properties as a Dynamic Link Library (.DLL).

Parameters — Allows you to specify parameter names (an optional list of variables separated by commas, with the entire list surrounded by parentheses) for your procedure, which you can pass to it from a calling procedure. You must specify the functionality for the parameters in embedded source code. See also: Procedure and Function Calls.

Window Operation Mode

This option allow you to override the window settings specified in the **Window Properties** dialog. This allows an additional access point to modify the window's operation mode. See also: WINDOW.

Use WINDOW Setting specifies no overrides to the window settings

Normal specifies application modal operation mode. The user must respond before moving to any other window in the application.

MDI specifies that the window conforms to standard MDI child behavior.

Modal specifies system modal operation. A system modal window takes complete control until the window is closed.

INI File Settings Checking the **Save and Restore Window Location** specifies that a window's location is stored in the application .INI file, and will open in that position the next time the procedure is called. This is available only if you enable INI File settings in the **Global Properties** dialog.

Files Accesses the **File Schematic Definition** dialog. You can define the procedure's access to variables or other files through the dialog.

Window Calls the Window Formatter, to visually design the window. See also: How to Customize Your Window

The ellipsis (...) button next to the **Window** button allows you to edit the WINDOW or APPLICATION structure at the source code level. Clarion for Windows allows you to easily switch back and forth between editing the window graphically, and editing the source code that describes it.

Tip: Take care when hand-editing code for any WINDOW which contains a control template. The Application Generator stores Template Language attributes which cannot be edited by hand.

Report The Report button is disabled for this procedure type.

Data Adds or edits local variables. Press this button and fill in the **Local Data** dialog. Any variables defined are local to the procedure. Define global variables by pressing the **Global** button in the **Application Tree** dialog.

The ellipsis (...) button next to the **Data** button allows you to view the memory variable declarations at the source code level.

Procedures Calls procedure made in hand-coded, embedded source.

Press this button to access the Called Procedu**res** dialog. To add a procedure, press the **Insert** button, and type a procedure name in the next dialog.

If procedure calls already exist, the names appear in the **Called Procedures** dialog. To add another, press the **Add** button. To delete one, press the **Delete** button. Additional buttons allow you to change the order of any procedures listed.

Tip: The purpose of the Procedure button is to add procedures called in embedded source code. The normal way to add template procedures to the Application Tree is to create a menu or toolbar command, add the procedure name via its Actions button, and let the Application Generator automatically add it to the tree.

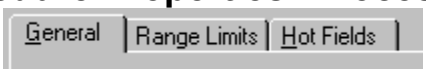
Embeds Displays the **Embedded Source** dialog. You can then select either a field specific event or window related action, then add executable source code to customize how the procedure will handle it.

After you choose an embed point in the **Embedded Source** dialog, you choose the code to execute. You can specify a call procedure, which is then added to the tree. You can write your own code with the text editor. Or, you can choose and customize a code template, which is a combination of pre-written code and prompts to "fill-in-the-blanks."

Embedded source gives you complete control over *all* the processing in your procedures. It's one of the most powerful tools Clarion provides you. See also: Adding Embedded Source Code.

Formulas	Accesses the **Formula Editor**, which allows you to create computed and/or conditional fields, which you can then reference in the controls you place in your windows and reports.
Extensions	Accesses extension and Control templates. Extensions, if applicable to the procedure, can be added or modified from this dialog. If any Control templates were placed in the window, this accesses the prompts for those Control templates. Optionally, you can specify whether or not the prompts for an extension or Control template should display on the procedure properties window.

Procedure Properties--Process



The Process procedure template sequentially processes a data file. You can specify a filter or range of on which records to perform the operation.. A predefined window contains a progress indicator to show the end user what percentage of the operation is complete.

Description — A short text description for the procedure, which appears next to the procedure name in the **Application Tree** dialog.

Press the ellipsis (...) button to edit a longer (up to 1000 characters) description.

Prototype — Allows you to optionally type a custom procedure prototype which the Application Generator places in the MAP section.

Module Name — The source code file to hold the code for the procedure. Select from the drop down list. By default, the Application Generator names modules by taking the first five characters of the .APP file name, then adding a three digit number for each module.

Export Procedure — Declares the procedure in the export file, enabling it to be called by another aplication. Note:This checkbox is only available when the target file specified in Application Properties as a Dynamic Link Library (.DLL).

Parameters — Allows you to specify parameter names (an optional list of variables separated by commas, with the entire list surrounded by parentheses) for your procedure, which you can pass to it from a calling procedure. You must specify the functionality for the parameters in embedded source code. See also: Procedure and Function Calls.

Window Message — The title displayed in the Processing records dialog.

Action for Process — The action to perform for each record processed.

No record action specifies no action to performed by the process template. Use embedded source to handle the action.

PUT record specifies that a record will be added.

DELETE record specifies that each record processed will be deleted.

Quick-Scan Records — Specifies buffered access behavior for ODBC, ASCII, DOS, or BASIC files. These file drivers read a buffer at a time (not a record), allowing for fast access. In a multi-user environment these buffers are not 100% trustworthy for subsequent access, because another user may change the file between accesses. As a safeguard, the driver rereads the buffers before each record access. To *disable* the reread, enable QUICKSCAN.

Range and Filter — Pressing this button accesses the Range and Filter dialog.

Record Filter — Type an expression to limit the contents of the browse list to only those records which match the filter expression. This filters all displayable records. When a

Record filter is used in conjunction with a Range Limit, only those records within the specified range are filtered. See also: **Using Range Limits and Filters**

Range Limit Field Type in the field name or press the ellipsis (...) button to select the field from the Component list. The Range Limit Field must be a component of the Access Key specified in the File Schematic dialog. The range limit is key-dependent; the generated source code uses the SET statement to find the first valid record.

Approx. Record Count This number is displayed in the progress dialog which appears during the process.

Range Limit Type When a field is selected for **Range Limit Field**, this specifies the method of determining the records for inclusion in the list box.

Current Value -- Signifies the value contained in the key field at the beginning of the ACCEPT loop. This is the value used for the range for the duration of the procedure.

Single Value -- Specifies a variable containing the limiting value. Only records matching the variable are included. Enter a variable in the **Range Limit Value** box which appears, or press the ellipses (...) button to select the variable from the File Schematic.

Range of Values -- Allows you to specify upper and lower limits. Enter a variable in the **Low Limit** and **High Limit Value** boxes which appears, or press the ellipses (...) button to select the variables from the File Schematic.

Files Accesses the **File Schematic Definition** dialog. You can define the procedure's access to variables or other files through the dialog.

Window The Window button is disabled for this procedure type.

Report The Report button is disabled for this procedure type.

Data Adds or edits local variables. Press this button and fill in the **Local Data** dialog. Any variables defined are local to the procedure. Define global variables by pressing the **Global** button in the **Application Tree** dialog.

The ellipsis (...) button next to the **Data** button allows you to view the memory variable declarations at the source code level.

Procedures Calls procedure made in hand-coded, embedded source.

Press this button to access the **Called Procedures** dialog. To add a procedure, press the **Insert** button, and type a procedure name in the next dialog.

If procedure calls already exist, the names appear in the **Called Procedures** dialog. To add another, press the **Add** button. To delete one, press the **Delete** button. Additional buttons allow you to change the order of any procedures listed.

Tip: The purpose of the Procedure button is to add procedures called in embedded source code. The normal way to add template procedures to the Application Tree is to create a menu or toolbar command, add the procedure name via its Actions button, and let the Application Generator automatically add it to the tree.

Embeds Displays the **Embedded Source** dialog. You can then select either a field specific

event or window related action, then add executable source code to customize how the procedure will handle it.

After you choose an embed point in the **Embedded Source** dialog, you choose the code to execute. You can specify a call procedure, which is then added to the tree. You can write your own code with the text editor. Or, you can choose and customize a code template, which is a combination of pre-written code and prompts to "fill-in-the-blanks."

Embedded source gives you complete control over *all* the processing in your procedures. It's one of the most powerful tools Clarion provides you. See also: Adding Embedded Source Code.

Formulas

Accesses the **Formula Editor**, which allows you to create computed and/or conditional fields, which you can then reference in the controls you place in your windows and reports.

Extensions

Accesses extension and Control templates. Extensions, if applicable to the procedure, can be added or modified from this dialog. If any Control templates were placed in the window, this accesses the prompts for those Control templates. Optionally, you can specify whether or not the prompts for an extension or Control template should display on the procedure properties window.

Procedure Properties--Menu

This template provides an SDI (Single Document Interface) window.

The predefined window contains only a single menu (File), containing a single command (Exit).

Description — A short text description for the procedure, which appears next to the procedure name in the **Application Tree** dialog.

Press the ellipsis (...) button to edit a longer (up to 1000 characters) description.

Prototype — Allows you to optionally type a custom procedure prototype which the Application Generator places in the MAP section.

Module Name — The source code file to hold the code for the procedure. Select from the drop down list. By default, the Application Generator names modules by taking the first five characters of the .APP file name, then adding a three digit number for each module.

Export Procedure — Declares the procedure in the export file, enabling it to be called by another aplication. Note:This checkbox is only available when the target file specified in Application Properties as a Dynamic Link Library (.DLL).

Parameters — Allows you to specify parameter names (an optional list of variables separated by commas, with the entire list surrounded by parentheses) for your procedure, which you can pass to it from a calling procedure. You must specify the functionality for the parameters in embedded source code. See also: Procedure and Function Calls.

Window Operation Mode

This option allow you to override the window settings specified in the **Window Properties** dialog. This allows an additional access point to modify the window's operation mode. See also: WINDOW.

Use WINDOW Setting specifies no overrides to the window settings

Normal specifies application modal operation mode. The user must respond before moving to any other window in the application.

MDI specifies that the window conforms to standard MDI child behavior.

Modal specifies system modal operation. A system modal window takes complete control until the window is closed.

INI File Settings — Checking the **Save and Restore Window Location** specifies that a window's location is stored in the application .INI file, and will open in that position the next time the procedure is called. This is available only if you enable INI File settings in the **Global Properties** dialog.

Files — Accesses the **File Schematic Definition** dialog. You can define the procedure's access to variables or other files through the dialog.

Window Calls the Window Formatter, to visually design the window. See also: How to Customize Your Window

The ellipsis (...) button next to the **Window** button allows you to edit the WINDOW or APPLICATION structure at the source code level. Clarion for Windows allows you to easily switch back and forth between editing the window graphically, and editing the source code that describes it.

Tip: Take care when hand-editing code for any WINDOW which contains a control template. The Application Generator stores Template Language attributes which cannot be edited by hand.

Report The Report button is disabled for this procedure type.

Data Adds or edits local variables. Press this button and fill in the **Local Data** dialog. Any variables defined are local to the procedure. Define global variables by pressing the **Global** button in the **Application Tree** dialog.

The ellipsis (...) button next to the **Data** button allows you to view the memory variable declarations at the source code level.

Procedures Calls procedure made in hand-coded, embedded source.

Press this button to access the **Called Procedures** dialog. To add a procedure, press the **Insert** button, and type a procedure name in the next dialog.

If procedure calls already exist, the names appear in the **Called Procedures** dialog. To add another, press the **Add** button. To delete one, press the **Delete** button. Additional buttons allow you to change the order of any procedures listed.

Tip: The purpose of the Procedure button is to add procedures called in embedded source code. The normal way to add template procedures to the Application Tree is to create a menu or toolbar command, add the procedure name via its Actions button, and let the Application Generator automatically add it to the tree.

Embeds Displays the **Embedded Source** dialog. You can then select either a field specific event or window related action, then add executable source code to customize how the procedure will handle it.

After you choose an embed point in the **Embedded Source** dialog, you choose the code to execute. You can specify a call procedure, which is then added to the tree. You can write your own code with the text editor. Or, you can choose and customize a code template, which is a combination of pre-written code and prompts to "fill-in-the-blanks."

Embedded source gives you complete control over *all* the processing in your procedures. It's one of the most powerful tools Clarion provides you. See also: Adding Embedded Source Code.

Formulas Accesses the **Formula Editor**, which allows you to create computed and/or conditional fields, which you can then reference in the controls you place in your windows and reports.

Extensions Accesses extension and Control templates. Extensions, if applicable to the procedure, can be added or modified from this dialog. If any Control templates were placed in the window, this accesses the prompts for those Control

templates. Optionally, you can specify whether or not the prompts for an extension or Control template should display on the procedure properties window.

Procedure Properties--Frame

This template provides an MDI (Multiple Document Interface) parent frame, containing a predefined shell menu. The menu provides useful items such as an Exit command, plus the standard editing and window management commands.

When creating an MDI application, the Frame should be the main procedure. Use the Initiate Thread code template to start new execution threads for each MDI child window which you want to appear inside the frame.

Description A short text description for the procedure, which appears next to the procedure name in the **Application Tree** dialog.

Press the ellipsis (...) button to edit a longer (up to 1000 characters) description.

Prototype Allows you to optionally type a custom procedure prototype which the Application Generator places in the MAP section.

Module Name The source code file to hold the code for the procedure. Select from the drop down list. By default, the Application Generator names modules by taking the first five characters of the .APP file name, then adding a three digit number for each module.

Export Procedure Declares the procedure in the export file, enabling it to be called by another aplication. Note:This checkbox is only available when the target file specified in Application Properties as a Dynamic Link Library (.DLL).

Parameters Allows you to specify parameter names (an optional list of variables separated by commas, with the entire list surrounded by parentheses) for your procedure, which you can pass to it from a calling procedure. You must specify the functionality for the parameters in embedded source code. See also: Procedure and Function Calls.

Window Operation Mode

This option allow you to override the window settings specified in the **Window Properties** dialog. This allows an additional access point to modify the window's operation mode. See also: WINDOW.

Use WINDOW Setting specifies no overrides to the window settings

Normal specifies application modal operation mode. The user must respond before moving to any other window in the allpication.

MDI specifies that the window conforms to standard MDI child behavior.

Modal specifies system modal operation. A system modal window takes complete control until the window is closed.

INI File Settings Checking the **Save and Restore Window Location** specifies that a window's location is stored in the allpication's .INI file, and will open in that position the next time the procedure is called. This is available only if you enable INI File settings in the **Global Properties** dialog.

Files Accesses the **File Schematic Definition** dialog. You can define the procedure's

access to variables or other files through the dialog.

Window Calls the Window Formatter, to visually design the window. See also: How to Customize Your Window

The ellipsis (...) button next to the **Window** button allows you to edit the WINDOW or APPLICATION structure at the source code level. Clarion for Windows allows you to easily switch back and forth between editing the window graphically, and editing the source code that describes it.

Tip: Take care when hand-editing code for any WINDOW which contains a control template. The Application Generator stores Template Language attributes which cannot be edited by hand.

Report The Report button is disabled for this procedure type.

Data Adds or edits local variables. Press this button and fill in the **Local Data** dialog. Any variables defined are local to the procedure. Define global variables by pressing the **Global** button in the **Application Tree** dialog.

The ellipsis (...) button next to the **Data** button allows you to view the memory variable declarations at the source code level.

Procedures Calls procedure made in hand-coded, embedded source.

Press this button to access the **Called Procedures** dialog. To add a procedure, press the **Insert** button, and type a procedure name in the next dialog.

If procedure calls already exist, the names appear in the **Called Procedures** dialog. To add another, press the **Add** button. To delete one, press the **Delete** button. Additional buttons allow you to change the order of any procedures listed.

Tip: The purpose of the Procedure button is to add procedures called in embedded source code. The normal way to add template procedures to the Application Tree is to create a menu or toolbar command, add the procedure name via its Actions button, and let the Application Generator automatically add it to the tree.

Embeds Displays the **Embedded Source** dialog. You can then select either a field specific event or window related action, then add executable source code to customize how the procedure will handle it.

After you choose an embed point in the **Embedded Source** dialog, you choose the code to execute. You can specify a call procedure, which is then added to the tree. You can write your own code with the text editor. Or, you can choose and customize a code template, which is a combination of pre-written code and prompts to "fill-in-the-blanks."

Embedded source gives you complete control over *all* the processing in your procedures. It's one of the most powerful tools Clarion provides you. See also: Adding Embedded Source Code.

Formulas Accesses the **Formula Editor**, which allows you to create computed and/or conditional fields, which you can then reference in the controls you place in your windows and reports.

Extensions Accesses extension and Control templates. Extensions, if applicable to the

procedure, can be added or modified from this dialog. If any Control templates were placed in the window, this accesses the prompts for those Control templates. Optionally, you can specify whether or not the prompts for an extension or Control template should display on the procedure properties window.

Procedure Properties--External

The External Procedure Template declares a procedure is contained in an external library (*.LIB only) or object file. The Application Generator writes no source code. The project system links in the external file as a module.

After selecting the External template type from the **Select Procedure Type** dialog, choose OBJ or LIB from the **Select Module Type** dialog.

Type the file name of the external library or object file in the **Module Name** field. Optionally type parameter declarations in the **Prototype** field.

Description — A short text description for the procedure, which appears next to the procedure name in the **Application Tree** dialog.

Press the ellipsis (...) button to edit a longer (up to 1000 characters) description.

Prototype — Allows you to optionally type a custom procedure prototype which the Application Generator places in the MAP section.

Module Name — The source code file to hold the code for the procedure. Select from the drop down list. By default, the Application Generator names modules by taking the first five characters of the .APP file name, then adding a three digit number for each module. See also: See also: Using DLLs not created in Clarion for Windows

The MODULE name for an External procedure should *not* be modified. (901)

Export Procedure — Declares the procedure in the export file, enabling it to be called by another aplication. Note:This checkbox is only available when the target file specified in Application Properties as a Dynamic Link Library (.DLL).

Files — This button is not valid for this procedure type.

Window — The Window button is disabled for this procedure type.

Report — The Report button is disabled for this procedure type.

Data — This button is not valid for this procedure type.

Procedures — Calls procedure made in hand-coded, embedded source.

Press this button to access the **Called Procedures** dialog. To add a procedure, press the **Insert** button, and type a procedure name in the next dialog.

If procedure calls already exist, the names appear in the **Called Procedures** dialog. To add another, press the **Add** button. To delete one, press the **Delete** button. Additional buttons allow you to change the order of any procedures listed.

Tip: The purpose of the Procedure button is to add procedures called in embedded source code. The normal way to add template procedures to the Application Tree is to create a menu or toolbar command, add the procedure name via its Actions button, and let the Application Generator automatically add it to the tree.

Embeds — This button is not valid for this procedure type.

Formulas	This button is not valid for this procedure type.
Extensions	This button is not valid for this procedure type.

Procedure Properties--Viewer

The Viewer procedure template provides a prepopulated window which allows you to view, search, and print an ASCII (text) file.

Description A short text description for the procedure, which appears next to the procedure name in the **Application Tree** dialog.

Press the ellipsis (...) button to edit a longer (up to 1000 characters) description.

Prototype Allows you to optionally type a custom procedure prototype which the Application Generator places in the MAP section.

Module Name The source code file to hold the code for the procedure. Select from the drop down list. By default, the Application Generator names modules by taking the first five characters of the .APP file name, then adding a three digit number for each module.

Export Procedure Declares the procedure in the export file, enabling it to be called by another aplication. Note:This checkbox is only available when the target file specified in Application Properties as a Dynamic Link Library (.DLL).

Parameters Allows you to specify parameter names (an optional list of variables separated by commas, with the entire list surrounded by parentheses) for your procedure, which you can pass to it from a calling procedure. You must specify the functionality for the parameters in embedded source code. See also: Procedure and Function Calls.

Window Operation Mode

This option allow you to override the window settings specified in the **Window Properties** dialog. This allows an additional access point to modify the window's operation mode. See also: WINDOW.

Use WINDOW Setting specifies no overrides to the window settings

Normal specifies application modal operation mode. The user must respond before moving to any other window in the application.

MDI specifies that the window conforms to standard MDI child behavior.

Modal specifies system modal operation. A system modal window takes complete control until the window is closed.

INI File Settings Checking the **Save and Restore Window Location** specifies that a window's location is stored in the application .INI file, and will open in that position the next time the procedure is called. This is available only if you enable INI File settings in the **Global Properties** dialog.

Files Accesses the **File Schematic Definition** dialog. You can define the procedure's access to variables or other files through the dialog.

Window Calls the Window Formatter, to visually design the window. See also: How to Customize Your Window

The ellipsis (...) button next to the **Window** button allows you to edit the

WINDOW or APPLICATION structure at the source code level. Clarion for Windows allows you to easily switch back and forth between editing the window graphically, and editing the source code that describes it.

Tip: Take care when hand-editing code for any WINDOW which contains a control template. The Application Generator stores Template Language attributes which cannot be edited by hand.

Report The Report button is disabled for this procedure type.

Data Adds or edits local variables. Press this button and fill in the **Local Data** dialog. Any variables defined are local to the procedure. Define global variables by pressing the **Global** button in the **Application Tree** dialog.

The ellipsis (...) button next to the **Data** button allows you to view the memory variable declarations at the source code level.

Procedures Calls procedure made in hand-coded, embedded source.

Press this button to access the **Called Procedures** dialog. To add a procedure, press the **Insert** button, and type a procedure name in the next dialog.

If procedure calls already exist, the names appear in the **Called Procedures** dialog. To add another, press the **Add** button. To delete one, press the **Delete** button. Additional buttons allow you to change the order of any procedures listed.

Tip: The purpose of the Procedure button is to add procedures called in embedded source code. The normal way to add template procedures to the Application Tree is to create a menu or toolbar command, add the procedure name via its Actions button, and let the Application Generator automatically add it to the tree.

Embeds Displays the **Embedded Source** dialog. You can then select either a field specific event or window related action, then add executable source code to customize how the procedure will handle it.

After you choose an embed point in the **Embedded Source** dialog, you choose the code to execute. You can specify a call procedure, which is then added to the tree. You can write your own code with the text editor. Or, you can choose and customize a code template, which is a combination of pre-written code and prompts to "fill-in-the-blanks."

Embedded source gives you complete control over *all* the processing in your procedures. It's one of the most powerful tools Clarion provides you. See also: Adding Embedded Source Code.

Formulas Accesses the **Formula Editor**, which allows you to create computed and/or conditional fields, which you can then reference in the controls you place in your windows and reports.

Extensions Accesses extension and Control templates. Extensions, if applicable to the procedure, can be added or modified from this dialog. If any Control templates were placed in the window, this accesses the prompts for those Control templates. Optionally, you can specify whether or not the prompts for an extension or Control template should display on the procedure properties window.

See also:

ASCII Box

ASCII Search Button

ASCIIPrint Button

CloseButton

Procedure Properties--Source 

The Source Procedure template provides an elegant and simple way to add hand code to your application. It provides two points at which to embed your code: the data section, and the code section.

The template simply declares the procedure, handles any optional parameters, places the embedded data declarations in the data section, begins the CODE section, then places any embedded executable code in the CODE section:

Description — A short text description for the procedure, which appears next to the procedure name in the **Application Tree** dialog.

Press the ellipsis (...) button to edit a longer (up to 1000 characters) description.

Prototype — Allows you to optionally type a custom procedure prototype which the Application Generator places in the MAP section.

Module Name — The source code file to hold the code for the procedure. Select from the drop down list. By default, the Application Generator names modules by taking the first five characters of the .APP file name, then adding a three digit number for each module.

Export Procedure — Declares the procedure in the export file, enabling it to be called by another aplication. Note:This checkbox is only available when the target file specified in Application Properties as a Dynamic Link Library (.DLL).

Parameters — Allows you to specify parameter names (an optional list of variables separated by commas, with the entire list surrounded by parentheses) for your procedure, which you can pass to it from a calling procedure. You must specify the functionality for the parameters in embedded source code. See also: Procedure and Function Calls.

Files — Accesses the **File Schematic Definition** dialog. You can define the procedure's access to variables or other files through the dialog.

Window — The Window button is disabled for this procedure type.

Report — The Report button is disabled for this procedure type.

Data — Adds or edits local variables. Press this button and fill in the **Local Data** dialog. Any variables defined are local to the procedure. Define global variables by pressing the **Global** button in the **Application Tree** dialog.

The ellipsis (...) button next to the **Data** button allows you to view the memory variable declarations at the source code level.

Procedures — Calls procedure made in hand-coded, embedded source.

Press this button to access the **Called Procedures** dialog. To add a procedure, press the **Insert** button, and type a procedure name in the next dialog.

If procedure calls already exist, the names appear in the **Called Procedures** dialog. To add another, press the **Add** button. To delete one, press the **Delete** button. Additional buttons allow you to change the order of any procedures listed.

Tip: **The purpose of the Procedure button is to add procedures called in embedded source code. The normal way to add template procedures to the Application Tree is to create a menu or toolbar command, add the procedure name via its Actions button, and let the Application Generator automatically add it to the tree.**

Embeds

Displays the **Embedded Source** dialog. You can then select either a field specific event or window related action, then add executable source code to customize how the procedure will handle it.

After you choose an embed point in the **Embedded Source** dialog, you choose the code to execute. You can specify a call procedure, which is then added to the tree. You can write your own code with the text editor. Or, you can choose and customize a code template, which is a combination of pre-written code and prompts to "fill-in-the-blanks."

Embedded source gives you complete control over *all* the processing in your procedures. It's one of the most powerful tools Clarion provides you. See also: Adding Embedded Source Code.

Formulas

This button is not valid for this procedure type.

Extensions

This button is not valid for this procedure type.

Procedure Properties--Report

This procedure enables you to create reports. Press the **Report** button in the **Procedure Properties** dialog to create your report. The procedure template includes a window to show the progress of the report processing. The **Procedure Properties** dialog also includes a checkbox to specify whether you wish to generate a print preview function for your report.

Description — A short text description for the procedure, which appears next to the procedure name in the **Application Tree** dialog.

Press the ellipsis (...) button to edit a longer (up to 1000 characters) description.

Prototype — Allows you to optionally type a custom procedure prototype which the Application Generator places in the MAP section.

Module Name — The source code file to hold the code for the procedure. Select from the drop down list. By default, the Application Generator names modules by taking the first five characters of the .APP file name, then adding a three digit number for each module.

Export Procedure — Declares the procedure in the export file, enabling it to be called by another aplication. Note:This checkbox is only available when the target file specified in Application Properties as a Dynamic Link Library (.DLL).

Parameters — Allows you to specify parameter names (an optional list of variables separated by commas, with the entire list surrounded by parentheses) for your procedure, which you can pass to it from a calling procedure. You must specify the functionality for the parameters in embedded source code. See also: Procedure and Function Calls.

Window Message — The message to display in the progress dialog.

Files — Accesses the **File Schematic Definition** dialog. You can define the procedure's access to variables or other files through the dialog.

Window — The Window button is disabled for this procedure type.

Report — Press this button to call the Report Formatter to visually design the window. See also: How to Use the Report Formatter -- An Overview

The ellipsis (...) button next to the **Report** button allows you to edit the REPORT structure at the source code level. Clarion for Windows allows you to easily switch back and forth between editing the report graphically, and editing the source code that describes it.

Data — Adds or edits local variables. Press this button and fill in the **Local Data** dialog. Any variables defined are local to the procedure. Define global variables by pressing the **Global** button in the **Application Tree** dialog.

The ellipsis (...) button next to the **Data** button allows you to view the memory variable declarations at the source code level.

Procedures — Calls procedure made in hand-coded, embedded source.

Press this button to access the **Called Procedures** dialog. To add a procedure, press the **Insert** button, and type a procedure name in the next dialog.

If procedure calls already exist, the names appear in the **Called Procedures** dialog. To add another, press the **Add** button. To delete one, press the **Delete** button. Additional buttons allow you to change the order of any procedures listed.

Tip: The purpose of the Procedure button is to add procedures called in embedded source code. The normal way to add template procedures to the Application Tree is to create a menu or toolbar command, add the procedure name via its Actions button, and let the Application Generator automatically add it to the tree.

Embeds Displays the **Embedded Source** dialog. You can then select either a field specific event or window related action, then add executable source code to customize how the procedure will handle it.

After you choose an embed point in the **Embedded Source** dialog, you choose the code to execute. You can specify a call procedure, which is then added to the tree. You can write your own code with the text editor. Or, you can choose and customize a code template, which is a combination of pre-written code and prompts to "fill-in-the-blanks."

Embedded source gives you complete control over *all* the processing in your procedures. It's one of the most powerful tools Clarion provides you. See also: Adding Embedded Source Code.

Formulas Accesses the **Formula Editor**, which allows you to create computed and/or conditional fields, which you can then reference in the controls you place in your windows and reports.

Extensions Accesses extension and Control templates. Extensions, if applicable to the procedure, can be added or modified from this dialog. If any Control templates were placed in the window, this accesses the prompts for those Control templates. Optionally, you can specify whether or not the prompts for an extension or Control template should display on the procedure properties window.

Print Preview Check this box to enable previewing of a report before printing.

Quick-Scan Records Specifies buffered access behavior for ODBC, ASCII, DOS, or BASIC files. These file drivers read a buffer at a time (not a record), allowing for fast access. In a multi-user environment these buffers are not 100% trustworthy for subsequent access, because another user may change the file between accesses. As a safeguard, the driver rereads the buffers before each record access. To *disable* the reread, enable QUICKSCAN.

Access the following prompts by pressing the **Range and Filter** button.

Record Filter Type an expression to limit the contents of the browse list to only those records which match the filter expression. This filters all displayable records. When a Record filter is used in conjunction with a Range Limit, only those records within the specified range are filtered. See also: **Using Range Limits and Filters**

Range Limit Field Type in the field name or press the ellipsis (...) button to select the field from the Component list. The Range Limit Field must be a component of the report's Access Key. The range limit is key-dependent; the generated source code uses the SET

statement to find the first valid record.

Range Limit Type When a field is selected for **Range Limit Field**, this specifies the method of determining the records for inclusion in the list box.

Current Value -- Signifies the value contained in the key field at the beginning of the ACCEPT loop. This is the value used for the range for the duration of the procedure.

Single Value -- Specifies a variable containing the limiting value. Only records matching the variable are included. Enter a variable in the **Range Limit Value** box which appears, or press the ellipses (...) button to select the variable from the File Schematic.

Range of Values -- Allows you to specify upper and lower limits. Enter a variable in the **Low Limit** and **High Limit Value** boxes which appears, or press the ellipses (...) button to select the variables from the File Schematic.

Request and Response

One of the biggest concerns of template design is inter-procedure communication. The added dimension of multi-threading only serves to make the resolution of this problem more crucial.
In a generic template-driven system, it is impossible to require that parameters be supported in templates. It's never certain if a Browse will be calling a Form, or if it calls a Report, etc. In fact, with Control Templates, a form can also *be* a browse, and an ASCII viewer. To require users to know all of the different parameters and their values is unreasonable. Further, building in support for functions overcomplicates the templates to the point of unusability. Again, add in the complications of multithreading and the system is unusable and unmaintainable.
Using global variables is acceptable, with the THREAD attribute insuring that the variable itself is safe within a thread. Unfortunately, the value of any global variable must be called into question as soon as any EMBED point is encountered, as the value could change with another procedure call, etc.
Any communications variable must therefore have as little happen from the time it is assigned a value and the time that value is interpreted. This time is referred to as the *Span* of the variable. The shorter span, the better the integrity of the system. The communications variable should also be considered suspect as soon as possible. The amount of time that the variable is considered to have a valid value is referred to as *Live Time*. If a variable has a short live time, its less likely to be subject to misinterpretation, and again system integrity benefits.
To this end, we've implemented a *Request and Response* system in the Clarion for Windows templates. This system was created to maintain the integrity of interprocedure communications in a fully generated system. In other words, if no embedded source is used and no hand-coded modules are used, confidence in system integrity is high.
There are three components to the Request and Response system:

Global Variables: GlobalRequest and GlobalResponse. In GENERATED code, there are no points between the place that either variable is assigned a value and the place that that value is interpreted. These variables are defined as:

```
GlobalRequest     LONG,THREAD
GlobalResponse    LONG,THREAD
```

NOTE: If you are creating an application that consists of more than one AppGen created DLL, you MUST check the "Generate Internal Global Data as EXTERNAL" check box for all DLLs except one. Likewise, you MUST check the "Generate Internal Global Data as EXTERNAL" check box for each APP creating an .EXE.

Local Variables: LocalRequest, LocalResponse, OriginalRequest. LocalRequest and OriginalRequest are assigned value immediately after the CODE statement. LocalResponse is assigned a value before a bit of code signals the exit of the procedure. Right before the procedure RETURNs, GlobalResponse is assigned the value of LocalResponse.

Enumerated EQUATEs:
These are primarily to increase readability of the code. The actual numbers themselves are inconsequential, with one exception; Request values less than 0 are reverved for use in multi-page systems.

```
InsertRecord      EQUATE (1)      !  Add a record to table
ChangeRecord      EQUATE (2)      !  Change the current record
DeleteRecord      EQUATE (3)      !  Delete the current record
SelectRecord      EQUATE (4)      !  Select the current record
RequestCompleted  EQUATE (1)      !  Update Completed
RequestCancelled  EQUATE (2)      !  Update Aborted
```

Using Range Limits and Filters

There are many times that you will want to view, process, or report a sub-set of records from a file. There are two ways to do this:

Range Limit
Filters

Each method has its tradeoffs. Range Limits are much faster to process, but they require that a procedure or control use a limited key as the primary key. Filters are more flexible, since they don't require any special key manipulation, but they are much slower. In any procedure that does sequential processing, you can specify a Range Limit Field, and the type of Range Limit you want to use. The types provided are:

Current Value -Value Limited Keyed Access

The key element specified in the Range Limit Field prompt is the final Fixed Key Element. With this kind of Range Limit, the value of all Fixed Key Elements are saved when the procedure is initialized. These values are used for the duration of the procedure.

Single Value-Value Limited Keyed Access

The key element specified in the Range Limit Field prompt is the final Fixed Key Element. With this kind of Range Limit, the values of all Fixed Key Elements EXCEPT the final Fixed Key Element are saved when the procedure is initialized. The final Fixed Key Element is assigned the value specified in the Range Limit value prompt. This value can be either a variable or fixed value. This value is reevaluated each time a page or entry is loaded or processed.

Range of Values -Ranged Key Access

The key element specified in the Range Limit Field prompt is the first Free Key Element. With this kind of Range Limit, the values of all Fixed Key Elements *except* the final Fixed Key Element are saved when the procedure is initialized. The Low Limit and High Limit Values are used to set the keys for sequential access and to evaluate each record read to insure it is within the valid range. These values can be either fixed values or variables. If variables are used, these variables are reevaluated each time a page or entry is loaded or processed.

Browse Box Only - File Relationship - Value Limited Key Access

All Fixed Key Elements are assigned values as defined in a relationship in the data dictionary. With this kind of Range Limit, it is possible to have multiple Browse Box control templates populated on a window, and as long as the relationships are defined and used, when a parent Browse Box goes out of range, all children (and grandchildren, etc.) Browse Boxes and Controls will automatically be reconstructed.

You *must* BIND any variables used in a filter expression. Add the variable to the **Hot Fields** list, and check the **Bind Field** box.

Global Properties

This dialog specifies application level options for file processing, .INI file support, plus allows you to define global variables.

This topic also provides help for two additional dialogs which you can access through the **Global Properties** dialog. They help you to optionally manage the file controls individually, rather than globally, and are explained below.

Program Author Allows you to add your own name, which is then added into the .APP file.

MSG The **Use Field Description as MSG() When MSG() is Blank** checkbox specifies the Application Generator will fill the MSG field in the Field Properties dialogs with the text you type in the Description field.

EXTERNAL The **Generate Global Data as EXTERNAL** checkbox specifies that any global variables should automatically receive the EXTERNAL attribute. This setting is most often used when creating an application that uses a dynamic link library (.DLL) which already has the Global Data declared.

NOTE: If you are creating an application that consists of more than one AppGen created DLL, you MUST check the "Generate Internal Global Data as EXTERNAL" check box for all DLLs except one. Likewise, you MUST check the "Generate Internal Global Data as EXTERNAL" check box for each APP creating an .EXE.

Data Press the **Data** button to open the **Global Data** dialog, which allows you to declare or edit global variables.

Embeds This provides access to the global data section, to the points at which the default application opens all files, and at the point when processing errors in case of a problem opening the files. The embed points include before opening the file, immediately after opening a file with no error, opening the file with an error, upon finding a bad key, and others.

To access the embed points, press the **Embed** button. As with any embed point, you can write your own custom code, call a procedure, or use a code template. The Application Generator, when generating code, places your code or calls your procedure at the next source code line following the point you pick from the Embedded Source dialog.

See also *Adding Embedded Source Code*, for more information on adding embedded source code to the code generated by the Application Generator.

INI File Settings

Allows you to specify that the application should create and use a standard .INI (initialization) file. This file is an ASCII file which stores variables for an application between session.

Use .INI file to save and restore program settings

You can automatically specify, for example, that the program remember where the end user placed the window(s) at the end of each session, by checking the **Use .INI File to Save and Restore Program Settings** box.

.INI File to use Once you check the box, optionally specify the .INI file name. By default, the

application creates a .INI file with the same file name as your application. If you wish to create a custom name, select the **Other** choice from the **.INI File to Use** drop down list, then type in a file name in the **Other File Name** box.

File Control Flags

The **File Control Flags** dialog allows you to override some of the settings in your dictionary, as well as define how procedures will access files. You can specify file attributes for all files, or individually.

Generate all File Declarations

Generates all file declarations in the dictionary, even if not specified in any procedure's file schematic.

When Done Specifies whether the application automatically closes each file when a procedure is finished.

Tip: One way in which you can design your application to be a "well-behaved" Windows application is not to hog system resources. One limited resource is file handles. You can "give back" file handles not in use with the Close Unused Files checkbox.

Enclose RI code in transaction frame

Enables rollback of data if an update fails.

Tip: If all files in a relation chain are using the same file system, and the file system supports transaction framing, and you do not want transaction framing around the RI code, you must clear the check box for *each* file in Individual File Overrides *and* in Global Settings. (0003)

LOGOUT() When your data dictionary includes a file driver which does not support the LOGOUT() function, which is used in the Referential Integrity checking routines, this enables a warning at compile time.

You should be sure that the **Issue Template Warning if LOGOUT() is Not Allowed** check box is *unchecked* for drivers such as dBase III. See also: *Database Drivers* .

File Attributes

Threaded Specifies whether your application should add the THREAD attribute to the FILE structure.

This is required for multiple-thread MDI access for browse and form procedure, to prevent record buffer conflicts when the end user changes focus from one thread to another.

You can use the file setting, as defined in the data dictionary, or optionally specify that all files be threaded, or none.

Create Specifies whether your application should allow the creation of a data file should it not exist, choose

You can use the file setting, as defined in the data dictionary, or optionally specify that all files be created, or none.

External Adds the EXTERNAL attribute. This is useful when working with dynamic link libraries. You can additionally specify which module should contain the declarations

File Access

Open Mode

To specify that your application always open a file in share mode, choose **Share** from the **File Open Mode** drop down box. To disable Shared mode access, choose **Open**.

Choose Other to specify exactly how the end user of your application accesses the file.

In the **Other Open Mode** options, which then appear, you can specify that your application opens the file in read only mode, write only, or read and write.

You can specify that other users, when your application has a file open, are denied write access, read access, read and write access, or are not denied any access.

Individual File Overrides

The Individual **File Overrides** button allows you to change the settings of the files you select. These include the file attributes, open mode, and what to do when done with the file.

Press the button to open a dialog which helps you to manage the overrides. Select a file, then press the **Edit** button. In the dialog which then opens, you can set File Control Flags for just that file.

Free Key Element

Any Element of a key whose value is fluid for the duration of a browse or report. No Free key Element may occur at any point in the key before the final Fixed Key Element.

Module Properties

This dialog allows you to specify settings for an individual source code document file. You must first view the Application Tree in module view to access this dialog. To do so, you choose **View**  **Module** View from the IDE menu. Then press the **Properties** button to open this dialog.

Name	Allows you to specify the file name for the module.
Description	Allows you to add a short description, which appears in the Application Tree when in Module View.
Type	Allows you to choose from the **Select Module Type** dialog.
Allow Repopulate	Specifies the Application Generator may move procedures from this and other modules.
Map Include File	Allows you to specify a source code file to include in the data declarations section of the module.

Module Properties

This dialog allows you to specify settings for an individual source code document file. You must first view the Application Tree in module view to access this dialog. To do so, you choose **View**  **Module** View from the IDE menu. Then press the **Properties** button to open this dialog.

Name	Allows you to specify the file name for the module.
Description	Allows you to add a short description, which appears in the Application Tree when in Module View.
Type	Allows you to choose from the **Select Module Type** dialog.
Allow Repopulate	Specifies the Application Generator may move procedures from this and other modules.
Map Include File	Allows you to specify a source code file to include in the data declarations section of the module.

Module Properties

This dialog allows you to specify settings for an individual source code document file. You must first view the Application Tree in module view to access this dialog. To do so, you choose **View**  **Module** View from the IDE menu. Then press the **Properties** button to open this dialog.

Name	Allows you to specify the file name for the module.
Description	Allows you to add a short description, which appears in the Application Tree when in Module View.
Type	Allows you to choose from the **Select Module Type** dialog.
Allow Repopulate	Specifies the Application Generator may move procedures from this and other modules.
Map Include File	Allows you to specify a source code file to include in the data declarations section of the module.

Module Properties

Use this Module template when inserting an External Dynamic Link Library. This add the DLL() attribute to exported procedures.

This dialog allows you to specify the settings for an External .DLL file. You must first view the Application Tree in module view to access this dialog. To do so, you select the **Module** tab from the IDE menu. Then press the **Properties** button to open this dialog.

Name	Allows you to specify the file name for the module.
Description	Allows you to add a short description, which appears in the Application Tree when in Module View.
Type	Allows you to choose from the **Select Module Type** dialog.
Allow Repopulate	Specifies the Application Generator may move procedures from this and other modules.
Map Include File	Allows you to specify a source code file to include in the data declarations section of the module.

Program Properties

This dialog allows you to specify settings for an individual source code document file. You must first view the Application Tree in module view to access this dialog. To do so, you choose **View**  **Module** View from the IDE menu. Then press the **Properties** button to open this dialog.

Name	Allows you to specify the file name for the module.
Description	Allows you to add a short description, which appears in the Application Tree when in Module View.
Type	Allows you to choose from the **Select Program Type** dialog.
Allow Repopulate	Specifies the Application Generator may move procedures from this and other modules.
Map Include File	Allows you to specify a source code file to include in the data declarations section of the module.

Refresh Window routine

The Refresh Window routine is used by the templates to keep the values displayed current. This routine is called whenever a control's value is modified, or in the case of a list box, a different record is highlighted. The routine reevaluates the display conditions for all window controls, and displays the correct value.

Alpha distribution

The Alpha distribution divides the scroll bar into 26 segments--one for each letter of the alphabet. Moving the thumb to a segment displays the first record beginning with the segment corresponding letter. For example, moving the thumb to the third segment displays the first record beginning with the letter **c**. If no records begin with **c**, the record with the closest higher value is highlighted.

Last Names distribution

The Last Names distribution divides the scroll bar into 100 segments. Each of these segments is assigned a value based on the distribution of names in an average U.S. telephone book. Each segment is assigned a value and positioning the thumb at that segment displays the first record matching that value. If no records begin with the value, the record with the next closest higher value is highlighted.
The pre-defined Last name distribution is a more accurate method than Alpha when displaying names because names are not evenly distributed alphabetically.
For example, the Browse Box control template uses these values
<null>
ALB
AME
ARN
BAK
BAT
BEN
...

Positioning the thumb on the third segment, highlights the first record beginning with the letters AME. If no records match, the next highest next closest higher value is highlighted.

Fixed Thumb

A fixed thumb positions the thumb (or elevator bar) in the center of the scroll bar. Clicking in the scroll bar above the thumb moves up one page at a time. Clicking in the scroll bar below the thumb moves down one page at a time.

Movable Thumb

A movable thumb positions the thumb (or elevator bar) at the top of the scroll bar when the browse box is initialized. Clicking in the scroll bar above the thumb moves up one page at a time. Clicking in the scroll bar below the thumb moves down one page at a time. Draging the the thumb up or down to a position, highlights the closest matching record for that position in the scroll bar. This is dependant on the type of vertical scroll bar behavior you specify.
See Alpha, Last Names, Custom, or Runtime.

Custom

This allows you to specify the break points for distribution along the scroll bar. This is useful when you have data with a skewed distribution. Insert the values for each point in the list. This divides the length of the scroll bar into segments. Each value you insert in the list creates a segment. For example, if you specify ten values, the scroll bar is divided into ten segments. Positioning the thumb on the third segment, highlights the first record beginning with the value of the value of the third item you've inserted in the list. If no records match, the next highest next closest higher value is highlighted.
You can use numeric or string constants, or variables. String constants should be enclosed in single quotes (').

Runtime

The Browse Box is initialized and computes the values for 100 break points based on the first and last record in the Range.The distribution points are determined only when the file is opened, therefore there is no performance penalty. The lowest value in the key is subtracted from the highest value to estimate the range of numbers, 100 evenly distributed points in that range are determined and used to control the vertical scroll bar behavior.

Step

The user types in a single character to advance the cursor bar in the list box to the record that contains the nearest match in the key field. Use this type of locator only when the first Free Key Element is a STRING, CSTRING, or PSTRING. If no free key element is available, the application generator converts a step locator to None.

Entry Locator

An entry box holds the value for the locator. When the end user places a value in the entry box, pressing TAB or reselecting the list box moves the selection to the nearest matching record. If an entry control is not placed in the window, the application generator converts an Entry locator to a Step locator.

If you use the same field more than once as a locator, you must override the default locator. For example, if you have a multi-keyed browse which has an ascending key and a descending key on the same field. To use a separate controls (as on separate TABs) for each condition, check the override box and select the second instance from the drop-down list.

Incremental Locator

When the end user types one or more characters, the list box moves the selection to the nearest matching record. Backspace clears the characters, one-by-one, moving the highlight bar to the nearest matching record of the remaining characters. If a STRING control is placed in the window, the characters display, allowing the user to see the characters as they are entered or cleared. If an ENTRY control is placed in the window, the locator works like an **Entry** Locator when the entry control has focus.

If you use the same field more than once as a locator, you must override the default locator. For example, if you have a multi-keyed browse which has an ascending key and a descending key on the same field. To use a separate controls (as on separate TABs) for each condition, check the override box and select the second instance from the drop-down list.

Current Value

Signifies the value contained in the key field at the beginning of the ACCEPT loop. This is the value used for the range for the duration of the procedure.

Single Value

Specifies a variable containing the limiting value. Only records matching the variable are included. The variable is reevaluated whenever the window is refreshed (See also: Refresh Window routine). Enter a variable in the Range Limit Value box which appears, or press the ellipses (...) button to select the variable from the File Schematic.

Range of Values

Allows you to specify upper and lower limits. Enter a variable in the Low Limit and High Limit Value boxes which appears, or press the ellipses (...) button to select the variables from the File Schematic.

File Relationship

Allows you to choose a range limiting file from a 1:MANY relationship. The Range Limiting field must be the "One" side of a One-to-Many Relationship with the Browse Box's Primary File. The relation's linking key must be the same as the Access Key for the Browse Box. Enter a file in the Related File box, or press the ellipses (...) button to select it from the File Schematic.

Fixed Key Element

An element of a KEY which has a fixed value for th displayable records. For example, when setting a range limit of a single value, all key components up to and including the range limit field have a fixed value. Any key components following the Fixed Key Elements are Free Key Elements.

Application Wizard utility template

This wizard creates a complete application from an existing dictionary. It creates a Frame containing a menu with options calling all procedures it creates. It also creates Browse and Report procedures each specified file, with associated Form (Update) procedures.

To use the Application Wizard:

1. Optionally, in File Manager, choose **File** **Create Directory**, type a subdirectory name and press **OK**.

Or else, use the DOS prompt, and the MkDir command.

2. Choose **File** **New** (or press the

button on the toolbar)**.**

The **New** file dialog appears.

3. Choose **Application** by CLICKING on the tab.

4. Type a name for the .APP file in the **Application File** field. If you want to use the Quick Start wizard, check the box below the file list. See Using the Quick Start Wizard.

Type a legal DOS filename. Clarion automatically adds the .APP extension.

The **Application Properties** dialog appears. This dialog allows you to define the essential files for the application.

5. Name the .DCT file the application will use in the **Dictionary File** field, or press the ellipsis (**...**) button to select the file in the **Select Dictionary** dialog.

See How to Create a Data Dictionary for information on creating your application's data dictionary. The **Select Dictionary** dialog is a standard **Open File** dialog.

The Application Generator does *not* require a data dictionary to generate an application, if you *uncheck* the **Require a dictionary** box in the **Application Options** dialog.

6. Do Not rename the first procedure from MAIN.

7. Choose the **Destination Type** from the drop down list.

This defines the type of target file for your application. Choose from **Executable** (.EXE), **Library** (.LIB), or **Dynamic Link Library** (.DLL).

8. Optionally, type a name for the application's .HLP file in the **Help File** field, or use the ellipsis (**...**) button to select the file in the Open File dialog.

The Application Generator does *not* require that the .HLP file exist at this point. You can leave the field blank for now, then fill in the field at a later time.

The Application Generator allows you to name the help topics in your application without determining that the help file exists. You are responsible for creating a .HLP file that contains the context strings and keywords that you optionally enter as HLP attributes for the various controls and dialogs.

9. Accept the default Clarion template in the **Application Template** field.

The selected application template controls code generation.

10. Check the **Application Wizard** box to use the wizard to create a complete application based on the selected dictionary and a few answers you specify.

11. Press the **OK** button.

The Application Wizard dialogs appear.

12. Answer the question(s) in each dialog, then press the **Next** button. On the last dialog, the **Finish**

button is enabled. If you are satisfied with your answers, press the **Finish** button.

The Application Wizard creates the .APP file based on the dictionary and the answers you provided, then displays the **Application Tree** dialog for your new application.

You can control some of the wizard options in the Data Dictionary by specifying **Options** for Files, Fields, Keys, or relations. See Using Wizard Options for more information.

Browse Procedure Wizard utility template

This wizard creates a multi-keyed Browse Procedure from an existing dictionary file definition. It also creates associated Form (Update) procedures, if you specify that updates are allowed.

To use the Browse Procedure Wizard:

1. Highlight a ToDo Procedure in the Procedure Tree and press enter.

The Select Procedure dialog appears.

2. Select **Browse** from the list of Procedure templates.

3. Check the **Procedure Wizard** box to use the wizard to create the procedure based on the selected dictionary file and a few answers you specify.

4. Press the **Select** button.

The **Procedure Wizard** dialogs appear.

5. Answer the question(s) in each dialog, then press the **Next** button. On the last dialog, the **Finish** button is enabled. If you are satisfied with your answers, press the **Finish** button.

The Procedure Wizard creates the procedure(s) based on the dictionary file and the answers you provided, then displays the Procedure Properties dialog for your new procedure.

You can control some of the wizard options in the Data Dictionary by specifying **Options** for Files, Fields, Keys, or relations. See Using Wizard Options for more information.

Form Wizard utility template

This wizard creates an update Form Procedure from an existing dictionary file definition.

To use the Form Procedure Wizard:

1. Highlight a ToDo Procedure in the Procedure Tree and press enter.

The Select Procedure dialog appears.

2. Select **Form** from the list of Procedure templates.

3. Check the **Procedure Wizard** box to use the wizard to create the procedure based on the selected dictionary file and a few answers you specify.

4. Press the **Select** button.

The **Procedure Wizard** dialogs appear.

5. Answer the question(s) in each dialog, then press the **Next** button. On the last dialog, the **Finish** button is enabled. If you are satisfied with your answers, press the **Finish** button.

The Procedure Wizard creates the procedure based on the dictionary file and the answers you provided, then displays the Procedure Properties dialog for your new procedure.

You can control some of the wizard options in the Data Dictionary by specifying **Options** for Files, Fields, Keys, or relations. See Using Wizard Options for more information.

Report Wizard utility template

This wizard creates a Report Procedure from an existing dictionary file definition.

To use the Report Procedure Wizard:

1. Highlight a ToDo Procedure in the Procedure Tree and press enter.

The Select Procedure dialog appears.

2. Select **Report** from the list of Procedure templates.

3. Check the **Procedure Wizard** box to use the wizard to create the procedure based on the selected dictionary file and a few answers you specify.

4. Press the **Select** button.

The **Procedure Wizard** dialogs appear.

5. Answer the question(s) in each dialog, then press the **Next** button. On the last dialog, the **Finish** button is enabled. If you are satisfied with your answers, press the **Finish** button.

The Procedure Wizard creates the procedure based on the dictionary file and the answers you provided, then displays the Procedure Properties dialog for your new procedure.

You can control some of the wizard options in the Data Dictionary by specifying **Options** for Files, Fields, Keys, or relations. See Using Wizard Options for more information.

Menus

Dictionary Editor
Application Generator
Text Editor
Window Formatter
Report Formatter
Database Manager

Dictionary Editor Menu Commands

File Edit Version Project Setup Window Help

The Data Dictionary is the central repository information concerning your application's data. The Dictionary file--.DCT--stores file names, file structures, file relations, file aliases and views, field names, lengths, and data types, field validity checks, field entry pictures, keys, indexes, plus much more such as status bar help messages by field and default prompt values by field.

The Application Generator uses the Data Dictionary to generate source code, such as file declarations, which it places in the data section of the generated source code files. It also uses the dictionary to provide, for example, entry pictures when formatting entry dialogs for the end user.

The following lists the menu commands available from within the Dictionary Editor. Many dialog also have Help buttons which you can press to view a help topic specifically about that dialog (the F1 key calls the same topic when the dialog is open).

See also: How to Create a Data Dictionary

Note that some of the commands, most notably on the Project and Setup menus, do not specifically reference Dictionary Editor functions. Because the Project System and the Registries are always active, their menu commands must be accessible.

File Menu

New	Opens the **New** dialog, which allows you to create a new dictionary or other type of file.
Open	Calls the **Open File** dialog, allowing you to open a Dictionary file. See also: Opening a Clarion Database Developer 3.0 Data Dictionary
Pick	Calls the **Pick** dialog, listing the most recently used files in a list box.
Close	Closes the currently active Dictionary.
Save	Saves the currently active Dictionary.
Save As	Saves the currently active Dictionary under a new name which you specify.
Save All	Saves all the currently open Dictionaries.
Print	Prints the currently active document, if for example, a text document is open.
Print Setup	Calls the **Printer Setup** dialog, allowing you to configure your printer.
Import File	Allows you to add a file definition from an existing data file to the current dictionary. Just choose the file and driver, in the Import File dialog which this command opens. **Tip: To import a file definition for an ODBC source which stores multiple tables in the same file (such as Microsoft Access), be sure the Data Source is correctly specified in the ODBC.INI file. Then select the Data Source and the table in the dialogs that appear. The import will add all fields except memos., and you must also manually define the keys.**
Import Text	Allows you to import a dictionary stored in .TXD (text) format. This is provided for compatibility with Clarion for DOS.

Export T**e**xt	Allows you to export a dictionary stored in .TXD (text) format. This is provided for compatibility with Clarion for DOS.
Browse *file*	Loads the Database Manager to browse the currently selected file.
Browse Database	Loads the Database Manager
Exit	Quits the program.

Edit Menu

Cut	Deletes the currently selected file definition, field, or key from the dictionary and places it in the clipboard.
Copy	Places a copy of the currently selected field, or key from the dictionary into the clipboard.
Paste	Pastes the previously copied file definition, field, or key from the dictionary from the clipboard into the currently active dictionary.
Add File	Adds a new file to the currently active dictionary.
Add Alias	Adds a new alias to the currently active dictionary.
Add View	Adds a new view to the currently active dictionary.
Properties	Depending on the dialog, calls the **Properties** dialog for the selected file, alias, field, key, etc., from the currently active dictionary.
Fields/Keys	Calls the **Field/Key Definitions** dialog for the currently selected file.
Delete...	Depending on the dialog, deletes the selected file, alias, field, key, etc., from the currently active dictionary. The actual commands vary according to the buttons in the active dialog which "delete" something to a list in the dialog.
Add Relation	Adds a new file relation to the currently active dictionary.
Relation Properties	Calls the **Properties** dialog for the selected file relation.
Delete Relation	Deletes the selected file relation from the currently active dictionary.
Dictionary Properties	Opens the **Dictionary Properties** dialog for the active dictionary.

Version Menu

Checkpoint	Adds one revision step to the current dictionary. The Dictionary Editor automatically places an internal version number in your dictionary file. A new dictionary automatically starts with version 1.0. You can see the version number/revision number on the caption bar of the **Dictionary** dialog. The **Dictionary Properties** dialog also displays the original creation date and time, and the last modified date and time. You should increase the version number, manually, whenever you make significant changes to a dictionary; for example, when you're working on version #2 of your application. The revision number (r. #) on the caption bar increases by one.
Revert	Rolls back to a previous version. Choose the revision to revert to by selecting it with the spin control in the **Previous Revision** dialog.

Project Menu

Set	Calls the **Open File** dialog, allowing you to change the active .APP or .PRJ.
New	Calls the **New Project File** dialog, allowing you to create a new project.
Load	Opens the **Project Tree** dialog for hand coded projects, **or Application Tree**

dialog for generated projects.

Edit Opens the **Project Tree** dialog, allowing you to add or edit component files in the current project.

Make Compiles and links the currently active application or project, which is named on the caption bar.

Run Executes the currently active application or project, which is named on the caption bar.

Debug Loads the Debugger and prepares the active application or project, listed on the caption bar, for debugging.

Make Statistics Calls the **Make Statistics** dialog. Allows you to view a statistical profile of the most recent make. Data on the size of each module size, including code and data size, will appear in the dialog.

Auto Make Before Run Toggles the Project System setting which forces a recompile each time you choose the Run command.

File Save Before Run Toggles the Project System setting which saves the source code file each time you choose the Run command.

Minimize on Run Toggles the Project System setting which minimizes CW before displaying the application each time you choose the Run command.

Wait for Termination on Run
Toggles the Project System setting which suspends CW until after you terminate the application upon executing it with the Run command.

Setup Menu

Editor Options Calls the **Editor Options** dialog, which allows you to customize the appearance and behavior of the Text Editor.

Dictionary Options Calls the **Dictionary Options** dialog, which allows you to specify default settings for the Dictionary Editor.

Application Options Calls the **Application Options** dialog, which allows you to specify default settings for the Application Generator.

Template Registry Calls the **Template Registry** dialog, which allows you to register and manage templates.

Database Driver Registry
Calls the **Database Driver Registry**, which allows you to register database drivers.

VBX Custom Control Registry
Calls the **VBX Custom Control Registry** dialog, which allows you to register VBX controls.

Edit Redirection File Loads the Redirection File in a document window, ready for editing.

Window Menu

Tile Vertically Arranges open document windows side by side in a vertical orientation.

Tile Horizontally Arranges open document windows side by side in a horizontal orientation.

Cascade Arranges open document windows in overlapped fashion so that all caption bars are visible

Arrange Icons Arranges iconized windows along the bottom of the Clarion for Windows Application frame.

(Window List)	Lists all open document windows by their caption bar text according to the order they were opened. Choosing a window from the list brings the window to the top.

Help Menu

Contents	Opens the Windows Help application and displays a list of main topics.
Search for Help On	Opens the **Search** dialog in the Windows Help application, allowing you to search for help topics containing a specific keyword.
How to Use Help	Opens the Windows Help application and displays instructions for using the Help system.
About Clarion	Displays the program name, version, registration, and copyright information.

Application Generator Menu Commands

File Edit Application Procedure Project Setup Window Help

The Application Generator generates your application's code based on the predefined templates you choose from the template registry.

The following lists the menu commands available from the Application Generator. Many dialogs also have Help buttons which you can press to view a help topic specifically about that dialog (the F1 key calls the same topic when the dialog is open).

Note that some of the commands, most notably on the Project and Setup menus, do not specifically reference Application Generator functions. Because the Project System and the Registries are always active, their menu commands must be accessible.

File Menu

New Opens the **New** dialog, which allows you to create a new application file, a new dictionary file, a new source file, or other type of file. You cannot create a new .APP file until you close the current one. You may invoke the **Quick Start Wizard** to help create a new .APP.

Open Calls the **Open File** dialog, allowing you to open another application, dictionary, source or other file (you must first close the current .APP before opening another).

Pick Calls the **Pick** dialog, listing the most recently used files by category.

Close Closes the currently active .APP file.

Save Saves the currently active .APP file.

Save As Saves the currently active Application under a new name which you specify.

Save All Saves all the currently open files.

Print Prints the currently active document, if for example, a text document is open.

Print Setup Calls the Printer Setup dialog, allowing you to configure your printer.

Import From Application

Allows you to import a procedure from another .APP file. Select the file from the **Open File** dialog. Then choose a procedure (or procedures) from the **Select Items to Import** dialog.

You can select an item by DOUBLE-CLICKING on it. A check mark appears to indicate the item is selected. Select additional items DOUBLE-CLICKING. De-select an item by DOUBLE-CLICKING a previously selected item. Note: Both applications must use the same dictionary.

Warning: When importing, CW1.5 converts the incoming .APP to CW1.5 format, consequently it can no longer be opened using an older version of CW.

See also: TXA Import Considerations

Import Text Imports the procedures defined in a .TXA (text) file, created with the **Export Text** (see below) command.

Export Text Allows you to create a .TXA (text) file from the current application.

Selective Export Allows you to create a .TXA (text) file containing only the selected procedure.

Browse Database Allows you to browse and edit a database file defined in the current dictionary. Select the file from the **Pick** file dialog, or the **Open File** dialog after specifying a database driver.

Exit Quits the program.

Edit Menu

Properties Calls the currently selected procedure's **Procedure Properties** dialog. Equivalent to the **Properties** button.

Window Calls the **Window Formatter** to visually design a window for the selected procedure.

Report Calls the **Report Formatter** to visually design a report for the selected procedure.

Data Calls the **Local Data** dialog to manage memory variables local to the selected procedure. Press the **Properties** or **Insert** button to define variables using the Data Dictionary's **Field Properties** dialog.

Embeds Calls the **Embedded Source** dialog to manage embedded source code for the selected procedure.

Extensions Calls the **Extension and Control Templates** dialog to manage template generated code added to the selected procedure.

Find Allows you to search for a procedure by name. This can be very useful in a large application with dozens of procedures. Type a string to search for in the **Search for Procedure** dialog.

Find Next Allows you to search for another procedure, using the same search string as the previous search. If you did not search previously, the **Search for Procedure** dialog appears.

Edit by Name Allows you to type the name of a procedure in **the Edit Procedure by Name** dialog, then opens the **Procedure Properties** dialog of the procedure you typed in. This can be very useful in a large application with many procedures.

Delete Equivalent to the **Delete** button. Deletes the currently selected procedure, leaving it as a ToDo item in your Application Tree. To remove it completely, remove the statement that calls the procedure.

Application Menu

Properties Displays the **Application Properties** dialog for specifying changes to the .APP file.

Global Properties Displays the **Global Properties** dialog. Equivalent to using the **Global** button in the Application Tree dialog. Set file handling and other application defaults.

Change Dictionary Allows you to name a new data dictionary for the application. Type a file name in the **Select New Dictionary** dialog, or press the ellipsis (...) button to choose a new dictionary file from the **Open File** dialog.

If your procedures already reference fields in one dictionary, the Application Generator can only match fields from the new dictionary if both the FILE structure prefix and the RECORD fields are exactly the same. The **New Dictionary** dialog provides a warning message.

Insert Module Specifies a new MODULE for generated source code. You can also specify an external .LIB or .OBJ file to add to the project

Template Utility Calls add-in utilities, including CW Wizards. Write your own or install third-party utilities. A simple example is provided in the Template Language Reference.

Redistribute Procedures
Allows you to change the number of procedures per module. Specify the new number in the **Select Procedures per Module** dialog. The Application Generator then redistributes the procedures among the modules, according to the new procedures per module number.

Repopulate Modules Allows you to change the number of procedures per module, but still keep related procedures together in the same module. Specify the new number in the **Select Procedures per Module** dialog. The Application Generator then redistributes the procedures among the modules, according to the new procedures per module number. Your application may execute slightly faster if you group procedures which commonly execute together in the same module

Renumber Modules Renumbers the modules created by the Application Generator. This is useful for large projects from which procedures have been deleted.

Delete Empty Modules
Removes empty generated source code modules from the project. This is useful for large projects from which procedures have been deleted.

Delete Empty Libraries
Removes empty external source code modules, .LIB files, and .OBJ files from the project. This is useful for large projects from which procedures have been deleted.

Procedure Menu

New Adds a procedure not connected to the procedure tree.

Rename Allows you to change the name of the currently selected procedure. Type a new name in the **Rename** dialog box. Don't forget to change the calling statement too.

Copy Copies the currently selected procedure to a new procedure, which you name.

Change Module Allows you to move the currently selected procedure from one source module to another. Select the destination in **the Select Destination Module** dialog. Your application may execute slightly faster if you group procedures which commonly execute together in the same module.

Change Template Type
Allows you to change the procedure type for the currently selected procedure. Select a new procedure template in **the Select Procedure Type** dialog.

Project Menu

Set Calls the **Open File** dialog, allowing you to change the active .APP or .PRJ.

New Calls the **New Project File** dialog, allowing you to create a new project.

Load Opens the **Project Tree** dialog for hand coded projects, **or Application Tree** dialog for generated projects.

Edit Opens the **Project Tree** dialog, allowing you to add or edit component files in the current project.

Make Compiles and links the currently active application or project, which is named on the caption bar.

Run Executes the currently active application or project, which is named on the caption bar.

Debug Loads the Debugger and prepares the active application or project, listed on the caption bar, for debugging.

Make Statistics Calls the **Make Statistics** dialog. Allows you to view a statistical profile of the

most recent make. Data on the size of each module size, including code and data size, will appear in the dialog.

Auto Make Before Run
Toggles the Project System setting which forces a recompile each time you choose the Run command.

File Save Before Run Toggles the Project System setting which saves the source code file each time you choose the Run command.

Minimize on Run Toggles the Project System setting which minimizes CW before displaying the application each time you choose the Run command.

Wait for Termination on Run
Toggles the Project System setting which suspends CW until after you terminate the application upon executing it with the Run command.

Generate Generates code for all modules that have changed since last code generation.

Generate All Generates code for all modules.

Note: If you use the DOS command line, or File Manager to delete one of the .CLW files in the current project, please execute this command to regenerate the file. When executing a Make, the Application Generator attempts to regenerate only those source code files which were changed within the Application Generator. (1351)

Properties Opens the **Project Tree** dialog, allowing you to add or edit component files in the current project.

Setup Menu

Editor Options Calls the **Editor Options** dialog, which allows you to customize the appearance and behavior of the Text Editor.

Dictionary Options Calls the **Dictionary Options** dialog, which allows you to specify default settings for the Dictionary Editor.

Application Options Calls the **Application Options** dialog, which allows you to specify default settings for the Application Generator.

Template Registry Calls the **Template Registry** dialog, which allows you to register and manage templates.

Database Driver Registry Calls the **Database Driver Registry** dialog, which allows you to register database drivers.

VBX Custom Control Registry Calls the **VBX Custom Control Registry** dialog, which allows you to register VBX controls.

Edit Redirection File Loads the Redirection File in a document window, ready for editing.

Window Menu

Tile Vertically Arranges open document windows side by side in a vertical orientation.

Tile Horizontally Arranges open document windows side by side in a horizontal orientation.

Cascade Arranges open document windows in overlapped fashion so that all caption bars are visible

Arrange Icons Arranges iconized windows along the bottom of the Clarion for Windows Application frame.

(Window List) Lists all open document windows by their caption bar text according to the order they were opened. Choosing a window from the list brings the window to the top.

Help Menu

Contents	Opens the Windows Help application and displays a list of main topics.
Search for Help On	Opens the **Search** dialog in the Windows Help application, allowing you to search for help topics containing a specific keyword.
How to Use Help	Opens the Windows Help application and displays instructions for using the Help system.
About Clarion	Displays the program name, version, registration, and copyright information.

Window Formatter Menu Commands

The **Window Formatter** helps you visually design Window elements--windows and controls--on screen. The **Window Formatter** automatically generates and places the language structures and source code that describe these elements in your .APP file or source code document. See also: How to Customize Your Window

Exit!

Exits the **Window Formatter**. You are prompted to save or discard any changes.

Edit Menu

Undo Reverses the most recent editing action.

Redo Reverses the most recent undo action.

Properties Opens the **Properties** dialog for the currently selected window or control.

Embeds Opens the **Embedded Source** dialog for the currently selected window or control. Allows you to manage embedded source at embed points associated with the window or control.

Font Opens the **Select Font** dialog for the selected window or control. Choose typeface, size, style, color, etc. from standard drop down lists.

Key Opens the **Input Key** dialog for the selected control. Establish a hot key, or key combination, that gives immediate focus to the control, or for buttons, initiates the button's action.

Alert Opens the **Alert Keys** dialog for the selected control. Add or delete one or more keys, or key combinations, that will generate an event:ALERT when the control has focus.

Position Opens the **Properties** dialog to the **Position** tab for the selected window or control. Specify default positioning, size, and/or exact x and y coordinates.

List Box Format Opens the **List Box Formatter** for the selected list box control. Add, delete, resize, and reorder the fields in the list box. Format the fields or groups of fields.

Actions Opens the **Properties** dialog to the **Actions** tab for the selected control. Specify a variety code options depending on the type of control and the template associated with the control.

Delete Deletes the currently selected window or control.

Duplicate Places a copy of the currently selected control or controls in the window under construction. Only the control is duplicated, associated template code is not duplicated.

Set Tab Order Opens the **Ordering Type** dialog, which allows you to visually specify the tab-stop order of the controls in the window.

Manual - Select this radio button, then press the OK button to specify the tab-stop order by CLICK on the controls. A small box with a number inside appears on each control, indicating the current order. CLICK on the controls to change the order to the order you wish.

Automatic - Select this radio button, then press the **OK** button to specify that the **Window Formatter** should set the tab-stop order based on the position of the controls. Choose **Horizontally** or **Vertically** from the options below.

Reselect the **Set Tab Order** menu command to toggle back to normal editing

mode.

Control Templates Opens the **Edit Control Templates** dialog, which allows you to access the **Prompts** dialogs of any control templates in the window. This is equivalent to RIGHT-CLICKING a control template, then choosing **Actions** from the popup menu.

Set Control Order Opens the **Order Controls** dialog for the window. Allows you to move controls between tabs, and specify the tab-stop order of the controls in the window by reordering a list of controls.

Control Menu

Push Button Allows you to place a BUTTON control on the window under construction. See also the **Button Properties** dialog.

Radio Button Allows you to place a RADIO control on the window under construction. See also the **Radio Button Properties** dialog.

Check Box Allows you to place a CHECKBOX control on the window under construction. See also the **Check Box Properties** dialog.

Entry Field Allows you to place an ENTRY control on the window under construction. See also the **Entry Properties** dialog.

Text Field Allows you to place a TEXT control on the window under construction. See also the **Text Properties** dialog.

Spin Box Allows you to place a SPIN control on the window under construction. See also the **Spin Properties** dialog.

String Allows you to place a STRING control on the window under construction. See also the **String Properties** dialog.

Prompt Allows you to place a PROMPT control on the window under construction. See also the **Prompt Properties** dialog.

Group Box Allows you to place a GROUP control (group box) on the window under construction. See also the **Group Properties** dialog.

Option Box Allows you to place an OPTION control (OPTION structure, which appears as a group box with radio buttons) on the window under construction. See also the **Option Properties** dialog.

List Box Allows you to place a LIST control (list box, or drop down list box) on the window under construction. See also the **List Properties** dialog.

Combo Box Allows you to place a COMBO control (combo box, or drop down combo box) on the window under construction. See also the **Combo Properties** dialog.

Ellipse Allows you to place an ELLIPSE control on the window under construction. See also the **Ellipse Properties** dialog.

Line

Rectangle Allows you to place a BOX control on the window under construction. See also the **Box Properties** dialog.

Image Allows you to place an IMAGE control (graphic image) on the window under construction. See also the **Image Properties** dialog.

Region Allows you to place a REGION control on the window under construction. See also the **Region Properties** dialog.

Progress Bar Allows you to place a PROGRESS control on the window under construction. See also the **Progress Properties** dialog.

Property Sheet Allows you to place a SHEET control on the window under construction. See also

the **Sheet Properties** dialog.

Tab Control Allows you to place a TAB control on the window under construction. See also the **Tab Properties** dialog.

Custom Control Allows you to place a CUSTOM control (Visual Basic custom control) on the window under construction. See also the **Custom Control Properties** dialog.

Alignment Menu

The Alignment menu provides commands for spacing and sizing the controls within the window. You may place two or more controls so that their 'edges' match up with each other. You may also spread the controls out, or make all of them the same size.

To do so, first select two or more controls. Select the first by clicking on it. Select the second and subsequent controls by pressing the CTRL key, then clicking on the second control while the shift key remains pressed. Lasso multiple controls by CTRL+CLICK+DRAGGING to form a box around the controls.

Align Left Aligns the left borders of the selected controls with the left border of the last control selected (red handles).

Align Right Aligns the right borders of the selected controls with the right border of the last control selected (red handles).

Align Top Aligns the top borders of the selected controls with the top border of the last control selected (red handles).

Align Bottom Aligns the bottom borders of the selected controls with the bottom border of the last control selected (red handles).

Align Horizontally Along a horizontal axis, aligns the centers of the selected controls with the center of the last control selected (red handles).

Align Vertically Along a vertical axis, aligns the centers of the selected controls with the center of the last control selected (red handles).

Spread Horizontally Equalizes the horizontal spaces between the selected controls.

Spread Vertically Equalizes the vertical spaces between the selected controls.

Make Same Size Makes all selected controls the same height and width as the last control selected (red handles).

Make Same Height Makes all selected controls the same height as the last control selected (red handles).

Center Horizontally As a group (relative positions of selected controls don't change), centers the selected controls vertically within the window.

Center Vertically As a group (relative positions of selected controls don't change), centers the selected controls horizontally within the window.

Menu Menu

New Menu Calls the Menu Editor, allowing you to create a menu for the window under construction. See also: How to Create a New Menu

Menu Editor Calls the Menu Editor, allowing you to edit an existing menu for the window under construction.

Delete Menu Allows you to delete an existing menu for the window under construction.

Toolbar Menu

New Toolbar Adds a tool bar to the window under construction. See also **the Toolbar Properties** dialog. See also: How to Add a Tool Bar

Delete Toolbar Deletes the existing tool bar for the window under construction.

Populate Menu

Field Places an entry control for a data dictionary field or memory variable, and an associated prompt. When you CHOOSE **Populate** ä **Field,** the **File Schematic Definition** dialog appears. Select a field or variable, then CLICK in the window.

The CLICK places the prompt for the control as well as the control. If you pre-formatted the field, on the **Window** tab of the **Field Properties** dialog (for example, specifying a spin control), the control you specified appears, rather than an entry box.

Multiple Fields Places an entry control for a data dictionary field or memory variable, and an associated prompt. When you CHOOSE **Populate** ä **Field,** the **File Schematic Definition** dialog appears. Select a field or variable, then CLICK in the window.

The CLICK places the prompt for the control as well as the control. If you pre-formatted the field, on the **Window** tab of the **Field Properties** dialog (for example, specifying a spin control), the control you specified appears, rather than an entry box.

After placing the first field, the **File Schematic Definition** dialog appears again, ready for you to select another field. When all fields are placed, press the **Cancel** button in this dialog to return to normal editing.

Control Template Allows you to add a control template to the window under construction. Select one from the Select a **Control Template** dialog.

A control template adds a control or controls to the window, plus the code to maintain them. For example, the Browse Box control template places a list box in the window, allows you to choose the fields for the list, and adds all the executable code for managing the list box.

Once the control template is placed, you can specify its properties and actions by RIGHT-CLICKING and selecting **Properties** or **Actions** from the popup menu, or pressing the respective buttons on **the Window Formatter** toolbar.

Options Menu

Show Toolbox Toggles display of the Controls tool box, which allows you to choose a control type and place it in a window. The tool icons available match those on the (see also)**Controls** menu and the (see also)**Populate** menus, described above.

Show Alignbox Toggles display of the Align tool box, which provides tool buttons for executing the align commands which appear on the (see also)**Alignment** menu, described above.

Show Propertybox Toggles display of the Property tool box, which provides tools for setting some common control properties such as caption text, field equate label/use variable, and font.

Show Fieldsbox Toggles display of the Fields tool box, which places an entry control and prompt for a data dictionary field, when you DOUBLE-CLICK the field in the list box. The list contains all the data dictionary fields defined in the **File Schematic** for this procedure.

Grid Settings Opens the **Grid Settings** dialog, which allows you to toggle the Snap to Grid function as well as set the size of the grid on the sample window.

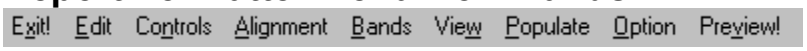
VBX Control Registry Opens the **VBX Custom Control Registry** dialog, which allows you to make manage the VBX control libraries available for use in **the Window Formatter**.

Preview!

Allows you to display an active window identical to the one the end user sees. This allows you to see how the window behaves, and how, for example, it looks with the 3D option set.

To exit **Preview!** mode, press ESC, or press any DEFAULT button control in the previewed window.

Report Formatter Menu Commands



The **Report Formatter** helps you visually design report elements--variable strings and other controls--on screen. The **Report Formatter** automatically generates and places the language structures and source code that describe these elements in your .APP file or source code document.

See also (see also)How to Use the Report Formatter -- An Overview

(see also)How the Print Engine Processes Report Sections at Runtime

Exit!

Exits the **Report Formatter**. You are prompted to save or discard any changes.

Edit Menu

Next Band	Moves focus to the next report band.
Delete Band	Deletes the selected report band and all controls in it.
Report Properties	Opens the **Report Properties** dialog to set paper size, orientation, measurement units, etc.
Selected Properties	Opens the Properties dialog for the selected control or report band.
Font	Opens the **Select Font** dialog to specify font, size, style, script, and color from drop down list boxes. Shows you a sample of the text design you have chosen.
Position	Opens the Properties dialog to the **Position** tab to set default position, width, and height, or specific x and y coordinates for the selected control or band.
List Box Format	Opens the **List Box Formatter** for the selected list box. Add, delete, resize, reorder and format the fields in the list box.
Delete Control	*To delete a control*, select it and choose the **Delete Control** command, or select it and press the DELETE key.
Duplicate	Not implemented in this release.
Set Control Order	Opens the **Order Control** dialog, which displays all controls on the report in a hierarchical list. Reorder the controls by selecting a control and pressing the and buttons to move the control up or down within the list.

Controls Menu

Check Box	Allows you to place a CHECKBOX control in the selected band. See also the **Check Box Properties** dialog.
Radio Button	Allows you to place a RADIO control in the selected band. See also the **Radio Button Properties** dialog.
Text Field	Allows you to place a TEXT control on the selected band. See also the **Text Properties** dialog.
String	Allows you to place a STRING control on the selected band. See also the **String Properties** dialog.
Group Box	Allows you to place a GROUP control (group box) in the selected band. See also the **Group Properties** dialog.
Option Box	Allows you to place an OPTION control (OPTION structure, which appears as a group box with radio buttons) in the selected band. See also the **Option Properties** dialog.

List Box	Allows you to place a LIST control (list box) in the selected band. See also the **List Properties** dialog.
Ellipse	Allows you to place an ELLIPSE control in the selected band. See also the **Ellipse Properties** dialog.
Line	Allows you to place a LINE control in the selected band. See also the **Line Properties** dialog.
Rectangle	Allows you to place a BOX control in the selected band. See also the **Box Properties** dialog.
Image	Allows you to place an IMAGE control (graphic image) in the selected band. See also the **Image Properties** dialog.
Custom Control	Allows you to place a CUSTOM control (Visual Basic custom control) in the selected band. See also the **Custom Control Properties** dialog.

Alignment Menu

The Alignment menu provides commands for spacing and sizing the controls within the report. You may place two or more controls so that their 'edges' match up with each other. You may also spread the controls out, or make all of them the same size.

To do so, first select two or more controls. Select the first by clicking on it. Select the second and subsequent controls by pressing the CTRL key, then clicking on the second control while the shift key remains pressed.

Align Left	Aligns the left borders of the selected controls with the left border of the last control selected (red handles).
Align Right	Aligns the right borders of the selected controls with the right border of the last control selected (red handles).
Align Top	Aligns the top borders of the selected controls with the top border of the last control selected (red handles).
Align Bottom	Aligns the bottom borders of the selected controls with the bottom border of the last control selected (red handles).
Align Horizontally	Along a horizontal axis, aligns the centers of the selected controls with the center of the last control selected (red handles).
Align Vertically	Along a vertical axis, aligns the centers of the selected controls with the center of the last control selected (red handles).
Spread Horizontally	Equalizes the horizontal spaces between the selected controls.
Spread Vertically	Equalizes the vertical spaces between the selected controls.
Make Same Size	Makes all selected controls the same height and width as the last control selected (red handles).
Make Same Height	Makes all selected controls the same height as the last control selected (red handles).
Center Horizontally	Centers the selected controls horizontally in the band.
Center Vertically	Centers the selected controls vertically in the band.

Bands Menu

Page Header	Adds a header band to your report.

The HEADER structure traditionally prints at the top of each page of the report. Typically, you place the report title, graphics and other "introductory" elements in the HEADER.

Page Footer Adds a footer band to your report.

The FOOTER structure traditionally prints at the bottom of the report. Typically, you place a page number, or totals in the FOOTER.

Page Form Adds a form band to your report.

The FORM structure prints as a "background layer." Typically, you may display "overlays" such as graphics and field labels in the FORM layer, then print the actual foreground data in the DETAIL. The FORM remains constant from page to page.

Detail Adds a detail band to your report..

The DETAIL structure is the "body" of the report. It contains the basic data, either in table or record format.

Break Group Adds a new detail, break, group header and group footer bands. See also the (see also)**Break Properties** dialog. Place the crosshair where you want the new group of bands to appear, and CLICK. The **Break Properties** dialog appears. Specify the variable to break on and press **OK.**

A Group BREAK structure can have nested BREAK structures, each with their own HEADER, DETAIL, and FOOTER structures. See (see also) How to Set Report Group Breaks, (see also) How to Sort Reports.

Group Header Adds a new Group Header band to the currently selected break section.

Group Footer Adds a new Group Footer band to the currently selected break section.

Surrounding Break Adds a break group around an existing detail. Place the crosshair on the detail you want to break on, and CLICK. The (see also)**Break Properties** dialog appears. Specify the variable to break on and press **OK**

View Menu

Page Layout View Allows you to reposition and resize your report bands by dragging handles. All bands display together on a representation of the page.

Band View Allows you to edit your report and place controls separately, in each individual band.

Expand Bands Expands or contracts all report bands at once.

Populate Menu

Dictionary Field Allows you to place a string variable control tied to a data dictionary field or memory variable. The **Select Field** dialog appears. Select a field or variable, then CLICK in the window.

Multiple Fields Allows you to place a string variable control tied to a data dictionary field or memory variable. The **Select Field** dialog appears. Select a field or variable, then CLICK in the window.

After placing the first field, the **Select Field** dialog appears again, ready for you to place another field. When all fields are placed, press the **Cancel** button in this dialog to return to normal editing.

Control Template Allows you to add a control template to the window under construction, if any are available. Select one from the **Select a Control Template** dialog.

Once the control template is placed, you can specify its properties and actions by

RIGHT-CLICKING and selecting **Properties** or **Actions** from the popup menu.

Option Menu

Zoom In Magnifies the "view" in (see also)**Preview**! mode.

Zoom Out Reduces the "view" in **Preview**! mode.

Snap to Grid Toggles the setting which forces new controls to align with the report grid.

Grid Size Opens the **Grid Settings** dialog, which allows you to set the size of the grid which helps align the controls you place in the window.

Show Toolbox Toggles display of the (see also)Controls toolbox, which allows you to choose a control type and place it in the report. The tool icons available match those on the (see also)Controls menu and the (see also)Populate menu, described above.

Show Alignbox Toggles display of the (see also)Align toolbox, which provides tool buttons for executing the align commands which appear on the (see also)Alignment menu, described above.

Show Propertybox Toggles display of the (see also)Property toolbox, which provides tools for setting some common control properties such as caption text, field equate label/use variable, and font.

Show Fieldsbox Toggles display of the (see also)Fields toolbox, which places a control for a data dictionary field in the currently selected report band, when you DOUBLE-CLICK the field name in the list box. The list contains all the data dictionary fields defined in the **File Schematic** for this procedure.

Preview!

Opens the (see also)**Preview Print Details** dialog which lets you generate "filler" data for your report. The data have no values, but serve as placeholders, so you can get a feel for the appearance of your finished report. Fonts, sizes, colors, and positions of report controls are all displayed

You can quickly "preview" alternative layouts for DETAILs, HEADERs, and FOOTERs, and you can see the effects of the (see also)page breaking options you have chosen, all without actually compiling or running your report. See also: Using Print Preview.

Text Editor Menu Commands

The Text Editor is a full function programmer's editor featuring Multiple Document Interface support, auto-indent, search-and-replace, and color coded syntax highlighting.

The following lists all menu commands available from within the Text Editor.

You can also get help for a keyword within a document window by placing the insertion point at the keyword, and pressing the F1 key. This allows you to quickly look up help for a Clarion language statement, function or attribute.

Note that some of the commands, most notably on the Project and Setup menus, do not specifically reference Text Editor functions. Because the Project System and the Registries are always active, their menu commands must be accessible.

File Menu

New	Opens the **New** dialog, which allows you to create a new source code or other type of file.
Open	Calls the **Open File** dialog, allowing you to open a source code document.
Pick	Calls the **Pick** dialog, listing the most recently used files in a list box.
Close	Closes the currently active source code document.
Save	Saves the currently active source code document.
Save As	Saves the currently active source code document under a new name which you specify.
Save All	Saves all the currently open source code documents.
Print	Prints the currently active source code document.
Print Setup	Calls the **Printer Setup** dialog, allowing you to configure your printer.
Import File	Calls the **Open File** dialog, allowing you to insert the contents of a file into the currently active source code document, at the insertion point.
Export Block	Saves the currently selected text in a new source code document under a new name which you specify.
Browse Database	Allows you to browse and edit a database file defined in the current dictionary. Select the file from the **Pick** file dialog, or the **Open File** dialog after specifying a database driver.
Exit	Quits the program.

Edit Menu

Undo	Reverses the most recent editing action.
Cut	Deletes the currently selected text from the document and places it in the clipboard.
Copy	Places a copy of the currently selected text from the document into the clipboard.
Paste	Pastes text from the clipboard into the currently active document, at the insertion point.
Select All	Selects all text in the currently active document.
Goto Line	Calls the **GoTo Line** dialog, in which you can enter a line number. After pressing

	the **OK** button, the cursor moves to the beginning of the line you specify.
Goto Next Error	Moves the insertion point to the next compiler error. The error appears at the bottom of the window. This command is disabled except following a compile which generated errors.
Goto Previous Error	Moves the insertion point to the previous compiler error. The error appears at the bottom of the window. This command is disabled except following a compile which generated errors.
Set/Clear Tabstop	Places a custom tab stop at the insertion point.
Duplicate Line	Places a copy of the current line at the line immediately following it. You do not need to select the entire line. The insertion point merely needs to be anywhere within the line you wish to copy.
Toggle Case	Changes the case of the next character following the insertion point.
Delete Line	Deletes the entire line at which the insertion point is located.
Delete Word	Deletes a current word at which the insertion point is located.
Format Structure	Calls the **Window Formatter** allowing you to edit or create a structure.

Search Menu

Find	Calls the **Find** dialog. It allows you to type in a word, then find the next occurrence forwards or backwards from the current position of the insertion point. The keyboard accelerator is ALT+F3. Type the word or phrase to search for in the **Find What** box. Optionally indicate whether you wish the search to **Match Whole Word Only**, and/or **Match Case**. Choose a forward search (Down) or backwards search (Up), then press the **Find Next** button. The **Find** dialog is modeless. This means that the dialog will remain on screen so that you may easily search again. This makes it easy to repeat a search several times quickly, using the **Find Next** button.
Replace	Calls the **Replace** dialog, which allows you to change specific text to something else. You may make the changes one at a time, throughout a selected text block, or throughout the entire document. Type the original word or phrase to replace in the **Find What** box. Type the replacement text in the **Replace With** box. Optionally indicate whether you wish the search to **Match Whole Word Only**, and/**or Match Case**. Press the **Find Next** button to find the next occurrence of the word or phrase. Press the **Replace** button to replace it, once found. Press the **Replace All** button to replace all instances within the document. The **Replace** dialog is modeless. This means that the dialog will remain on screen so that you may easily "replace" again. This makes it easy to repeat an operation several times quickly, using the **Find Next** button.
Find Next	Searches for the text most recently searched for, moving toward the top of the document.
Find Previous	Searches for the text most recently searched for, moving toward the beginning of the document.
Find Marked Text	Finds the next occurrence of the currently selected text. This is equivalent to executing the **Find** command, typing the currently selected text in the **Find What** box, and specifying a forward search.

Project Menu

Set Calls the **Open File** dialog, allowing you to change the active .APP or .PRJ.

New Calls the **New Project File** dialog, allowing you to create a new project.

Load Opens the **Project Tree** dialog for hand coded projects, **or Application Tree** dialog for generated projects.

Edit Opens the **Project Tree** dialog, allowing you to add or edit component files in the current project.

Make Compiles and links the currently active application or project, which is named on the caption bar.

Run Executes the currently active application or project, which is named on the caption bar.

Debug Loads the Debugger and prepares the active application or project, listed on the caption bar, for debugging.

Make Statistics Calls the **Make Statistics** dialog. Allows you to view a statistical profile of the most recent make. Data on the size of each module size, including code and data size, will appear in the dialog.

Auto Make Before Run Toggles the Project System setting which forces a recompile each time you choose the Run command.

File Save Before Run Toggles the Project System setting which saves the source code file each time you choose the Run command.

Minimize on Run Toggles the Project System setting which minimizes CW before displaying the application each time you choose the Run command.

Wait for Termination on Run Toggles the Project System setting which suspends CW until after you terminate the application upon executing it with the Run command.

Setup Menu

Editor Options Calls the **Editor Options** dialog, which allows you to customize the appearance and behavior of the Text Editor.

Dictionary Options Calls the **Dictionary Options** dialog, which allows you to specify default settings for the Dictionary Editor.

Application Options Calls the **Application Options** dialog, which allows you to specify default settings for the Application Generator.

Template Registry Calls the **Template Registry** dialog, which allows you to register and manage templates.

Database Driver Registry Calls the **Database Driver Registry**, which allows you to register database drivers.

VBX Custom Control Registry Calls the **VBX Custom Control Registry** dialog, which allows you to register VBX controls.

Edit Redirection File Loads the Redirection File in a document window, ready for editing.

Window Menu

Tile Vertically Arranges open document windows side by side in a vertical orientation.

Tile Horizontally Arranges open document windows side by side in a horizontal orientation.

Cascade Arranges open document windows in overlapped fashion so that all caption bars are visible

Arrange Icons	Arranges iconized windows along the bottom of the Clarion for Windows Application frame.
(Window List)	Lists all open document windows by their caption bar text according to the order they were opened. Choosing a window from the list brings the window to the top.

Help Menu

Contents	Opens the Windows Help application and displays a list of main topics.
Search for Help On	Opens the **Search** dialog in the Windows Help application, allowing you to search for help topics containing a specific keyword.
How to Use Help	Opens the Windows Help application and displays instructions for using the Help system.
About Clarion	Displays the program name, version, registration, and copyright information.

Database Manager Menu Commands

File Browse Edit Column Project Setup Window Help

The Database Manager allows you direct access to data files without requiring you to create an application. Database Manager thus allows you free access to your data files. The only entry constraint is the picture assigned to a column. For example, if a field has a @n3 picture token, only numbers can be entered. If the picture is changed to @s3, then any character can be entered. This allows you to create files for testing purposes.

The Database Manager offers neither data Validity Checking nor Referential Integrity Constraints. This is a programmer's tool; there are no controls to prevent you from making changes that could compromise the integrity of the database.

The following lists the menu commands available from within the Database Manager. Many dialogs also have Help buttons which you can press to view a help topic specifically about that dialog (the F1 key calls the same topic when the dialog is open).

Note that some of the commands, most notably on the Project and Setup menus, do not specifically reference Database Manager functions. Because the Project System and the Registries are always active, their menu commands must be accessible.

File Menu

New	Opens the **New** dialog, which allows you to create a new application file, a new dictionary file, a new source file, or other type of file. You cannot create a new .APP file until you close the current one. You may invoke the **Quick Start Wizard** to help create a new .APP.
Open	Calls the **Open File** dialog, allowing you to open another application, dictionary, source or other file (you must first close the current .APP before opening another).
Pick	Calls the **Pick** dialog, listing the most recently used files by category.
Close	Closes the currently active data file.
Save	Saves the currently active data file.
Save As	Saves the currently active data file under a new name which you specify.
Save All	Saves all currently open files.
Print	Prints the currently active document, if for example, a text document is open.
Print Setup	Calls the **Printer Setup** dialog, allowing you to configure your printer.
Save As Source	Allows you to create a source code document containing a FILE declaration for the current database file.
Convert File	Opens the **Convert File** dialog, which creates source code, and a project file for creating an application to convert the data file from one format to another.
File Statistics	Opens the **File Statistics** dialog, which provides information about the data file.
Browse Database	Allows you to browse and edit a database file. Select the file from the **Pick** file dialog, or the **Open File** dialog after specifying a database driver.
Exit	Quits the program.

Browse Menu

Order	Opens the **Select File Order** dialog, which allows you to choose the active key.

Query by Example	Opens the **Query by Example** dialog, which allows you to filter the data file, then display only the records that meet the criteria you specify by entering example values or expressions in this dialog.
Send Driver String	Opens the **Send Driver String** dialog, which allows you to execute a SEND command to the file driver.

Edit Menu

Change	Activates an edit control which appears at the currently selected field and record
Insert	Inserts a blank record at the end of the file, and then activates an edit control which appears in the first field.
Delete	Deletes the current record, first requesting confirmation.
Undelete	Undeletes a record.
Hold	Places a HOLD on the current record.
Release	RELEASEs the previously held record.
Search	Opens the **Search** dialog, which allows you to search for the first record containing a value you specify. You may limit the search to one field, or all fields,
Find Next	Repeats the most recent search.
Locate	Opens the **Locate** dialog, which allows you to search for the first record containing the value you specify in the key field(s).
Edit Memo	Opens the **Edit Memo** dialog, which allows you to edit a memo in ASCII Text.
Hex Edit Memo	Opens the **Hex Edit Memo** dialog, which allows you to edit a memo in Hexadecimal format.
OEM Conversion	Specifies string data is converted from OEM ASCII to ANSI when read from disk and ANSI to OEM ASCII before writing to disk.

Column Menu

Hide	Hides the currently selected column.
Show	Re-displays a previously hidden column.
Picture	Opens the **Picture of Field (fieldname)** dialog, which allows you to specify a different display picture for the current column.
Justify	Opens the **Justify Field (fieldname)** dialog, which allows you to select a different justification style for the text in the current column.
Reformat	Opens the **Reformat Fields** dialog, which allows you to change the field order in the window, and to hide or unhide fields from view..
Header	Opens the **Header Type** dialog, which allows you to specify the contents of the header line for the browse window. You can specify, for example, the field label, or the field picture.

Project Menu

Set	Calls the **Open File** dialog, allowing you to change the active .APP or .PRJ.
New	Calls the **New Project File** dialog, allowing you to create a new project.
Load	Opens the **Project Tree** dialog for hand coded projects, **or Application Tree** dialog for generated projects.
Edit	Opens the **Project Tree** dialog, allowing you to add or edit component files in the current project.

Make Compiles and links the currently active application or project, which is named on the caption bar.

Run Executes the currently active application or project, which is named on the caption bar.

Debug Loads the Debugger and prepares the active application or project, listed on the caption bar, for debugging.

Make Statistics Calls the **Make Statistics** dialog. Allows you to view a statistical profile of the most recent make. Data on the size of each module size, including code and data size, will appear in the dialog.

Auto Make Before Run Toggles the Project System setting which forces a recompile each time you choose the Run command.

File Save Before Run Toggles the Project System setting which saves the source code file each time you choose the Run command.

Minimize on Run Toggles the Project System setting which minimizes CW before displaying the application each time you choose the Run command.

Wait for Termination on Run Toggles the Project System setting which suspends CW until after you terminate the application upon executing it with the Run command.

Setup Menu

Editor Options Calls the **Editor Options** dialog, which allows you to customize the appearance and behavior of the Text Editor.

Dictionary Options Calls the **Dictionary Options** dialog, which allows you to specify default settings for the Dictionary Editor.

Application Options Calls the **Application Options** dialog, which allows you to specify default settings for the Application Generator.

Template Registry Calls the **Template Registry** dialog, which allows you to register and manage templates.

Database Driver Registry Calls the **Database Driver Registry**, which allows you to register database drivers.

VBX Custom Control Registry Calls the **VBX Custom Control Registry** dialog, which allows you to register VBX controls.

Edit Redirection File Loads the Redirection File in a document window, ready for editing.

Window Menu

Tile Vertically Arranges open document windows side by side in a vertical orientation.

Tile Horizontally Arranges open document windows side by side in a horizontal orientation.

Cascade Arranges open document windows in overlapped fashion so that all caption bars are visible

Arrange Icons Arranges iconized windows along the bottom of the Clarion for Windows Application frame.

Show Headers Hides or displays column headers above database fields.

Show Deleted Hides or displays database records marked for deletion.

Use QBE This menu toggle allows you to enable or disable your QBE filter. When checked, the records displayed match the filter criteria.

(Window List) Lists all open document windows by their caption bar text according to the order

they were opened. Choosing a window from the list brings the window to the top.

Help Menu

Contents	Opens the Windows Help application and displays a list of main topics.
Search for Help On	Opens the **Search** dialog in the Windows Help application, allowing you to search for help topics containing a specific keyword.
How to Use Help	Opens the Windows Help application and displays instructions for using the Help system.
About Clarion	Displays the program name, version, registration, and copyright information.

Glossary

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

All definitions should be considered general terms, except where otherwise indicated. The context for definitions marked (Clarion) pertain to the Clarion language or the Clarion for Windows IDE. Likewise for (SQL), which applies to generalized Structured Query Language usage.

-A-

ANSI character set
API
append
applet
application
application generator
application tree
application window
array
ASCII character set
assignment statement
attribute
auto-increment field

-B-

background priority
binary memo
bind
bitmap
Boolean
Border or Line Color
break field
breakpoint
BringWindowToTop
Browse
built-in
button

-C-

calculated field
cascading menu
case sensitive
case structure
character string
check box
child window
Clarion standard date
click
Client Server Architecture
clipboard
Close
code section
color dialog
column
combo box

command
comment
commit
common file dialog
compiler directive
concatenate
concurrency checking
conditional statement
constant
control
control alignment
control menu
control properties
cool switch
criteria
current directory
current record
cursor

-D-

data dictionary
data file
data section
data type
data validation
database
database administrator
database definition file
database design
database driver
database integrity
dBase format
DBMS
DDE
debug
deep assignment
default
default button
default window position
delimiter
dependent entity
desktop
DETAIL structure
dialog box
dialog unit
disabled
document
DOS buffer
double-click
drag
drag and drop
drop-down list
dynamic link library

-E-

-F-

-G-

-H-

-I-

-J-

-K-

-L-

-M-

multi-user database

-N-

natural join
nested queries
nesting
non-Windows application
normalization
null value

-O-

ODBC
ODBC Administrator
ODBC Control Panel applet
ODBC driver
one-to-many relationship
one-to-one relationship
option structure
origin
orphan
outer join
overlay

-P-

page footer
page header
page overflow
palette
parameter
PCX image
pel
pen
picture token
pixel
point size
pointer
prefix
primary key
print job
print structure
printer driver
printer font
procedure
program MAP
project system
prompt
property assignment syntax
prototype
PUT statement

-Q-

query
Query by Example
queue

-R-

radio button
range constraint
raster font
read only
RECORD
redirection file
reference variable
referential integrity
region
registry
relationship
report form
resource file
restore button
rich text format (RTF)
ROLLBACK
ROUTINE
run time library

-S-

schema
scope
scroll bar
select
selected event
sequential access
server
SET statement
SHARE.EXE
sort
source code file
spin control
SQL
stack memory
statement
static text
static variable
status bar
standard behavior
stream mode
swap file
syntax
system colors
system date

-T-

tab order
table
tabular report
tag
target file
task
template procedure

text control
text file
text justification
third normal form
thread
thumb
timer
token
toolbar
transaction
tree control

-U-
untyped parameter
USE variable

-V-
validity check
VBX control
VCR controls
vector font
vector graphic
view
virtual table

-W-
watch variable
widow
window frame
window pane
WinExec

-X-
X axis

-Y-
Y axis

-Z-

ACCEPT loop

(Clarion) An event handling loop beginning with the ACCEPT statement. The loop transparently processes the Windows messages and related events which affect the application's window. A single ACCEPT loop automatically gets end user input for all controls within a given window.

accepted event

(Clarion) An event generated when an end user interacts with a window control, such as when moving the focus to a field, that results in the event being reported in the ACCEPT look.

access key

(Clarion) A specified key or index to set the order for processing records in a procedure.

active window

The document or active window which currently has the focus; Windows sends the next keyboard or mouse action to the ACCEPT loop of the active window.

alias

An alternate name for a data file, which allows multiple, independent operations on it. Clarion provides a separate record buffer for each alias, increasing the performance of the separate operations.

ANSI character set

Character set standardized by the American National Standards Institute. Many ANSI characters are different then the corresponding ASCII character set. The ANSI set contains more non-English characters. The standard Microsoft Windows character set is the ANSI character set.

API

Application Programming Interface; generally refers to the Windows API. Allows applications to dynamically link function calls to the three main Windows libraries (USER.EXE, GDI.EXE, and KERNEL.EXE), plus the external libraries such as MMSYSTEM.DLL. Just about everything that every Windows program does is accomplished via the API.

append

Add a record to a data file, usually without updating a key or index.

applet

A small, single purpose application; applets are not necessarily stand alone executable programs. The "programs" managed by the Windows Control Panel, for example, are called applets, though they are actually dynamic link libraries with specialized entry points. The accessories which ship with Windows are also known as applets.

application

A computer program designed for a specific type of work; the terms "application" and "program" are interchangeable. In general, when referring to a Windows program, "application" is the preferable term.

application generator

A program which combines prewritten, generalized executable code modules or fragments to create an application.

(Clarion) The part of the IDE which manages pre-written template procedures, obtains customizations from the developer, and generates Clarion language source code files.

application tree

(Clarion) An Application Generator dialog which graphically depicts the hierarchy of procedures for an application.

application window

In a Multiple Document Interface application, the parent window, usually containing no controls, in which all child document windows appear.

array

A ordered series or group of dimensioned values or data items.

ASCII character set

Character set standardized as the American Standard Code for Information Interchange. The standard IBM PC character set.

assignment statement

A statement placing a value in a variable; for example, A = 6 places the value 6 in variable "A."

attribute

(Clarion) A modifier to a data declaration which specifies an optional property.

auto-increment field

(Clarion) A key field which stores a value which increases with each successive record, and is generally not available to the end user. The application places the value in the field immediately upon appending the record.

background priority

A measure, expressed in a ratio, for the amount of CPU processing time allocated to a program or task which does not currently have system focus. In the Windows 16-bit environment, all multitasking is cooperative; therefore, all background processing is dependent on all executing applications properly yielding at regular intervals.

band view

(Clarion) A specialized layout mode within the Report Formatter. Displays the contents of each part of the report structure in separate panes.

binary memo

(Clarion) A memo field suitable for holding non-ASCII contents, such as images.

bind

(Clarion) A statement which allows a variable name to be used in a dynamic expression which is assembled and processed at runtime.

bitmap

A binary file representation of a graphic or picture; raster format defines the image by absolute pixels. Popular bitmap formats supported by Clarion for Windows include .BMP, .GIF, .ICO, .PCX, .JPG. Sometimes refers specifically to the .BMP file format, an uncompressed, but widely supported file format.

Boolean

A logical expression which evaluates to true or false, one or zero.

Border or Line Color

The color designated for the outside line of a graphical control.

break field

(Clarion) A field or variable monitored when processing a report structure. When the value in the field changes while sequentially processing records, the print engine processes the next element in the report structure (usually the group footer).

breakpoint

A debugger stopping point, relative to a source or disassembly code statement. The application executes up to the breakpoint, then halts and turns execution over to the debugger, which can then examine variables and expressions to search for bugs.

BringWindowToTop

Windows API function for forcing a window to always display on top of all other windows on the desktop. Implemented in Clarion for Windows by the TOOLBAR attribute.

Browse

A specialized listbox procedure dedicated to displaying database records arranged in columns and rows.

built-in

(Clarion) Default map definitions, as provided in source code format in the BUILTINS.CLW file.

button

A control that initiates a command, or selects an option. An end user chooses a button by clicking with the mouse.

calculated field

A field created via an expression which may include one or more database fields.

cascading menu

A hierarchical submenu, sometimes called a child menu. Parent menus that lead to cascading menus usually have a right-pointing triangle at the right side of the menu item, to cue the user to the submenu.

case sensitive

A characteristic indicating whether a command treats text typed with capital (uppercase) letters differently than those typed with lower case, or a combination of both.

case structure

A control structure which branches execution to a statement (or group of statements) based upon a single condition or expression.

character string

An alphanumeric data type.

check box

A control consisting of a small square or diamond, in which an end user indicates a on/off, yes/no, or true/false choice.

child window

An MDI document window displaying a document or view within the main application window.

Clarion standard date

(Clarion) The number of days elapsed since December 28, 1800; the valid range is from Jan. 1, 1801 through Dec. 31, 2099.

click

To place the mouse pointer on a control or window, then press and release the left mouse button.

Client Server Architecture

A network configuration by which linked workstations request services from a dedicated program running on a server.

clipboard

A temporary storage area in memory for holding data, maintained by Windows.

Close

To normally terminate processing of a window or file.

code section

(Clarion) The portion of source code containing executable code statements.

color dialog

Standard Windows dialog for choosing color.

column

(SQL) Generally refers to a list of database field contents arranged by records.

combo box

A window control consisting of a synchronized edit box and list box.

command

An executable code statement or program instruction.

comment

Text inserted in a source code file to annotate or explain the code. Clarion language comments begin with the exclamation point (!) character. Each comment terminates at the end of the line it appears on.

commit

Terminates a successful transaction and commits it to disk.

common file dialog

A standard Windows dialog for displaying drives, directories, and file names. The Clarion FILEDIALOG function displays the dialog and returns a file name to the calling application.

compiler directive

An instruction directing a compiler to build an application to meet a certain condition.

concatenate

Append two string data elements to form a longer string comprised of both.

concurrency checking

The process of guarding against two users updating the same record at the same time. Usually consists of checking the record on disk still contains the same values as when it was first retrieved for updating.

conditional statement

An IF statement which branches subsequent execution based on a logical condition.

constant

A static value.

control

A window or report object which displays data and/or processes user input.

control alignment

The "Snap-To" behavior, as found in the Window and Report Formatters, by which you may "line up" window and report elements.

control menu

Contains commands for resizing, repositioning, or closing a window.

control properties

(Clarion) Attributes which determine the appearance and functionality of a window or report control.

cool switch

The Windows procedure for switching between active applications by holding down the ALT key and pressing the TAB key.

criteria

(SQL) An expression containing a condition which limits the records for processing.

current directory

The default DOS subdirectory, in which Windows or DOS searches for files not identified with a fully qualified file name.

current record

(Clarion) The current database record in the record buffer.

cursor

The mouse pointer. Changing the cursor "shape" can indicate the type of action or selection the end user can effect on a given control or window.

data dictionary

(Clarion) ASCII file describing the individual data files which comprise the database, their structure, keys, relations, and other information describing how an application will process the contents of the database.

data file

Generally, a collection of data elements in an organized format, usually arranged by records (rows) and fields (columns).

data section

(Clarion) The section of source code containing variable and data structure declarations, such as FILE, WINDOW, REPORT, and QUEUE.

data type

A physical description of the type of storage supported by a variable; what sort of values it can hold.

data validation

An expression or the process of checking data against a condition prior to accepting the data for entry into the database.

database

A structured collection of data, contained in one or more data files, plus the key files and other information which describes the order and relations of the data elements.

database administrator

(DBA) A person responsible for designing and maintaining a multi-user database system.

database definition file

(*.DDF). A Btrieve file, separate from the data file, containing the database structure. Equivalent to the header contained internally in most other PC database file formats.

database design

The process of planning and describing the most efficient application or system for storing and managing data for a specific project.

database driver

A collection of functions and procedures contained in a dynamic link library, supporting low level access to a specific database file format.

database integrity

Under the relational model, database integrity consists of two general rules:

1.

Each database file or table must have a primary key serving as a unique identifier for all records.

2.

When a table has a foreign key matching the primary key of another table, each value in the foreign key must either equal a value in the primary key of the other table, or be null.

dBase format

PC database file format popularized by dBase III.

DBMS

Database Management System: generic term for a program that enables a system to perform all the functions associated with managing a database.

DDE

Dynamic Data Exchange: a message protocol for exchanging data between Windows applications.

debug

To test, diagnose and (hopefully) solve software bugs. The Clarion debugger offers two general modes:

1.

Hard mode debugging, in which all keyboard and mouse input goes to the debugger first, before being sent to the application. This effectively suspends all other applications which may have been running prior to starting the debugger in hard mode.

2.

Soft mode debugging, in which the debuggee runs as a normal windows application.

deep assignment

(Clarion) Automatically assigns multiple components from one data structure to another, between elements with the same labels (but different prefixes).

default

An assumed state or action, which the end user accepts or executes with little or no action.

default button

A command button which is activated by default when the user presses the enter button.

default window position

The default location at which a new window appears unless a position is specified. The top left corner of the new window is usually below and to the right of the top left corner of the last window, when it first appeared.

delimiter

A character marking the boundaries of one database field from another.

dependent entity

(SQL) A set of data elements dependent on other related entities in the database to identify them..

desktop

The screen area in which all windows, dialog boxes, and icons appear.

DETAIL structure

(Clarion) The portion of a report structure which usually conveys the main data within the printed report. The application loops through, updates, and prints the detail band controls with the contents of all the records being processed.

dialog box

A window, usually not resizeable, which usually requires additional information to be input by the user.

dialog unit

Special fractional measurement units, based on the system font. Windows automatically calculates the horizontal measurement unit in fourths of the average system character width, and the vertical in eighths of character height. The net effect supports a proportional placement of dialog box elements regardless of the resolution Windows is running in.

disabled

A window, menu, or control visible but prevented from gaining focus.

document

Any file which stores data associated with an application.

DOS buffer

A (normally) small amount of memory maintained by the operating system for short-term storage of data transferred to/from a disk drive. The size is set by the BUFFERS setting in the CONFIG.SYS file, where one unit equals 512 bytes.

double-click

To press and release the left mouse button twice, quickly. Executes the default action on a selection.

drag

To press the left mouse button, then move the mouse while continuing to hold the button down. Usually a visual cue indicates a process such as moving a selected object, or rubber-banding a region. Releasing the button completes the action.

drag and drop

To select an object in a window or dialog box, press down the left mouse button, move the mouse while continuing to hold the button down, then release the button when the pointer is on top of another object. When drag and drop is supported by the program(s), the action generally indicates the dropped object is to be processed in some way by the recipient object.

drop-down list

A listbox control which only displays only the current selection when closed. When the user opens the list box, it expands to include additional choices.

dynamic link library

An external file containing functions and procedures which the application may call at runtime, also referred to as a .DLL. When an application calls a .DLL without specifying a path to the file, Windows automatically attempts to load the file from the current directory, the Windows\System directory, and directories listed in the PATH environment variable.

embedded source

(Clarion) Executable code statements, written by the developer, and inserted into generated source at predefined points within a procedure generated by the Application Generator.

enabled

Normal window, menu. or control state allowing focus and/or user input.

encryption

The storage on disk of data in scrambled or encrypted form, such that an unauthorized user may not access the data in an intelligible format.

equi-join

(SQL) A join which takes two database files (or tables) and creates a new, wider table consisting of all possible concatenated records (or rows), where there are matching values in the join fields.

event

An action which triggers a Windows message to the application's message queue. Clarion for Windows handles most of the actual messages internally.

event driven programming

A programming paradigm which describes how an application will respond to possible actions selected and defined by the end user.

Excel format

File format used by the Microsoft Excel spreadsheet application. Note: an ODBC driver exists for this format, and is available in the Microsoft ODBC 2.0 Software Development Kit.

exclusive access

Opening a DOS file so that no other user in a multi-user environment may update the same file.

executable

A standard .EXE application file capable of being launched by the Microsoft Windows shell.

expand

To decompress, usually for installation purposes, a compressed file.

expression

A mathematical formula containing any valid combination of variables, functions, operators, and constants.

extension

A file name suffix; up to three characters in the DOS file system. Windows 3.1 matches document files to their application via the [Extensions] section in the WIN.INI file.

external name

(Clarion) An attribute which holds the native format name (such as a DOS file name) for a given data element. The Clarion source code refers to the file by the Clarion label.

external procedure

(Clarion) A procedure contained in an external library, such as a library file linked at the time the application is built, or a .DLL, linked at run time.

field

A basic data element or category which names all the values in a column of data within a database file or table.

field equate label

(Clarion) A symbolic constant which references an integer, which references a window control.

field event

(Clarion) An event generated and processed within an ACCEPT loop, specific to a control in a window structure.

file handle

An operating system pointer to a file. The "FILES=" line in the CONFIG.SYS file sets the system limit on the total number of allowable open files at one time.

fill color

The color designated for the inside of a graphical control.

filter

An expression which isolates a subset of records for an operation.

focus

A visual cue indicating the window control which will receive the next action resulting from user input.

font

The family name of related type face files. For example, "Times New Roman" is the font name, and "Times New Roman plain," "Times New Roman Italic," "Times New Roman Bold," and "Times New Roman Bold Italic" are the styles, which are stored in separate files.

font dialog

A standard Windows dialog for picking a typeface, style, size, and optionally, the text color.

font style

Character formatting applied to a font face, such as bold, italic, or bold italic.

foreground priority

A measure, expressed in a ratio, for the amount of CPU processing time allocated to a program or task which currently has system focus.

foreign key

(SQL) A key in one table (database file) whose values match the primary key of another table.

form letter

A mailmerge document containing "boiler-plate" text, in which controls reference fields from which to obtain information when creating letters to individuals.

form report style

A report format generally containing one record per page, with field labels and values arranged in a vertical format.

format string

(Clarion) A string specifying the display format for a list box or drop down list box control.

formatter

(Clarion) A specialized window which allows you to visually define the formatting for a data structure in "WYSIWYG" fashion.

function

(Clarion) A specialized procedure which returns a value. The function declaration may optionally define parameters which are passed when calling the function. A function may be used within computed or conditional fields.

GDI

Abbreviation for Graphics Device Interface, the Microsoft Windows dynamic link library responsible for outputting text and images to the screen and printer.

GIF image

Graphics Interchange File format; an image format popularized by CompuServe. Generally acknowledged to offer the best compression ration for 256 color or less images. Attention: should you utilize the word "GIF" anywhere within an application or program, you must add a trademark notice: "GIF (Graphics Interchange Format) is a trademark of CompuServe Information Services."

global variable

(Clarion) A variable accessible from all levels of a program. Global variables are allocated memory that is not released until the entire program finishes execution.

graph

A graphical representation of related data elements, on screen or paper.

Graphical User Interface

(GUI) An operating system or program environment relying heavily on images to present information to the user and to gather the user's input.

grayed

A visual cue to the user that the window, menu, or control is unavailable or disabled.

grid snap

A series of coordinates, represented by dots, such as those used by the Clarion Window and Report Formatters, to force controls to exact positioning.

group

(Clarion) A compound data structure which allows you to reference its component variables with a single label.

groupbox

A rectangular line frame with a label at upper left, used to define related controls.

handle

In Windows, an integer serving as a pointer to the memory location for a given object, most commonly a handle to a window (HWND). The handle has approximately the same importance to most API functions as the zip code on a first class letter. In Clarion, its functionality is implemented via field equate labels. You *can* obtain the actual handle to a window or control by examining PROP:handle. The property is read only.

help context string

A unique identifier for a topic or page in a help file, which can be passed to the help engine.

help system

Comprised of the Windows help application (WINHELP.EXE) and a help document (*.HLP) distributed by individual applications. When displaying help, both the application which called it, and WINHELP.EXE are running.

help topic

A page in a Windows help document.

help compiler

A utility available from Microsoft for converting a Rich-Text-Format (.RTF) document into a Windows help (.HLP) document.

hide

Prevent a control or window from displaying on screen; the control exists but is not seen by the end user.

I-beam

A special cursor usually indicating the end user can type text into an edit control.

I/O

Input/Output. The process of moving information into and out of the system.

icon

A graphical representation of a physical object in the system, such as a printer. Also, any small image representing an action, concept or program, as when an icon appears on a command button. The normal icon file format carries the .ICO extension; one of its main features is built-in support for transparency. This enables you to display a small picture without obliterating the background.

IDE

Integrated Development Environment; a complete compiler product which includes tools for producing source code, creating resources, compiling, linking, and debugging an application.

identifier

A label uniquely identifying a variable or other program element.

implicit variable

(Clarion) A specialized variable not declared within the data structure of an application, nor defined before its first use. The compiler creates them when it first encounters them (usually within executable code) and automatically initializes them to zero.

include file

An external source file read and preprocessed at compile time. In Clarion for Windows, the Equates and other files in the LIBSRC subdirectory are the default include files.

independent entity

(SQL) A set of data elements sharing a set of properties independent of other related entities in the database. Independent entities have unique identifiers, and therefore, primary keys.

index file

An external key file ordered according to the contents of a specified field or expression. An index file usually must be manually updated when adding, deleting, or changing records.

INI file

A Windows Initialization file in ASCII format. The .INI file is divided into sections separated by an identifier enclosed in square brackets. Variables and their values follow, each pair separated by a carriage return, with an equal sign between the variable name and its value. Values may be stored as strings or integers.

insertion point

The point in a document at which the next characters typed by the end user will appear.

interface

The communication between the computer and the user; it presents information to the user and accepts the user's input.

ISAM

Indexed Sequential Access Method; a database organization in which data files are ordered by keys, and may be retrieved in the sequence of the keys.

join

A join takes two database files (or tables) and creates a new, wider table consisting of all possible concatenated records (or rows).

JPG image

A true-color graphics file format featuring 24-bit color storage. It usually provides for adjustable lossy compression, which allows for greater compression but loss of some resolution.

key

An indexed file ordered according to the contents of a specified field or fields. Keys are usually dynamically updated whenever the value in a key field changes.

key-in template picture

(Clarion) A formatting option, which when combined with the MASK attribute, restricts and verifies end user keyboard input according to a specified character pattern applied upon a variable.

keyboard accelerator

A hot-key combination which directly executes a command.

keyword

A reserved word or Clarion language statement.

label

(Clarion) A unique identifier for a variable, procedure, function, routine, or data structure.

library file

A precompiled file (.LIB) containing procedures or functions which may be statically linked to the executable and utilized by a program.

license file

A proprietary key file distributed by a VBX vendor only to a licensed user of the VBX library. The license file allows an IDE to incorporate the VBX control within a window or dialog box. This file is *not* redistributable to the end user.

listbox

A window control presenting data arranged in rows, and optionally, columns.

literal

A constant referred to in source code by its value. For example, the literal "MyString" refers to a seven byte data item containing ASCII codes for the letters in "MyString."

local data

Data created by, residing in memory specific to, and accessible only to a specific procedure or function.

lock

A concurrency control mechanism to prevent more than one user from updating the same record at the same time. Within Clarion, the HOLD statement arms record locking.

locked field or record

A field or record currently being updated by one user within a multi-user database, such that an attempt by another user to update the same record at the same time will fail.

logical operator

A true/false or bitwise comparison of two values; logical operators are: =, >, <, <>, >=, <+, NOT, AND, OR, and XOR.

lookup table

A database file on one side of a one to many relation, upon which a variable is searched for, and a corresponding field in the related table is returned.

LOOP structure

(Clarion) A control structure which repeats the execution of the statements it encloses for a specified count.

many-to-many relationship

A connection between two data entities in which there may exist many corresponding values in the foreign key in one database file or table, to many corresponding values in the foreign key of another table. Usually implemented via a "join" file breaking them into two 1:Many relations.

many-to-one relationship

A connection between two data entities in which there may exist many corresponding values in the foreign key in one database file or table, to only one value in the primary key of another "look-up" table. The relationship implicitly describes the direction of the relation. For example, the relation of cities to states implies many cities may belong to the same state. Also called a child-parent relation.

MASK

(Clarion) Specifies pattern editing of user input, converting data to a predefined format. The pattern is specified for an individual control, and enabled when the MASK attribute is added to the window in which the control appears.

maximize box

A window control which resizes a window to full size of the desktop, or if a child window, to the full size of the client area of the application window.

Media Control Interface

The multimedia API support component of Microsoft Windows. Managed by the MMSYSTEM.DLL library and related driver files; abbreviated as MCI.

memo

A free-form, variable length text field, suitable for storing very long strings. In most PC file formats, the memo is stored in a file separate from the fixed-length database fields. A binary memo field is a specialized type of memo field suitable for storing binary information such as graphics.

menu

An element of the user interface listing available actions which the end user may effect upon a document or selected portion of a document.

message box

A standard windows element, usually consisting of a short message string, an OK button, often a standard icon such as "stop" or "information." It may optionally contain additional buttons such as "Cancel," and "Retry."

message queue

The "place" in which Windows holds all messages for an application, which the application checks on a regular basis. The messages consist of everything the application needs to know regarding the user interface--keyboard, mouse and menu events; the system--shutdown messages, and all the other operations which may affect the application. Clarion processes the entire messaging process transparently in the ACCEPT loop.

metafile

In Windows, the representation of a graphic or line art in vector format; defines the image as a series of lines and curves, allowing for smooth resizing. Clarion for Windows supports the .WMF (Windows Metafile) vector format. The metafile is actually a stored collection of the commands which instruct the GDI (Windows Graphics Device Interface) to display the graphic on the output device.

minimize box

A window control which resizes a window to iconic size, usually at the bottom of the desktop, or if a child window, to iconic size, usually at the bottom of the application window.

mnemonic access key

The underlined letter in the command names on Microsoft Windows menus. When a user activates a pulldown menu, the key executes the command.

modal window

A dialog or window which prevents the end user from activating controls from any other of the application's windows (or of any other application, if system modal), until processing of the modal window is completed and the window closed.

modeless dialog

A dialog which remains open even while the user "works" in another of the application's document windows. The modeless dialog remains available, so that the user can utilize its functionality; as in a Search dialog, as practiced by most applications.

module

(Clarion) A source or library file for a given project.

multi-tasking

The capability of an operating system to execute multiple programs at the same time. Pre-emptive multi-tasking allots percentages of CPU time to each individual task, with the operating system automatically switching to the next task at the end of its time allotment. Cooperative multi-tasking, supported by Windows 3.1, relies upon the currently executing program to finish a task, or part of one, then yield to the next program. See also *Thread*.

multiple selection

An extended listbox selection, signifying the user has marked more than one item for a subsequent action.

multilevel index

To speed up access to a rage table or data file, a multilevel index functions as an index to an index. For example, index level one could contain pointers to four subindexes which respectively index entries beginning with A-E, F-L, M-R, and S-Z. This example describes a classic B-TREE index structure.

Multiple Document Interface

(MDI) A Windows programming convention which allows an application to manage several documents, or views of documents, each in its own child window, all in an application frame window.

multi-user database

A database system designed so that more than one user can access a file or record at the same time. The system requires concurrency checking so that two users don't attempt to update the same record at the same time.

natural join

(SQL) A join which takes two database files (or tables) and creates a new, wider table consisting of all possible concatenated records (or rows), where the new table contains two identical columns, one of which is dropped.

nested queries

(SQL) A single query consisting of both an outer and inner query. Allows for more efficient retrieval of data from large tables by combining multiple operations into one.

nesting

Placing one operation inside another, such as nesting a function within another by specifying the nested function as a parameter of the first.

non-Windows application

Any application which doesn't require the Windows environment. Typically, a DOS program.

normalization

The representation of data entities in their simplest forms, for the purpose of quickest access and most efficient storage. The normalization process includes the elimination of redundant data groups, and the elimination of redundant data elements.

null value

A zero or empty value.

ODBC

The Open Database Connectivity standard supported by many Windows applications. Provides a standard API for accessing multiple database file formats via replaceable file drivers, and Client/Server support. The ODBC SDK is published by Microsoft.

ODBC Administrator

A redistributable Microsoft application for adding, maintaining or deleting individual ODBC drivers within a system. Usually located in the Windows\System directory, the executable file name is ODBCADM.EXE.

ODBC Control Panel applet

A Windows Control Panel interface to the ODBC administrator.

ODBC driver

A driver library containing the individual functions supporting standard ODBC calls for a particular file format.

one-to-many relationship

A connection between two data entities in which there may exist one corresponding value in the primary key of one database file or table, to many identical values in the foreign key of another table. The relationship implicitly describes the direction of the relation. For example, the relation of states to cities implies a state may have many cities. Also called a parent-child relation.

one-to-one relationship

A connection between two data entities in which there may exist one and only one corresponding value in the primary key of one database file or table, to a single identical value in the foreign key of another table. For example, the relation of customer name to internet address. The data is usually split into two separate tables for storage savings; all customers have names, but only a minority have internet addresses.

option structure

(Clarion) A structure containing mutually exclusive controls, such as radio buttons.

origin

The upper left corner of a window or control, expressed in x,y coordinates (0,0).

orphan

A portion of text or data separated from its complementary preceding data by a page break.

outer join

(SQL) A join which includes all records from one database file, and only those records from another in which the values in a selected field (or fields) match those in the first.

overlay

(Clarion) A variable or field sharing the same location as another. Acts as a data "re-declaration, and provides more efficient storage. Most useful in "either/or" situations when a variable and its overlay are of similar types but utilize different pictures.

page footer

The section of a report composed after the detail.

page header

The section of a report composed before the detail.

page overflow

In Clarion, the point at which the report library composes enough data to complete a page; the library will either send the page to the Windows spooler at that point, or first check to verify there are no "widows," if the application so specifies.

palette

The table of available colors which a given window may user for painting.

parameter

An argument or optional variable passed to a procedure.

PCX image

A standard graphics file format, offering moderate compression, originally developed by the Zsoft corporation. The Windows Paintbrush accessory supports this format.

pel

Equivalent to pixel; abbreviation for picture element. The smallest screen unit addressed in graphic mode; a dot.

pen

In Windows, the active drawing or painting element; you can set its color, size, etc.

picture token

(Clarion) A formatting string, which specifies a specific "picture" or masking format for displaying and editing variables. The picture token begins with the "@" character.

pixel

Equivalent to pel; abbreviation for picture element. The smallest screen unit addressed in graphic mode; a dot.

point size

A measurement expressed in points; one point equals 1/72nd inch, or 1/28 centimeter.

pointer

The mouse cursor. Or, an index entry which locates or "points" to the corresponding data record.

prefix

(Clarion) A short identifying string for a data structure. Provides a method for resolving variable names when, for example, two database files include fields whose names are the same.

primary key

(SQL) A database field or expression which uniquely identifies each record in the table or database file.

print job

One complete task sent to the Windows print spooler (accessible from Print Manager).

print structure

(Clarion) The parts of a report structure, which include the group break structure, detail, header, footer, and form.

printer driver

An external library file containing low level instructions and functions by which the Windows GDI library sends specific commands to the printer.

printer font

A typeface resident in the printer's RAM.

procedure

(Clarion) A set of executable statements which may be executed repeatedly.

program MAP

(Clarion) The "layout" of modules, procedures and functions, which the compiler uses to logically assemble the file. The MAP structure contains the prototypes which declare the functions, procedures, and external source modules used in a PROGRAM or MEMBER module.

project system

(Clarion) The IDE component which tracks the modules which comprise the application to be built, including source code and external libraries. The Project System also stores the various pragma , compiler and linking options.

prompt

A text label which normally appears near a screen control, to identify the control.

property assignment syntax

(Clarion) Specific language format for setting or retrieving the value of a control property.

prototype

To define the parameter(s) and return data types for a procedure or function. Within Clarion, prototypes are defined within the MAP structure.

PUT statement

(Clarion) A statement which executes an update to a given record, and writes it to disk.

query

(SQL) An operation upon a database table which results in another table or subset of the first.

Query by Example

A query built by "filling-in the blanks" in a form representing the fields in a database table. The end user types in "example elements" which represent the possible answers to the query.

queue

(Clarion) A specialized memory structure containing a doubly-linked list of values.

radio button

A control for eliciting a mutually exclusive choice from an end user.

range constraint

A bounds for a database operation limiting the operation to a set of records for which a given field falls within specified starting and ending values.

raster font

A bitmapped typeface, stored as a pattern of dots.

read only

(Clarion) A field or variable which is displayed but not modified.

RECORD

(Clarion) A data structure representing one row in a database table.

redirection file

(Clarion) A list of alternate subdirectories to search for source code, object or library files.

reference variable

(Clarion) An indirection to another data variable (the target). The reference variable label can substitute for the target variable anyplace in executable code. Depending upon the target data type, the reference variable may contain the address in memory of the target, or a more complex internal data structure.

referential integrity

The process by which an application "follows through" on an update to a key field in one file, to check its related record in another file. This maintains valid parent-child relationships within the database. The Application Generator can automatically generate the executable code to support referential integrity constraints when you select options in the Relate dialog.

region

A specialized control whose sole function is to provide a reference for a screen area in x,y coordinates.

registry

A specialized initialization file storing various values and parameters in binary format.

relationship

A logical link between records in data files based upon a duplicate (linking) field.

report form

(Clarion) A report element defined once, when first composing the report, then printed on all pages of the report.

resource file

An external file containing data for a window control, such as an icon file.

restore button

A window control which resizes a window from a maximized state to the last size prior to maximizing.

rich text format (RTF)

A common word processing file format, originally designed for transportability between word processing systems across different operating systems. The default format for the source document for the Windows help file format.

ROLLBACK

(Clarion) To restore an earlier state of a database, undoing the effect of one or more active transactions. Restores data held in a temporary file managed by the file driver.

ROUTINE

(Clarion) A series of executable statements local to a procedure or function. Following execution of the ROUTINE, program control returns to the calling procedure or function.

run time library

A dynamic link library providing essential support for basic application functions. For example, the Clarion runtime library provides all the "housekeeping" functions such as checking message queues, and managing the allocation and deallocation of all device contexts (for windows and reports).

schema

The map or catalog of a database describing its files or tables, fields, and relations.

scope

A range of records selected for a given operation. Also, the "boundaries" beyond which a given variable is unavailable to another procedure or function.

scroll bar

Standard window control for changing the view of data within a window, displaying more of a document or application controls than currently visible.

select

To indicate to the system that the next command should act upon an on screen object, by placing the mouse cursor over it and pressing the left mouse button.

SELECT statement

(SQL) A statement setting the fields and tables for viewing, and for subsequent operations.

(Clarion) Sets the next control to receive input focus.

selected event

An event generated and sent to the ACCEPT loop when a control obtains focus.

sequential access

The ability to manipulate all the records in a database file or table in the sequence defined by the key or index.

server

A remote computer providing data storage or services to other linked computers.

SET statement

A Clarion language statement preparing a file for sequential processing upon a group of records.

SHARE.EXE

The MS-DOS executable responsible for supporting multi-user access to a single file.

sort

Physically rearrange all database records in a specified order, and store the results in a new database file or table.

source code file

(Clarion) A text file containing Clarion language statements in a structured format, which the compiler can compile and link into an executable program.

spin control

A specialized edit box control, with two "increaser" and "decreaser" controls, linked to an array of values. When the end user increases or decreases the control, it updates to display the next value in the array.

SQL

Structured Query Language; a database language for maintaining a relational database; most often utilized in mainframe and client/server applications.

stack memory

A portion of memory which usually stores the most recent parameter data utilized by procedures and commands executed by a program or application.

statement

A single executable command.

static text

A window control which displays a string constant, and never receives focus; primarily used for labeling other controls or displaying information and instructions.

static variable

(Clarion) A persistent variable, which maintains its value from one use within a procedure to the next.

status bar

An area of a window, usually found at the bottom, in which the program can display prompts and information.

standard behavior

(STD) (Clarion) A predefined set of operations associated with a menu command; the actions are automatically supported by the run-time library, without requiring specific code on the part of the application.

stream mode

A special mode for several of the Clarion database drivers which optimizes file input/output.

swap file

A system file maintained by Windows for maintaining virtual memory as required by the system.

syntax

A rule specifying the specific format of a language statement.

system colors

The default colors shared by all custom Windows palettes.

system date

The date maintained by the system clock.

tab order

The sequence in which each control in a window gains focus upon a TAB key press.

table

(SQL) A structured collection of data, consisting of a row of fields or column headings plus zero or more rows of data. Each row contains exactly one value for each of the fields. Within Clarion, the table corresponds to a specific FILE, ALIAS, or VIEW structure.

tabular report

A listing of data labels and their corresponding values, arranged in a row of column labels, followed by additional rows of data arranged by column.

tag

For file drivers (such as FoxPro and dBase IV) supporting multiple indexes within the same index file, the indicator marking an individual index.

target file

Indicates to the project system the name of the application or library file to be built.

task

A currently executing Windows application.

template procedure

(Clarion) A pre-written source code module written in the Clarion Template Language, containing "boiler-plate" Clarion language code, instructions for processing it at code generation, plus a user interface for gathering the customization instructions from the developer.

text control

A multi-line edit control which automatically supports word wrap.

text file

An ASCII file.

text justification

A paragraph alignment style which lines up the edges of the paragraph at left, right, left and right, or centers the entire line.

third normal form

A test or measure of how closely a database meets relational theory tests for data normalization.

thread

In a multi-threaded operating system such as Windows NT, the thread is the basic entity to which the operating system allocates a slice of CPU time. The thread has access to the same code, data, and system resources as the task (program) which started it. Clarion START threads do not receive separate "timeslices" from Windows 3.1; the run time library "slices" the Clarion thread and "divides" it among the Clarion START threads.

thumb

The box control on a scroll bar.

timer

A Windows resource which can automatically send a message to an application at pre-defined intervals.

token

A structured symbol or series of symbols, recognized and parsed by the compiler. Operators, and variable names are examples of tokens.

toolbar

A horizontal or vertically arranged group of command buttons, and/or other controls, generally remaining accessible the entire time a program executes.

transaction

The logical event during which an input or entry to a database record, held for sequential management with other entries, is written to disk. Failure of any of the disk writes during the transaction would compromise the integrity of the database.

tree control

Displays a logically hierarchical list of items in collapsible outline format. In Clarion for Windows, a small square filled with a plus or minus symbol, followed by a folder, represents an expandable tree control.

untyped parameter

(Clarion) Within a function prototype, specifies the data type of a parameter is to be resolved at tun time.

USE variable

(Clarion) An attribute indicating a variable whose value should display in a window or report control.

validity check

An executable code procedure which checks end user input against an expression defining acceptable values for a given field.

VBX control

A custom window control for processing end user input or displaying data.

VCR controls

A set of icons designed for use in navigating a browse or list; the images on the controls bearing a similarity to the controls on a video cassette recorder.

vector font

A scalable typeface, such as a TrueType font.

vector graphic

A binary file representation of a graphic or line art; defines the image as a series of lines and curves, allowing for smooth resizing. Clarion for Windows supports the .WMF (Windows Metafile) vector format.

view

A virtual file containing selected fields from one or more related database files.

virtual table

A data table or view which exists in memory only, constructed from one or more tables or data files which may exist on disk.

watch variable

A variable designated for monitoring by the Debugger.

widow

A portion of text or data separated from its complementary following data by a page break.

window frame

The window boundary. Dialog window frames are not resizeable. End users can resize other windows by dragging the frame.

window pane

A specialized window which acts as a "part" of a greater window. This allows an end user to divide an active window into separate sections which may then be scrolled independently or in sync.

WinExec

The standard Windows API function for calling another application. Supported in Clarion via the RUN statement.

X axis

The horizontal axis. Used for locating controls; the leftmost pixel in a window is position zero.

Y axis

The vertical axis. Used for locating controls; the upper pixel in a window is position zero.

Syntax

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

-A-

-B-

-C-

CHAIN (execute another program)
CHANGE (change control field value)
CHOICE (return relative item position)
CHORD (draw a section of an ellipse)
CHR (return character from ASCII)
CLIP (return string without trailing spaces)
CLIPBOARD (return windows clipboard contents)
CLOCK (return system time)
CLOSE (close a data file)
CLOSE (close a VIEW)
CLOSE (close an active report structure)
CLOSE (close window)
CODE (begin executable statements)
COLORDIALOG (return chosen color)
COMMAND (return command line)
COMMIT (terminate successful transaction)
COMPILE (specify source to be compiled)
CONTENTS (return contents of USE variable)
COPY (copy a data file)
COS (return cosine)
CREATE (create an empty data file)
CREATE (create new control)
CYCLE (go to top of loop)

-D-

DATE (return standard date)
DAY (return day of month)
DDEAPP (return server application)
DDECHANNEL (return DDE channel number)
DDECLIENT (return DDE client channel)
DDECLOSE (terminate DDE server link)
DDEEXECUTE (send command to DDE server)
DDEITEM (return server item)
DDEPOKE (send unsolicited data to DDE server)
DDEQUERY (return registered DDE servers)
DDEREAD (get data from DDE server)
DDESERVER (return DDE server channel)
DDETOPIC (return server topic)
DDEVALUE (return data value sent to server)
DDEWRITE (provide data to DDE client)

-E-

-F-

-G-

-H-

-I-

-J-

-K-

-L-

-M-

-N-

-O-

-P-

-Q-

-R-

-S-

-T-

TAN (return tangent)
THREAD (return current execution thread)
TITLE (print MODULE title)
TODAY (return system date)
TYPE (write string to screen)

-U-
UNBIND (free runtime expression string variable)
UNHIDE (show hidden control)
UNLOCK (unlock a locked data file)
UPDATE (write from screen to USE variables)
UPPER (return upper case)

-V-
VAL (return ASCII value)

-W-
WATCH (automatic file concurrency check)
WATCH (automatic view concurrency check)

-X-

-Y-
YEAR (return year of date)
YIELD (allow event processing)

-Z-

Data Attributes

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

-A-

ABSOLUTE (set fixed-position printing)

ALONE (set to print without page header, footer, or form)

ALRT (set control "hot" keys)

ALRT (set window "hot" keys)

APPLICATION (declare an MDI frame window)

AUTO (uninitialized local variable)

AUTO (set USE variable automatic re-display)

AT (set control position and size in report)

AT (set control position and size in window)

AT (set detail print area)

AT (set print structure position and size)

AT (set window position and size)

AVE (set total average)

-B-

BFLOAT4 (four-byte signed floating point)

BFLOAT8 (eight-byte signed floating point)

BINARY (MEMO contains binary data)

BINDABLE (set dynamic expression string variables)

BINDABLE (set runtime expression string RECORD variables)

BOX (declare a report box control)

BOX (declare a window box control)

BOXED (set report controls group border)

BOXED (set window controls group border)

BREAK (declare group break structure)

BUTTON (declare a pushbutton control)

BYTE (one-byte unsigned integer)

-C-

CAP, UPR (set display case)

CAP, UPR (set print case)

CENTER (set position and size)

CHECK (declare a report checkbox control)

CHECK (declare a window checkbox control)

CHECK (set on/off ITEM)

CLASS (set .VBX custom control class)

CNT (set total count)

-D-

-E-

-F-

-G-

-H-

-I-

-J-

-K-

KEY (declare dynamic file access index)

KEY (set control execution keycode)

-L-

LANDSCAPE (set page orientation)

LEFT, RIGHT, CENTER, DECIMAL (set display justification)

LEFT, RIGHT, CENTER, DECIMAL (set print justification)

LIKE (inherited data type)

LINE (declare a report line control)

LINE (declare a window line control)

LIST (declare a report list control)

LIST (declare a window list control)

LONG (four-byte signed integer)

-M-

MARK (set multiple selection mode)

MASK (set pattern editing data entry)

MAX (set maximize control)

MAX (set total maximum)

MAXIMIZE (set window open maximized)

MDI (set MDI child window)

MEMO (declare a text field)

MENU (declare a menu box)

MENUBAR (declare a pulldown menu)

META (set .VBX to print as .WMF)

MIN (set total minimum)

MODAL (set system modal window)

MSG (set control status bar message)

MSG (set window status bar message)

MM, THOUS, POINTS (set report coordinate measure)

-N-

NAME (set external name)

NAME (set queue variable external name)

NAME (set filename)

NAME (set variable´s external name)

NOBAR (set no highlight bar)

NOCASE (case insensitive KEY or INDEX)

NOFRAME, DOUBLE, RESIZE (set window border)

-O-

-P-

-Q-

-R-

-S-

-T-

-U-

-V-

-W-

-X-

-Y-

-Z-

SPREAD (set evenly spaced TAB controls)

SPREAD

The **SPREAD** attribute specifies a SHEET's TAB controls are evenly spaced.

WIZARD (set "tabless" SHEET control)

WIZARD

The **WIZARD** attribute specifies a SHEET control that does not display its TAB controls. This allows the program to direct the user through each TAB in a specified sequence (usually with "Next" and "Previous" buttons.

BLOB (declare a variable-length memo field)

label **BLOB** [,**BINARY**] [,**NAME( )**]

BLOB Declares a variable-length string which may be greater than 64K (in both 16 and 32-bit applications).

BINARY Declares the BLOB a storage area for binary data.

NAME Specifies the disk filename for the BLOB field. (Use of this parameter is file driver dependent.)

BLOB (Binary Large OBject) declares a string field which is completely variable-length and may be greater than 64K in size (in both 16 and 32-bit applications). A BLOB must be declared before the RECORD structure. Memory for a BLOB is dynamically allocated and de-allocated as necessary. Generally, up to 255 BLOB fields may be declared in a FILE structure. The exact number of BLOB fields and their manner of storage on disk is file driver dependent.

A BLOB may not be accessed "as a whole;" you must use "string slicing" syntax to access one piece (up to 64K) at a time. A BLOB may not be used in the same manner as a variable (may not be named as a control's USE attribute, etc.).You can use PROP:Handle to get the windows handle to the BLOB entity. This provides the only mechanism to assign one BLOB to another: get the handle of both BLOB entities and then assign one BLOB's handle to the other BLOB's handle. The SIZE function returns the number of bytes contained in the BLOB field for the current record in memory. You can also get (and set) the size of a BLOB using PROP:BlobSize.

Example:

```
Names      FILE,DRIVER('TopSpeed')
NbrKey     KEY(Names:Number)
Notes        BLOB            !Can be larger than 64K
Rec        RECORD
Name         STRING(20)
Number       SHORT
           . .
ArcNames FILE,DRIVER('TopSpeed')
NbrKey     KEY(ArcNames:Number)
Notes        BLOB
Rec        RECORD
Name         STRING(20)
Number       SHORT
           . .
  CODE
  OPEN(Names)
  CREATE(ArcNames)
  SET(Names)
  LOOP
    NEXT(Names)
    IF ERRORCODE() THEN BREAK.
    ArcNames:Rec = Names:Rec        !Assign record data to Archive file
    ArcNames:Notes{PROP:Handle} = Names:Notes{PROP:Handle}   !Assign BLOB to Archive
    ADD(ArcNames)
  END
```

DIRECTORY (get file directory)

DIRECTORY(*queue, path, attributes* **)**

DIRECTORY Gets a file directory listing (just like the DIR command in DOS).

queue The label of the QUEUE structure that will receive the directory listing. This must be exactly the same structure as the ff_:queue structure in the EQUATES.CLW file.

path A string constant, variable, or expression that specifies the path and filenames directory listing to get. This may include the wildcard characters (* and ?).

attributes An integer constant, variable, or expression that specifies the attributes of the files to place in the *queue*.

The **DIRECTORY** procedure returns a directory listing of all files in the *path* with the specified *attributes* into the specified *queue*.

The *queue* parameter must name a QUEUE with a structure that begins the same as the following structure contained in EQUATES.CLW:

```
ff_:queue   QUEUE,PRE(ff_),TYPE
name        STRING(13)
date        LONG
time        LONG
size        LONG
attrib      BYTE
          END
```

Your QUEUE may contain more fields, but must begin with these five fields. It will receive the returned information about each file in the *path* that has the *attributes* you specify. The date and time fields will contain standard Clarion date and time information (the conversion from the operating system's storage format to Clarion standard format is automatic).

The *attributes* parameter is a bitmap which specifies what filenames to place in the *queue*. The following equates are contained in EQUATES.CLW:

```
ff_:NORMAL       EQUATE(0)
ff_:READONLY     EQUATE(1)
ff_:HIDDEN       EQUATE(2)
ff_:SYSTEM       EQUATE(4)
ff_:DIRECTORY    EQUATE(10H)
ff_:ARCHIVE      EQUATE(20H)      ! NOT Win95 compatible
```

The *attributes* bitmap is a non-exclusive OR filter: if you add the equates, you get files with any of the attributes you specify. This means that, when you just set the ff_:NORMAL attribute, you only get files (no sub-directories) without the read-only, hidden, system, or archive attributes set. If you add ff_:DIRECTORY to ff_:NORMAL, you will get files AND sub-directories from the *path*.

Example:

```
DirectoryList  PROCEDURE

AllFiles      QUEUE,PRE(FIL)
name          STRING(13)
date          LONG
time          LONG
size          LONG
attrib        BYTE
            END
LP          LONG
```

```
Recs        LONG

  CODE
  DIRECTORY(AllFiles,'*.*',ff_:DIRECTORY)   !Get all files and directories
  Recs = RECORDS(AllFiles)
  LOOP LP = 1 to Recs
    GET(AllFiles,LP)
    IF BAND(FIL:Attrib,ff_:DIRECTORY) AND FIL:Name <> '..' AND FIL:Name <> '.' THEN
      CYCLE           !Let sub-directory entries stay
    ELSE
      DELETE(AllFiles)   !Get rid of all other entries
    END
```

REJECTCODE (return reject code number)

REJECTCODE( )

The **REJECTCODE** function returns the code number for the reason any EVENT:Rejected that was posted. If no EVENT:Rejected was posted, REJECTCODE returns zero. The EQUATES.CLW file contains equates for the values returned by REJECTCODE:

```
REJECT:RangeHigh          ! Above the top range on a SPIN
REJECT:RangeLow     ! Below the bottom range on a SPIN
REJECT:Range                    ! Other range error
REJECT:Invalid            ! Invalid input
```

Return Data Type: LONG

Example:

```
CASE EVENT()
OF EVENT:Rejected
  EXECUTE REJECTCODE()
    MESSAGE('Input invalid -- out of range -- too high')
    MESSAGE('Input invalid -- out of range -- too low')
    MESSAGE('Input invalid -- out of range')
    MESSAGE('Input invalid')
  END
END
```

PAPER (set report paper size)

PAPER([*type*] [,*width*] [,*height*]**)**

PAPER Defines the paper size for the report.

type An integer constant or EQUATE that specifies a standard Windows paper size. EQUATES for these are contained in the EQUATES.CLW file.

width An integer constant or constant expression that specifies the width of the paper.

height An integer constant or constant expression that specifies the height of the paper.

The **PAPER** attribute on a REPORT structure defines the paper size for the report. The *width* and *height* parameters are only required when PAPER:Custom is selected as the *type*.

The values contained in the *width*, and *height* parameters default to dialog units unless the THOUS, MM, or POINTS attribute is also present. Dialog units are defined as one-quarter the average character width by one-eighth the average character height. The size of a dialog unit is dependent upon the size of the default font for the report. This measurement is based on the font specified in the FONT attribute of the report, or the printers default font.

Example:

```
CustRpt1 REPORT,AT(1000,1000,6500,9000),THOUS,PAPER(PAPER:Custom,8500,7000)
                                        ! print on 8.5" x 7" paper
         !report declarations
         END

CustRpt2 REPORT,AT(72,72,468,648),POINTS,PAPER(PAPER:A4)
                                        ! print on A4 size paper
         !report declarations
         END
```

Example Source for Global Program Section

The following code was generated from the Clarion Program template.

Comments appear in **red** at the default embed points.

```
  PROGRAM

!Embed: Before Global INCLUDEs
  INCLUDE('Equates.CLW')
  INCLUDE('TplEqu.CLW')
  INCLUDE('Keycodes.CLW')
  INCLUDE('Errors.CLW')
!Embed: After Global INCLUDEs
  MAP
    MODULE('EMBED001.clw')
      Main
    END
    MODULE('EMBED002.clw')
      BrowseNames
    END
    MODULE('EMBED003.clw')
      PrintNAM:KeyNumber
    END
    MODULE('EMBED004.clw')
      PrintNAM:KeyZip
    END
    MODULE('EMBED005.clw')
      UpdateNames
    END
    MODULE('EMBED_SF.CLW')
      CheckOpen(FILE,<BYTE>,<BYTE>)
      ReportPreview(QUEUE)
      StandardWarning(LONG,<STRING>,<STRING>,<STRING>,<STRING>),LONG,PROC
      SetupStringStops(STRING,STRING,LONG,<LONG>)
      NextStringStop,STRING
      SetupRealStops(REAL,REAL)
      NextRealStop,REAL
      INIRestoreWindow(STRING,STRING)
      INISaveWindow(STRING,STRING)
    END
    MODULE('EMBED_RU.CLW')
      RIUpdate:Names,LONG
      RISnap:Names
    END
    MODULE('EMBED_RD.CLW')
      RIDelete:Names,LONG
    END
    !Embed: Inside the Global Map
  END


GlobalRequest     LONG(0),THREAD
GlobalResponse    LONG(0),THREAD
!Embed: Before File Declarations
Names               FILE,DRIVER('TOPSPEED'),PRE(NAM),CREATE,THREAD
KeyNumber             KEY(NAM:Number),NOCASE,OPT
KeyZip              KEY(NAM:Zip),DUP,NOCASE
```

```
Record                RECORD,PRE()
Number                   DECIMAL(3)
FirstName                STRING(20)
LastName                 STRING(20)
Address                  STRING(20)
City                     STRING(20)
State                    STRING(2)
Zip                    DECIMAL(5)
                       END
                     END
Names::Used          LONG,THREAD

!Embed: After File Declarations
Sort:Name STRING(ScrollSort:Name)
Sort:Name:Array STRING(3),DIM(100),OVER(Sort:Name)
Sort:Alpha STRING(ScrollSort:Alpha)
Sort:Alpha:Array STRING(2),DIM(100),OVER(Sort:Alpha)
!Embed: Global Data
LocateOnPosition   EQUATE(1)
LocateOnValue     EQUATE(2)
LocateOnTop       EQUATE(3)
FillBackward        EQUATE(1)
FillForward       EQUATE(2)
RefreshOnPosition EQUATE(1)
RefreshOnQueue     EQUATE(2)
RefreshOnTop       EQUATE(3)
RefreshOnBottom    EQUATE(4)
RefreshOnCurrent   EQUATE(5)
  CODE
  !Embed: Program Setup
  Main
  !Embed: Program End
```

Example Source for Browse Procedure

The following code was generated from the Browse procedure template.

Comments appear in **red** at the default embed points.

```
                MEMBER('Prog.clw')        ! This is a MEMBER module
!Embed: Compile Module Declarations
!Embed: Module Data Section
!Embed: INTERNAL Top of SetupBrowseBehavior GROUP
!Embed: INTERNAL Bottom of SetupBrowseBehavior GROUP
!Embed: Gather Template Symbols
BrowseNames PROCEDURE

CurrentTab          STRING(80)
LocalRequest        LONG,AUTO
OriginalRequest       LONG,AUTO
LocalResponse       LONG,AUTO
WindowOpened        LONG
WindowInitialized     LONG
ForceRefresh        LONG,AUTO
RecordFiltered      LONG
!Embed: Data Section, Before Window Declaration

BRW1::View:Browse     VIEW(Names)
                    PROJECT(NAM:Number)
                    PROJECT(NAM:FirstName)
                    PROJECT(NAM:LastName)
                    PROJECT(NAM:Address)
                    PROJECT(NAM:City)
                    PROJECT(NAM:State)
                    PROJECT(NAM:Zip)
                  END

Queue:Browse:1      QUEUE,PRE()                 ! Browsing Queue
BRW1::NAM:Number      LIKE(NAM:Number)          ! Queue Display field
BRW1::NAM:FirstName     LIKE(NAM:FirstName)       ! Queue Display field
BRW1::NAM:LastName    LIKE(NAM:LastName)        ! Queue Display field
BRW1::NAM:Address       LIKE(NAM:Address)         ! Queue Display field
BRW1::NAM:City        LIKE(NAM:City)            ! Queue Display field
BRW1::NAM:State         LIKE(NAM:State)           ! Queue Display field
BRW1::NAM:Zip         LIKE(NAM:Zip)             ! Queue Display field
!Embed: End of list QUEUE 1
BRW1::Position        STRING(512)               ! Queue POSITION information
                  END                       ! END (Browsing Queue)
BRW1::CurrentScroll BYTE                      ! Queue position of scroll thumb
BRW1::ScrollRecordCount LONG                    ! Queue position of scroll thumb
BRW1::Sort1:KeyDistribution LIKE(NAM:Zip),DIM(100)
BRW1::Sort1:LowValue LIKE(NAM:Zip)              ! Queue position of scroll thumb
BRW1::Sort1:HighValue LIKE(NAM:Zip)             ! Queue position of scroll thumb
BRW1::Sort2:KeyDistribution LIKE(NAM:Number),DIM(100)
BRW1::Sort2:LowValue LIKE(NAM:Number)           ! Queue position of scroll thumb
BRW1::Sort2:HighValue LIKE(NAM:Number)          ! Queue position of scroll thumb
BRW1::CurrentEvent LONG                     !
BRW1::CurrentChoice LONG                      !
BRW1::RecordCount   LONG                      !
BRW1::SortOrder     BYTE                      !
BRW1::LocateMode    BYTE                    !
BRW1::RefreshMode   BYTE                      !
```

```
BRW1::LastSortOrder BYTE                    !
BRW1::FillDirection BYTE                    !
BRW1::AddQueue     BYTE                    !
BRW1::Changed      BYTE                    !
BRW1::RecordStatus BYTE                    ! Flag for Range/Filter test
BRW1::ItemsToFill   LONG                     ! Controls records retrieved
BRW1::MaxItemsInList LONG                      ! Retrieved after window opened
BRW1::HighlightedPosition STRING(512)            ! POSITION of located record
BRW1::NewSelectPosted BYTE                     ! Queue position of located record
BRW1::PopupText     STRING(128)                !
QuickWindow        WINDOW('Browse the Names File'),AT(,,358,188),FONT('MS Sans
Serif',8,,),IMM,HLP('BrowseNames'),SYSTEM,GRAY,MDI
                   LIST,AT(8,20,342,124),MSG('Browsing Records'),USE(?
Browse:1),IMM,HVSCROLL,FORMAT('12L|M~Number~@n03@80L|M~First Name~@S20@80L|M~Last
Name~@S20@80L|M~Address~@S20@' &|
       '80L|M~City~@S20@8L|M~State~@s2@20L|M~Zip~@n05@'),FROM(Queue:Browse:1)
                   BUTTON('&Insert'),AT(207,148,45,14),USE(?Insert:2)
                   BUTTON('&Change'),AT(256,148,45,14),USE(?Change:2),DEFAULT
                   BUTTON('&Delete'),AT(305,148,45,14),USE(?Delete:2)
                   SHEET,AT(4,4,350,162),USE(?CurrentTab)
                     TAB('NAM:KeyNumber')
                     END
                     TAB('NAM:KeyZip')
                     END
                   END
                   BUTTON('Close'),AT(260,170,45,14),USE(?Close)
                   BUTTON('Help'),AT(309,170,45,14),USE(?Help),STD(STD:Help)
                 END
!Embed: Data Section, After Window Declaration
  CODE
  !Embed: Initialize the Procedure
  LocalRequest = GlobalRequest
  OriginalRequest = GlobalRequest
  LocalResponse = RequestCancelled
  ForceRefresh = False
  CLEAR(GlobalRequest)
  CLEAR(GlobalResponse)
  !Embed: Procedure Setup
  IF KEYCODE() = MouseRight
    SETKEYCODE(0)
  END
  !Embed: Beginning of Procedure, Before Opening Files
  IF Names::Used = 0
    CheckOpen(Names,1)
    BIND(NAM:RECORD)
  END
  Names::Used += 1
  !Embed: Beginning of Procedure, After Opening Files
  !Embed: Before Opening the Window
  OPEN(QuickWindow)
  WindowOpened=True
  !Embed: After Opening the Window
  BRW1::AddQueue = True
  BRW1::RecordCount = 0
  IF LocalRequest <> SelectRecord
    !Embed: Browse Preparation, Request Normal Operation
  ELSE
    !Embed: Browse Preparation, Request to Select Record
  END
  ?Browse:1{Prop:VScroll} = False
```

```
!Embed: Preparing Window Alerts
?Browse:1{Prop:Alrt,252} = MouseLeft2
?Browse:1{Prop:Alrt,255} = InsertKey
?Browse:1{Prop:Alrt,254} = DeleteKey
?Browse:1{Prop:Alrt,253} = CtrlEnter
?Browse:1{Prop:Alrt,252} = MouseLeft2
!Embed: Preparing to Process the Window
ACCEPT
  !Embed: Accept Loop, Before CASE EVENT() handling
  CASE EVENT()
  !Embed: CASE EVENT() structure, before generated code
  OF EVENT:AlertKey
    !Embed: Window Event Handling AlertKey
  OF EVENT:PreAlertKey
    !Embed: Window Event Handling PreAlertKey
  OF EVENT:CloseWindow
    !Embed: Window Event Handling CloseWindow
  OF EVENT:CloseDown
    !Embed: Window Event Handling CloseDown
  OF EVENT:OpenWindow
    !Embed: Window Event Handling OpenWindow
    IF NOT WindowInitialized
      DO InitializeWindow
      WindowInitialized = True
    END
    SELECT(?Browse:1)
  OF EVENT:LoseFocus
    !Embed: Window Event Handling LoseFocus
  OF EVENT:GainFocus
    !Embed: Window Event Handling GainFocus
    ForceRefresh = True
    IF NOT WindowInitialized
      DO InitializeWindow
      WindowInitialized = True
    ELSE
      DO RefreshWindow
    END
  OF EVENT:Suspend
    !Embed: Window Event Handling Suspend
  OF EVENT:Resume
    !Embed: Window Event Handling Resume
  OF EVENT:Timer
    !Embed: Window Event Handling Timer
  OF EVENT:Move
    !Embed: Window Event Handling Move
  OF EVENT:Size
    !Embed: Window Event Handling Size
  OF EVENT:Restore
    !Embed: Window Event Handling Restore
  OF EVENT:Maximize
    !Embed: Window Event Handling Maximize
  OF EVENT:Iconize
    !Embed: Window Event Handling Iconize
  OF EVENT:Moved
    !Embed: Window Event Handling Moved
  OF EVENT:Sized
    !Embed: Window Event Handling Sized
  OF EVENT:Restored
    !Embed: Window Event Handling Restored
  OF EVENT:Maximized
```

```
  !Embed: Window Event Handling Maximized
OF EVENT:Iconized
  !Embed: Window Event Handling Iconized
ELSE
  !Embed: Other Window Event Handling
!Embed: CASE EVENT() structure, after generated code
END
!Embed: Accept Loop, After CASE EVENT() handling
!Embed: Accept Loop, Before CASE FIELD() handling
CASE FIELD()
!Embed: CASE FIELD() structure, before generated code
OF ?Browse:1
  !Embed: Control Handling, before event handling ?Browse:1
  CASE EVENT()
  OF EVENT:Accepted
    !Embed: Control Event Handling, before generated code ?Browse:1, Accepted
    !Embed: Internal Control Event Handling ?Browse:1, Accepted
    !Embed: Control Event Handling, after generated code ?Browse:1, Accepted
  OF EVENT:NewSelection
    !Embed: Control Event Handling, before generated code ?Browse:1, NewSelection
    !Embed: Internal Control Event Handling ?Browse:1, NewSelection
    DO BRW1::NewSelection
    !Embed: Control Event Handling, after generated code ?Browse:1, NewSelection
  OF EVENT:ScrollUp
    !Embed: Control Event Handling, before generated code ?Browse:1, ScrollUp
    !Embed: Internal Control Event Handling ?Browse:1, ScrollUp
    DO BRW1::ProcessScroll
    !Embed: Control Event Handling, after generated code ?Browse:1, ScrollUp
  OF EVENT:ScrollDown
    !Embed: Control Event Handling, before generated code ?Browse:1, ScrollDown
    !Embed: Internal Control Event Handling ?Browse:1, ScrollDown
    DO BRW1::ProcessScroll
    !Embed: Control Event Handling, after generated code ?Browse:1, ScrollDown
  OF EVENT:PageUp
    !Embed: Control Event Handling, before generated code ?Browse:1, PageUp
    !Embed: Internal Control Event Handling ?Browse:1, PageUp
    DO BRW1::ProcessScroll
    !Embed: Control Event Handling, after generated code ?Browse:1, PageUp
  OF EVENT:PageDown
    !Embed: Control Event Handling, before generated code ?Browse:1, PageDown
    !Embed: Internal Control Event Handling ?Browse:1, PageDown
    DO BRW1::ProcessScroll
    !Embed: Control Event Handling, after generated code ?Browse:1, PageDown
  OF EVENT:ScrollTop
    !Embed: Control Event Handling, before generated code ?Browse:1, ScrollTop
    !Embed: Internal Control Event Handling ?Browse:1, ScrollTop
    DO BRW1::ProcessScroll
    !Embed: Control Event Handling, after generated code ?Browse:1, ScrollTop
  OF EVENT:ScrollBottom
    !Embed: Control Event Handling, before generated code ?Browse:1, ScrollBottom
    !Embed: Internal Control Event Handling ?Browse:1, ScrollBottom
    DO BRW1::ProcessScroll
    !Embed: Control Event Handling, after generated code ?Browse:1, ScrollBottom
  OF EVENT:Locate
    !Embed: Control Event Handling, before generated code ?Browse:1, Locate
    !Embed: Internal Control Event Handling ?Browse:1, Locate
    !Embed: Control Event Handling, after generated code ?Browse:1, Locate
  OF EVENT:AlertKey
    !Embed: Control Event Handling, before generated code ?Browse:1, AlertKey
    !Embed: Internal Control Event Handling ?Browse:1, AlertKey
```

```
    DO BRW1::AlertKey
    !Embed: Control Event Handling, after generated code ?Browse:1, AlertKey
  OF EVENT:PreAlertKey
    !Embed: Control Event Handling, before generated code ?Browse:1, PreAlertKey
    !Embed: Internal Control Event Handling ?Browse:1, PreAlertKey
    !Embed: Control Event Handling, after generated code ?Browse:1, PreAlertKey
  OF EVENT:ScrollDrag
    !Embed: Control Event Handling, before generated code ?Browse:1, ScrollDrag
    !Embed: Internal Control Event Handling ?Browse:1, ScrollDrag
    DO BRW1::ScrollDrag
    !Embed: Control Event Handling, after generated code ?Browse:1, ScrollDrag
  OF EVENT:Selected
    !Embed: Control Event Handling, before generated code ?Browse:1, Selected
    !Embed: Internal Control Event Handling ?Browse:1, Selected
    !Embed: Control Event Handling, after generated code ?Browse:1, Selected
  ELSE
    !Embed: Other Control Event Handling ?Browse:1
  END
  !Embed: Control Handling, after event handling ?Browse:1
OF ?Insert:2
  !Embed: Control Handling, before event handling ?Insert:2
  CASE EVENT()
  OF EVENT:Accepted
    !Embed: Control Event Handling, before generated code ?Insert:2, Accepted
    DO SyncWindow
    !Embed: Internal Control Event Handling ?Insert:2, Accepted
    DO BRW1::ButtonInsert
    !Embed: Control Event Handling, after generated code ?Insert:2, Accepted
  OF EVENT:Selected
    !Embed: Control Event Handling, before generated code ?Insert:2, Selected
    !Embed: Internal Control Event Handling ?Insert:2, Selected
    !Embed: Control Event Handling, after generated code ?Insert:2, Selected
  ELSE
    !Embed: Other Control Event Handling ?Insert:2
  END
  !Embed: Control Handling, after event handling ?Insert:2
OF ?Change:2
  !Embed: Control Handling, before event handling ?Change:2
  CASE EVENT()
  OF EVENT:Accepted
    !Embed: Control Event Handling, before generated code ?Change:2, Accepted
    DO SyncWindow
    !Embed: Internal Control Event Handling ?Change:2, Accepted
    DO BRW1::ButtonChange
    !Embed: Control Event Handling, after generated code ?Change:2, Accepted
  OF EVENT:Selected
    !Embed: Control Event Handling, before generated code ?Change:2, Selected
    !Embed: Internal Control Event Handling ?Change:2, Selected
    !Embed: Control Event Handling, after generated code ?Change:2, Selected
  ELSE
    !Embed: Other Control Event Handling ?Change:2
  END
  !Embed: Control Handling, after event handling ?Change:2
OF ?Delete:2
  !Embed: Control Handling, before event handling ?Delete:2
  CASE EVENT()
  OF EVENT:Accepted
    !Embed: Control Event Handling, before generated code ?Delete:2, Accepted
    DO SyncWindow
    !Embed: Internal Control Event Handling ?Delete:2, Accepted
```

```
    DO BRW1::ButtonDelete
    !Embed: Control Event Handling, after generated code ?Delete:2, Accepted
  OF EVENT:Selected
    !Embed: Control Event Handling, before generated code ?Delete:2, Selected
    !Embed: Internal Control Event Handling ?Delete:2, Selected
    !Embed: Control Event Handling, after generated code ?Delete:2, Selected
  ELSE
    !Embed: Other Control Event Handling ?Delete:2
  END
  !Embed: Control Handling, after event handling ?Delete:2
OF ?CurrentTab
  !Embed: Control Handling, before event handling ?CurrentTab
  CASE EVENT()
  OF EVENT:Accepted
    !Embed: Control Event Handling, before generated code ?CurrentTab, Accepted
    DO RefreshWindow
    !Embed: Internal Control Event Handling ?CurrentTab, Accepted
    !Embed: Control Event Handling, after generated code ?CurrentTab, Accepted
  OF EVENT:NewSelection
    !Embed: Control Event Handling, before generated code ?CurrentTab, NewSelection
    DO RefreshWindow
    !Embed: Internal Control Event Handling ?CurrentTab, NewSelection
    !Embed: Control Event Handling, after generated code ?CurrentTab, NewSelection
  OF EVENT:TabChanging
    !Embed: Control Event Handling, before generated code ?CurrentTab, TabChanging
    DO RefreshWindow
    !Embed: Internal Control Event Handling ?CurrentTab, TabChanging
    !Embed: Control Event Handling, after generated code ?CurrentTab, TabChanging
  OF EVENT:Selected
    !Embed: Control Event Handling, before generated code ?CurrentTab, Selected
    DO RefreshWindow
    !Embed: Internal Control Event Handling ?CurrentTab, Selected
    !Embed: Control Event Handling, after generated code ?CurrentTab, Selected
  ELSE
    !Embed: Other Control Event Handling ?CurrentTab
  END
  !Embed: Control Handling, after event handling ?CurrentTab
OF ?Close
  !Embed: Control Handling, before event handling ?Close
  CASE EVENT()
  OF EVENT:Accepted
    !Embed: Control Event Handling, before generated code ?Close, Accepted
    DO SyncWindow
    !Embed: Internal Control Event Handling ?Close, Accepted
    LocalResponse = RequestCancelled
    POST(Event:CloseWindow)
    !Embed: Control Event Handling, after generated code ?Close, Accepted
  OF EVENT:Selected
    !Embed: Control Event Handling, before generated code ?Close, Selected
    !Embed: Internal Control Event Handling ?Close, Selected
    !Embed: Control Event Handling, after generated code ?Close, Selected
  ELSE
    !Embed: Other Control Event Handling ?Close
  END
  !Embed: Control Handling, after event handling ?Close
OF ?Help
  !Embed: Control Handling, before event handling ?Help
  CASE EVENT()
  OF EVENT:Accepted
    !Embed: Control Event Handling, before generated code ?Help, Accepted
```

```
      DO SyncWindow
      !Embed: Internal Control Event Handling ?Help, Accepted
      !Embed: Control Event Handling, after generated code ?Help, Accepted
    OF EVENT:Selected
      !Embed: Control Event Handling, before generated code ?Help, Selected
      !Embed: Internal Control Event Handling ?Help, Selected
      !Embed: Control Event Handling, after generated code ?Help, Selected
    ELSE
      !Embed: Other Control Event Handling ?Help
    END
    !Embed: Control Handling, after event handling ?Help
  !Embed: CASE FIELD() structure, after generated code
  END
  !Embed: Accept Loop, After CASE FIELD() handling
 END
 DO ProcedureReturn
!---------------------------------------------------------------------------
ProcedureReturn ROUTINE
 !Embed: End of Procedure, Before Closing Files
 Names::Used -= 1
 IF Names::Used = 0 THEN CLOSE(Names).
 !Embed: End of Procedure, After Closing Files
 !Embed: Before Closing the Window
 IF WindowOpened
  CLOSE(QuickWindow)
 END
 !Embed: After Closing the Window
 !Embed: End of Procedure
 IF LocalResponse
  GlobalResponse = LocalResponse
 ELSE
  GlobalResponse = RequestCancelled
 END
 RETURN
!---------------------------------------------------------------------------
InitializeWindow ROUTINE
 !Embed: Window Initialization Code
 DO RefreshWindow
!---------------------------------------------------------------------------
RefreshWindow ROUTINE
 IF QuickWindow{Prop:AcceptAll} THEN EXIT.
 !Embed: Refresh Window routine, before lookups
 !Embed: Lookup Related Records
 !Embed: Refresh Window routine, after lookups
 !Embed: After Refresh Window for a control ?Browse:1
 DO BRW1::SelectSort
 !Embed: After Refresh Window for a control ?Browse:1
 !Embed: Refresh Window routine, before DISPLAY()
 ?Browse:1{Prop:VScrollPos} = BRW1::CurrentScroll
 DISPLAY()
 ForceRefresh = False
!---------------------------------------------------------------------------
SyncWindow ROUTINE
 !Embed: Sync Record routine, before lookups
 !Embed: Lookup Related Records
 !Embed: Sync Record routine, after lookups
 DO BRW1::GetRecord
!---------------------------------------------------------------------------
!Embed: Procedure Routines
!---------------------------------------------------------------------
```

```
BRW1::ValidateRecord ROUTINE
  !Embed: Start of Validate Record ROUTINE
  BRW1::RecordStatus = Record:OutOfRange
  !Embed: Validate Record: Range Checking 1
  BRW1::RecordStatus = Record:Filtered
  !Embed: Validate Record: Filter Checking 1
  !Embed: After Range and Filter Check 1
  BRW1::RecordStatus = Record:OK
  !Embed: End of Validate Record ROUTINE
  EXIT
!---------------------------------------------------------------------
BRW1::SelectSort ROUTINE
  BRW1::LastSortOrder = BRW1::SortOrder
  BRW1::Changed = False
  IF CHOICE(?CurrentTab) = 2
    BRW1::SortOrder = 1
  ELSE
    BRW1::SortOrder = 2
  END
  IF BRW1::SortOrder <> BRW1::LastSortOrder OR BRW1::Changed OR ForceRefresh
    DO BRW1::GetRecord
    DO BRW1::Reset
    IF BRW1::LastSortOrder = 0
      IF LocalRequest = SelectRecord
        BRW1::LocateMode = LocateOnValue
        DO BRW1::LocateRecord
      ELSE
        BRW1::RefreshMode = RefreshOnTop
        DO BRW1::RefreshPage
        DO BRW1::PostNewSelection
      END
    ELSE
      IF BRW1::Changed
        BRW1::RefreshMode = RefreshOnTop
        DO BRW1::RefreshPage
        DO BRW1::PostNewSelection
      ELSE
        BRW1::LocateMode = LocateOnValue
        DO BRW1::LocateRecord
      END
    END
    IF BRW1::RecordCount
      GET(Queue:Browse:1,BRW1::CurrentChoice)
      DO BRW1::FillBuffer
    END
    DO BRW1::InitializeBrowse
  END
!---------------------------------------------------------------------
BRW1::InitializeBrowse ROUTINE
  !Embed: Start of Initialize Browse ROUTINE
  SETCURSOR(Cursor:Wait)
  DO BRW1::Reset
  !Embed: Before Browse Total Loop 1
  LOOP
    NEXT(BRW1::View:Browse)
    IF ERRORCODE()
      IF ERRORCODE() = BadRecErr
        BREAK
      ELSE
        StandardWarning(Warn:RecordFetchError,'Names')
```

```
      POST(Event:CloseWindow)
      EXIT
    END
  END
  DO BRW1::ValidateRecord
  EXECUTE(BRW1::RecordStatus)
    BREAK
    CYCLE
  END
  DO BRW1::FillQueue
  !Embed: Browse Total Loop 1
END
!Embed: After Browse Total Loop 1
SETCURSOR()
DO BRW1::Reset
LOOP
  PREVIOUS(BRW1::View:Browse)
  IF ERRORCODE()
    IF ERRORCODE() = BadRecErr
      BREAK
    ELSE
      StandardWarning(Warn:RecordFetchError,'Names')
      POST(Event:CloseWindow)
      EXIT
    END
  END
  DO BRW1::ValidateRecord
  EXECUTE(BRW1::RecordStatus)
    BREAK
    CYCLE
  END
  BREAK
END
CASE BRW1::SortOrder
OF 1
  BRW1::Sort1:HighValue = NAM:Zip
OF 2
  BRW1::Sort2:HighValue = NAM:Number
END
DO BRW1::Reset
LOOP
  NEXT(BRW1::View:Browse)
  IF ERRORCODE()
    IF ERRORCODE() = BadRecErr
      BREAK
    ELSE
      StandardWarning(Warn:RecordFetchError,'Names')
      POST(Event:CloseWindow)
      EXIT
    END
  END
  DO BRW1::ValidateRecord
  EXECUTE(BRW1::RecordStatus)
    BREAK
    CYCLE
  END
  BREAK
END
CASE BRW1::SortOrder
OF 1
```

```
    BRW1::Sort1:LowValue = NAM:Zip
    SetupRealStops(BRW1::Sort1:LowValue,BRW1::Sort1:HighValue)
    LOOP BRW1::ScrollRecordCount = 1 TO 100
      BRW1::Sort1:KeyDistribution[BRW1::ScrollRecordCount] = NextRealStop()
    END
  OF 2
    BRW1::Sort2:LowValue = NAM:Number
    SetupRealStops(BRW1::Sort2:LowValue,BRW1::Sort2:HighValue)
    LOOP BRW1::ScrollRecordCount = 1 TO 100
      BRW1::Sort2:KeyDistribution[BRW1::ScrollRecordCount] = NextRealStop()
    END
  END
  !Embed: End of Initialize Browse ROUTINE
!---------------------------------------------------------------------------
BRW1::FillBuffer ROUTINE
  !Embed: Start of Fill Buffer ROUTINE
  NAM:Number = BRW1::NAM:Number
  NAM:FirstName = BRW1::NAM:FirstName
  NAM:LastName = BRW1::NAM:LastName
  NAM:Address = BRW1::NAM:Address
  NAM:City = BRW1::NAM:City
  NAM:State = BRW1::NAM:State
  NAM:Zip = BRW1::NAM:Zip
  !Embed: Start of Fill Buffer ROUTINE
!---------------------------------------------------------------------------
BRW1::FillQueue ROUTINE
  !Embed: Format an element of the browse queue 1
  BRW1::NAM:Number = NAM:Number
  BRW1::NAM:FirstName = NAM:FirstName
  BRW1::NAM:LastName = NAM:LastName
  BRW1::NAM:Address = NAM:Address
  BRW1::NAM:City = NAM:City
  BRW1::NAM:State = NAM:State
  BRW1::NAM:Zip = NAM:Zip
  BRW1::Position = POSITION(BRW1::View:Browse)
  !Embed: End of Format an element of the browse queue 1
!---------------------------------------------------------------------------
BRW1::PostNewSelection ROUTINE
  IF NOT BRW1::NewSelectPosted
    BRW1::NewSelectPosted = True
    POST(Event:NewSelection,?Browse:1)
  END
!---------------------------------------------------------------------------
BRW1::NewSelection ROUTINE
  BRW1::NewSelectPosted = False
  IF KEYCODE() = MouseRight
    BRW1::PopupText = ''
    IF BRW1::RecordCount
      !Embed: INTERNAL Browse Box Popup with Records
      IF BRW1::PopupText
        BRW1::PopupText = '&Insert|&Change|&Delete|-|' & BRW1::PopupText
      ELSE
        BRW1::PopupText = '&Insert|&Change|&Delete'
      END
    ELSE
      !Embed: INTERNAL Browse Box Popup with No Records
      IF BRW1::PopupText
        BRW1::PopupText = '&Insert|~&Change|~&Delete|-|' & BRW1::PopupText
      ELSE
        BRW1::PopupText = '&Insert|~&Change|~&Delete'
```

```
      END
    END
    EXECUTE(POPUP(BRW1::PopupText))
      !Embed: INTERNAL Browse Box Popup Handling
      POST(Event:Accepted,?Insert:2)
      POST(Event:Accepted,?Change:2)
      POST(Event:Accepted,?Delete:2)
      !Embed: INTERNAL Browse Box Popup Handling
    END
  ELSIF BRW1::RecordCount
    BRW1::CurrentChoice = CHOICE(?Browse:1)
    GET(Queue:Browse:1,BRW1::CurrentChoice)
    DO BRW1::FillBuffer
    IF BRW1::RecordCount = ?Browse:1{Prop:Items}
      IF ?Browse:1{Prop:VScroll} = False
        ?Browse:1{Prop:VScroll} = True
      END
      CASE BRW1::SortOrder
      OF 1
        LOOP BRW1::CurrentScroll = 1 TO 100
          IF BRW1::Sort1:KeyDistribution[BRW1::CurrentScroll] => NAM:Zip
            IF BRW1::CurrentScroll <= 1
              BRW1::CurrentScroll = 0
            ELSIF BRW1::CurrentScroll = 100
              BRW1::CurrentScroll = 100
            ELSE
            END
            BREAK
          END
        END
      OF 2
        LOOP BRW1::CurrentScroll = 1 TO 100
          IF BRW1::Sort2:KeyDistribution[BRW1::CurrentScroll] => NAM:Number
            IF BRW1::CurrentScroll <= 1
              BRW1::CurrentScroll = 0
            ELSIF BRW1::CurrentScroll = 100
              BRW1::CurrentScroll = 100
            ELSE
            END
            BREAK
          END
        END
      END
    ELSE
      IF ?Browse:1{Prop:VScroll} = True
        ?Browse:1{Prop:VScroll} = False
      END
    END
    DO RefreshWindow
  END
!---------------------------------------------------------------------------
BRW1::ProcessScroll ROUTINE
  IF BRW1::RecordCount
    BRW1::CurrentEvent = EVENT()
    CASE BRW1::CurrentEvent
    OF Event:ScrollUp OROF Event:ScrollDown
      DO BRW1::ScrollOne
    OF Event:PageUp OROF Event:PageDown
      DO BRW1::ScrollPage
    OF Event:ScrollTop OROF Event:ScrollBottom
```

```
      DO BRW1::ScrollEnd
    END
    ?Browse:1{Prop:SelStart} = BRW1::CurrentChoice
    DO BRW1::PostNewSelection
  END
!---------------------------------------------------------------------------
BRW1::ScrollOne ROUTINE
  IF BRW1::CurrentEvent = Event:ScrollUp
    !Embed: INTERNAL: Start of Scroll Up ROUTINE
  ELSE
    !Embed: INTERNAL: Start of Scroll Down ROUTINE
  END
  IF BRW1::CurrentEvent = Event:ScrollUp AND BRW1::CurrentChoice > 1
    BRW1::CurrentChoice -= 1
    EXIT
  ELSIF BRW1::CurrentEvent = Event:ScrollDown AND BRW1::CurrentChoice <
BRW1::RecordCount
    BRW1::CurrentChoice += 1
    EXIT
  END
  BRW1::ItemsToFill = 1
  BRW1::FillDirection = BRW1::CurrentEvent - 2
  DO BRW1::FillRecord
  IF BRW1::CurrentEvent = Event:ScrollUp
    !Embed: INTERNAL: End of Scroll Up ROUTINE
  ELSE
    !Embed: INTERNAL: End of Scroll Down ROUTINE
  END
!---------------------------------------------------------------------------
BRW1::ScrollPage ROUTINE
  IF BRW1::CurrentEvent = Event:PageUp
    !Embed: INTERNAL: Start of Page Up ROUTINE
  ELSE
    !Embed: INTERNAL: Start of Page Down ROUTINE
  END
  BRW1::ItemsToFill = ?Browse:1{Prop:Items}
  BRW1::FillDirection = BRW1::CurrentEvent - 4
  DO BRW1::FillRecord                       ! Fill with next read(s)
  IF BRW1::ItemsToFill
    IF BRW1::CurrentEvent = Event:PageUp
      BRW1::CurrentChoice -= BRW1::ItemsToFill
      IF BRW1::CurrentChoice < 1
        BRW1::CurrentChoice = 1
      END
    ELSE
      BRW1::CurrentChoice += BRW1::ItemsToFill
      IF BRW1::CurrentChoice > BRW1::RecordCount
        BRW1::CurrentChoice = BRW1::RecordCount
      END
    END
  END
  IF BRW1::CurrentEvent = Event:PageUp
    !Embed: INTERNAL: End of Page Up ROUTINE
  ELSE
    !Embed: INTERNAL: End of Page Down ROUTINE
  END
!---------------------------------------------------------------------------
BRW1::ScrollEnd ROUTINE
  IF BRW1::CurrentEvent = Event:ScrollTop
    !Embed: INTERNAL: Start of Scroll Top ROUTINE
```

```
  ELSE
    !Embed: INTERNAL: Start of Scroll Bottom ROUTINE
  END
  FREE(Queue:Browse:1)
  BRW1::RecordCount = 0
  DO BRW1::Reset
  BRW1::ItemsToFill = ?Browse:1{Prop:Items}
  IF BRW1::CurrentEvent = Event:ScrollTop
    BRW1::FillDirection = FillForward
  ELSE
    BRW1::FillDirection = FillBackward
  END
  DO BRW1::FillRecord                       ! Fill with next read(s)
  IF BRW1::CurrentEvent = Event:ScrollTop
    BRW1::CurrentChoice = 1
  ELSE
    BRW1::CurrentChoice = BRW1::RecordCount
  END
  IF BRW1::CurrentEvent = Event:ScrollTop
    !Embed: INTERNAL: End of Scroll Top ROUTINE
  ELSE
    !Embed: INTERNAL: End of Scroll Bottom ROUTINE
  END
!---------------------------------------------------------------------------
BRW1::AlertKey ROUTINE
  !Embed: INTERNAL: Start of Alert Key ROUTINE
  IF BRW1::RecordCount
    CASE KEYCODE()                            ! What keycode was hit
    !Embed: AlertKey routine, inside CASE KEYCODE 1
    OF MouseLeft2
      !Embed: Browse Double Click Handler 1
      !Embed: INTERNAL Browse Box Double Click Handler
      POST(Event:Accepted,?Change:2)
      DO BRW1::FillBuffer
    !Embed: Browse Key Handling
    OF InsertKey
      POST(Event:Accepted,?Insert:2)
    !Embed: Browse Key Handling
    OF DeleteKey
      POST(Event:Accepted,?Delete:2)
    OF CtrlEnter
      POST(Event:Accepted,?Change:2)
    END                                     ! END (What keycode was hit)
  ELSE
    CASE KEYCODE()                            ! What keycode was hit
    !Embed: Browse Key Handling
    OF InsertKey
      POST(Event:Accepted,?Insert:2)
    END
  END
  DO BRW1::PostNewSelection
  !Embed: INTERNAL: End of Alert Key ROUTINE
!---------------------------------------------------------------------------
BRW1::ScrollDrag ROUTINE
  IF ?Browse:1{Prop:VScrollPos} <= 1
    POST(Event:ScrollTop,?Browse:1)
  ELSIF ?Browse:1{Prop:VScrollPos} = 100
    POST(Event:ScrollBottom,?Browse:1)
  ELSE
    CASE BRW1::SortOrder
```

```
    OF 1
      NAM:Zip = BRW1::Sort1:KeyDistribution[?Browse:1{Prop:VScrollPos}]
      BRW1::LocateMode = LocateOnValue
      DO BRW1::LocateRecord
    OF 2
      NAM:Number = BRW1::Sort2:KeyDistribution[?Browse:1{Prop:VScrollPos}]
      BRW1::LocateMode = LocateOnValue
      DO BRW1::LocateRecord
    END
  END
!---------------------------------------------------------------------
BRW1::FillRecord ROUTINE
  IF BRW1::FillDirection = FillForward
    !Embed: Start of Fill Forward ROUTINE
    !Embed: Start of Fill Record ROUTINE, Reading Forward
  ELSE
    !Embed: Start of Fill Backward ROUTINE
    !Embed: Start of Fill Record ROUTINE, Reading Backward
  END
  IF BRW1::RecordCount
    IF BRW1::FillDirection = FillForward
      GET(Queue:Browse:1,BRW1::RecordCount)     ! Get the first queue item
    ELSE
      GET(Queue:Browse:1,1)                   ! Get the first queue item
    END
    RESET(BRW1::View:Browse,BRW1::Position)     ! Reset for sequential processing
    NEXT(BRW1::View:Browse)
  END
  LOOP WHILE BRW1::ItemsToFill
    IF BRW1::FillDirection = FillForward
      NEXT(BRW1::View:Browse)
    ELSE
      PREVIOUS(BRW1::View:Browse)
    END
    IF ERRORCODE()
      IF ERRORCODE() = BadRecErr
        BREAK
      ELSE
        StandardWarning(Warn:RecordFetchError,'Names')
        POST(Event:CloseWindow)
        EXIT
      END
    END
    DO BRW1::ValidateRecord
    EXECUTE(BRW1::RecordStatus)
      BEGIN
        IF BRW1::FillDirection = FillForward
          GET(Queue:Browse:1,BRW1::RecordCount)  ! Get the first queue item
        ELSE
          GET(Queue:Browse:1,1)               ! Get the first queue item
        END
        DO BRW1::FillBuffer
        BREAK
      END
      CYCLE
    END
    IF BRW1::AddQueue
      IF BRW1::RecordCount = ?Browse:1{Prop:Items}
        IF BRW1::FillDirection = FillForward
          GET(Queue:Browse:1,1)               ! Get the first queue item
```

```
      ELSE
        GET(Queue:Browse:1,BRW1::RecordCount)  ! Get the first queue item
      END
      DELETE(Queue:Browse:1)
      BRW1::RecordCount -= 1
    END
    DO BRW1::FillQueue
    IF BRW1::FillDirection = FillForward
      ADD(Queue:Browse:1)
    ELSE
      ADD(Queue:Browse:1,1)
    END
    BRW1::RecordCount += 1
  END
  BRW1::ItemsToFill -= 1
 END
 BRW1::AddQueue = True
 IF BRW1::FillDirection = FillForward
   !Embed: End of Fill Forward ROUTINE
   !Embed: End of Fill Record ROUTINE, Reading Forward
 ELSE
   !Embed: End of Fill Backward ROUTINE
   !Embed: End of Fill Record ROUTINE, Reading Backward
 END
 EXIT
!---------------------------------------------------------------------------
BRW1::LocateRecord ROUTINE
  !Embed: Start of Locate Record ROUTINE
  IF BRW1::LocateMode = LocateOnPosition
    BRW1::HighlightedPosition = POSITION(BRW1::View:Browse)
    RESET(BRW1::View:Browse,BRW1::HighlightedPosition)
  ELSE
    CLOSE(BRW1::View:Browse)
    CASE BRW1::SortOrder
    OF 1
      IF BRW1::LocateMode = LocateOnValue
        SET(NAM:KeyZip,NAM:KeyZip)
      ELSE
        SET(NAM:KeyZip)
      END
      BRW1::View:Browse{Prop:Filter} = ''
    OF 2
      IF BRW1::LocateMode = LocateOnValue
        SET(NAM:KeyNumber,NAM:KeyNumber)
      ELSE
        SET(NAM:KeyNumber)
      END
      BRW1::View:Browse{Prop:Filter} = ''
    END
    OPEN(BRW1::View:Browse)
  END
  FREE(Queue:Browse:1)
  BRW1::RecordCount = 0
  BRW1::ItemsToFill = 1
  BRW1::FillDirection = FillForward          ! Fill with next read(s)
  BRW1::AddQueue = False
  DO BRW1::FillRecord                       ! Fill with next read(s)
  BRW1::AddQueue = True
  IF BRW1::ItemsToFill
    BRW1::RefreshMode = RefreshOnBottom
```

```
    DO BRW1::RefreshPage
  ELSE
    BRW1::RefreshMode = RefreshOnPosition
    DO BRW1::RefreshPage
  END
  DO BRW1::PostNewSelection
  BRW1::LocateMode = 0
  EXIT
!------------------------------------------------------------------------
BRW1::RefreshPage ROUTINE
  !Embed: Start of Refresh Page ROUTINE
  SETCURSOR(Cursor:Wait)
  IF BRW1::RefreshMode = RefreshOnPosition
    BRW1::HighlightedPosition = POSITION(BRW1::View:Browse)
    RESET(BRW1::View:Browse,BRW1::HighlightedPosition)
    BRW1::RefreshMode = RefreshOnTop
  ELSIF RECORDS(Queue:Browse:1)
    GET(Queue:Browse:1,BRW1::CurrentChoice)
    IF ERRORCODE()
      GET(Queue:Browse:1,RECORDS(Queue:Browse:1))
    END
    BRW1::HighlightedPosition = BRW1::Position
    GET(Queue:Browse:1,1)
    RESET(BRW1::View:Browse,BRW1::Position)
    BRW1::RefreshMode = RefreshOnCurrent
  ELSE
    BRW1::HighlightedPosition = ''
    DO BRW1::Reset
  END
  FREE(Queue:Browse:1)
  BRW1::RecordCount = 0
  BRW1::ItemsToFill = ?Browse:1{Prop:Items}
  IF BRW1::RefreshMode = RefreshOnBottom
    BRW1::FillDirection = FillBackward
  ELSE
    BRW1::FillDirection = FillForward
  END
  DO BRW1::FillRecord                        ! Fill with next read(s)
  IF BRW1::HighlightedPosition
    IF BRW1::ItemsToFill
      IF NOT BRW1::RecordCount
        DO BRW1::Reset
      END
      IF BRW1::RefreshMode = RefreshOnBottom
        BRW1::FillDirection = FillForward
      ELSE
        BRW1::FillDirection = FillBackward
      END
      DO BRW1::FillRecord
    END
  END
  IF BRW1::RecordCount
    IF BRW1::HighlightedPosition
      LOOP BRW1::CurrentChoice = 1 TO BRW1::RecordCount
        GET(Queue:Browse:1,BRW1::CurrentChoice)
        IF BRW1::Position = BRW1::HighlightedPosition THEN BREAK.
      END
    ELSE
      IF BRW1::RefreshMode = RefreshOnBottom
        BRW1::CurrentChoice = RECORDS(Queue:Browse:1)
```

```
      ELSE
        BRW1::CurrentChoice = 1
      END
    END
    ?Browse:1{Prop:Selected} = BRW1::CurrentChoice
    DO BRW1::FillBuffer
    !Embed: Browse Box, Records Found 1
    ?Change:2{Prop:Disable} = 0
    ?Delete:2{Prop:Disable} = 0
  ELSE
    CLEAR(NAM:Record)
    BRW1::CurrentChoice = 0
    !Embed: Browse Box, No Records Found 1
    ?Change:2{Prop:Disable} = 1
    ?Delete:2{Prop:Disable} = 1
  END
  SETCURSOR()
  !Embed: End of Refresh Page ROUTINE
  BRW1::RefreshMode = 0
  EXIT
BRW1::Reset ROUTINE
  CLOSE(BRW1::View:Browse)
  CASE BRW1::SortOrder
  OF 1
    SET(NAM:KeyZip)
    BRW1::View:Browse{Prop:Filter} = ''
  OF 2
    SET(NAM:KeyNumber)
    BRW1::View:Browse{Prop:Filter} = ''
  END
  OPEN(BRW1::View:Browse)
!-----------------------------------------------------------------------
!-----------------------------------------------------------------------
BRW1::GetRecord ROUTINE
  IF BRW1::RecordCount
    BRW1::CurrentChoice = CHOICE(?Browse:1)
    GET(Queue:Browse:1,BRW1::CurrentChoice)
    WATCH(BRW1::View:Browse)
    REGET(BRW1::View:Browse,BRW1::Position)
  END
!---------------------------------------------------------------------
BRW1::ButtonInsert ROUTINE
  GET(Names,0)
  CLEAR(NAM:Record,0)
  LocalRequest = InsertRecord
  !Embed: Browse Box, Before Insert 2
  DO BRW1::CallUpdate
  !Embed: Browse Box, After Insert 2
  IF GlobalResponse = RequestCompleted
    BRW1::LocateMode = LocateOnValue
    DO BRW1::LocateRecord
  ELSE
    BRW1::RefreshMode = RefreshOnQueue
    DO BRW1::RefreshPage
  END
  DO BRW1::InitializeBrowse
  DO BRW1::PostNewSelection
  SELECT(?Browse:1)
  LocalRequest = OriginalRequest
  DO RefreshWindow
```

```
!---------------------------------------------------------------
BRW1::ButtonChange ROUTINE
  LocalRequest = ChangeRecord
  !Embed: Browse Box, Before Change 2
  DO BRW1::CallUpdate
  !Embed: Browse Box, After Change 2
  IF GlobalResponse = RequestCompleted
    BRW1::LocateMode = LocateOnValue
    DO BRW1::LocateRecord
  ELSE
    BRW1::RefreshMode = RefreshOnQueue
    DO BRW1::RefreshPage
  END
  DO BRW1::InitializeBrowse
  DO BRW1::PostNewSelection
  SELECT(?Browse:1)
  LocalRequest = OriginalRequest
  DO RefreshWindow
!---------------------------------------------------------------
BRW1::ButtonDelete ROUTINE
  LocalRequest = DeleteRecord
  !Embed: Browse Box, Before Delete 2
  DO BRW1::CallUpdate
  !Embed: Browse Box, After Delete 2
  DELETE(Queue:Browse:1)
  BRW1::RecordCount -= 1
  BRW1::RefreshMode = RefreshOnQueue
  DO BRW1::RefreshPage
  DO BRW1::InitializeBrowse
  DO BRW1::PostNewSelection
  SELECT(?Browse:1)
  LocalRequest = OriginalRequest
  DO RefreshWindow
!---------------------------------------------------------------
BRW1::CallUpdate ROUTINE
  !Embed: Browse Box, before calling the update procedure 2
  CLOSE(BRW1::View:Browse)
  GlobalRequest = LocalRequest
  UpdateNames
  LocalResponse = GlobalResponse
  DO BRW1::Reset
  !Embed: Browse Box, returning from the update procedure 2
```

Example Source for Form Procedure

The following code was generated from the Form procedure template.

Comments appear in **red** at the default embed points.

```
                  MEMBER('Prog.clw')        ! This is a MEMBER module
!Embed: Compile Module Declarations
!Embed: Compile Module Declarations
!Embed: Module Data Section
!Embed: Module Data Section
!Embed: Gather Template Symbols
UpdateNames PROCEDURE

CurrentTab          STRING(80)
LocalRequest        LONG,AUTO
OriginalRequest       LONG,AUTO
LocalResponse       LONG,AUTO
WindowOpened        LONG
WindowInitialized     LONG
ForceRefresh        LONG,AUTO
ActionMessage       CSTRING(40)
RecordChanged       BYTE,AUTO
!Embed: Data Section, Before Window Declaration
SAV::NAM:Record       STRING(SIZE(NAM:Record))
QuickWindow         WINDOW('Update the Names File'),AT(,,151,140),FONT('MS Sans
Serif',8,,),IMM,HLP('UpdateNames'),SYSTEM,GRAY,MDI
                    SHEET,AT(4,4,143,114),USE(?CurrentTab)
                      TAB('General')
                        PROMPT('&Number:'),AT(8,20),USE(?NAM:Number:Prompt)
                        ENTRY(@n03),AT(61,20,40,10),USE(NAM:Number)
                        PROMPT('&First Name:'),AT(8,34),USE(?NAM:FirstName:Prompt)
                        ENTRY(@S20),AT(61,34,80,10),USE(NAM:FirstName)
                        PROMPT('&Last Name:'),AT(8,48),USE(?NAM:LastName:Prompt)
                        ENTRY(@S20),AT(61,48,80,10),USE(NAM:LastName)
                        PROMPT('&Address:'),AT(8,62),USE(?NAM:Address:Prompt)
                        ENTRY(@S20),AT(61,62,80,10),USE(NAM:Address)
                        PROMPT('&City:'),AT(8,76),USE(?NAM:City:Prompt)
                        ENTRY(@S20),AT(61,76,80,10),USE(NAM:City)
                        PROMPT('&State:'),AT(8,90),USE(?NAM:State:Prompt)
                        ENTRY(@s2),AT(61,90,40,10),USE(NAM:State)
                        PROMPT('&Zip:'),AT(8,104),USE(?NAM:Zip:Prompt)
                        ENTRY(@n05),AT(61,104,40,10),USE(NAM:Zip)
                      END
                    END
                    BUTTON('OK'),AT(4,122,45,14),USE(?OK),DEFAULT
                    BUTTON('Cancel'),AT(53,122,45,14),USE(?Cancel)
                    BUTTON('Help'),AT(102,122,45,14),USE(?Help),STD(STD:Help)
                  END
!Embed: Data Section, After Window Declaration
  CODE
  !Embed: Initialize the Procedure
  LocalRequest = GlobalRequest
  OriginalRequest = GlobalRequest
  LocalResponse = RequestCancelled
  ForceRefresh = False
  CLEAR(GlobalRequest)
  CLEAR(GlobalResponse)
  !Embed: Procedure Setup
```

```
IF KEYCODE() = MouseRight
  SETKEYCODE(0)
END
!Embed: Beginning of Procedure, Before Opening Files
IF Names::Used = 0
  CheckOpen(Names,1)
  BIND(NAM:RECORD)
END
Names::Used += 1
!Embed: Beginning of Procedure, After Opening Files
RISnap:Names
SAV::NAM:Record = NAM:Record
IF LocalRequest = InsertRecord
  !Embed: On Insert, before record is primed
  DO PrimeFields
  !Embed: On Insert, after record is primed
END
IF LocalRequest = DeleteRecord
  IF StandardWarning(Warn:StandardDelete) = Button:OK
    LOOP
      LocalResponse = RequestCancelled
      SETCURSOR(Cursor:Wait)
      IF RIDelete:Names()
        SETCURSOR()
        CASE StandardWarning(Warn:DeleteError)
        OF Button:Yes
          CYCLE
        OF Button:No OROF Button:Cancel
          BREAK
        END
      ELSE
        SETCURSOR()
        LocalResponse = RequestCompleted
      END
      BREAK
    END
  END
  DO ProcedureReturn
END
!Embed: Before Opening the Window
OPEN(QuickWindow)
WindowOpened=True
!Embed: After Opening the Window
!Embed: Preparing Window Alerts
!Embed: Preparing to Process the Window
CASE LocalRequest
OF InsertRecord
  ActionMessage = 'Adding a Names Record'
OF ChangeRecord
  ActionMessage = 'Changing a Names Record'
OF DeleteRecord
END
QuickWindow{Prop:Text} = ActionMessage
ACCEPT
  !Embed: Accept Loop, Before CASE EVENT() handling
  CASE EVENT()
  !Embed: CASE EVENT() structure, before generated code
  OF EVENT:AlertKey
    !Embed: Window Event Handling AlertKey
  OF EVENT:PreAlertKey
```

```
  !Embed: Window Event Handling PreAlertKey
OF EVENT:CloseWindow
  !Embed: Window Event Handling CloseWindow
  IF LocalResponse <> RequestCompleted
  END
OF EVENT:CloseDown
  !Embed: Window Event Handling CloseDown
  IF LocalResponse <> RequestCompleted
  END
OF EVENT:OpenWindow
  !Embed: Window Event Handling OpenWindow
  IF NOT WindowInitialized
    DO InitializeWindow
    WindowInitialized = True
  END
  SELECT(?NAM:Number:Prompt)
OF EVENT:LoseFocus
  !Embed: Window Event Handling LoseFocus
OF EVENT:GainFocus
  !Embed: Window Event Handling GainFocus
  ForceRefresh = True
  IF NOT WindowInitialized
    DO InitializeWindow
    WindowInitialized = True
  ELSE
    DO RefreshWindow
  END
OF EVENT:Suspend
  !Embed: Window Event Handling Suspend
OF EVENT:Resume
  !Embed: Window Event Handling Resume
OF EVENT:Timer
  !Embed: Window Event Handling Timer
OF EVENT:Move
  !Embed: Window Event Handling Move
OF EVENT:Size
  !Embed: Window Event Handling Size
OF EVENT:Restore
  !Embed: Window Event Handling Restore
OF EVENT:Maximize
  !Embed: Window Event Handling Maximize
OF EVENT:Iconize
  !Embed: Window Event Handling Iconize
OF EVENT:Moved
  !Embed: Window Event Handling Moved
OF EVENT:Sized
  !Embed: Window Event Handling Sized
OF EVENT:Restored
  !Embed: Window Event Handling Restored
OF EVENT:Maximized
  !Embed: Window Event Handling Maximized
OF EVENT:Iconized
  !Embed: Window Event Handling Iconized
ELSE
  !Embed: Other Window Event Handling
  IF EVENT() = Event:Completed
    !Embed: When completed, before writing to disk
    CASE LocalRequest
    OF InsertRecord
      ADD(Names)
```

```
      CASE ERRORCODE()
      OF NoError
        LocalResponse = RequestCompleted
        POST(Event:CloseWindow)
      OF DupKeyErr
        IF DUPLICATE(NAM:KeyNumber)
          IF StandardWarning(Warn:DuplicateKey,'NAM:KeyNumber')
            SELECT(?NAM:Number:Prompt)
            CYCLE
          END
        END
      ELSE
        IF StandardWarning(Warn:InsertError)
          SELECT(?NAM:Number:Prompt)
          CYCLE
        END
      END
    OF ChangeRecord
      LOOP
        LocalResponse = RequestCancelled
        SETCURSOR(Cursor:Wait)
        IF RIUpdate:Names()
          SETCURSOR()
          CASE StandardWarning(Warn:UpdateError)
          OF Button:Yes
            CYCLE
          OF Button:No
            POST(Event:CloseWindow)
            BREAK
          OF Button:Cancel
            DISPLAY
            SELECT(?NAM:Number:Prompt)
            BREAK
          END
        ELSE
          SETCURSOR()
          LocalResponse = RequestCompleted
          POST(Event:CloseWindow)
        END
        BREAK
      END
    END
  END
!Embed: CASE EVENT() structure, after generated code
END
!Embed: Accept Loop, After CASE EVENT() handling
!Embed: Accept Loop, Before CASE FIELD() handling
CASE FIELD()
!Embed: CASE FIELD() structure, before generated code
OF ?CurrentTab
  !Embed: Control Handling, before event handling ?CurrentTab
  CASE EVENT()
  OF EVENT:Accepted
    !Embed: Control Event Handling, before generated code ?CurrentTab, Accepted
    DO RefreshWindow
    !Embed: Internal Control Event Handling ?CurrentTab, Accepted
    !Embed: Control Event Handling, after generated code ?CurrentTab, Accepted
  OF EVENT:NewSelection
    !Embed: Control Event Handling, before generated code ?CurrentTab, NewSelection
    DO RefreshWindow
```

```
    !Embed: Internal Control Event Handling ?CurrentTab, NewSelection
    !Embed: Control Event Handling, after generated code ?CurrentTab, NewSelection
  OF EVENT:TabChanging
    !Embed: Control Event Handling, before generated code ?CurrentTab, TabChanging
    DO RefreshWindow
    !Embed: Internal Control Event Handling ?CurrentTab, TabChanging
    !Embed: Control Event Handling, after generated code ?CurrentTab, TabChanging
  OF EVENT:Selected
    !Embed: Control Event Handling, before generated code ?CurrentTab, Selected
    DO RefreshWindow
    !Embed: Internal Control Event Handling ?CurrentTab, Selected
    !Embed: Control Event Handling, after generated code ?CurrentTab, Selected
  ELSE
    !Embed: Other Control Event Handling ?CurrentTab
  END
  !Embed: Control Handling, after event handling ?CurrentTab
OF ?NAM:Number:Prompt
  !Embed: Control Handling, before event handling ?NAM:Number:Prompt
  CASE EVENT()
  ELSE
    !Embed: Other Control Event Handling ?NAM:Number:Prompt
  END
  !Embed: Control Handling, after event handling ?NAM:Number:Prompt
OF ?NAM:Number
  !Embed: Control Handling, before event handling ?NAM:Number
  CASE EVENT()
  OF EVENT:Accepted
    !Embed: Control Event Handling, before generated code ?NAM:Number, Accepted
    !Embed: Internal Control Event Handling ?NAM:Number, Accepted
    !Embed: Control Event Handling, after generated code ?NAM:Number, Accepted
  OF EVENT:Rejected
    !Embed: Control Event Handling, before generated code ?NAM:Number, Rejected
    !Embed: Internal Control Event Handling ?NAM:Number, Rejected
    !Embed: Control Event Handling, after generated code ?NAM:Number, Rejected
  OF EVENT:Selected
    !Embed: Control Event Handling, before generated code ?NAM:Number, Selected
    !Embed: Internal Control Event Handling ?NAM:Number, Selected
    !Embed: Control Event Handling, after generated code ?NAM:Number, Selected
  ELSE
    !Embed: Other Control Event Handling ?NAM:Number
  END
  !Embed: Control Handling, after event handling ?NAM:Number
OF ?NAM:FirstName:Prompt
  !Embed: Control Handling, before event handling ?NAM:FirstName:Prompt
  CASE EVENT()
  ELSE
    !Embed: Other Control Event Handling ?NAM:FirstName:Prompt
  END
  !Embed: Control Handling, after event handling ?NAM:FirstName:Prompt
OF ?NAM:FirstName
  !Embed: Control Handling, before event handling ?NAM:FirstName
  CASE EVENT()
  OF EVENT:Accepted
    !Embed: Control Event Handling, before generated code ?NAM:FirstName, Accepted
    !Embed: Internal Control Event Handling ?NAM:FirstName, Accepted
    !Embed: Control Event Handling, after generated code ?NAM:FirstName, Accepted
  OF EVENT:Rejected
    !Embed: Control Event Handling, before generated code ?NAM:FirstName, Rejected
    !Embed: Internal Control Event Handling ?NAM:FirstName, Rejected
    !Embed: Control Event Handling, after generated code ?NAM:FirstName, Rejected
```

```
  OF EVENT:Selected
    !Embed: Control Event Handling, before generated code ?NAM:FirstName, Selected
    !Embed: Internal Control Event Handling ?NAM:FirstName, Selected
    !Embed: Control Event Handling, after generated code ?NAM:FirstName, Selected
  ELSE
    !Embed: Other Control Event Handling ?NAM:FirstName
  END
  !Embed: Control Handling, after event handling ?NAM:FirstName
OF ?NAM:LastName:Prompt
  !Embed: Control Handling, before event handling ?NAM:LastName:Prompt
  CASE EVENT()
  ELSE
    !Embed: Other Control Event Handling ?NAM:LastName:Prompt
  END
  !Embed: Control Handling, after event handling ?NAM:LastName:Prompt
OF ?NAM:LastName
  !Embed: Control Handling, before event handling ?NAM:LastName
  CASE EVENT()
  OF EVENT:Accepted
    !Embed: Control Event Handling, before generated code ?NAM:LastName, Accepted
    !Embed: Internal Control Event Handling ?NAM:LastName, Accepted
    !Embed: Control Event Handling, after generated code ?NAM:LastName, Accepted
  OF EVENT:Rejected
    !Embed: Control Event Handling, before generated code ?NAM:LastName, Rejected
    !Embed: Internal Control Event Handling ?NAM:LastName, Rejected
    !Embed: Control Event Handling, after generated code ?NAM:LastName, Rejected
  OF EVENT:Selected
    !Embed: Control Event Handling, before generated code ?NAM:LastName, Selected
    !Embed: Internal Control Event Handling ?NAM:LastName, Selected
    !Embed: Control Event Handling, after generated code ?NAM:LastName, Selected
  ELSE
    !Embed: Other Control Event Handling ?NAM:LastName
  END
  !Embed: Control Handling, after event handling ?NAM:LastName
OF ?NAM:Address:Prompt
  !Embed: Control Handling, before event handling ?NAM:Address:Prompt
  CASE EVENT()
  ELSE
    !Embed: Other Control Event Handling ?NAM:Address:Prompt
  END
  !Embed: Control Handling, after event handling ?NAM:Address:Prompt
OF ?NAM:Address
  !Embed: Control Handling, before event handling ?NAM:Address
  CASE EVENT()
  OF EVENT:Accepted
    !Embed: Control Event Handling, before generated code ?NAM:Address, Accepted
    !Embed: Internal Control Event Handling ?NAM:Address, Accepted
    !Embed: Control Event Handling, after generated code ?NAM:Address, Accepted
  OF EVENT:Rejected
    !Embed: Control Event Handling, before generated code ?NAM:Address, Rejected
    !Embed: Internal Control Event Handling ?NAM:Address, Rejected
    !Embed: Control Event Handling, after generated code ?NAM:Address, Rejected
  OF EVENT:Selected
    !Embed: Control Event Handling, before generated code ?NAM:Address, Selected
    !Embed: Internal Control Event Handling ?NAM:Address, Selected
    !Embed: Control Event Handling, after generated code ?NAM:Address, Selected
  ELSE
    !Embed: Other Control Event Handling ?NAM:Address
  END
  !Embed: Control Handling, after event handling ?NAM:Address
```

```
OF ?NAM:City:Prompt
  !Embed: Control Handling, before event handling ?NAM:City:Prompt
  CASE EVENT()
  ELSE
    !Embed: Other Control Event Handling ?NAM:City:Prompt
  END
  !Embed: Control Handling, after event handling ?NAM:City:Prompt
OF ?NAM:City
  !Embed: Control Handling, before event handling ?NAM:City
  CASE EVENT()
  OF EVENT:Accepted
    !Embed: Control Event Handling, before generated code ?NAM:City, Accepted
    !Embed: Internal Control Event Handling ?NAM:City, Accepted
    !Embed: Control Event Handling, after generated code ?NAM:City, Accepted
  OF EVENT:Rejected
    !Embed: Control Event Handling, before generated code ?NAM:City, Rejected
    !Embed: Internal Control Event Handling ?NAM:City, Rejected
    !Embed: Control Event Handling, after generated code ?NAM:City, Rejected
  OF EVENT:Selected
    !Embed: Control Event Handling, before generated code ?NAM:City, Selected
    !Embed: Internal Control Event Handling ?NAM:City, Selected
    !Embed: Control Event Handling, after generated code ?NAM:City, Selected
  ELSE
    !Embed: Other Control Event Handling ?NAM:City
  END
  !Embed: Control Handling, after event handling ?NAM:City
OF ?NAM:State:Prompt
  !Embed: Control Handling, before event handling ?NAM:State:Prompt
  CASE EVENT()
  ELSE
    !Embed: Other Control Event Handling ?NAM:State:Prompt
  END
  !Embed: Control Handling, after event handling ?NAM:State:Prompt
OF ?NAM:State
  !Embed: Control Handling, before event handling ?NAM:State
  CASE EVENT()
  OF EVENT:Accepted
    !Embed: Control Event Handling, before generated code ?NAM:State, Accepted
    !Embed: Internal Control Event Handling ?NAM:State, Accepted
    !Embed: Control Event Handling, after generated code ?NAM:State, Accepted
  OF EVENT:Rejected
    !Embed: Control Event Handling, before generated code ?NAM:State, Rejected
    !Embed: Internal Control Event Handling ?NAM:State, Rejected
    !Embed: Control Event Handling, after generated code ?NAM:State, Rejected
  OF EVENT:Selected
    !Embed: Control Event Handling, before generated code ?NAM:State, Selected
    !Embed: Internal Control Event Handling ?NAM:State, Selected
    !Embed: Control Event Handling, after generated code ?NAM:State, Selected
  ELSE
    !Embed: Other Control Event Handling ?NAM:State
  END
  !Embed: Control Handling, after event handling ?NAM:State
OF ?NAM:Zip:Prompt
  !Embed: Control Handling, before event handling ?NAM:Zip:Prompt
  CASE EVENT()
  ELSE
    !Embed: Other Control Event Handling ?NAM:Zip:Prompt
  END
  !Embed: Control Handling, after event handling ?NAM:Zip:Prompt
OF ?NAM:Zip
```

```
  !Embed: Control Handling, before event handling ?NAM:Zip
  CASE EVENT()
  OF EVENT:Accepted
    !Embed: Control Event Handling, before generated code ?NAM:Zip, Accepted
    !Embed: Internal Control Event Handling ?NAM:Zip, Accepted
    !Embed: Control Event Handling, after generated code ?NAM:Zip, Accepted
  OF EVENT:Rejected
    !Embed: Control Event Handling, before generated code ?NAM:Zip, Rejected
    !Embed: Internal Control Event Handling ?NAM:Zip, Rejected
    !Embed: Control Event Handling, after generated code ?NAM:Zip, Rejected
  OF EVENT:Selected
    !Embed: Control Event Handling, before generated code ?NAM:Zip, Selected
    !Embed: Internal Control Event Handling ?NAM:Zip, Selected
    !Embed: Control Event Handling, after generated code ?NAM:Zip, Selected
  ELSE
    !Embed: Other Control Event Handling ?NAM:Zip
  END
  !Embed: Control Handling, after event handling ?NAM:Zip
OF ?OK
  !Embed: Control Handling, before event handling ?OK
  CASE EVENT()
  OF EVENT:Accepted
    !Embed: Control Event Handling, before generated code ?OK, Accepted
    DO SyncWindow
    !Embed: Internal Control Event Handling ?OK, Accepted
    IF OriginalRequest = ChangeRecord OR OriginalRequest = InsertRecord
      SELECT()
    ELSE
      POST(EVENT:Completed)
    END
    !Embed: Control Event Handling, after generated code ?OK, Accepted
  OF EVENT:Selected
    !Embed: Control Event Handling, before generated code ?OK, Selected
    !Embed: Internal Control Event Handling ?OK, Selected
    !Embed: Control Event Handling, after generated code ?OK, Selected
  ELSE
    !Embed: Other Control Event Handling ?OK
  END
  !Embed: Control Handling, after event handling ?OK
OF ?Cancel
  !Embed: Control Handling, before event handling ?Cancel
  CASE EVENT()
  OF EVENT:Accepted
    !Embed: Control Event Handling, before generated code ?Cancel, Accepted
    DO SyncWindow
    !Embed: Internal Control Event Handling ?Cancel, Accepted
    LocalResponse = RequestCancelled
    POST(Event:CloseWindow)
    !Embed: Control Event Handling, after generated code ?Cancel, Accepted
  OF EVENT:Selected
    !Embed: Control Event Handling, before generated code ?Cancel, Selected
    !Embed: Internal Control Event Handling ?Cancel, Selected
    !Embed: Control Event Handling, after generated code ?Cancel, Selected
  ELSE
    !Embed: Other Control Event Handling ?Cancel
  END
  !Embed: Control Handling, after event handling ?Cancel
OF ?Help
  !Embed: Control Handling, before event handling ?Help
  CASE EVENT()
```

```
      OF EVENT:Accepted
        !Embed: Control Event Handling, before generated code ?Help, Accepted
        DO SyncWindow
        !Embed: Internal Control Event Handling ?Help, Accepted
        !Embed: Control Event Handling, after generated code ?Help, Accepted
      OF EVENT:Selected
        !Embed: Control Event Handling, before generated code ?Help, Selected
        !Embed: Internal Control Event Handling ?Help, Selected
        !Embed: Control Event Handling, after generated code ?Help, Selected
      ELSE
        !Embed: Other Control Event Handling ?Help
      END
      !Embed: Control Handling, after event handling ?Help
    !Embed: CASE FIELD() structure, after generated code
    END
    !Embed: Accept Loop, After CASE FIELD() handling
  END
  DO ProcedureReturn
!---------------------------------------------------------------------------
ProcedureReturn ROUTINE
  !Embed: End of Procedure, Before Closing Files
  Names::Used -= 1
  IF Names::Used = 0 THEN CLOSE(Names).
  !Embed: End of Procedure, After Closing Files
  !Embed: Before Closing the Window
  IF WindowOpened
    CLOSE(QuickWindow)
  END
  !Embed: After Closing the Window
  !Embed: End of Procedure
  IF LocalResponse
    GlobalResponse = LocalResponse
  ELSE
    GlobalResponse = RequestCancelled
  END
  RETURN
!---------------------------------------------------------------------------
InitializeWindow ROUTINE
  !Embed: Window Initialization Code
  DO RefreshWindow
!---------------------------------------------------------------------------
RefreshWindow ROUTINE
  IF QuickWindow{Prop:AcceptAll} THEN EXIT.
  !Embed: Refresh Window routine, before lookups
  !Embed: Lookup Related Records
  !Embed: Lookup Related Records
  !Embed: Refresh Window routine, after lookups
  !Embed: Refresh Window routine, before DISPLAY()
  DISPLAY()
  ForceRefresh = False
!---------------------------------------------------------------------------
SyncWindow ROUTINE
  !Embed: Sync Record routine, before lookups
  !Embed: Lookup Related Records
  !Embed: Sync Record routine, after lookups
!---------------------------------------------------------------------------
!Embed: Procedure Routines
PrimeFields ROUTINE
  NAM:Record = SAV::NAM:Record
  !Embed: Prime record fields on Insert
```

SAV::NAM:Record = NAM:Record

Example Source for Frame Procedure

The following code was generated from the Frame procedure template.

Comments appear in **red** at the default embed points.

```
                MEMBER('Prog.clw')        ! This is a MEMBER module
!Embed: Compile Module Declarations
!Embed: Compile Module Declarations
!Embed: Module Data Section
!Embed: Module Data Section
!Embed: Gather Template Symbols
Main PROCEDURE

LocalRequest       LONG,AUTO
OriginalRequest      LONG,AUTO
LocalResponse      LONG,AUTO
WindowOpened       LONG
WindowInitialized    LONG
ForceRefresh       LONG,AUTO
CurrentTab         STRING(80)
!Embed: Data Section, Before Window Declaration
AppFrame         APPLICATION('Application'),AT(,,400,220),FONT('MS Sans
Serif',8,,),STATUS(-1,80,120,45),SYSTEM,MAX,RESIZE,IMM
                   MENUBAR
                     MENU('&File')
                       ITEM('&Print Setup ...'),USE(?PrintSetup),MSG('Setup
printer'),STD(STD:PrintSetup)
                       ITEM,SEPARATOR
                       ITEM('E&xit'),USE(?Exit),MSG('Exit this
application'),STD(STD:Close)
                     END
                     MENU('&Edit')
                       ITEM('Cu&t'),USE(?Cut),MSG('Remove item to Windows
Clipboard'),STD(STD:Cut)
                       ITEM('&Copy'),USE(?Copy),MSG('Copy item to Windows
Clipboard'),STD(STD:Copy)
                       ITEM('&Paste'),USE(?Paste),MSG('Paste contents of Windows
Clipboard'),STD(STD:Paste)
                     END
                     MENU('&Browse')
                       ITEM('Browse the Names file'),USE(?BrowseNames),MSG('Browse
Names')
                     END
                     MENU('&Reports'),USE(?ReportMenu),MSG('Report data')
                       MENU('Report the Names file'),USE(?PrintNames)
                         ITEM('Print by NAM:KeyNumber key'),USE(?
PrintNAM:KeyNumber),MSG('Print ordered by the NAM:KeyNumber key')
                         ITEM('Print by NAM:KeyZip key'),USE(?
PrintNAM:KeyZip),MSG('Print ordered by the NAM:KeyZip key')
                       END
                     END
                     MENU('&Window'),MSG('Create and Arrange
windows'),STD(STD:WindowList)
                       ITEM('T&ile'),USE(?Tile),MSG('Make all open windows
visible'),STD(STD:TileWindow)
                       ITEM('&Cascade'),USE(?Cascade),MSG('Stack all open
windows'),STD(STD:CascadeWindow)
```

```
                      ITEM('&Arrange Icons'),USE(?Arrange),MSG('Align all window
icons'),STD(STD:ArrangeIcons)
                    END
                    MENU('&Help'),MSG('Windows Help')
                      ITEM('&Contents'),USE(?Helpindex),MSG('View the contents of the
help file'),STD(STD:HelpIndex)
                      ITEM('&Search for Help On...'),USE(?HelpSearch),MSG('Search for
help on a subject'),STD(STD:HelpSearch)
                      ITEM('&How to Use Help'),USE(?HelpOnHelp),MSG('How to use Windows
Help'),STD(STD:HelpOnHelp)
                    END
                  END
                END
!Embed: Data Section, After Window Declaration
  CODE
  !Embed: Initialize the Procedure
  LocalRequest = GlobalRequest
  OriginalRequest = GlobalRequest
  LocalResponse = RequestCancelled
  ForceRefresh = False
  CLEAR(GlobalRequest)
  CLEAR(GlobalResponse)
  !Embed: Procedure Setup
  IF KEYCODE() = MouseRight
    SETKEYCODE(0)
  END
  !Embed: Beginning of Procedure, Before Opening Files
  !Embed: Beginning of Procedure, After Opening Files
  !Embed: Before Opening the Window
  OPEN(AppFrame)
  WindowOpened=True
  !Embed: After Opening the Window
  !Embed: Preparing Window Alerts
  !Embed: Preparing to Process the Window
  ACCEPT
    !Embed: Accept Loop, Before CASE EVENT() handling
    CASE EVENT()
    !Embed: CASE EVENT() structure, before generated code
    OF EVENT:AlertKey
      !Embed: Window Event Handling AlertKey
    OF EVENT:PreAlertKey
      !Embed: Window Event Handling PreAlertKey
    OF EVENT:CloseWindow
      !Embed: Window Event Handling CloseWindow
    OF EVENT:CloseDown
      !Embed: Window Event Handling CloseDown
    OF EVENT:OpenWindow
      !Embed: Window Event Handling OpenWindow
      IF NOT WindowInitialized
        DO InitializeWindow
        WindowInitialized = True
      END
    OF EVENT:LoseFocus
      !Embed: Window Event Handling LoseFocus
    OF EVENT:GainFocus
      !Embed: Window Event Handling GainFocus
      ForceRefresh = True
      IF NOT WindowInitialized
        DO InitializeWindow
        WindowInitialized = True
```

```
  ELSE
    DO RefreshWindow
  END
OF EVENT:Suspend
  !Embed: Window Event Handling Suspend
OF EVENT:Resume
  !Embed: Window Event Handling Resume
ELSE
  !Embed: Other Window Event Handling
!Embed: CASE EVENT() structure, after generated code
END
!Embed: Accept Loop, After CASE EVENT() handling
CASE ACCEPTED()
OF
  !Embed: Control Event Handling, before generated code ,
  !Embed: Internal Control Event Handling ,
  !Embed: Control Event Handling, after generated code ,
OF ?PrintSetup
  !Embed: Control Event Handling, before generated code ?PrintSetup, Accepted
  DO SyncWindow
  !Embed: Internal Control Event Handling ?PrintSetup, Accepted
  !Embed: Control Event Handling, after generated code ?PrintSetup, Accepted
OF
  !Embed: Control Event Handling, before generated code ,
  !Embed: Internal Control Event Handling ,
  !Embed: Control Event Handling, after generated code ,
OF ?Exit
  !Embed: Control Event Handling, before generated code ?Exit, Accepted
  DO SyncWindow
  !Embed: Internal Control Event Handling ?Exit, Accepted
  !Embed: Control Event Handling, after generated code ?Exit, Accepted
OF
  !Embed: Control Event Handling, before generated code ,
  !Embed: Internal Control Event Handling ,
  !Embed: Control Event Handling, after generated code ,
OF ?Cut
  !Embed: Control Event Handling, before generated code ?Cut, Accepted
  DO SyncWindow
  !Embed: Internal Control Event Handling ?Cut, Accepted
  !Embed: Control Event Handling, after generated code ?Cut, Accepted
OF ?Copy
  !Embed: Control Event Handling, before generated code ?Copy, Accepted
  DO SyncWindow
  !Embed: Internal Control Event Handling ?Copy, Accepted
  !Embed: Control Event Handling, after generated code ?Copy, Accepted
OF ?Paste
  !Embed: Control Event Handling, before generated code ?Paste, Accepted
  DO SyncWindow
  !Embed: Internal Control Event Handling ?Paste, Accepted
  !Embed: Control Event Handling, after generated code ?Paste, Accepted
OF
  !Embed: Control Event Handling, before generated code ,
  !Embed: Internal Control Event Handling ,
  !Embed: Control Event Handling, after generated code ,
OF ?BrowseNames
  !Embed: Control Event Handling, before generated code ?BrowseNames, Accepted
  DO SyncWindow
  START(BrowseNames,50000)
  LocalRequest = OriginalRequest
  DO RefreshWindow
```

```
      !Embed: Internal Control Event Handling ?BrowseNames, Accepted
      !Embed: Control Event Handling, after generated code ?BrowseNames, Accepted
    OF ?ReportMenu
      !Embed: Control Event Handling, before generated code ?ReportMenu,
      !Embed: Internal Control Event Handling ?ReportMenu,
      !Embed: Control Event Handling, after generated code ?ReportMenu,
    OF ?PrintNames
      !Embed: Control Event Handling, before generated code ?PrintNames,
      !Embed: Internal Control Event Handling ?PrintNames,
      !Embed: Control Event Handling, after generated code ?PrintNames,
    OF ?PrintNAM:KeyNumber
      !Embed: Control Event Handling, before generated code ?PrintNAM:KeyNumber,
Accepted
      DO SyncWindow
      START(PrintNAM:KeyNumber,50000)
      LocalRequest = OriginalRequest
      DO RefreshWindow
      !Embed: Internal Control Event Handling ?PrintNAM:KeyNumber, Accepted
      !Embed: Control Event Handling, after generated code ?PrintNAM:KeyNumber,
Accepted
    OF ?PrintNAM:KeyZip
      !Embed: Control Event Handling, before generated code ?PrintNAM:KeyZip, Accepted
      DO SyncWindow
      START(PrintNAM:KeyZip,50000)
      LocalRequest = OriginalRequest
      DO RefreshWindow
      !Embed: Internal Control Event Handling ?PrintNAM:KeyZip, Accepted
      !Embed: Control Event Handling, after generated code ?PrintNAM:KeyZip, Accepted
    OF
      !Embed: Control Event Handling, before generated code ,
      !Embed: Internal Control Event Handling ,
      !Embed: Control Event Handling, after generated code ,
    OF ?Tile
      !Embed: Control Event Handling, before generated code ?Tile, Accepted
      DO SyncWindow
      !Embed: Internal Control Event Handling ?Tile, Accepted
      !Embed: Control Event Handling, after generated code ?Tile, Accepted
    OF ?Cascade
      !Embed: Control Event Handling, before generated code ?Cascade, Accepted
      DO SyncWindow
      !Embed: Internal Control Event Handling ?Cascade, Accepted
      !Embed: Control Event Handling, after generated code ?Cascade, Accepted
    OF ?Arrange
      !Embed: Control Event Handling, before generated code ?Arrange, Accepted
      DO SyncWindow
      !Embed: Internal Control Event Handling ?Arrange, Accepted
      !Embed: Control Event Handling, after generated code ?Arrange, Accepted
    OF
      !Embed: Control Event Handling, before generated code ,
      !Embed: Internal Control Event Handling ,
      !Embed: Control Event Handling, after generated code ,
    OF ?Helpindex
      !Embed: Control Event Handling, before generated code ?Helpindex, Accepted
      DO SyncWindow
      !Embed: Internal Control Event Handling ?Helpindex, Accepted
      !Embed: Control Event Handling, after generated code ?Helpindex, Accepted
    OF ?HelpSearch
      !Embed: Control Event Handling, before generated code ?HelpSearch, Accepted
      DO SyncWindow
      !Embed: Internal Control Event Handling ?HelpSearch, Accepted
```

```
  !Embed: Control Event Handling, after generated code ?HelpSearch, Accepted
OF ?HelpOnHelp
  !Embed: Control Event Handling, before generated code ?HelpOnHelp, Accepted
  DO SyncWindow
  !Embed: Internal Control Event Handling ?HelpOnHelp, Accepted
  !Embed: Control Event Handling, after generated code ?HelpOnHelp, Accepted
END
!Embed: Accept Loop, Before CASE FIELD() handling
CASE FIELD()
!Embed: CASE FIELD() structure, before generated code
OF ?PrintSetup
  !Embed: Control Handling, before event handling ?PrintSetup
  CASE EVENT()
  OF EVENT:Accepted
  ELSE
    !Embed: Other Control Event Handling ?PrintSetup
  END
  !Embed: Control Handling, after event handling ?PrintSetup
OF ?Exit
  !Embed: Control Handling, before event handling ?Exit
  CASE EVENT()
  OF EVENT:Accepted
  ELSE
    !Embed: Other Control Event Handling ?Exit
  END
  !Embed: Control Handling, after event handling ?Exit
OF ?Cut
  !Embed: Control Handling, before event handling ?Cut
  CASE EVENT()
  OF EVENT:Accepted
  ELSE
    !Embed: Other Control Event Handling ?Cut
  END
  !Embed: Control Handling, after event handling ?Cut
OF ?Copy
  !Embed: Control Handling, before event handling ?Copy
  CASE EVENT()
  OF EVENT:Accepted
  ELSE
    !Embed: Other Control Event Handling ?Copy
  END
  !Embed: Control Handling, after event handling ?Copy
OF ?Paste
  !Embed: Control Handling, before event handling ?Paste
  CASE EVENT()
  OF EVENT:Accepted
  ELSE
    !Embed: Other Control Event Handling ?Paste
  END
  !Embed: Control Handling, after event handling ?Paste
OF ?BrowseNames
  !Embed: Control Handling, before event handling ?BrowseNames
  CASE EVENT()
  OF EVENT:Accepted
  ELSE
    !Embed: Other Control Event Handling ?BrowseNames
  END
  !Embed: Control Handling, after event handling ?BrowseNames
OF ?ReportMenu
  !Embed: Control Handling, before event handling ?ReportMenu
```

```
  CASE EVENT()
  OF EVENT:Accepted
  ELSE
    !Embed: Other Control Event Handling ?ReportMenu
  END
  !Embed: Control Handling, after event handling ?ReportMenu
OF ?PrintNames
  !Embed: Control Handling, before event handling ?PrintNames
  CASE EVENT()
  OF EVENT:Accepted
  ELSE
    !Embed: Other Control Event Handling ?PrintNames
  END
  !Embed: Control Handling, after event handling ?PrintNames
OF ?PrintNAM:KeyNumber
  !Embed: Control Handling, before event handling ?PrintNAM:KeyNumber
  CASE EVENT()
  OF EVENT:Accepted
  ELSE
    !Embed: Other Control Event Handling ?PrintNAM:KeyNumber
  END
  !Embed: Control Handling, after event handling ?PrintNAM:KeyNumber
OF ?PrintNAM:KeyZip
  !Embed: Control Handling, before event handling ?PrintNAM:KeyZip
  CASE EVENT()
  OF EVENT:Accepted
  ELSE
    !Embed: Other Control Event Handling ?PrintNAM:KeyZip
  END
  !Embed: Control Handling, after event handling ?PrintNAM:KeyZip
OF ?Tile
  !Embed: Control Handling, before event handling ?Tile
  CASE EVENT()
  OF EVENT:Accepted
  ELSE
    !Embed: Other Control Event Handling ?Tile
  END
  !Embed: Control Handling, after event handling ?Tile
OF ?Cascade
  !Embed: Control Handling, before event handling ?Cascade
  CASE EVENT()
  OF EVENT:Accepted
  ELSE
    !Embed: Other Control Event Handling ?Cascade
  END
  !Embed: Control Handling, after event handling ?Cascade
OF ?Arrange
  !Embed: Control Handling, before event handling ?Arrange
  CASE EVENT()
  OF EVENT:Accepted
  ELSE
    !Embed: Other Control Event Handling ?Arrange
  END
  !Embed: Control Handling, after event handling ?Arrange
OF ?Helpindex
  !Embed: Control Handling, before event handling ?Helpindex
  CASE EVENT()
  OF EVENT:Accepted
  ELSE
    !Embed: Other Control Event Handling ?Helpindex
```

```
      END
      !Embed: Control Handling, after event handling ?Helpindex
    OF ?HelpSearch
      !Embed: Control Handling, before event handling ?HelpSearch
      CASE EVENT()
      OF EVENT:Accepted
      ELSE
        !Embed: Other Control Event Handling ?HelpSearch
      END
      !Embed: Control Handling, after event handling ?HelpSearch
    OF ?HelpOnHelp
      !Embed: Control Handling, before event handling ?HelpOnHelp
      CASE EVENT()
      OF EVENT:Accepted
      ELSE
        !Embed: Other Control Event Handling ?HelpOnHelp
      END
      !Embed: Control Handling, after event handling ?HelpOnHelp
    !Embed: CASE FIELD() structure, after generated code
    END
    !Embed: Accept Loop, After CASE FIELD() handling
  END
  DO ProcedureReturn
!---------------------------------------------------------------------------
ProcedureReturn ROUTINE
  !Embed: End of Procedure, Before Closing Files
  !Embed: End of Procedure, After Closing Files
  !Embed: Before Closing the Window
  IF WindowOpened
    CLOSE(AppFrame)
  END
  !Embed: After Closing the Window
  !Embed: End of Procedure
  IF LocalResponse
    GlobalResponse = LocalResponse
  ELSE
    GlobalResponse = RequestCancelled
  END
  RETURN
!---------------------------------------------------------------------------
InitializeWindow ROUTINE
  !Embed: Window Initialization Code
  DO RefreshWindow
!---------------------------------------------------------------------------
RefreshWindow ROUTINE
  IF AppFrame{Prop:AcceptAll} THEN EXIT.
  !Embed: Refresh Window routine, before lookups
  !Embed: Lookup Related Records
  !Embed: Refresh Window routine, after lookups
  !Embed: Refresh Window routine, before DISPLAY()
  DISPLAY()
  ForceRefresh = False
!---------------------------------------------------------------------------
SyncWindow ROUTINE
  !Embed: Sync Record routine, before lookups
  !Embed: Lookup Related Records
  !Embed: Sync Record routine, after lookups
!---------------------------------------------------------------------------
!Embed: Procedure Routines
```

Example Source for Menu Procedure

The following code was generated from the Menu procedure template.

Comments appear in **red** at the default embed points.

```
                  MEMBER('Prog.clw')        ! This is a MEMBER module
!Embed: Compile Module Declarations
!Embed: Compile Module Declarations
!Embed: Module Data Section
!Embed: Module Data Section
!Embed: Gather Template Symbols
MenuProc PROCEDURE

LocalRequest       LONG,AUTO
OriginalRequest      LONG,AUTO
LocalResponse      LONG,AUTO
WindowOpened       LONG
WindowInitialized    LONG
ForceRefresh       LONG,AUTO
CurrentTab         STRING(80)
!Embed: Data Section, Before Window Declaration
MenuWindow         WINDOW('Caption'),AT(0,0,260,146)
                   MENUBAR
                     MENU('&File'),USE(?FileMenu)
                       ITEM('Item&2'),USE(?Item2),SEPARATOR
                       ITEM('E&xit'),USE(?Exit),STD(STD:Close)
                     END
                   END
                 END
!Embed: Data Section, After Window Declaration
  CODE
  !Embed: Initialize the Procedure
  LocalRequest = GlobalRequest
  OriginalRequest = GlobalRequest
  LocalResponse = RequestCancelled
  ForceRefresh = False
  CLEAR(GlobalRequest)
  CLEAR(GlobalResponse)
  !Embed: Procedure Setup
  IF KEYCODE() = MouseRight
    SETKEYCODE(0)
  END
  !Embed: Beginning of Procedure, Before Opening Files
  !Embed: Beginning of Procedure, After Opening Files
  !Embed: Before Opening the Window
  OPEN(MenuWindow)
  WindowOpened=True
  !Embed: After Opening the Window
  !Embed: Preparing Window Alerts
  !Embed: Preparing to Process the Window
  ACCEPT
    !Embed: Accept Loop, Before CASE EVENT() handling
    CASE EVENT()
    !Embed: CASE EVENT() structure, before generated code
    OF EVENT:AlertKey
      !Embed: Window Event Handling AlertKey
    OF EVENT:PreAlertKey
      !Embed: Window Event Handling PreAlertKey
```

```
OF EVENT:CloseWindow
  !Embed: Window Event Handling CloseWindow
OF EVENT:CloseDown
  !Embed: Window Event Handling CloseDown
OF EVENT:OpenWindow
  !Embed: Window Event Handling OpenWindow
  IF NOT WindowInitialized
    DO InitializeWindow
    WindowInitialized = True
  END
OF EVENT:LoseFocus
  !Embed: Window Event Handling LoseFocus
OF EVENT:GainFocus
  !Embed: Window Event Handling GainFocus
  ForceRefresh = True
  IF NOT WindowInitialized
    DO InitializeWindow
    WindowInitialized = True
  ELSE
    DO RefreshWindow
  END
OF EVENT:Suspend
  !Embed: Window Event Handling Suspend
OF EVENT:Resume
  !Embed: Window Event Handling Resume
ELSE
  !Embed: Other Window Event Handling
!Embed: CASE EVENT() structure, after generated code
END
!Embed: Accept Loop, After CASE EVENT() handling
CASE ACCEPTED()
OF ?FileMenu
  !Embed: Control Event Handling, before generated code ?FileMenu,
  !Embed: Internal Control Event Handling ?FileMenu,
  !Embed: Control Event Handling, after generated code ?FileMenu,
OF ?Item2
  !Embed: Control Event Handling, before generated code ?Item2,
  !Embed: Internal Control Event Handling ?Item2,
  !Embed: Control Event Handling, after generated code ?Item2,
OF ?Exit
  !Embed: Control Event Handling, before generated code ?Exit, Accepted
  DO SyncWindow
  !Embed: Internal Control Event Handling ?Exit, Accepted
  !Embed: Control Event Handling, after generated code ?Exit, Accepted
END
!Embed: Accept Loop, Before CASE FIELD() handling
CASE FIELD()
!Embed: CASE FIELD() structure, before generated code
OF ?FileMenu
  !Embed: Control Handling, before event handling ?FileMenu
  CASE EVENT()
  OF EVENT:Accepted
  ELSE
    !Embed: Other Control Event Handling ?FileMenu
  END
  !Embed: Control Handling, after event handling ?FileMenu
OF ?Item2
  !Embed: Control Handling, before event handling ?Item2
  CASE EVENT()
  OF EVENT:Accepted
```

```
      ELSE
        !Embed: Other Control Event Handling ?Item2
      END
      !Embed: Control Handling, after event handling ?Item2
    OF ?Exit
      !Embed: Control Handling, before event handling ?Exit
      CASE EVENT()
      OF EVENT:Accepted
      ELSE
        !Embed: Other Control Event Handling ?Exit
      END
      !Embed: Control Handling, after event handling ?Exit
    !Embed: CASE FIELD() structure, after generated code
    END
    !Embed: Accept Loop, After CASE FIELD() handling
  END
  DO ProcedureReturn
!---------------------------------------------------------------------------
ProcedureReturn ROUTINE
  !Embed: End of Procedure, Before Closing Files
  !Embed: End of Procedure, After Closing Files
  !Embed: Before Closing the Window
  IF WindowOpened
    CLOSE(MenuWindow)
  END
  !Embed: After Closing the Window
  !Embed: End of Procedure
  IF LocalResponse
    GlobalResponse = LocalResponse
  ELSE
    GlobalResponse = RequestCancelled
  END
  RETURN
!---------------------------------------------------------------------------
InitializeWindow ROUTINE
  !Embed: Window Initialization Code
  DO RefreshWindow
!---------------------------------------------------------------------------
RefreshWindow ROUTINE
  IF MenuWindow{Prop:AcceptAll} THEN EXIT.
  !Embed: Refresh Window routine, before lookups
  !Embed: Lookup Related Records
  !Embed: Refresh Window routine, after lookups
  !Embed: Refresh Window routine, before DISPLAY()
  DISPLAY()
  ForceRefresh = False
!---------------------------------------------------------------------------
SyncWindow ROUTINE
  !Embed: Sync Record routine, before lookups
  !Embed: Lookup Related Records
  !Embed: Sync Record routine, after lookups
!---------------------------------------------------------------------------
!Embed: Procedure Routines
```

Example Source for Process Procedure

The following code was generated from the Process procedure template.

Comments appear in **red** at the default embed points.

```
                MEMBER('Prog.clw')        ! This is a MEMBER module
!Embed: Compile Module Declarations
!Embed: Compile Module Declarations
!Embed: Module Data Section
!Embed: Module Data Section
!Embed: Gather Template Symbols
process PROCEDURE
!Embed: Declaration Section
RejectRecord       LONG,AUTO
LocalRequest       LONG,AUTO
LocalResponse      LONG,AUTO
WindowOpened       LONG,AUTO
RecordsToProcess   LONG,AUTO
RecordsProcessed   LONG,AUTO
RecordsPerCycle      LONG,AUTO
RecordsThisCycle   LONG,AUTO
PercentProgress      BYTE
RecordStatus       BYTE,AUTO
!---------------------------------------------------------------------------
Process:View       VIEW(Names)
                  END
Progress:Thermometer BYTE
ProgressWindow WINDOW('Progress...'),AT(,,142,59),CENTER,TIMER(1),GRAY,DOUBLE
     PROGRESS,USE(Progress:Thermometer),AT(15,15,111,12),RANGE(0,100)
     STRING(''),AT(0,3,141,10),USE(?Progress:UserString),CENTER
     STRING(''),AT(0,30,141,10),USE(?Progress:PctText),CENTER
     BUTTON('Cancel'),AT(45,42,50,15),USE(?Progress:Cancel)
   END
  CODE
  !Embed: Procedure Setup
  LocalRequest = GlobalRequest
  LocalResponse = RequestCancelled
  CLEAR(GlobalRequest)
  CLEAR(GlobalResponse)
  !Embed: Beginning of Procedure, Before Opening Files
  IF Names::Used = 0
    CheckOpen(Names,1)
    BIND(NAM:RECORD)
  END
  Names::Used += 1
  !Embed: Beginning of Procedure, After Opening Files
  !Embed: Preparing to Process the Window
  RecordsToProcess = BYTES(Names)
  RecordsPerCycle = 1000
  RecordsProcessed = 0
  PercentProgress = 0
  !Embed: Before Opening Progress Window
  OPEN(ProgressWindow)
  ProgressWindow{Prop:Text} = 'Processing Records'
  ?Progress:PctText{Prop:Text} = '0% Completed'
  ?Progress:UserString{Prop:Text}=''
  !Embed: Before Turning QuickScan On
  SEND(Names,'QUICKSCAN=on')
```

```
!Embed: After Turning QuickScan On
ACCEPT
  CASE EVENT()
  OF Event:OpenWindow
    !Embed: Window Event: Open Window, before setting up for reading
    !Embed: Before SET() issued
    SET(Names)
    OPEN(Process:View)
    !Embed: Before first record retrieval
    LOOP
      DO GetNextRecord
      DO ValidateRecord
      CASE RecordStatus
        OF Record:Ok
          BREAK
        OF Record:OutOfRange
          LocalResponse = RequestCancelled
          BREAK
      END
    END
    IF LocalResponse = RequestCancelled
      POST(Event:CloseWindow)
      CYCLE
    END
    !Embed: After first record retrieval
    !Embed: Window Event: Open Window, after setting up for read
  OF Event:Timer
    RecordsThisCycle = 0
    LOOP WHILE RecordsThisCycle < RecordsPerCycle
      !Embed: Activity for each record
      !Embed: Error checking after record Action
      LOOP
        !Embed: Before subsequent record retrieval
        DO GetNextRecord
        !Embed: After subsequent record retrieval
        DO ValidateRecord
        CASE RecordStatus
          OF Record:OutOfRange
            LocalResponse = RequestCancelled
            BREAK
          OF Record:OK
            BREAK
        END
      END
      IF LocalResponse = RequestCancelled
        LocalResponse = RequestCompleted
        BREAK
      END
      LocalResponse = RequestCancelled
    END
    IF LocalResponse = RequestCompleted
      ?Progress:PctText{Prop:Text} = 'Process Completed'
      DISPLAY(?Progress:PctText)
      POST(Event:CloseWindow)
    END
  END
  CASE FIELD()
  OF ?Progress:Cancel
    CASE Event()
    OF Event:Accepted
```

```
        LocalResponse = RequestCancelled
        POST(Event:CloseWindow)
      END
    END
  END
  !Embed: Before Turning QuickScan Off
  IF SEND(Names,'QUICKSCAN=off').
  !Embed: After Turning QuickScan Off
  DO ProcedureReturn
ProcedureReturn ROUTINE
  !Embed: End of Procedure, Before Closing Files
  Names::Used -= 1
  IF Names::Used = 0 THEN CLOSE(Names).
  !Embed: End of Procedure, After Closing Files
  !Embed: End of Procedure
  IF LocalResponse
    GlobalResponse = LocalResponse
  ELSE
    GlobalResponse = RequestCancelled
  END
  RETURN
!---------------------------------------------------------------------------
ValidateRecord     ROUTINE
  RecordStatus = Record:OutOfRange
  IF LocalResponse = RequestCancelled THEN EXIT.
  !Embed: Validate Record: Range Checking
  RecordStatus = Record:Filtered
    !Embed: Validate Record: Filter Checking
  RecordStatus = Record:OK
  EXIT
GetNextRecord ROUTINE
  !Embed: Top of GetNextRecord ROUTINE
  NEXT(Process:View)
  !Embed: GetNextRecord ROUTINE, after NEXT
  IF ERRORCODE()
    IF ERRORCODE() <> BadRecErr
      StandardWarning(Warn:RecordFetchError,'Names')
    END
    LocalResponse = RequestCancelled
    !Embed: GetNextRecord ROUTINE, NEXT failed
    EXIT
  ELSE
    LocalResponse = RequestCompleted
    !Embed: GetNextRecord ROUTINE, NEXT succeeds
  END
  RecordsProcessed += BYTES(Names)
  RecordsThisCycle += BYTES(Names)
  IF PercentProgress < 100
    PercentProgress = (RecordsProcessed / RecordsToProcess)*100
    IF PercentProgress > 100
      PercentProgress = 100
    END
    IF PercentProgress <> Progress:Thermometer THEN
      Progress:Thermometer = PercentProgress
      ?Progress:PctText{Prop:Text} = FORMAT(PercentProgress,@N3) & '% Completed'
      DISPLAY()
    END
  END

!Embed: Procedure Routines
```

Example Source for Report Procedure

The following code was generated from the Report procedure template.

Comments appear in **red** at the default embed points.

```
                MEMBER('Prog.clw')        ! This is a MEMBER module
!Embed: Compile Module Declarations
!Embed: Compile Module Declarations
!Embed: Module Data Section
!Embed: Module Data Section
!Embed: Gather Template Symbols
PrintNAM:KeyNumber PROCEDURE
!Embed: Declaration Section
RejectRecord      LONG,AUTO
LocalRequest      LONG,AUTO
LocalResponse     LONG,AUTO
WindowOpened      LONG,AUTO
RecordsToProcess  LONG,AUTO
RecordsProcessed  LONG,AUTO
RecordsPerCycle     LONG,AUTO
RecordsThisCycle  LONG,AUTO
PercentProgress     BYTE
RecordStatus      BYTE,AUTO
!-----------------------------------------------------------------------------
Process:View      VIEW(Names)
                  PROJECT(NAM:Address)
                  PROJECT(NAM:City)
                  PROJECT(NAM:FirstName)
                  PROJECT(NAM:LastName)
                  PROJECT(NAM:Number)
                  PROJECT(NAM:State)
                  PROJECT(NAM:Zip)
                END
!Embed: Data Section, Before Report Declaration
report            REPORT,AT(1000,2000,6000,7000),PRE(RPT),FONT('Arial',10,,),THOUS
                    HEADER,AT(1000,1000,6000,1000)
                    END
detail              DETAIL
                      STRING(@n03),AT(125,80),USE(NAM:Number)
                      STRING(@S20),AT(125,240),USE(NAM:FirstName)
                      STRING(@S20),AT(125,400),USE(NAM:LastName)
                      STRING(@S20),AT(125,560),USE(NAM:Address)
                      STRING(@S20),AT(125,720),USE(NAM:City)
                      STRING(@s2),AT(125,880),USE(NAM:State)
                      STRING(@n05),AT(125,1040),USE(NAM:Zip)
                    END
                    FOOTER,AT(1000,9000,6000,1000)
                    END
                  END
!Embed: Data Section, After Report Declaration
Progress:Thermometer BYTE
ProgressWindow WINDOW('Progress...'),AT(,,142,59),CENTER,TIMER(1),GRAY,DOUBLE
     PROGRESS,USE(Progress:Thermometer),AT(15,15,111,12),RANGE(0,100)
     STRING(''),AT(0,3,141,10),USE(?Progress:UserString),CENTER
     STRING(''),AT(0,30,141,10),USE(?Progress:PctText),CENTER
     BUTTON('Cancel'),AT(45,42,50,15),USE(?Progress:Cancel)
   END
PrintSkipDetails BOOL,AUTO
```

```
PrintPreviewQueue QUEUE,PRE
PrintPreviewImage  STRING(80)
                END
  CODE
  !Embed: Procedure Setup
  LocalRequest = GlobalRequest
  LocalResponse = RequestCancelled
  CLEAR(GlobalRequest)
  CLEAR(GlobalResponse)
  !Embed: Beginning of Procedure, Before Opening Files
  IF Names::Used = 0
    CheckOpen(Names,1)
    BIND(NAM:RECORD)
  END
  Names::Used += 1
  !Embed: Beginning of Procedure, After Opening Files
  !Embed: Preparing to Process the Window
  RecordsToProcess = RECORDS(Names)
  RecordsPerCycle = 25
  RecordsProcessed = 0
  PercentProgress = 0
  !Embed: Before Opening Progress Window
  OPEN(ProgressWindow)
  Progress:Thermometer = 0
  ?Progress:PctText{Prop:Text} = '0% Completed'
  ProgressWindow{Prop:Text} = 'Generating Report'
  ?Progress:UserString{Prop:Text}=''
  !Embed: After Opening Progress Window
  !Embed: Before Turning QuickScan On
  SEND(Names,'QUICKSCAN=on')
  !Embed: After Turning QuickScan On
  ACCEPT
    CASE EVENT()
    OF Event:OpenWindow
      !Embed: Window Event: Open Window, before setting up for reading
      !Embed: Before SET() issued
      SET(NAM:KeyNumber)
      OPEN(Process:View)
      !Embed: Before first record retrieval
      LOOP
        DO GetNextRecord
        DO ValidateRecord
        CASE RecordStatus
          OF Record:Ok
            BREAK
          OF Record:OutOfRange
            LocalResponse = RequestCancelled
            BREAK
        END
      END
      IF LocalResponse = RequestCancelled
        POST(Event:CloseWindow)
        CYCLE
      END
      !Embed: After first record retrieval
      !Embed: Window Event: Open Window, after setting up for read
      !Embed: Before Opening Report
      OPEN(report)
      !Embed: After Opening Report
      report{Prop:Preview} = PrintPreviewImage
```

```
  OF Event:Timer
    LOOP RecordsPerCycle TIMES
      !Embed: Before Lookups
      !Embed: Lookup Related Records
      !Embed: After Lookups
      !Embed: Before Printing Detail Section
      PrintSkipDetails = FALSE



      IF ~PrintSkipDetails THEN
        PRINT(RPT:detail)
      END

      !Embed: After Printing Detail Section
      LOOP
        !Embed: Before subsequent record retrieval
        DO GetNextRecord
        !Embed: After subsequent record retrieval
        DO ValidateRecord
        CASE RecordStatus
          OF Record:OutOfRange
            LocalResponse = RequestCancelled
            BREAK
          OF Record:OK
            BREAK
        END
      END
      IF LocalResponse = RequestCancelled
        LocalResponse = RequestCompleted
        BREAK
      END
      LocalResponse = RequestCancelled
    END
    IF LocalResponse = RequestCompleted
      POST(Event:CloseWindow)
    END
  END
  CASE FIELD()
  OF ?Progress:Cancel
    CASE Event()
    OF Event:Accepted
      LocalResponse = RequestCancelled
      POST(Event:CloseWindow)
    END
  END
END
!Embed: Before Turning QuickScan Off
IF SEND(Names,'QUICKSCAN=off').
!Embed: After Turning QuickScan Off
!Embed: Before Print Preview
IF LocalResponse = RequestCompleted
  ENDPAGE(report)
  ReportPreview(PrintPreviewQueue)
  IF GlobalResponse = RequestCompleted
    report{PROP:FlushPreview} = True
  END
END
!Embed: Before Closing Report
CLOSE(report)
```

```
  !Embed: After Closing Report
  FREE(PrintPreviewQueue)
  DO ProcedureReturn
ProcedureReturn ROUTINE
  !Embed: End of Procedure, Before Closing Files
  Names::Used -= 1
  IF Names::Used = 0 THEN CLOSE(Names).
  !Embed: End of Procedure, After Closing Files
  !Embed: End of Procedure
  IF LocalResponse
    GlobalResponse = LocalResponse
  ELSE
    GlobalResponse = RequestCancelled
  END
  RETURN
!---------------------------------------------------------------------------
ValidateRecord     ROUTINE
  RecordStatus = Record:OutOfRange
  IF LocalResponse = RequestCancelled THEN EXIT.
  !Embed: Validate Record: Range Checking
  RecordStatus = Record:Filtered
    !Embed: Validate Record: Filter Checking
  RecordStatus = Record:OK
  EXIT
GetNextRecord ROUTINE
  !Embed: Top of GetNextRecord ROUTINE
  NEXT(Process:View)
  !Embed: GetNextRecord ROUTINE, after NEXT
  IF ERRORCODE()
    IF ERRORCODE() <> BadRecErr
      StandardWarning(Warn:RecordFetchError,'Names')
    END
    LocalResponse = RequestCancelled
    !Embed: GetNextRecord ROUTINE, NEXT failed
    EXIT
  ELSE
    LocalResponse = RequestCompleted
    !Embed: GetNextRecord ROUTINE, NEXT succeeds
  END
  RecordsProcessed += 1
  RecordsThisCycle += 1
  IF PercentProgress < 100
    PercentProgress = (RecordsProcessed / RecordsToProcess)*100
    IF PercentProgress > 100
      PercentProgress = 100
    END
    IF PercentProgress <> Progress:Thermometer THEN
      Progress:Thermometer = PercentProgress
      ?Progress:PctText{Prop:Text} = FORMAT(PercentProgress,@N3) & '% Completed'
      DISPLAY()
    END
  END

!Embed: Procedure Routines
```

Example Source for Source Procedure

The following code was generated from the Source procedure template.

Comments appear in **red** at the default embed points.

```
                MEMBER('Prog.clw')        ! This is a MEMBER module
!Embed: Compile Module Declarations
!Embed: Compile Module Declarations
!Embed: Module Data Section
!Embed: Module Data Section
SrcProc            PROCEDURE                ! Declare Procedure
!Embed: Data Section
  CODE                                      ! Begin processed code
!Embed: Processed Code
```

Example Source for Viewer Procedure

The following code was generated from the Viewer procedure template.

Comments appear in **red** at the default embed points.

```
                MEMBER('Prog.clw')       ! This is a MEMBER module
!Embed: Compile Module Declarations
!Embed: Compile Module Declarations
!Embed: Module Data Section
!Embed: Module Data Section
!Embed: Gather Template Symbols
ViewProc PROCEDURE

LocalRequest       LONG,AUTO
OriginalRequest      LONG,AUTO
LocalResponse      LONG,AUTO
WindowOpened       LONG
WindowInitialized    LONG
ForceRefresh       LONG,AUTO
CurrentTab         STRING(80)
ASCIIFileSize      LONG
ASCIIBytesThisRead LONG
ASCIIBytesRead     LONG
ASCIIBytesThisCycle LONG
ASCIIPercentProgress BYTE
!Embed: Data Section, Before Window Declaration
Queue:ASCII        QUEUE
                  STRING(255)
                END
ASC1:FileName      STRING(80)
ASC1:CurrentFileName STRING(80)
ASC1:ASCIIFile       FILE,PRE(ASC1),DRIVER('ASCII'),NAME(ASCIIFileName)
                   RECORD,PRE()
STRING               STRING(255)
                   END
                  END
ASC1:ReportCounter LONG
ASC1:Report REPORT,AT(1000,2500,6000,6000),FONT('Fixedsys',9,,FONT:regular),THOUS
Detail             DETAIL,AT(,,,167)
                     STRING(@s255),AT(104,0,,),USE(Queue:ASCII)
                   END
                  END
ASC1:WholeWord     BYTE
ASC1:Matchcase     BYTE
ASC1:Direction     CSTRING('Down')
ASC1:SearchString  CSTRING(80)
ASC1:CurrentPointer LONG
ASC1:TextLocation  LONG
ASC1:SearchWindow WINDOW('Searching Text...'),AT(43,25,267,60),FONT('MS Sans
Serif',8,,),GRAY
                    PROMPT('Find What:'),AT(11,5,,),USE(?ASC1:TextPrompt)
                    ENTRY(@s20),AT(53,5,149,15),USE(ASC1:SearchString)
                    CHECK('Match &Whole Word Only'),AT(11,30,,),USE(ASC1:WholeWord)
                    CHECK('Match &Case'),AT(11,44,,),USE(ASC1:MatchCase)
                    OPTION('Direction'),AT(111,28,81,26),USE(ASC1:Direction),BOXED
                      RADIO('Up'),AT(117,39,,)
                      RADIO('Down'),AT(149,39,,)
                    END
```

```
                    BUTTON('Find'),AT(208,5,53,15),USE(?ASC1:Search),DEFAULT
                    BUTTON('Cancel'),AT(208,25,53,15),USE(?ASC1:CancelSearch)
                    END
ViewWindow          WINDOW('View an ASCII File'),AT(3,7,296,136),SYSTEM,GRAY,MAX,IMM
                    LIST,AT(5,5,285,110),FONT('FixedSys',9,,FONT:regular),USE(?
AsciiBox),HVSCROLL,FROM(Queue:ASCII)
                    BUTTON('&Print'),AT(135,120,35,10),USE(?ASCIIPrint)
                    BUTTON('&Find...'),AT(174,120,35,10),USE(?ASCIISearch)
                    BUTTON('Find Next'),AT(213,120,39,10),USE(?ASCIIRepeat)
                    BUTTON('&Close'),AT(255,120,35,10),USE(?Close)
                  END
!Embed: Data Section, After Window Declaration
Progress:Thermometer BYTE
ProgressWindow WINDOW('Progress...'),AT(,,142,59),CENTER,TIMER(1),GRAY,DOUBLE
     PROGRESS,USE(Progress:Thermometer),AT(15,15,111,12),RANGE(0,100)
     STRING(''),AT(0,3,141,10),USE(?Progress:UserString),CENTER
     STRING(''),AT(0,30,141,10),USE(?Progress:PctText),CENTER
     BUTTON('Cancel'),AT(45,42,50,15),USE(?Progress:Cancel)
   END
  CODE
  !Embed: Initialize the Procedure
  LocalRequest = GlobalRequest
  OriginalRequest = GlobalRequest
  LocalResponse = RequestCancelled
  ForceRefresh = False
  CLEAR(GlobalRequest)
  CLEAR(GlobalResponse)
  !Embed: Procedure Setup
  IF KEYCODE() = MouseRight
    SETKEYCODE(0)
  END
  !Embed: Beginning of Procedure, Before Opening Files
  !Embed: Beginning of Procedure, After Opening Files
  !Embed: Before Opening the Window
  OPEN(ViewWindow)
  WindowOpened=True
  !Embed: After Opening the Window
  !Embed: Preparing Window Alerts
  !Embed: Preparing to Process the Window
  ?ASCIIRepeat{Prop:Disable}=True
  ACCEPT
    !Embed: Accept Loop, Before CASE EVENT() handling
    CASE EVENT()
    !Embed: CASE EVENT() structure, before generated code
    OF EVENT:AlertKey
      !Embed: Window Event Handling AlertKey
    OF EVENT:PreAlertKey
      !Embed: Window Event Handling PreAlertKey
    OF EVENT:CloseWindow
      !Embed: Window Event Handling CloseWindow
    OF EVENT:CloseDown
      !Embed: Window Event Handling CloseDown
    OF EVENT:OpenWindow
      !Embed: Window Event Handling OpenWindow
      IF NOT WindowInitialized
        DO InitializeWindow
        WindowInitialized = True
      END
      SELECT(?AsciiBox)
    OF EVENT:LoseFocus
```

```
  !Embed: Window Event Handling LoseFocus
OF EVENT:GainFocus
  !Embed: Window Event Handling GainFocus
  ForceRefresh = True
  IF NOT WindowInitialized
    DO InitializeWindow
    WindowInitialized = True
  ELSE
    DO RefreshWindow
  END
OF EVENT:Suspend
  !Embed: Window Event Handling Suspend
OF EVENT:Resume
  !Embed: Window Event Handling Resume
ELSE
  !Embed: Other Window Event Handling
!Embed: CASE EVENT() structure, after generated code
END
!Embed: Accept Loop, After CASE EVENT() handling
!Embed: Accept Loop, Before CASE FIELD() handling
CASE FIELD()
!Embed: CASE FIELD() structure, before generated code
OF ?AsciiBox
  !Embed: Control Handling, before event handling ?AsciiBox
  CASE EVENT()
  OF EVENT:Accepted
    !Embed: Control Event Handling, before generated code ?AsciiBox, Accepted
    !Embed: Internal Control Event Handling ?AsciiBox, Accepted
    !Embed: Control Event Handling, after generated code ?AsciiBox, Accepted
  OF EVENT:NewSelection
    !Embed: Control Event Handling, before generated code ?AsciiBox, NewSelection
    !Embed: Internal Control Event Handling ?AsciiBox, NewSelection
    !Embed: Control Event Handling, after generated code ?AsciiBox, NewSelection
  OF EVENT:ScrollUp
    !Embed: Control Event Handling, before generated code ?AsciiBox, ScrollUp
    !Embed: Internal Control Event Handling ?AsciiBox, ScrollUp
    !Embed: Control Event Handling, after generated code ?AsciiBox, ScrollUp
  OF EVENT:ScrollDown
    !Embed: Control Event Handling, before generated code ?AsciiBox, ScrollDown
    !Embed: Internal Control Event Handling ?AsciiBox, ScrollDown
    !Embed: Control Event Handling, after generated code ?AsciiBox, ScrollDown
  OF EVENT:PageUp
    !Embed: Control Event Handling, before generated code ?AsciiBox, PageUp
    !Embed: Internal Control Event Handling ?AsciiBox, PageUp
    !Embed: Control Event Handling, after generated code ?AsciiBox, PageUp
  OF EVENT:PageDown
    !Embed: Control Event Handling, before generated code ?AsciiBox, PageDown
    !Embed: Internal Control Event Handling ?AsciiBox, PageDown
    !Embed: Control Event Handling, after generated code ?AsciiBox, PageDown
  OF EVENT:ScrollTop
    !Embed: Control Event Handling, before generated code ?AsciiBox, ScrollTop
    !Embed: Internal Control Event Handling ?AsciiBox, ScrollTop
    !Embed: Control Event Handling, after generated code ?AsciiBox, ScrollTop
  OF EVENT:ScrollBottom
    !Embed: Control Event Handling, before generated code ?AsciiBox, ScrollBottom
    !Embed: Internal Control Event Handling ?AsciiBox, ScrollBottom
    !Embed: Control Event Handling, after generated code ?AsciiBox, ScrollBottom
  OF EVENT:Locate
    !Embed: Control Event Handling, before generated code ?AsciiBox, Locate
    !Embed: Internal Control Event Handling ?AsciiBox, Locate
```

```
    !Embed: Control Event Handling, after generated code ?AsciiBox, Locate
  OF EVENT:Selected
    !Embed: Control Event Handling, before generated code ?AsciiBox, Selected
    !Embed: Internal Control Event Handling ?AsciiBox, Selected
    !Embed: Control Event Handling, after generated code ?AsciiBox, Selected
  ELSE
    !Embed: Other Control Event Handling ?AsciiBox
  END
  !Embed: Control Handling, after event handling ?AsciiBox
OF ?ASCIIPrint
  !Embed: Control Handling, before event handling ?ASCIIPrint
  CASE EVENT()
  OF EVENT:Accepted
    !Embed: Control Event Handling, before generated code ?ASCIIPrint, Accepted
    DO SyncWindow
    !Embed: Internal Control Event Handling ?ASCIIPrint, Accepted
    SETCURSOR(CURSOR:Wait)
    ASC1:ReportCounter = 0
    OPEN(ASC1:Report)
    LOOP
      ASC1:ReportCounter += 1
      IF ASC1:ReportCounter > RECORDS(Queue:ASCII)
        BREAK
      END
      GET(Queue:ASCII,ASC1:ReportCounter)
      PRINT(DETAIL)
    END
    CLOSE(ASC1:Report)
    SETCURSOR
    !Embed: Control Event Handling, after generated code ?ASCIIPrint, Accepted
  OF EVENT:Selected
    !Embed: Control Event Handling, before generated code ?ASCIIPrint, Selected
    !Embed: Internal Control Event Handling ?ASCIIPrint, Selected
    !Embed: Control Event Handling, after generated code ?ASCIIPrint, Selected
  ELSE
    !Embed: Other Control Event Handling ?ASCIIPrint
  END
  !Embed: Control Handling, after event handling ?ASCIIPrint
OF ?ASCIISearch
  !Embed: Control Handling, before event handling ?ASCIISearch
  CASE EVENT()
  OF EVENT:Accepted
    !Embed: Control Event Handling, before generated code ?ASCIISearch, Accepted
    DO SyncWindow
    !Embed: Internal Control Event Handling ?ASCIISearch, Accepted
    OPEN(ASC1:SearchWindow)
    ACCEPT
      CASE EVENT()
      OF Event:OpenWindow
        CLEAR(ASC1:SearchString)
      OF Event:CloseWindow
        CLOSE(ASC1:SearchWindow)
      OF Event:Accepted
        CASE FIELD()
        OF ?ASC1:Search
          LocalResponse = RequestCompleted
          POST(Event:CloseWindow)
        OF ?ASC1:CancelSearch
          LocalResponse = RequestCancelled
          POST(Event:CloseWindow)
```

```
          END
        END
      END
      IF LocalResponse = RequestCompleted
        DO ASC1:FindText
        ?ASCIIRepeat{Prop:Disable}=False
      ELSE
        ?ASCIIRepeat{Prop:Disable}=True
      END
      !Embed: Control Event Handling, after generated code ?ASCIISearch, Accepted
    OF EVENT:Selected
      !Embed: Control Event Handling, before generated code ?ASCIISearch, Selected
      !Embed: Internal Control Event Handling ?ASCIISearch, Selected
      !Embed: Control Event Handling, after generated code ?ASCIISearch, Selected
    ELSE
      !Embed: Other Control Event Handling ?ASCIISearch
    END
    !Embed: Control Handling, after event handling ?ASCIISearch
  OF ?ASCIIRepeat
    !Embed: Control Handling, before event handling ?ASCIIRepeat
    CASE EVENT()
    OF EVENT:Accepted
      !Embed: Control Event Handling, before generated code ?ASCIIRepeat, Accepted
      DO SyncWindow
      !Embed: Internal Control Event Handling ?ASCIIRepeat, Accepted
      DO ASC1:FindText
      !Embed: Control Event Handling, after generated code ?ASCIIRepeat, Accepted
    OF EVENT:Selected
      !Embed: Control Event Handling, before generated code ?ASCIIRepeat, Selected
      !Embed: Internal Control Event Handling ?ASCIIRepeat, Selected
      !Embed: Control Event Handling, after generated code ?ASCIIRepeat, Selected
    ELSE
      !Embed: Other Control Event Handling ?ASCIIRepeat
    END
    !Embed: Control Handling, after event handling ?ASCIIRepeat
  OF ?Close
    !Embed: Control Handling, before event handling ?Close
    CASE EVENT()
    OF EVENT:Accepted
      !Embed: Control Event Handling, before generated code ?Close, Accepted
      DO SyncWindow
      !Embed: Internal Control Event Handling ?Close, Accepted
      LocalResponse = RequestCancelled
      POST(Event:CloseWindow)
      !Embed: Control Event Handling, after generated code ?Close, Accepted
    OF EVENT:Selected
      !Embed: Control Event Handling, before generated code ?Close, Selected
      !Embed: Internal Control Event Handling ?Close, Selected
      !Embed: Control Event Handling, after generated code ?Close, Selected
    ELSE
      !Embed: Other Control Event Handling ?Close
    END
    !Embed: Control Handling, after event handling ?Close
  !Embed: CASE FIELD() structure, after generated code
  END
  !Embed: Accept Loop, After CASE FIELD() handling
 END
 DO ProcedureReturn
!---------------------------------------------------------------------------
ProcedureReturn ROUTINE
```

```
  !Embed: End of Procedure, Before Closing Files
  !Embed: End of Procedure, After Closing Files
  !Embed: Before Closing the Window
  IF WindowOpened
    CLOSE(ViewWindow)
  END
  !Embed: After Closing the Window
  !Embed: End of Procedure
  IF LocalResponse
    GlobalResponse = LocalResponse
  ELSE
    GlobalResponse = RequestCancelled
  END
  RETURN
!---------------------------------------------------------------------------
InitializeWindow ROUTINE
  !Embed: Window Initialization Code
  DO RefreshWindow
!---------------------------------------------------------------------------
RefreshWindow ROUTINE
  IF ViewWindow{Prop:AcceptAll} THEN EXIT.
  !Embed: Refresh Window routine, before lookups
  !Embed: Lookup Related Records
  !Embed: Refresh Window routine, after lookups
  !Embed: After Refresh Window for a control ?Close
  ASC1:FileName = 'c:\win95\win.ini'
  IF ASC1:FileName <> ASC1:CurrentFileName
    ASCIIFileName = ASC1:FileName
    DO ASC1:FillQueue
    ASC1:CurrentFileName = ASC1:Filename
  END
  !Embed: After Refresh Window for a control ?Close
  !Embed: Refresh Window routine, before DISPLAY()
  DISPLAY()
  ForceRefresh = False
!---------------------------------------------------------------------------
SyncWindow ROUTINE
  !Embed: Sync Record routine, before lookups
  !Embed: Lookup Related Records
  !Embed: Sync Record routine, after lookups
!---------------------------------------------------------------------------
!Embed: Procedure Routines
ASC1:FillQueue ROUTINE
  FREE(Queue:ASCII)
  IF NOT ASCIIFileName
    ?AsciiBox{Prop:Disable} = True
    EXIT
  ELSE
    ?AsciiBox{Prop:Disable} = False
  END
  OPEN(ASC1:ASCIIFile,10h)
  IF ERRORCODE()
    DISABLE(?AsciiBox)
    !Embed: ASCII Box, File not found 1
    IF StandardWarning(Warn:FileLoadError,ASC1:FileName)
      EXIT
    END
  ELSE
    ENABLE(?AsciiBox)
    !Embed: ASCII Box, File found 1
```

```
  END
  ASCIIFileSize = BYTES(ASC1:ASCIIFile)
  IF ASCIIFileSize = 0
    CLOSE(ASC1:ASCIIFile)
    DISABLE(?AsciiBox)
    IF StandardWarning(Warn:FileZeroLength,ASC1:FileName)
      EXIT
    END
    EXIT
  END
  OPEN(ProgressWindow)
  ASCIIPercentProgress = 0
  ASCIIBytesRead = 0
  ProgressWindow{Prop:Text} = 'Reading File'
  Progress:Thermometer = 0
  ?Progress:PctText{Prop:Text} = '0% Completed'
  ?Progress:UserString{Prop:Text} = ''
  ACCEPT
    CASE EVENT()
    OF Event:OpenWindow
      SET(ASC1:ASCIIFile)
    OF Event:Timer
      ASCIIBytesThisCycle = 0
      LOOP WHILE ASCIIBytesThisCycle < 20000
        NEXT(ASC1:ASCIIFile)
        IF ERRORCODE()
          LocalResponse = RequestCompleted
          BREAK
        END
        ASCIIBytesThisRead = BYTES(ASC1:ASCIIFile)
        ASCIIBytesThisCycle += ASCIIBytesThisRead
        ASCIIBytesRead += ASCIIBytesThisRead
        Queue:ASCII = ASC1:String
        ADD(Queue:ASCII)
      END
      IF ASCIIPercentProgress < 100
        ASCIIPercentProgress = (ASCIIBytesRead/ASCIIFileSize)*100
        IF ASCIIPercentProgress > 100
          ASCIIPercentProgress = 100
        END
        IF Progress:Thermometer <> ASCIIPercentProgress THEN
          Progress:Thermometer = ASCIIPercentProgress
          ?Progress:PctText{Prop:Text} = FORMAT(ASCIIPercentProgress,@N3) & '% Completed
('  & ASCIIBytesRead & ') bytes'
          DISPLAY(?Progress:Thermometer)
          DISPLAY(?Progress:PctText)
        END
      END
      IF LocalResponse = RequestCompleted
        LocalResponse = RequestCancelled
        POST(Event:CloseWindow)
      END
    END
    CASE FIELD()
    OF ?Progress:Cancel
      CASE EVENT()
      OF Event:Accepted
        IF StandardWarning(Warn:ConfirmCancelLoad,ASC1:FileName)=Button:OK
          POST(Event:CloseWindow)
        END
```

```
      END
    END
  END
  CLOSE(ProgressWindow)
  CLOSE(ASC1:ASCIIFile)
!-----------------------------------------------------------------
ASC1:FindText ROUTINE
  ASC1:CurrentPointer = CHOICE(?AsciiBox)
  SETCURSOR(CURSOR:Wait)
  LOOP
    IF ASC1:Direction = 'Down'
      ASC1:CurrentPointer += 1
      IF ASC1:CurrentPointer = RECORDS(Queue:ASCII)
        IF StandardWarning(Warn:EndOfASCIIQueue,'Down') = Button:Yes
          ASC1:CurrentPointer = 1
        ELSE
          BREAK
        END
      END
    ELSE
      ASC1:CurrentPointer -= 1
      IF ASC1:CurrentPointer = 0
        IF StandardWarning(Warn:EndOfASCIIQueue,'Up') = Button:Yes
          ASC1:CurrentPointer = RECORDS(Queue:ASCII)
        ELSE
          BREAK
        END
      END
    END
    GET(Queue:ASCII,ASC1:CurrentPointer)
    IF ASC1:MatchCase
      ASC1:TextLocation = INSTRING(ASC1:SearchString,Queue:ASCII,1,1)
    ELSE
      ASC1:TextLocation = INSTRING(UPPER(ASC1:SearchString),UPPER(Queue:ASCII),1,1)
    END
    IF NOT ASC1:TextLocation
      CYCLE
    END
    IF ASC1:WholeWord
      IF ASC1:TextLocation > 1
        IF SUB(Queue:ASCII,ASC1:TextLocation-1,1)
          CYCLE
        END
      END
      IF ASC1:TextLocation+LEN(CLIP(ASC1:SearchString))<LEN(CLIP(Queue:ASCII))
        IF SUB(Queue:ASCII,ASC1:TextLocation+LEN(CLIP(ASC1:SearchString)),1)
          CYCLE
        END
      END
    END
    ?AsciiBox{Prop:SelStart} = ASC1:CurrentPointer
    BREAK
  END
  SETCURSOR
```

Example Source for Window Procedure

The following code was generated from the Window procedure template.

Comments appear in **red** at the default embed points.

```
                MEMBER('Prog.clw')        ! This is a MEMBER module
!Embed: Compile Module Declarations
!Embed: Compile Module Declarations
!Embed: Module Data Section
!Embed: Module Data Section
!Embed: Gather Template Symbols
WindProc PROCEDURE

LocalRequest       LONG,AUTO
OriginalRequest      LONG,AUTO
LocalResponse      LONG,AUTO
WindowOpened       LONG
WindowInitialized    LONG
ForceRefresh       LONG,AUTO
!Embed: Data Section, Before Window Declaration
window             WINDOW('Caption'),AT(,,185,92),SYSTEM,GRAY,RESIZE,MDI
                   END
!Embed: Data Section, After Window Declaration
  CODE
  !Embed: Initialize the Procedure
  LocalRequest = GlobalRequest
  OriginalRequest = GlobalRequest
  LocalResponse = RequestCancelled
  ForceRefresh = False
  CLEAR(GlobalRequest)
  CLEAR(GlobalResponse)
  !Embed: Procedure Setup
  IF KEYCODE() = MouseRight
    SETKEYCODE(0)
  END
  !Embed: Beginning of Procedure, Before Opening Files
  !Embed: Beginning of Procedure, After Opening Files
  !Embed: Before Opening the Window
  OPEN(window)
  WindowOpened=True
  !Embed: After Opening the Window
  !Embed: Preparing Window Alerts
  !Embed: Preparing to Process the Window
  ACCEPT
    !Embed: Accept Loop, Before CASE EVENT() handling
    CASE EVENT()
    !Embed: CASE EVENT() structure, before generated code
    OF EVENT:AlertKey
      !Embed: Window Event Handling AlertKey
    OF EVENT:PreAlertKey
      !Embed: Window Event Handling PreAlertKey
    OF EVENT:CloseWindow
      !Embed: Window Event Handling CloseWindow
    OF EVENT:CloseDown
      !Embed: Window Event Handling CloseDown
    OF EVENT:OpenWindow
      !Embed: Window Event Handling OpenWindow
      IF NOT WindowInitialized
```

```
      DO InitializeWindow
      WindowInitialized = True
    END
  OF EVENT:LoseFocus
    !Embed: Window Event Handling LoseFocus
  OF EVENT:GainFocus
    !Embed: Window Event Handling GainFocus
    ForceRefresh = True
    IF NOT WindowInitialized
      DO InitializeWindow
      WindowInitialized = True
    ELSE
      DO RefreshWindow
    END
  OF EVENT:Suspend
    !Embed: Window Event Handling Suspend
  OF EVENT:Resume
    !Embed: Window Event Handling Resume
  ELSE
    !Embed: Other Window Event Handling
  !Embed: CASE EVENT() structure, after generated code
  END
  !Embed: Accept Loop, After CASE EVENT() handling
  !Embed: Accept Loop, Before CASE FIELD() handling
  CASE FIELD()
  !Embed: CASE FIELD() structure, before generated code
  !Embed: CASE FIELD() structure, after generated code
  END
  !Embed: Accept Loop, After CASE FIELD() handling
 END
 DO ProcedureReturn
!---------------------------------------------------------------------------
ProcedureReturn ROUTINE
 !Embed: End of Procedure, Before Closing Files
 !Embed: End of Procedure, After Closing Files
 !Embed: Before Closing the Window
 IF WindowOpened
   CLOSE(window)
 END
 !Embed: After Closing the Window
 !Embed: End of Procedure
 IF LocalResponse
   GlobalResponse = LocalResponse
 ELSE
   GlobalResponse = RequestCancelled
 END
 RETURN
!---------------------------------------------------------------------------
InitializeWindow ROUTINE
 !Embed: Window Initialization Code
 DO RefreshWindow
!---------------------------------------------------------------------------
RefreshWindow ROUTINE
 IF window{Prop:AcceptAll} THEN EXIT.
 !Embed: Refresh Window routine, before lookups
 !Embed: Lookup Related Records
 !Embed: Refresh Window routine, after lookups
 !Embed: Refresh Window routine, before DISPLAY()
 DISPLAY()
 ForceRefresh = False
```

```
!-----------------------------------------------------------------------------
SyncWindow ROUTINE
  !Embed: Sync Record routine, before lookups
  !Embed: Lookup Related Records
  !Embed: Sync Record routine, after lookups
!-----------------------------------------------------------------------------
!Embed: Procedure Routines
```

Example Source for ToDo Procedure

The following code was generated from the ToDo procedure template.

Comments appear in **red** at the default embed points.

```
                MEMBER('Prog.clw')        ! This is a MEMBER module
!Embed: Compile Module Declarations
!Embed: Module Data Section
MyToDo PROCEDURE
  CODE
  IF StandardWarning(Warn:ProcedureToDo,' MyToDo ')
    RETURN
  END
```

Example Source for...

Global Program Section

Browse Procedure

Form Procedure

Frame Procedure

Menu Procedure

Process Procedure

Report Procedure

Source Procedure

Viewer Procedure

Window Procedure

ToDo Procedure

Grid Size Dialog

This allows you to set the spacing between grid points. When you place a new control in the report, you can then optionally force it to the grid points with the **Option**  **Snap to Grid** command.

Set the spacing by typing in values for the **Horizontal** and **Vertical** spacing. The measurement units depend on the default for the report, as set in the **Report Properties** dialog. The **Grid Size** dialog tells you the minimum value (1/20th inch, or 2 millimeters).

Preview Print Details Dialog

This allows you to choose a report section to preview.

Because you can nest many sections of various types within a single report, you have to select a section before actually previewing it. This way, the Report Formatter knows what part(s) you want to compose on the screen.

Select a section from the **Details** list, then press the **Add** button to move it to the **Selected Details** list. Press the **OK** button to preview.

Database Drivers

Clarion for Windows achieves database independence with its built-in driver technology, enabling you to access data from virtually any file system. Many file drivers are available and more are being added. All of the file drivers read and write in the file system's native format without temporary files or import/export routines.

Often, your application's main purpose is accessing existing data in its original format. For those times, you just plug in the appropriate file driver. For the times when you're not "locked into" a particular file system, this appendix provides tips on the file drivers best suited for different jobs. You can choose the right tool depending on the type, size, and nature of the data files necessary for your application.

The commands for accessing data from different systems are the same; simply choose the correct file driver from the drop down list within your Data Dictionary, and don't worry about it.

See Also:

Supported File Systems

Optimizing File Drivers with Driver Strings

Supported File Systems

ASCII Files

Basic Files

Btrieve Files

Clarion Files

Clipper Files

dBase III Files

dBase IV Files

DOS Files

FoxPro and FoxBase Files

TopSpeed Database Files

Optimizing File Drivers with Driver Strings

There are settings you can specify, with driver strings (the second parameter of the DRIVER attribute), to optimize the way your application creates, reads, and writes data files for a specific driver. To specify a driver string with the Data Dictionary, type it in the **Options** field in the *New File Properties* dialog, as described in the *Using the Dictionary* chapter. To send a driver string in executable code (after the application initializes the driver), use the SEND() function, described in the *Language Reference*.

Driver strings are all preceded by a forward slash character (/). SEND function commands can take two formatsone with an equal sign to modify a switch setting, the other without an equal sign to return the value of the switch.

Driver strings are sent to the file driver when the file is opened. The SEND function sends a command to modify a setting. Some driver strings have no effect after the file is open, so the SEND function syntax to modify the setting is not listed. However, the SEND function to return the value of the switch is valid for all driver strings.

ASCII Files

The ASCII driver reads and writes standard ASCII files without field delimiters. This is often used for mainframe data import/export via an ASCII flat-file. A carriage-return/line-feed delimits records. The ASCII driver does not support keys.

Files:	**CWASC16.LIB**	Windows Export Library (16-bit)
	CWASC32.LIB	Windows Export Library (32-bit)
	CLASC16.LIB	Windows Static Link Library (16-bit)
	CLASC32.LIB	Windows Static Link Library (32-bit)
	CWASC16.DLL	Windows Dynamic Link Library (16-bit)
	CWASC32.DLL	Windows Dynamic Link Library (32-bit)

Tip: **Due to its lack of relational features and security (anyone can view and change an ASCII file using Notepad), it's unlikely you'll use the ASCII driver to store large data files. But it can help you create a text file vieweruse it to open a file, and read it in to a multi-line edit or listbox control!**

ASCII:Supported Data Types

STRING
GROUP

ASCII:File Specifications/Maximums

File Size:	**4,294,967,295 bytes**
Records per File:	4,294,967,295 bytes
Record Size:	65,520 bytes
Field Size:	65,520 bytes
Fields per Record:	65,520 bytes
Keys/Indexes per File:	n/a
Key Size:	n/a
Memo fields per File:	n/a
Memo Field Size:	n/a
Open Data Files:	Operating system dependent

ASCII:Driver Strings and SEND functions

Driver strings (the second parameter of the DRIVER attribute) are all preceded by a forward slash character (/). SEND function commands can take two formatsone with an equal sign modifies a switch setting and return the value of the previous switch setting; the other format (without an equal sign) returns the value of the switch.

Driver strings are sent to the file driver when the file is opened. The SEND function sends a command to modify a setting after the file is open. Some driver strings have no effect after the file is open, so the SEND function syntax to modify the setting is not listed. However, the SEND function syntax to return the value of the switch is listed for all driver strings.

/FILEBUFFERS=n

Specifies a value for the number of buffers used to read and write to the file. The ASCII driver allocates internal buffers of 512 bytes, or the size of the record, whichever is larger. The default number of buffers is 2 for files opened denying write access to other users, and 1 for all other open modes. Use the optional driver string to increase the buffers should you find access to records is slow.

SEND(file, 'FILEBUFFERS')

Returns a STRING containing the number of bytes in the buffers in STRING format

/TAB= n

Specifies TAB/SPACE expansion. The ASCII driver expands TABs (ASCII character 9) to spaces when reading. The value indicates the number of spaces with which to replace the tab, subject to the guidelines below. The default value is 8.

If n > 0 , spaces replace each tab until the character pointer moves to the next multiple of *n*. For example, with the default of 8, if the TAB character is the third character in the record, 6 spaces replace the TAB.

If n = 0, the driver removes tabs without replacement.

If n < 0, the driver removes tabs with the positive value of *n* spaces. For example, "TAB=-4" causes 4 spaces to replace every tab, regardless of the position of the tab in the record.

If n = -100, tabs remain as tabs; the driver *does not* replace them with spaces.

SEND(file, 'TAB')

Returns the number of spaces which replace the tab character in the form of a STRING.

/ENDOFRECORD=n,<m>

Specifies the end of record delimiter.

n represents the number of characters that make up the end-of-record separator.

m represents the ASCII code(s) for the end-of-record characters, separated by commas. The default is 2,13,10, indicating 2 characters mark the end-of-record, namely, carriage return (13) and line feed (10).

SEND(file,'ENDOFRECORD')

Returns the end of record delimiter in the form of a STRING.

Tip: Mainframes frequently use a carriage return to delimit records. You can use /ENDOFRECORD to read these files.

/QUICKSCAN=on|off

SEND(file,'QUICKSCAN=on|off')

Specifies buffered access behavior. The ASCII driver reads a buffer at a time (not a record), allowing for fast access. In a multi-user environment these buffers are not 100% trustworthy for subsequent access, because another user may change the file between accesses. As a safeguard, the driver rereads the buffers before each record access. To *disable* the reread, set QUICKSCAN to ON. The default is ON for files opened denying write access to other users, and OFF for all other open modes.

SEND(file,'QUICKSCAN')

Returns the Quickscan setting (ON or OFF) in the form of a STRING(3).

/CLIP=on|off

The driver automatically removes trailing spaces from a record before writing it to file. To disable this feature, set CLIP to OFF. The default is ON.

SEND(file,'CLIP')

Returns the CLIP setting (ON or OFF) in the form of a STRING(3).

ASCII:Unsupported Functions and Attributes

Memos:	**NOMEMO()**
Transaction Processing:	**COMMIT(), LOGOUT(), ROLLBACK()**
Key Processing:	**BUILD(key), BUILD(index)** **GET(file,key), GET(key,keypointer)** **RESET(key,string)** **SET(file,key), SET(key), SET(key,key), SET(key,keypointer), SET(key,key,filepointer)** **DUPLICATE()** **POINTER(key)** **POSITION(key)** **RECORDS(key)** **REGET(key,string)**
Record Locking:	**HOLD(), RELEASE()**
File Buffering:	**STREAM()**
File Information:	**RECORDS(file)**
Sequential Processing:	**PREVIOUS(), BOF(), SKIP()**
File Manipulation:	**BUILD(), DELETE(), PACK(), WATCH(), REGET()**

ASCII:Miscellaneous

POSITION(file) returns a STRING(4).

Basic Files

The BASIC file driver reads and writes comma delimited ASCII files. Quotes (" ") surround strings, commas delimit fields, and a carriage-return/line-feed delimits records. The original BASIC programming language defined this file format. The Basic driver does not support keys.

Tip: The Basic file format provides a good choice for a common file format for sharing data with spreadsheet programs. A common file extension used for these files is *.CSV, which stands for "comma separated values."

Files:	**CWBAS16.LIB**	Windows Export Library (16-bit)
	CWBAS32.LIB	Windows Export Library (32-bit)
	CLBAS16.LIB	Windows Static Link Library (16-bit)
	CLBAS32.LIB	Windows Static Link Library (32-bit)
	CWBAS16.DLL	Windows Dynamic Link Library (16-bit)
	CWBAS32.DLL	Windows Dynamic Link Library (32-bit)

Basic:Supported Data Types

BYTE DECIMAL
SHORT PDECIMAL
USHORT STRING
LONG CSTRING
ULONG PSTRING
SREAL DATE
REAL TIME
BFLOAT4 GROUP
BFLOAT8

Basic:File Specifications/Maximums

File Size:	**4,294,967,295 bytes**
Records per File:	4,294,967,295 bytes
Record Size:	65,520 bytes
Field Size:	65,520 bytes
Fields per Record:	65,520 bytes
Keys/Indexes per File:	n/a
Key Size:	n/a
Memo fields per File:	0
Memo Field Size:	n/a
Open Data Files:	Operating system dependent

Basic:Driver Strings and SEND functions

Driver strings (the second parameter of the DRIVER attribute) are all preceded by a forward slash character (/). SEND function commands can take two formatsone with an equal sign modifies a switch setting and return the value of the previous switch setting; the other format (without an equal sign) returns the value of the switch.

Driver strings are sent to the file driver when the file is opened. The SEND function sends a command to modify a setting after the file is open. Some driver strings have no effect after the file is open, so the SEND function syntax to modify the setting is not listed. However, the SEND function syntax to return the value of the switch is listed for all driver strings.

/FILEBUFFERS=n Specifies a value for the number of buffers used to read and write to the file.

The Basic driver allocates internal buffers of 512 bytes, or the size of your record, whichever is larger, to store the retrieved data. The default number of buffers is 2 for files opened denying write access to other users, and 1 for all other open modes. Use the optional driver string to increase the buffers should you find access to records is slow.

SEND(file, 'FILEBUFFERS')

Returns the number of buffers in the form of a STRING.

/ENDOFRECORD=n,<m> Specifies the end of record delimiter.

n represents the number of characters of the end-of-record separator.

m represents the ASCII code(s) for the end-of-record characters, separated by commas. The default is 2,13,10 indicating 2 characters mark the end-of-record, namely, carriage return (13) and line feed (10).

SEND(file,'ENDOFRECORD')

Returns the end of record delimiter in the form of a STRING.

Tip: Mainframes frequently use a carriage return to delimit records. You can use /ENDOFRECORD to read these files.

/QUICKSCAN=on|off

SEND(file,'QUICKSCAN=ON|OFF')

Specifies buffered access behavior.

The Basic driver reads a buffer at a time (not a record), allowing for fast access. In a multi-user environment these buffers are not 100% trustworthy for subsequent access, because another user may change the database between accesses. As a safeguard, the driver rereads the buffers before each record access. To *disable* the reread, set QUICKSCAN to ON. The default is ON for files opened denying write access to other users, and OFF for all other open modes.

SEND(file,'QUICKSCAN')

Returns the Quickscan setting (ON or OFF) in the form of a STRING(3).

/FIELDDELIMITER=n,<m> Specifies the end-of-field separator.

n represents the number of characters that make up the end-of-field separator.

m represents the ASCII code(s) for the end-of-field characters, separated by commas. The default is 1,44, which indicates the "comma" character.

SEND(file,'FIELDDELIMITER')

Returns the value of the field delimiter in the form of a STRING.

/COMMA=n

Specifies a single character end-of-field separator. *n* represents the ASCII code for the end-of-field character. The default is 44, which is equivalent to "/FIELDDELIMITER=1,44."

SEND(file,'COMMA')

Returns the ASCII code for the single character end-of-field delimiter in the form of a STRING.

Tip: TAB-delimited values are a common format compatible with the Windows clipboard. Using the BASIC file driver string /COMMA=9 allows you to read Windows clipboard files

/QUOTE=n

Specifies a single character string delimiter. *n* represents the ASCII code. The default is 34, the ASCII value for the quotation mark character.

SEND(file,'QUOTE')

Returns the ASCII code value of the single character string delimiter in the form of a STRING.

/ALWAYSQUOTE=on|off

For compatibility with Basic format data files created by products which do *not* place string values in quotes, set ALWAYSQUOTE to off.

When the contents of a string field includes the comma or quote character(s), and ALWAYSQUOTE is off, the Basic driver automatically places quotes around the string when writing to file. This also applies to delimiter characters specified with FIELDDELIMITER, or COMMA. For example, with the defaults in use and ALWAYSQUOTE off, a STRING field containing the value *1313 Mockingbird Lane, Apt. 33* is automatically stored as: "1313 Mockingbird Lane, Apt. 33"

SEND(file,'ALWAYSQUOTE')

Returns the ALWAYSQUOTE setting (ON or OFF) in the form of a STRING(3).

Basic:Unsupported Functions and Attributes

Memos:	**NOMEMO()**
Transaction Processing:	**COMMIT(), LOGOUT(), ROLLBACK()**
Key Processing:	**BUILD(key), BUILD(index)** **GET(file,key), GET(key,keypointer)** **RESET(key,string)** **SET(file,key), SET(key), SET(key,key), SET(key,keypointer), SET(key,key,filepointer)** **DUPLICATE()** **POINTER(key)** **POSITION(key)** **RECORDS(key)** **REGET(key,string)**
Record Locking:	**HOLD(), RELEASE()**
File Buffering:	**STREAM()**
File Information:	**RECORDS(file)**
Sequential Processing:	**PREVIOUS(), BOF(), SKIP()**
File Manipulation:	**BUILD(), DELETE(), PACK(), WATCH(), REGET()**

Basic:Miscellaneous

The following demonstrates how to use the driver strings to create two popular file formats:

- Microsoft Word for Windows Mail Merge:

/ALWAYSQUOTE=OFF
/FIELDDELIMITER=2,13,7
/ENDOFRECORD=4,13,7,13,7

- TAB delimited format:

/COMMA=9

- POSITION(file) returns a STRING(4).

Btrieve Files

This file driver reads and writes Btrieve files, using low-level direct access.

Under Clarion for Windows, the Btrieve file driver is implemented by using .DLLs and an .EXE supplied by Btrieve Technologies, Inc. (BTI). For an application to use a Btrieve file driver, the following BTI files must accompany the executable:

16-bit

WBTR32.EXE

WBTRLOCL.DLL

WBTRCALL.DLL

WBTRVRES.DLL

32-bit

Filenames were not available at press time.

Contact BTI for more information.

LICENSE WARNING: A registered Clarion for Windows owner cannot redistribute the above BTI files outside of his/her organization without a license from BTI. In order to obtain a license, please contact:

Btrieve Technologies, Inc.

5918 West Courtyard Drive, Suite 400

Austin, Texas 78730

Phone: (512)794-1719

For Client/Server-based Btrieve, Netware Btrieve is a server-based version of Btrieve that runs on a Novell server. The Btrieve requester program BREQUEST.EXE must be loaded at each workstation before Windows is started.

A single file normally holds the data and all keys. Data filenames default to a *.DAT file extension. By default, the driver stores memos in a separate file, or optionally in the data file itself, given the appropriate driver string.

KEYs are dynamic, and automatically update when the data file changes.

INDEXes are stored separately from data files. INDEX files receive a temporary file name, and are deleted when the program terminates normally. INDEXes are staticthey are not automatically updated when the data file changes. The BUILD statement creates or updates index files.

The Btrieve file format stores minimal file structure information in the file. The driver validates your description against the information in the file. It *is* possible to successfully open a Btrieve file that has key definitions that do not exactly match your definition. You must make certain that your file and key definitions accurately match the Btrieve file.

Files:	**CWBTRV16.LIB**	Windows Export Library (16-bit)
	CWBTRV32.LIB	Windows Export Library (32-bit)
	CLBTRV16.LIB	Windows Static Link Library (16-bit)
	CLBTRV32.LIB	Windows Static Link Library (32-bit)
	CWBTRV16.DLL	Windows Dynamic Link Library (16-bit)

An Owner Name is similar to a password. An encrypted Btrieve file uses the owner name as the encryption key.

Btrieve:Data Types

Clarion data type	Btrieve data type
BYTE	STRING (1 byte)
SHORT	INTEGER (2 bytes)
LONG	INTEGER (4 bytes)
SREAL	FLOAT (4 bytes)
REAL	FLOAT (8 bytes)
BFLOAT4	BFLOAT (4 bytes)
BFLOAT8	BFLOAT (8 bytes)
PDECIMAL	DECIMAL
STRING	STRING
CSTRING	ZSTRING
PSTRING	LSTRING
DATE	DATE
TIME	TIME
USHORT	UNSIGNED BINARY (2 bytes)
ULONG	UNSIGNED BINARY (4 bytes)
MEMO	STRING,LVAR or NOTE (see below)
BYTE,NAME('LOGICAL')	LOGICAL*
USHORT,NAME('LOGICAL')	LOGICAL*
PDECIMAL,NAME('MONEY')	MONEY*
STRING(@N0n-),NAME('STS')	SIGNED TRAILING SEPERATE*
DECIMAL	

Notes:

- You can store Clarion DECIMAL types in a Btrieve file. However, you cannot build a key or index using the field.
- If you want to create a file with LOGICAL or MONEY field types, you must specify an external name of LOGICAL or MONEY, respectively. If you are accessing an existing file, the NAME attribute is not required.

LOGICAL may be declared as a BYTE or USHORT, depending on whether it is a one or two byte LOGICAL:

```
LogicalField1   BYTE!One byte LOGICAL
LogicalField2   USHORT !Two byte LOGICAL
```

MONEY may be declared as a PDECIMAL(x,2), where *x* is the total number of digits to be stored:

```
MoneyField     PDECIMAL(7,2)  !Store up to 99999.99
```

- Btrieve NUMERIC fields are not fully supported by the driver. Btrieve NUMERIC is stored as a string with the last character holding a digit and an implied sign.. The possible values for this last character are:

```
          1 2 3 4 5 6 7 8 9 0
Positive:    A B C D E F G H I {
Negative:  J K L M N O P Q R }
```

To access a NUMERIC field you must define a STRING(@N0x), where x is one less than the digits in the NUMERIC, and a STRING(1) to hold the sign indicator. The Btrieve driver does not maintain this sign field, the application must be written to directly handle it.

For example to access a NUMERIC(7) you would have:

```
NumericGroup     GROUP          !Store -999999 to 999999
Number                       STRING(@N06)   !Numbers
Sign               STRING(1)      !Sign indicator
                 END
```

Btrieve:File Specifications/Maximums:

File Size : 4,000,000,000 bytes
Records per File : Limited by the size of the file
Record Size
Client-based : 65,520 bytes variable length
Server based : 54K variable length
Field Size : 65,520 bytes
Fields per Record : 65,520 bytes
Keys/Indexes per File: 24 with NLM5
256 with NLM6.
Client Btrieve v6.15
Page Size Max Key Segments
512 8
1,024 23
1,536 24
2,048 54
4,096 119
This is the total number of components. If you have a multicomponent key built from three fields, this counts as three indexes when counting the number of allowed indexes.
Key Size : 255 bytes
Memo fields per File: System memory dependent
Memo field size : 65,520 bytes
Open Files : Operating system dependent

Btrieve:Driver Strings and SEND functions

Driver strings (the second parameter of the DRIVER attribute) are all preceded by a forward slash character (/). SEND function commands can take two formatsone with an equal sign modifies a switch setting and return the value of the previous switch setting; the other format (without an equal sign) returns the value of the switch.

Driver strings are sent to the file driver when the file is opened. The SEND function sends a command to modify a setting after the file is open. Some driver strings have no effect after the file is open, so the SEND function syntax to modify the setting is not listed. However, the SEND function to return the value of the switch is valid for all driver strings.

/MEMO=SINGLE

To access existing Btrieve files created with the Btrieve LEM from Clarion 2.1, or files with variable length records set MEMO to SINGLE.

/MEMO=LVAR

To access a file with variable length records, use a SINGLE style MEMO whose size equals the maximum size of the variable length component of the record. To add/put records to this style file with binary data stored in the variable length section, use the ADD(file,length), APPEND(file,length) and PUT(file,pos,length) functions. The driver ignores the *pos* parameter in the PUT function, but initialize it to *0 (zero)* for future compatibility. The ADD, APPEND or PUT functions will remove all trailing spaces for text memos and NULL characters for binary memos before storing the record.

/MEMO=NOTE,<delimiter>

To access Xtrieve data files that have data type of Note or LVar, set the driver string to NOTE and LVAR respectively. With the NOTE data type, specify the end-of-field delimiter. Specify the ASCII value for the delimiter. NOTE and LVAR memos do not require the use of the size variants of ADD, APPEND and PUT, when storing records. The end of record marker is not necessary for a NOTE style memo. The driver automatically adds the end of record marker before storing the record and removes it before putting the memo data into the memo buffer.

As an example, "/MEMO=NOTE,141" indicates a file with an Xtrieve Notes field using CR/LF as the delimiter. For more information on the Xtrieve data types refer to the documentation supplied by Novell.

SEND(file,'MEMO')

Returns the MEMO setting: NORMAL,NOTE,LVAR or SINGLE

/PAGESIZE=<size>

Optionally sets the Btrieve Page size at file creation time. The keyword must be upper case. It must always be a multiple of 512, with a maximum of 4096. Larger page sizes usually result in more efficient disk storage. *Do not add spaces before or after* the equal sign.

SEND(file,'PAGESIZE')

Returns the page size in the form of a STRING.

/ALLOWREAD=[ON|OFF]

By default, a Btrieve file created with an owner name may be accessed *only* in read-only mode when the owner name is not known. To prevent *all* access to the file without the owner name, set ALLOWREAD to OFF.

SEND(file,'ALLOWREAD')

Returns the ALLOWREAD setting (ON or OFF) in the form of a STRING(3).

/COMPRESS=[ON|OFF]

Btrieve allows you to compress the data before storage. This allows for a smaller storage requirement, but reduces performance. When COMPRESS is ON,

CREATE creates a compressed Btrieve file.

SEND(file,'COMPRESS')

Returns the COMPRESS setting (ON or OFF) in the form of a STRING(3).

/PREALLOCATE=n When creating a Btrieve file, you can preallocate *n* pages of disk space for the file. The default is zero.

SEND(file,'PREALLOCATE')

Returns the number of pages of allocated disk space in the form of a STRING.

/FREESPACE=[0|10|20|30]

Specifies the percentage of free space to maintain on variable length pages. The default is zero.

SEND(file,'FREESPACE') Returns the percentage of free space to maintain on variable length pages in the form of a STRING.

/ACS=file_name When creating a Btrieve file you can specify an alternate collating sequence that STRING keys will be sorted by. This sorting sequence is normally obtained from the sort sequence you define in the INI file for your program. However, Btrieve supplies files for doing case insensitive sorts. To create your file using these sort sequences you specify the name of the sort file in the driver string.

For example. To use the alternate collating sequence file UPPER.ALT you would specify:

```
AFile   FILE,DRIVER('BTRIEVE','/ACS=UPPER.ALT'),CREATE
```

/APPENDBUFFER=size

SEND(file,'APPENDBUFFER=size')

By default APPEND adds records to the file one at a time. To get better performance over a network you can tell the driver to build up a buffer of records then send all of them to Btrieve at once. This is done using SEND(file,'APPENDBUFFER=size') where size is the number of records you want to allocate for the buffer. The maximum value of size of the buffer.

SEND(file,'APPENDBUFFER')

Returns the number of records that will fit in the buffer.

/BALANCEKEYS=[ON|OFF]')

When creating a Btrieve file, you can use this driver string to tell Btrieve that Btrieve that all keys associated with the file must be stored in a balances btree. This saves disk space, but will slow down file adds, deletes and updates where key values change.

SEND(file,'BALANCEKEYS')

Returns the BALANCEKEYS setting (ON or OFF) in the form of a STRING(3).

SEND(file,'FREEAPPENDBUFFER')

Frees up the memory used by the append buffer allocated by a call to SEND(file,APPENDBUFFER=size). Returns the number of records that fitted in the old buffer.

/LACS= With Btrieve v6.15 Btrieve added the feature of Local Alternate Collating Sequences. This allows your string key to sort based on the country code for the machine running your program. To use this feature you put '/LACS=' in your

driver string.

/LACS=country_ID,code_page

With Btrieve v6.15 Btrieve added the feature of User-Defined Alternate Collating Sequences. This allows your string key to sort based on the DOS country code and code page for a particular country. To use this feature you put '/LACS=country_id,codepage' in your driver string. Note that there must be no spaces surrounding the comma.

SEND(file,'LACS') Returns country_ID,code_page or the string ',' (if using machine-dependent LACS).

/TRUNCATE=[ON|OFF] When creating a Btrieve file, you can use this driver string to tell Btrieve to truncate trailing spaces. This forces the record to be stored as a variable length records.

SEND(file,'TRUNCATE')

Returns the TRUNCATE setting (ON or OFF) in the form of a STRING(3).

Btrieve:Unsupported/Modified Functions and Attributes

Key Attribute: **NOCASE**

NLM 5 does not support case insensitive indexing. When necessary, you must supply an alternate collating sequence which implements case insensitive sorting.

Btrieve supports an alternate collating sequence. However, NLM 6 does not support *both* NOCASE *and* an alternate collating sequence. If you specify both, the NOCASE attribute takes precedence. No error is returned from the SEND function.

Buffering Control: **STREAM, FLUSH**

There is no buffering control within the Btrieve driver.

File Locking: **LOCK()**

Btrieve does not support file locking. The driver does not return any error if you call it. If you require file locking, use LOGOUT.

Record Access: **GET(file, fileptr, len)**

File information: **BYTES()**

File updates: **PUT(file, fileptr)**

SET(file, filepointer), SET(key, keypointer)

If a file or key pointer has a value of zero, or any other value that does not exist in the file, the driver ignores the pointer parameter. Processing is set to either file or key order, and the record pointer is set to the first element.

SET(key, key, filepointer)

If the filepointer has a value of zero, or any other value that does not exist in the file, processing starts at the first key value whose position is greater than (or less than for PREVIOUS) the filepointer. Not passing a valid pointer is inefficient.

EOF(file), BOF(file)

These functions are supported, but not recommended. They cause more disk I/O than ERRORCODE(). Btrieve returns *eof* when reading past the last record. This requires the driver must read the next record, then the *next* to see if it's at the end of file, then goes back to the record you want.

ADD(file), PUT(file)

When using the LVAR and NOTE memo type, make certain that the memo has the appropriate structure. If the structure is incorrect and the driver calculates a length greater than the maximum memo size defined for that file, these functions fail and set errorcode to 57 - Invalid Memo File.***

DELETE(file) *when stepping through in record order*

Tip: Btrieve's DELETE destroys positioning information when processing in file order. The driver attempts to reposition to the appropriate record. This is not always possible and may require the driver to read from the start of the file. Using key order processing avoids this possible slow down.

LOGOUT()

Btrieve does not allow you to logout only certain files. When you issue a LOGOUT() call, all Btrieve files accessed during the transaction are logged out. This means the following code is illegal (as you cannot close a logged-out file:

```
LOGOUT(1,file1)
OPEN(file2)
CLOSE(file2)
```

APPEND()

Btrieve does not support non-key updates. To emulate APPEND() behavior, the driver drops all indexes possible when APPEND() is first called. Calling BUILD() immediately after

appending records rebuilds the dropped key fields.

BUILD()

If used after an APPEND(), but before a file is closed, this adds the keys dropped by APPEND(). In all other cases BUILD() rebuilds the file and keys. If you only want to rebuild keys, doing a BUILD(key) for each key is faster than BUILD(file).

BUILD(DynamicIndex, expression,filter)

Btrieve:Miscellaneous

The driver stores records less than 4K as fixed length. It stores records greater than 4K as variable length. The minimum record length is 4 bytes. One record can be held in each open file by each user.

The driver ignores any NAME attribute on a MEMO field. MEMO fields can reside either in a separate file, or in the data file if the driver string MEMO is set to SINGLE, LVAR or NOTE. If the driver string MEMO is not set, the separate MEMO file name is "MEM," preceded by the first five characters of the file's label, plus the file extension "DAT." Setting the driver string MEMO restricts you to one memo field per file.

Btrieve allows you to open a file in five different formats: NORMAL, ACCELERATED, READ-ONLY, VERIFY or EXCLUSIVE. The equivalent Clarion OPEN() states are:

Btrieve State	Clarion OPEN/SHARE access mode
ACCELERATED	Read/Write with FCB compatibility mode (2H)
READ-ONLY	Read Only (0H,10H,20H,30H,40H)
VERIFY	Write Only with FCB compatibility mode (1H)
EXCLUSIVE	Write Only with any Deny flag (11H,21H,31H,41H); Read/Write with Deny All, Read or Write (12H,22H,32H)
NORMAL	Read/Write with Deny None (42H)

Btrieve allows a file to have a specified owner. See the driver string /READONLY for details on setting this flag. The file may also be encrypted. This is set with the ENCRYPT attribute. A file can only be encrypted when an owner name is supplied.

Btrieve uses an unsigned long for its internal record pointer; negative values are stripped of their sign. We recommend the ULONG data type for your record pointer.

Calculating Page Size:

To determine the physical record length, add 8 bytes for each KEY that allows duplicates. Add 4 bytes if the file allows variable record lengths. Finally, allow 6 bytes for overhead per page.

For example: If the record size is 300 bytes and the file has three KEYs that allow Duplicates, the total record size is:

	300	*record size*
x	24	*overhead for three KEYs with the DUP attribute*
=	324	*physical record length*

A page size of 512 would only hold one such record, and 182 bytes per page would go unused (512 - 6 - 324). If the page size were 1024, three records could be stored per page and only 46 bytes would go unused (1024 - 6 - (324 * 3)).

You must load BTRIEVE.EXE with a page size equal to or greater than the largest page size of any file that you will be accessing.

When defining a file, the key definition does not need to exactly match the underlying file. For example, you can have a physical file with a single component STRING(20). You can define this as a key with two string components with a total length of 20. The rule is that the data types must match and the total size must match. However, if your Clarion definition does not exactly match the underlying file, the driver cannot optimize APPEND() or BUILD() statements.

- A Key's NAME attribute can add additional functionality.

KEY,NAME('MODIFIABLE=true|false')

Btrieve allows you to create a key that can not be changed once created. To use this feature you can use the name attribute on the key to set MODIFIABLE to FALSE. It defaults to TRUE.

KEY,NAME('ANYNULL')

Btrieve allows you to create a key that will not include a record if any key components are null. To create such a key you specify ANYNULL in the key name.

For example, to create a key that is non modifiable and excludes keys if any component is null:

```
Key1    KEY(+pre:field1,-pre:field2),NAME('ANYNULL MODIFIABLE=FALSE')
```

KEY,NAME('REPEATINGDUPLICATE')

By default Btrieve version 6 stores a reference to only the first record in a series of duplicate records in a key. The other occurrences of the duplicate key value are obtained by following a link list stored at the record. To create an index where all duplicate records are stored in the key you use the NAME('REPEATINGDUPLICATE'). This produces larger keys, but random access to duplicate records is faster. (This feature is only available for version 6 files.)

- POSITION(file) returns a STRING(4).
- POSITION(key) returns a STRING the size of the key fields + 4 bytes.

Clarion Files

The Clarion file driver is compatible with the file system used by Clarion Database Developer 3.0 and Clarion Professional Developer.

Keys and Indexes exist as separate files from the data file. Keys are dynamicthey are automatically updated as the data file changes. The default file extension for a key file is *.K##. Indexes are staticthey do not automatically update, but instead require the BUILD statement for updating.

The driver stores records as fixed length. It stores memo fields in a separate file. The memo file defaults to the first eight characters of the File Label plus an extension of .MEM.

Files:	**CWC2116.LIB**	Windows Export Library (16-bit)
	CWC2132.LIB	Windows Export Library (32-bit)
	CLC2116.LIB	Windows Static Link Library (16-bit)
	CLC2132.LIB	Windows Static Link Library (32-bit)
	CWC2116.DLL	Windows Dynamic Link Library (16-bit)
	CWC2132.DLL	Windows Dynamic Link Library (32-bit)

Tip: By avoiding the ASCII-only file formats of many other popular PC database application development systems, the Clarion file format provides a more secure means of storing data.

Clarion:Data Types

BYTE DECIMAL
SHORT STRING (255 byte maximum)
LONG MEMO
REAL GROUP

Clarion:Maximum File Specifications:

File Size:	**limited only by disk space**
Records per File :	4,294,967,295
Record Size:	65,520 bytes
Field Size :	65,520 bytes
Fields per Record:	65,520 bytes
Keys/Indexes per File:	251
Key Size :	245 bytes
Memo fields per File :	1
Memo Field Size:	65,520 bytes
Open Data Files:	Operating system dependent

Clarion:Driver Strings and SEND functions

Driver strings (the second parameter of the DRIVER attribute) are all preceded by a forward slash character (/). SEND function commands can take two formatsone with an equal sign modifies a switch setting and return the value of the previous switch setting; the other format (without an equal sign) returns the value of the switch.

Driver strings are sent to the file driver when the file is opened. The SEND function sends a command to modify a setting after the file is open. Some driver strings have no effect after the file is open, so the SEND function syntax to modify the setting is not listed. However, the SEND function syntax to return the value of the switch is listed for all driver strings.

RECOVER may not be used as a DRIVER stringyou may only use it with the SEND function.

SEND(file,'RECOVER=n')

The RECOVER string, when *n* is greater than 0, UNLOCKs a data file, or RELEASEs a held record in order to recover from a system crash.

n represents the number os seconds to wait before invoking the recovery process. When *n* is equal to 1, the recovery process is invoked immediately. When *n* is equal to 0, the recovery process is disarmed.

The SEND function returns a blank string.

To RELEASE a held record, you must read that record into memory. If there are multiple held records, loop through the entire file after SENDing the RECOVER= message to the driver.

SEND(file,'IGNORESTATUS=on|off')

/IGNORESTATUS=on|off When set *on*, the driver does *not* skip deleted records when accessing the file with GET(), NEXT(), and PREVIOUS() in file order. It also enables a PUT() on a deleted or held record. /IGNORESTATUS requires opening the file in exclusive mode.

SEND(file,'IGNORESTATUS')

Returns the **IGNORESTATUS** setting (ON or OFF) in the form of a STRING(3).

SEND(file,'DELETED')

For use only with the SEND command, when /IGNORESTATUS is on. Reports the status of the loaded record. If deleted, the return string is "ON;" if not, "OFF."

SEND(file,'HELD') For use only with the SEND command, when /IGNORESTATUS is on. Reports the status of the loaded record. If held, the return string is "ON;" if not, "OFF."

Clarion:Miscellaneous

- POSITION(file) returns a STRING(4).
- POSITION(key) returns a STRING the size of the key fields + 4 bytes.

Clarion:Unsupported Functions and Attributes

Record Access: **GET(file, fileptr, len), ADD(file, len), APPEND(file, len)**

The driver does not support variable length records.

File updates: **PUT(file, fileptr, len)**

The driver does not support variable length records.

EOF(file), BOF(file)

Although the driver supports these functions, we do not recommend their use. These functions must physically access the files in order to operate, adding considerable overhead. Instead, test the value returned by ERRORCODE() after each sequential access. NEXT() or PREVIOUS() post Error 33 (Record Not Available) if an attempt is made to access a record beyond the end or beginning of the file.

BUILD(DynamicIndex, expression,filter)

Clipper Files

The Clipper file driver is compatible with Clipper Summer '87 and Clipper 5.0. The default data file extension is *.DBF.

Keys and Indexes exist as separate files from the data file. Keys are dynamicthey automatically update as the data file changes. Indexes are staticthey do not automatically update, but instead require the BUILD statement for updating. The default file extension for the index file is *.NTX.

The driver stores records as fixed length. It stores memo fields in a separate file. The memo file name takes the first eight characters of the File Label plus an extension of .DBT.

Files:	**CWCLIP16.LIB**	Windows Export Library (16-bit)
	CWCLIP32.LIB	Windows Export Library (32-bit)
	CLCLIP16.LIB	Windows Static Link Library (16-bit)
	CLCLIP32.LIB	Windows Static Link Library (32-bit)
	CWCLIP16.DLL	Windows Dynamic Link Library (16-bit)
	CWCLIP32.DLL	Windows Dynamic Link Library (32-bit)

Tip: **As a popular xBase database application development system, Clipper provides a common file format for many installed business applications and their data files. Use the Clipper driver to access these files in their native format.**

Clipper:Data Types

The xBase file format stores all data as ASCII strings. You may either specify STRING types with declared pictures for each field, or specify native Clarion data types, which the driver converts automatically.

Clipper data type	Clarion data type	STRING w/ picture
Date	DATE	STRING(@D12)
*Numeric	REAL	STRING(@N-_p.d)
*Logical	BYTE	STRING(1)
Character	STRING	STRING
*Memo	MEMO	MEMO

If your application reads and writes to existing files, a pictured STRING will suffice. However, if your application *creates* a Clipper file, you may require additional information for these Clipper types:

To create a numeric field in the Data Dictionary, choose the REAL data type. In the External Name attribute, specify '*NumericFieldName*=N(*Precision*,*DecimalPlaces*)' where *NumericFieldName* is the name of the field, *Precision* is the precision of the field and *DecimalPlaces* is the number of decimal places.

If you're hand coding a native Clarion data type, add the NAME attribute using the same syntax. If you're hand coding a STRING with picture, STRING(@N-_9.2), NAME('*Number*'), where *Number* is the field name.

To create a logical field, using the data dictionary, choose the BYTE data type. There are no special steps; however, see the miscellaneous section for tips on reading the data from the field.

If you're hand coding a STRING with picture, add the NAME attribute: STRING(1), NAME('*LogFld* = L')**.**

To create a date field, using the data dictionary, choose the DATE data type, rather than LONG, which you usually use for the TopSpeed or Clarion file formats.

MEMO field declarations require the a pointer field in the file's record structure. Declare the pointer field as a STRING(10) or a LONG. This field will be stored in the .DBF file containing the offset of the memo in the .DBT file. The MEMO declaration must have a NAME() attribute naming the pointer field. An example file declaration follows:

```
File FILE, DRIVER('Clipper')
Memo1    MEMO(200),NAME('Notes')
Memo2    MEMO(200),NAME('Text')
Rec    RECORD
Mem1Ptr    LONG,NAME('Notes')
Mem2Ptr    STRING(10),NAME('Text')
         END
     END
```

Clipper:File Specifications/Maximums

File Size:	**2,000,000,000 bytes**
Records per File:	1,000,000,000 bytes
Record Size:	4,000 bytes (Clipper '87)
	8,192 bytes (Clipper 5.0)
Field Size	
Character:	254 bytes (Clipper '87)
	2048 bytes (Clipper 5.0)
Date:	8 bytes
Logical:	1 byte
Numeric:	20 bytes including decimal point
Memo:	65,520 bytes (see note)
Fields per Record:	255
Keys/Indexes per File:	No Limit
Key Sizes	
Character:	100 bytes
Numeric, Date:	8 bytes
Memo fields per File:	Dependent on available memory
Open Files:	Operating system dependent

Clipper:Driver Strings and SEND functions

Driver strings (the second parameter of the DRIVER attribute) are all preceded by a forward slash character (/). SEND function commands can take two formatsone with an equal sign modifies a switch setting and return the value of the previous switch setting; the other format (without an equal sign) returns the value of the switch.

Driver strings are sent to the file driver when the file is opened. The SEND function sends a command to modify a setting after the file is open. Some driver strings have no effect after the file is open, so the SEND function syntax to modify the setting is not listed. However, the SEND function syntax to return the value of the switch is listed for all driver strings.

/BUFFERS=n

Specify a value for the number of buffers used to read and write to the file.

The Clipper driver utilizes DOS buffering. The default is three buffers of 1024 bytes each. Increasing the number of buffers will not increase performance when a file is shared by multiple users.

SEND(file,'BUFFERS')

Returns the number of buffers in the form of a STRING.

/RECOVER

SEND(file,'RECOVER')

Equivalent to the Xbase RECALL command, which recovers records marked for deletion. When using the Clipper driver, the DELETE statement flags a record as "inactive." The driver does not remove the record until the PACK or BUILD command is executed.

/RECOVER is evaluated each time you open the file if you add the driver string to the data dictionary. When the driver recovers the records previously marked for deletion, you must manually rebuild keys and indexes with the BUILD statement.

SEND(file,'IGNORESTATUS=on|off')

/IGNORESTATUS=on|off

When set *on*, the driver does *not* skip deleted records when accessing the file with GET, NEXT, and PREVIOUS in file order. It also enables a PUT on a deleted or held record. /IGNORESTATUS requires opening the file in exclusive mode.

SEND(file,'IGNORESTATUS')

Returns the IGNORESTATUS setting (ON or OFF) in the form of a STRING(3).

SEND(file,'DELETED')

For use only with the SEND command, when IGNORESTATUS is on. Reports the status of the current record. If deleted, the return string is "ON;" if not, "OFF."

Clipper:Unsupported/Modified Functions & Attributes

Memos: **BINARY**

Clipper supports only text memos.

Keys: **NOCASE, OPT**

File: **ENCRYPT, OWNER, RECLAIM**

The Clipper driver cannot read encrypted Clipper files. To reclaim space from deleted records, call PACK(file).

Transaction Processing: **COMMIT(), LOGOUT(), ROLLBACK()**

The Clipper driver does not support any transaction logging.

Record Access: **GET(file, fileptr, len), ADD(file, len)**

Clipper does not support variable length records

File updates: **PUT(file, fileptr, len)**

Clipper does not support variable length records

EOF(file), BOF(file)

Although the driver supports these functions, we do not recommend their use. They must physically access the files and add overhead. Instead, test the value returned by ERRORCODE() after each sequential access. NEXT or PREVIOUS post Error 33 (Record Not Available) if an attempt is made to access a record beyond the end or beginning of the file.

ADD(file) VS. APPEND(file)

The ADD statement tests for duplicate keys before modifying the data file or its associated KEY files. Consequently it is slower than APPEND which performs no checks and does not update KEYs. When adding large amounts of data to a database use APPEND...BUILD in preference to ADD.

BUILD(key, str)

When building dynamic indexes, the *str* component may take one of two forms:

BUILD(DynNdx, '+Pre:FLD1, -Pre:FLD2')

This form specifies the names of the fields on which to build the index. The field names must appear as specified in the fields' NAME() attribute if supplied, or must be the label name. A prefix may be used for compatibility with the Clarion conventions but is ignored.

BUILD(DynNdx, 'T[Expression]')

This form specifies the type and Expression used to build an index, see

the miscellaneous section, below.

COPY(file, newname)

The COPY() command copies data and memo files using *newname*, which may specify a new file name or directory. Key or index files are copied if the *newname* is a subdirectory specification. To copy an index file to a new file, use a special form of the copy command:

COPY(file,'<index>|<newname>')

This returns *File Not Found* if an invalid index is passed. The COPY command assumes a default extension of ".NTX" for both the source and the target file names if none is specified. If you require a file name without an extension, terminate the name with a period. Given the file structure:

```
Clar2    FILE,CREATE,DRIVER('Clipper')
NumKey   KEY(Num),DUP
StrKey   KEY(Str1)
StrKey2    KEY(Str2)
AMemo      MEMO(100), NAME('mem')
Record   RECORD
Num        STRING(@n-_9.2)
STR1       STRING(2)
STR2       STRING(2)
Mem        STRING(10)
           .   .
```

The following commands copy this file definition to A:

```
COPY(Clar2,'A:\CLAR2')
COPY(Clar2,'StrKey|A:\STRKEY')
COPY(Clar2,'StrKey2|A:\STRKEY2')
COPY(Clar2,'NumKey|A:\NUMKEY')
```

After these calls, the following files would exist on drive A: CLAR2.DBF, CLAR2.DBT, STRKEY.NTX, STRKEY2.NTX, and NUMKEY.NTX.

DELETE(file)

When the driver deletes a record from a Clipper database, the record is not physically removed, instead the driver marks it inactive. Memo fields are not physically removed from the memo file, however they cannot be retrieved if they refer to an inactive record. Key values *are* removed from the index files. To remove records and memo fields permanently, execute a PACK(file).

Tip: To those programmers familiar with Clipper, this driver processes deleted records consistent with the way Clipper processes them after the SET DELETED ON command is issued. Records marked for deletion are ignored from processing by executable code statements, but remain in the data file.

HOLD(file), HOLD(file, timeout)

Clipper performs record locking by locking the entire record within the data file. This prevents read access to other processes. Therefore we recommend minimizing the amount of time for which a record is held.

POINTER(file), POINTER(key)

There is no distinction between file pointers and key pointers; they both contain the same value for any given record.

RECORDS(file), RECORDS(key)

Under Clipper, the RECORDS() function reports the same number of records for the data file and its keys and indexes. Usually there will be no difference in the number of records *unless* the INDEX is out of date. Because the DELETE statement does not physically remove records, *the number of records reported by the* RECORDS() *function includes inactive records*. Exercise care when using this function.

RENAME(file, newname)

The RENAME command copies the data and memo files using *newname*, which may specify a new file name or directory path. Key and index files must be renamed using the same syntax as the COPY command, above.

- POSITION(file) returns a STRING(12).
- POSITION(key) returns a STRING the size of the key fields + 4 bytes.
- **BUILD(DynamicIndex, expression,filter)** is not supported.

Clipper:Miscellaneous

Clipper allows a maximum of 254 characters to a character field.

Clipper allows a logical field to accept one of nine possible values (y,Y,n,N,t,T,f,F or a space character). The space character is neither true nor false. When using a logical field from a preexisting database in a logical expression, account for all these possibilities. Remember that when a STRING field is used as an expression, it is true if it contains any data and false if it is equal to zero or blank. Therefore, to evaluate a Logical field's truth, the expression should be true if the field contains any of the "true" characters (T,t,Y, or y). For example, if a Logical field were used to specify a product as taxable or nontaxable, the expression to evaluate its truth would be:

(If Condition):

Taxable='T' OR Taxable='t' OR Taxable='Y' OR Taxable='y'

Clarion for Windows supports MEMO fields up to a maximum of 64K. If you have an existing file which includes a memo greater than 64K, you can use the file but not modify the large MEMOs.

You can determine when your application encounters a large MEMO by detecting when the memo pointer variable is non-blank, but the memo appears to be blank. Error 47 (Bad Record Declaration) is posted. If you attempt to update such a record, any modification to the MEMO field is ignored.

Clipper supports a maximum of 10 characters in a field name. If you require more, use an External Name with 10 characters or less.

Clipper supports the use of expressions to define keys. Within the Dictionary Editor, you can place the expression in the external name field in the *Key Properties* dialog. The general format of the external name is :

'FileName=T[Expression]'

Where *FileName* represents the name of the index file (which can contain a path and file extension), and *T* represents the type of the index. Valid types are: C = character, D = date, and N = numeric. If the type is D or N then *Expression* can name only one field.

The expression may refer to multiple fields in the record, and contain xBase functions. Square brackets must enclose the expression. The currently supported functions appear below. If the driver encounters an unsupported Xbase function in a preexisting file, it posts error 76 'Invalid Index String' when the file is opened for keys and static indexes.

String expressions may use the '+' operator to concatenate multiple string arguments. Numeric expressions use the '+' or '-' operators with their conventional meanings. The maximum length of a Clipper expression is 250 characters.

Supported xBase commands

ALLTRIN(string) Removes leading and trailing spaces.

CTOD(string) Converts a string key to a date. The *string* format mm/dd/yy; the result takes the form 'yyyymmdd'. The yyyy element of the date defaults to the twentieth century. An invalid date results in a key containing blanks.

DELETED() Returns TRUE if the record is deleted.

DESCEND(string|date|numeric)

	Inverts the argument, and creates descending Clipper indexes.
DTOC(date)	Converts a date key to string format 'mm/dd/yy'
DTOS(date)	Converts a date key to string format 'yyyymmdd'
FIXED(float)	Converts a float key to a numeric.
FLOAT(numeric)	Converts a numeric key to a float.
IIF(bool,val1,val2)	Returns val1 if the first parameter is TRUE, otherwise returns val2.
LEFT(string, n)	Returns the leftmost *n* characters of the string key as a string of length *n*.
RIGHT(string, n)	Returns the rightmost *n* characters of the string key as a string of length *n*.
RTRIM(string)	Removes spaces from the right of a string.
STR(numeric [,length[, decimal places]])	converts a numeric to a string. The length of the string and the number of decimal places are optional. The default string length is 10, and the number of decimal places is 0.
SUBSTR(string,offset,n)	Returns a substring of the *string* key starting at *offset* and of *n* characters in length.
TRIM(string)	Removes spaces from the right of a string (identical to RTRIM).
UPPER(string)	Converts a string key to upper case.
VAL(string)	Converts a string key to a numeric.

dBase III Files

The dBase3 file driver is compatible with dBase III. The default data file extension is *.DBF.

Keys and Indexes exist as separate files from the data file. Keys are dynamicthey automatically update as the data file changes. Indexes are staticthey do not automatically update, but instead require the BUILD statement for updating. The default file extension for the index file is *.NDX.

The driver stores records as fixed length. It stores memo fields in a separate file. The memo file name takes the first eight characters of the File Label plus an extension of .DBT.

Files:	**CWDB316.LIB**	Windows Export Library (16-bit)
	CWDB332.LIB	Windows Export Library (32-bit)
	CLDB316.LIB	Windows Static Link Library (16-bit)
	CLDB332.LIB	Windows Static Link Library (32-bit)
	CWDB316.DLL	Windows Dynamic Link Library (16-bit)
	CWDB332.DLL	Windows Dynamic Link Library (32-bit)

Tip: dBase III is probably the most common file format for PC database applications. These days, even desktop publishing programs can import dBase III compatible .DBF files. If the main task of your application is to export data files for other applications about which you know nothing, you should consider this format.

dBase III:Data Types

The xBase file format stores all data as ASCII strings. You may either specify STRING types with declared pictures for each field, or specify native Clarion types, which the driver converts automatically.

dBase data type	Clarion data type	STRING w/ picture
Date	DATE	STRING(@D12)
*Numeric	REAL	STRING(@N-_p.d)
*Logical	BYTE	STRING(1)
Character	STRING	STRING
*Memo	MEMO	MEMO

If your application reads and writes to existing files, a pictured STRING will suffice. However, if your application *creates* a dBase III file, you may require additional information for these dBase III types:

To create a numeric field in the Data Dictionary, choose the REAL data type. In the External Name attribute, specify '*NumericFieldName*=N(*Precision*,*DecimalPlaces*)' where *NumericFieldName* is the name of the field, *Precision* is the precision of the field and *DecimalPlaces* is the number of decimal places.

If you're hand coding a native Clarion data type, add the NAME attribute using the same syntax. If you're hand coding a STRING with picture, STRING(@N-_9.2), NAME('*Number*'), where *Number* is the field name.

To create a logical field, using the data dictionary, choose the BYTE data type. There are no special steps; however, see the miscellaneous section for tips on reading the data from the field.

If you're hand coding a STRING with picture, add the NAME attribute: STRING(1), NAME('*LogFld* = L')**.**

To create a date field, using the data dictionary, choose the DATE data type, rather than LONG, which you usually use for the TopSpeed or Clarion file formats.

MEMO field declarations require the a pointer field in the file's record structure. Declare the pointer field as a STRING(10) or a LONG. This field will be stored in the .DBF file containing the offset of the memo in the .DBT file. The MEMO declaration must have a NAME() attribute naming the pointer field. An example file declaration follows:

```
File FILE, DRIVER('dBase3')
Memo1    MEMO(200),NAME('Notes')
Memo2    MEMO(200),NAME('Text')
Rec    RECORD
Mem1Ptr    LONG,NAME('Notes')
Mem2Ptr    STRING(10),NAME('Text')
             END
     END
```

dBase III:File Specifications/Maximums

File Size: **2,000,000,000 bytes**

```
Records per File:       1,000,000,000 bytes
Record Size:          4,000 bytes
Field Size
     Character:         254 bytes
     Date:              8 bytes
     Logical:         1 byte
     Numeric:         20 bytes including decimal point
     Memo:              64K (see note)
Fields per Record:    255
Keys/Indexes per File: No Limit
Key Sizes
     Character:         100 bytes
     Numeric, Date:     8 bytes
Memo fields per File:  Dependent on available memory
Open Files:           Operating system dependent
```

dBase III:Driver Strings and SEND functions

Driver strings (the second parameter of the DRIVER attribute) are all preceded by a forward slash character (/). SEND function commands can take two formatsone with an equal sign modifies a switch setting and return the value of the previous switch setting; the other format (without an equal sign) returns the value of the switch.

Driver strings are sent to the file driver when the file is opened. The SEND function sends a command to modify a setting after the file is open. Some driver strings have no effect after the file is open, so the SEND function syntax to modify the setting is not listed. However, the SEND function syntax to return the value of the switch is listed for all driver strings.

/BUFFERS=n

Specify a value for the number of buffers used to read and write to the file.

The dBase III driver utilizes DOS buffering. The default is three buffers of 1024 bytes each. Increasing the number of buffers will not increase performance when a file is shared by multiple users.

SEND(file,'BUFFERS')

Returns the number of buffers in the form of a STRING.

/RECOVER

SEND(file,'RECOVER')

Equivalent to the Xbase RECALL command, which recovers records marked for deletion. When using the dBase III driver, the DELETE statement flags a record as "inactive." The driver does not remove the record until the PACK command is executed.

/RECOVER is evaluated each time you open the file if you add the driver string to the data dictionary. When the driver recovers the records previously marked for deletion, you must manually rebuild keys and indexes with the BUILD statement.

SEND(file,'IGNORESTATUS=on|off')

/IGNORESTATUS=on|off

When set *on*, the driver does *not* skip deleted records when accessing the file with GET, NEXT, and PREVIOUS in file order. It also enables a PUT on a deleted or held record. /IGNORESTATUS requires opening the file in exclusive mode.

SEND(file,'IGNORESTATUS')

Returns the IGNORESTATUS setting (ON or OFF) in the form of a STRING(3).

SEND(file,'DELETED')

For use only with the SEND command, when IGNORESTATUS is on. Reports the status of the current record. If deleted, the return string is "ON;" if not, "OFF."

/OMNIS

Specifies OMNIS file header and file delimiter compatibility.

dBase III:Unsupported/Modified Functions & Attributes

Memos: **BINARY**

dBase III supports only text memos.

Keys: **DUP, NOCASE, OPT, ascending|descending**

File: **ENCRYPT, OWNER, RECLAIM**

The dBase III driver cannot read encrypted dBase III files. To reclaim space from deleted records, call PACK(file).

Transaction Processing: **COMMIT(), LOGOUT(), ROLLBACK()**

The dBase III driver does not support any transaction logging.

Record Access: **GET(file, fileptr, len), ADD(file, len)**

dBase III does not support variable length records

File updates: **PUT(file, fileptr, len)**

dBase III does not support variable length records

EOF(file), BOF(file)

Although the driver supports these functions, we do not recommend their use. They must physically access the files and add overhead. Instead, test the value returned by ERRORCODE() after each sequential access. NEXT or PREVIOUS post Error 33 (Record Not Available) if an attempt is made to access a record beyond the end or beginning of the file.

ADD(file) VS. APPEND(file)

The ADD statement tests for duplicate keys before modifying the data file or its associated KEY files. Consequently it is slower than APPEND which performs no checks and does not update KEYs. When adding large amounts of data to a database use APPEND...BUILD in preference to ADD.

BUILD(key, str)

When building dynamic indexes, the *str* component may take one of two forms:

BUILD(DynNdx, '+Pre:FLD1, -Pre:FLD2')

This form specifies the names of the fields on which to build the index. The field names must appear as specified in the fields' NAME() attribute if supplied, or must be the label name. A prefix may be used for compatibility with the Clarion conventions but is ignored.

BUILD(DynNdx, 'T[Expression]')

This form specifies the type and Expression used to build an index, see

the miscellaneous section, below.

COPY(file, newname)

The COPY() command copies data and memo files using *newname*, which may specify a new file name or directory. Key or index files are copied if the *newname* is a subdirectory specification. To copy an index file to a new file, use a special form of the copy command:

COPY(file,'<index>|<newname>')

This returns *File Not Found* if an invalid index is passed. The COPY command assumes a default extension of ".NDX" for both the source and the target file names if none is specified. If you require a file name without an extension, terminate the name with a period. Given the file structure:

```
Clar2  FILE,CREATE,DRIVER('dBase3')
NumKey KEY(Num),DUP
StrKey KEY(Str1)
StrKey2  KEY(Str2)
AMemo    MEMO(100), NAME('mem')
Record   RECORD
Num      STRING(@n_9.2)
STR1     STRING(2)
STR2     STRING(2)
  Mem    STRING(10)
         .     .
```

The following commands copy this file definition to A:

```
COPY(Clar2,'A:\CLAR2')
COPY(Clar2,'StrKey|A:\STRKEY')
COPY(Clar2,'StrKey2|A:\STRKEY2')
COPY(Clar2,'NumKey|A:\NUMKEY')
```

After these calls, the following files would exist on drive A: CLAR2.DBF, CLAR2.DBT, STRKEY.NDX, STRKEY2.NDX, and NUMKEY.NDX.

DELETE(file)

When the driver deletes a record from a dBase III database, the record is not physically removed, instead the driver marks it inactive. Memo fields are not physically removed from the memo file, however they cannot be retrieved if they refer to an inactive record. Key values *are* removed from the index files. To remove records and memo fields permanently, execute a PACK(file).

Tip: To those programmers familiar with dBase III, this driver processes deleted records consistent with the way dBase III processes them after the SET DELETED ON command is issued. Records marked for deletion are ignored from processing by executable code statements, but remain in the data file.

HOLD(file), HOLD(file, timeout)

dBase III performs record locking by locking the entire record within the data file. This prevents read access to other processes. Therefore we recommend minimizing the amount of time for which a record is held.

POINTER(file), POINTER(key)

There is no distinction between file pointers and key pointers; they both contain the same value for any given record.

RECORDS(file), RECORDS(key)

Under dBase III the RECORDS() function reports the same number of records for the data file and its keys and indexes. Usually there will be no difference in the number of records *unless* the INDEX is out of date. Because the DELETE statement does not physically remove records, *the number of records reported by the* RECORDS() *function includes inactive records*. Exercise care when using this function.

RENAME(file, newname)

The RENAME command copies the data and memo files using *newname*, which may specify a new file name or directory path. Key and index files must be renamed using the same syntax as the COPY command, above.

BUILD(DynamicIndex, expression,filter) is not supported.

dBase III:Miscellaneous

dBase III allows a maximum of 254 characters to a character field.

dBase III allows a logical field to accept one of nine possible values (y,Y,n,N,t,T,f,F or a space character). The space character is neither true nor false. When using a logical field from a preexisting database in a logical expression, account for all these possibilities. Remember that when a STRING field is used as an expression, it is true if it contains any data and false if it is equal to zero or blank. Therefore, to evaluate a Logical field's truth, the expression should be true if the field contains any of the "true" characters (T,t,Y, or y). For example, if a Logical field were used to specify a product as taxable or nontaxable, the expression to evaluate its truth would be:

(If Condition):

Taxable='T' OR Taxable='t' OR Taxable='Y' OR Taxable='y'

Clarion for Windows supports MEMO fields up to a maximum of 64K. If you have an existing file which includes a memo greater than 64K, you can use the file but not modify the large MEMOs.

You can determine when your application encounters a large MEMO by detecting when the memo pointer variable is non-blank, but the memo appears to be blank. Error 47 (Bad Record Declaration) is posted, and any modification to the MEMO field is ignored.

dBase III supports a maximum of 10 characters in a field name. If you require more, use an External Name with 10 characters or less.

dBase III supports the use of expressions to define keys. Within the Dictionary Editor, you can place the expression in the external name field in the *Key Properties* dialog. The general format of the external name is :

'FileName=T[Expression]'

Where *FileName* represents the name of the index file (which can contain a path and file extension), and *T* represents the type of the index. Valid types are: C = character, D = date, and N = numeric. If the type is D or N then *Expression* can name only one field.

The expression may refer to multiple fields in the record, and contain xBase functions. Square brackets must enclose the expression. The currently supported functions appear below. If the driver encounters an unsupported Xbase function in a preexisting file, it posts error 76 'Invalid Index String' when the file is opened for keys and static indexes.

String expressions may use the '+' operator to concatenate multiple string arguments. Numeric expressions use the '+' or '-' operators with their conventional meanings. The maximum length of a dBase III expression is 250 characters.

Supported xBase commands

ALLTRIN(string)	Removes leading and trailing spaces.
CTOD(string)	Converts a string key to a date. The *string* format mm/dd/yy; the result takes the form 'yyyymmdd'. The yyyy element of the date defaults to the twentieth century. An invalid date results in a key containing blanks.
DELETED()	Returns TRUE if the record is deleted.
DTOC(date)	Converts a date key to string format 'mm/dd/yy.'

DTOS(date)	Converts a date key to string format 'yyyymmdd.'
FIXED(float)	Converts a float key to a numeric.
FLOAT(numeric)	Converts a numeric key to a float.
IIF(bool,val1,val2)	Returns val1 if the first parameter is TRUE, otherwise returns val2.
LEFT(string, n)	Returns the leftmost *n* characters of the string key as a string of length *n*.
RIGHT(string, n)	Returns the rightmost *n* characters of the string key as a string of length *n*.
RTRIM(string)	Removes spaces from the right of a string.
STR(numeric [,length [, decimal places]])	Converts a numeric to a string. The length of the string and the number of decimal places are optional. The default string length is 10, and the number of decimal places is 0.
SUBSTR(string,offset,n)	Returns a substring of the *string* key starting at *offset* and of *n* characters in length.
TRIM(string)	Removes spaces from the right of a string (identical to RTRIM).
UPPER(string)	Converts a string key to upper case.
VAL(string)	Converts a string key to a numeric.


POSITION(file) returns a STRING(12).


POSITION(key) returns a STRING the size of the key fields + 4 bytes.

dBase IV Files

The dBase4 file driver is compatible with dBase IV. The default data file extension is *.DBF.

Keys and Indexes exist as separate files from the data file. Keys are dynamicthey automatically update as the data file changes. Indexes are staticthey do not automatically update, but instead require the BUILD statement for updating. The default file extension for the index file is *.NDX.

dBase IV supports multiple index files, whose extension is *.MDX. The miscellaneous section describes procedures for using .MDX files.

The driver stores records as fixed length. It stores memo fields in a separate file. The memo file name takes the first eight characters of the File Label plus an extension of .DBT.

Files:	**CWDB416.LIB**	Windows Export Library (16-bit)
	CWDB432.LIB	Windows Export Library (32-bit)
	CLDB416.LIB	Windows Static Link Library (16-bit)
	CLDB432.LIB	Windows Static Link Library (32-bit)
	CWDB416.DLL	Windows Dynamic Link Library (16-bit)
	CWDB432.DLL	Windows Dynamic Link Library (32-bit)

Tip: dBase IV was never as widely adopted as dBase III. Choose this driver only when you must share data with an end-user using dBase IV.

dBase IV:Data Types

The xBase file format stores all data as ASCII strings. You may either specify STRING types with declared pictures for each field, or specify native Clarion types, which the driver converts automatically.

dBase data type	Clarion data type	STRING w/ picture
Date	DATE	STRING(@D12)
*Numeric	REAL	STRING(@N-_p.d)
*Logical	BYTE	STRING(1)
Character	STRING	STRING
*Memo	MEMO	MEMO

If your application reads and writes to existing files, a pictured STRING will suffice. However, if your application *creates* a dBase IV file, you may require additional information for these dBase IV types:

To create a numeric field in the Data Dictionary, choose the REAL data type. In the External Name attribute, specify '*NumericFieldName*=N(*Precision*,*DecimalPlaces*)' where *NumericFieldName* is the name of the field, *Precision* is the precision of the field and *DecimalPlaces* is the number of decimal places.

If you're hand coding a native Clarion data type, add the NAME attribute using the same syntax. If you're hand coding a STRING with picture, STRING(@N-_9.2), NAME('*Number*'), where *Number* is the field name.

To create a logical field, using the data dictionary, choose the BYTE data type. There are no special steps; however, see the miscellaneous section for tips on reading the data from the field.

If you're hand coding a STRING with picture, add the NAME attribute: STRING(1), NAME('*LogFld* = L')**.**

To create a date field, using the data dictionary, choose the DATE data type, rather than LONG, which you usually use for the TopSpeed or Clarion file formats.

MEMO field declarations require the a pointer field in the file's record structure. Declare the pointer field as a STRING(10) or a LONG. This field will be stored in the .DBF file containing the offset of the memo in the .DBT file. The MEMO declaration must have a NAME() attribute naming the pointer field. An example file declaration follows:

```
File FILE, DRIVER('dBase4')
Memo1    MEMO(200),NAME('Notes')
Memo2    MEMO(200),NAME('Text')
Rec    RECORD
Mem1Ptr    LONG,NAME('Notes')
Mem2Ptr    STRING(10),NAME('Text')
           END
     END
```

dBase IV:File Specifications/Maximums

File Size: **2,000,000,000 bytes**

```
Records per File:       1,000,000,000 bytes
Record Size:          4,000 bytes
Field Size
    Character:          254 bytes
    Date:             8 bytes
    Logical:            1 byte
    Numeric:            20 bytes including decimal point
   Float:           20 bytes including decimal point
    Memo:             64K (see note)
Fields per Record:    255
Keys/Indexes per File:
          .NDX:       No Limit
          .MDX      47 tags per .MDX files
Key Sizes
     Character:         100 bytes
     Numeric, Date:     8 bytes
Memo fields per File:  Dependent on available memory
Open Files:           Operating system dependent
```

dBase IV:Driver Strings and SEND functions

Driver strings (the second parameter of the DRIVER attribute) are all preceded by a forward slash character (/). SEND function commands can take two formatsone with an equal sign modifies a switch setting and return the value of the previous switch setting; the other format (without an equal sign) returns the value of the switch.

Driver strings are sent to the file driver when the file is opened. The SEND function sends a command to modify a setting after the file is open. Some driver strings have no effect after the file is open, so the SEND function syntax to modify the setting is not listed. However, the SEND function syntax to return the value of the switch is listed for all driver strings.

/BUFFERS=n

Specify a value for the number of buffers used to read and write to the file.

The dBase IV driver utilizes DOS buffering. The default is three buffers of 1024 bytes each. Increasing the number of buffers will not increase performance when a file is shared by multiple users.

SEND(file,'BUFFERS')

Returns the number of buffers in the form of a STRING.

/RECOVER

SEND(file,'RECOVER')

Equivalent to the Xbase RECALL command, which recovers records marked for deletion. When using the dBase IV driver, the DELETE statement flags a record as "inactive." The driver does not remove the record until the PACK command is executed.

/RECOVER is evaluated each time you open the file if you add the driver string to the data dictionary. When the driver recovers the records previously marked for deletion, you must manually rebuild keys and indexes with the BUILD statement.

SEND(file,'IGNORESTATUS=on|off')

/IGNORESTATUS=on|off

When set *on*, the driver does *not* skip deleted records when accessing the file with GET, NEXT, and PREVIOUS in file order. It also enables a PUT on a deleted or held record. /IGNORESTATUS requires opening the file in exclusive mode.

SEND(file,'IGNORESTATUS')

Returns the IGNORESTATUS setting (ON or OFF) in the form of a STRING(3).

SEND(file,'DELETED')

For use only with the SEND command, when IGNORESTATUS is on. Reports the status of the current record. If deleted, the return string is "ON;" if not, "OFF."

dBase IV:Unsupported/Modified Functions & Attributes

Memos: **BINARY**

dBase IV supports only text memos.

Keys: **OPT**

File: **ENCRYPT, OWNER, RECLAIM**

The dBase IV driver cannot read encrypted dBase IV files. To reclaim space from deleted records, call PACK(file).

Transaction Processing: **COMMIT(), LOGOUT(), ROLLBACK()**

The dBase IV driver does not support any transaction logging.

Record Access: **GET(file, fileptr, len), ADD(file, len)**

dBase IV does not support variable length records

File updates:

PUT(file, fileptr, len)

dBase IV does not support variable length records

EOF(file), BOF(file)

Although the driver supports these functions, we do not recommend their use. They must physically access the files and add overhead. Instead, test the value returned by ERRORCODE() after each sequential access. NEXT or PREVIOUS post Error 33 (Record Not Available) if an attempt is made to access a record beyond the end or beginning of the file.

ADD(file) VS. **APPEND(file)**

The ADD statement tests for duplicate keys before modifying the data file or its associated KEY files. Consequently it is slower than APPEND which performs no checks and does not update KEYs. When adding large amounts of data to a database use APPEND...BUILD in preference to ADD.

BUILD(key, str)

When building dynamic indexes, the *str* component may take one of two forms:

BUILD(DynNdx, '+Pre:FLD1, -Pre:FLD2')

This form specifies the names of the fields on which to build the index. The field names must appear as specified in the fields' NAME() attribute if supplied, or must be the label name. A prefix may be used for compatibility with the Clarion conventions but is ignored.

BUILD(DynNdx, 'T[Expression]')

This form specifies the type and Expression used to build an index, see the miscellaneous section, below.

COPY(file, newname)

The COPY() command copies data and memo files using *newname*, which may specify a new file name or directory. Key or index files are copied if the *newname* is a subdirectory specification. To copy an index file to a new file, use a special form of the copy command:

COPY(file,'<index>|<newname>')

This returns *File Not Found* if an invalid index is passed. The COPY command assumes a default extension of ".NDX" for both the source and the target file names if none is specified. If you require a file name without an extension, terminate the name with a period. Given the file structure:

```
Clar2    FILE,CREATE,DRIVER('dBase3')
NumKey   KEY(Num),DUP
StrKey   KEY(Str1)
StrKey2    KEY(Str2)
AMemo      MEMO(100), NAME('mem')
Record   RECORD
Num        STRING(@n-_9.2)
STR1       STRING(2)
STR2       STRING(2)
Mem        STRING(10)
           .   .
```

The following commands copy this file definition to A:

```
COPY(Clar2,'A:\CLAR2')
COPY(Clar2,'StrKey|A:\STRKEY')
COPY(Clar2,'StrKey2|A:\STRKEY2')
COPY(Clar2,'NumKey|A:\NUMKEY')
```

After these calls, the following files would exist on drive A: CLAR2.DBF, CLAR2.DBT, STRKEY.NDX, STRKEY2.NDX, and NUMKEY.NDX.

DELETE(file)

When the driver deletes a record from a dBase IV database, the record is not physically removed, instead the driver marks it inactive. Memo fields are not physically removed from the memo file, however they cannot be retrieved if they refer to an inactive record. Key values *are* removed from the index files. To remove records and memo fields permanently, execute a PACK(file).

Tip: To those programmers familiar with dBase IV, this driver processes deleted records in a consistent manner with the way dBase IV processes them after the SET DELETED ON command is issued. Records marked for deletion are ignored from processing by executable code statements, but remain in the data file.

HOLD(file), HOLD(file, timeout)

dBase IV performs record locking by locking the entire record within the data file. This prevents read access to other processes. Therefore we recommend minimizing the amount of time for which a record is held.

POINTER(file), POINTER(key)

There is no distinction between file pointers and key pointers; they both contain the same value for any given record.

RECORDS(file), RECORDS(key)

Under dBase IV the RECORDS() function reports the same number of records for the data file and its keys and indexes. Usually there will be no difference in the number of records *unless* the INDEX is out of date. Because the DELETE statement does not physically remove records, *the number of records reported by the* RECORDS() *function includes inactive records*. Exercise care when using this function.

RENAME(file, newname)

The RENAME command copies the data and memo files using *newname*, which may specify a new file name or directory path. Key and index files must be renamed using the same syntax as the COPY command, above.

- POSITION(file) returns a STRING(12).
- POSITION(key) returns a STRING containing the size of the key fields + 4 bytes.
- **BUILD(DynamicIndex, expression,filter)** is not supported.

dBase IV:Miscellaneous

dBase IV allows a maximum of 254 characters to a character field.

dBase IV allows a logical field to accept one of 11 possible values (1,0,y,Y,n,N,t,T,f,F or a space character). The space character is neither true nor false. When using a logical field from a preexisting database in a logical expression, account for all these possibilities. Remember that when a STRING field is used as an expression, it is true if it contains any data and false if it is equal to zero or blank. Therefore, to evaluate a Logical field's truth, the expression should be true if the field contains any of the "true" characters (T,t,Y, or y). For example, if a Logical field were used to specify a product as taxable or nontaxable, the expression to evaluate its truth would be:

(If Condition):

Taxable='1' OR Taxable='T' OR Taxable='t' OR Taxable='Y' OR Taxable='y'

Clarion for Windows supports MEMO fields up to a maximum of 64K. If you have an existing file which includes a memo greater than 64K, you can use the file but not modify the large MEMOs.

You can determine when your application encounters a large MEMO by detecting when the memo pointer variable is non-blank, but the memo appears to be blank. Error 47 (Bad Record Declaration) is posted, and any modification to the MEMO field is ignored.

dBase IV supports a maximum of 10 characters in a field name. If you require more, use an External Name with 10 characters or less.

dBase IV supports the use of expressions to define keys. Within the Dictionary Editor, you can place the expression in the external name field in the *Key Properties* dialog. The general format of the external name is :

'FileName=T[Expression]'

Where *FileName* represents the name of the index file (which can contain a path and file extension), and *T* represents the type of the index. Valid types are: C = character, D = date, and N = numeric. If the type is D or N then *Expression* can name only one field.

Multiple-index (.MDX) files require the NAME() attribute on a KEY or INDEX to specify the storage type of the key and any expression used to generate the key values. The general format of the NAME() attribute on a KEY or INDEX is:

NAME('TagName|FileName[PageSize]=T[Expression],FOR[Expression]')

The following documents the parameters for the NAME() attribute:

TagName	Specifies the name of an index tag within a multiple index file. If omitted the driver creates a dBase IV style .NDX file using the name specified in FileName.
FileName	Specifies the name of the index file, which may contain a path and extension.
PageSize	Specifies that when creating a .MDX file, (if a TagName is specified), a number in the range 2-32 specifying the number of 512-byte blocks in each index page. This value is only used when creating the file. If you specify multiple values via declarations for different tags in the same .MDX file, the largest value will be selected. The default value is 2.
T	Specifies the type of the index,. Legal types are C = character, D = date, N = numeric. If the type is D or N then *Expression* may name *only one* field.
Expression	Specifies an expression to generate the index. It may refer to multiple fields, and invoke multiple xBase functions. The functions currently supported are listed

below. Square brackets must enclose the expression.

Elements of the NAME() attribute may be omitted from the right. When specifying an Expression, you must also specify the type and name. If the Expression is omitted, the driver determines the Expression from the key fields when the file is created, or from the index file when opened.

If the type is omitted, the driver determines the index type from the first key component when the file is created, or from the index file when opened.

If the NAME() attribute is omitted altogether, the index file name is determined from the key label. The path defaults to the same location as the .DBF.

Tag names are limited to 9 characters in length. If the supplied name is too long it is automatically truncated.

Specify all field names in the NAME() attribute without a prefix.

 dBase IV additionally supports the use of the Xbase FOR statement in expressions to define keys. The expressions supported in the FOR condition must be a simple condition of the form:

expression comparison_op expression

comparison_op may be one of the following: <, <=, =<, <>, =, =>, >= or >.

The expression may refer to multiple fields in the record, and contain xBase functions. Square brackets must enclose the expression. The currently supported functions appear below. If the driver encounters an unsupported Xbase function in a preexisting file, it posts error 76 'Invalid Index String' when the file is opened for keys and static indexes.

String expressions may use the '+' operator to concatenate multiple string arguments. Numeric expressions use the '+' or '-' operators with their conventional meanings. The maximum length of a dBase IV expression is 250 characters.

Supported xBase commands

ALLTRIN(string) Removes leading and trailing spaces.

CTOD(string) Converts a string key to a date. The *string* format mm/dd/yy; the result takes the form 'yyyymmdd'. The yyyy element of the date defaults to the twentieth century. An invalid date results in a key containing blanks.

DELETED() Returns TRUE if the record is deleted.

DTOC(date) Converts a date key to string format 'mm/dd/yy.'

DTOS(date) Converts a date key to string format 'yyyymmdd.'

FIXED(float) Converts a float key to a numeric.

FLOAT(numeric) Converts a numeric key to a float.

IIF(bool,val1,val2) Returns val1 if the first parameter is TRUE, otherwise returns val2.

LEFT(string, n) Returns the leftmost *n* characters of the string key as a string of length *n*.

RIGHT(string, n) Returns the rightmost *n* characters of the string key as a string of length *n*.

RTRIM(string) Removes spaces from the right of a string.

STR(numeric [,length[, decimal places]])
converts a numeric to a string. The length of the string and the number of decimal places are optional. The default string length is 10, and the number of decimal places is 0.

SUBSTR(string,offset,n) Returns a substring of the *string* key starting at *offset* and of *n* characters in length.

TRIM(string) Removes spaces from the right of a string (identical to RTRIM).

UPPER(string) Converts a string key to upper case.

VAL(string) Converts a string key to a numeric.

DOS Files

The DOS file driver reads and writes any binary, byte-addressable files. Neither fields nor records are delimited. When reading a record, the driver reads the number of bytes defined in the file's RECORD structure, unless a length parameter is specified in the GET statement.

The DOS driver supports the length parameter for the ADD, APPEND, GET, and PUT statements; this allows for variable length records in a DOS file.

The POINTER function returns the relative byte position within the file of the beginning of the last record accessed by an ADD, APPEND, GET, or NEXT statement.

This file driver performs forward sequential processing *only*. No key or transaction processing functions are supported, and the PREVIOUS statement is not supported.

Tip: Due to its limitations, the main function of this driver is as a disk editor for binary files.

Files:	**CWDOS16.LIB**	Windows Export Library (16-bit)
	CWDOS32.LIB	Windows Export Library (32-bit)
	CLDOS16.LIB	Windows Static Link Library (16-bit)
	CLDOS32.LIB	Windows Static Link Library (32-bit)
	CWDOS16.DLL	Windows Dynamic Link Library (16-bit)
	CWDOS32.DLL	Windows Dynamic Link Library (32-bit)

DOS:Data Types

BYTE DECIMAL
SHORT PDECIMAL
USHORT STRING
LONG CSTRING
ULONG PSTRING
SREAL DATE
REAL TIME
BFLOAT4 GROUP
BFLOAT4

DOS:File Specifications/Maximums

File Size : 4,294,967,295
Records per File : 4,294,967,295
Record Size : 64K
Field Size : 64K
Fields per Record : 64K
Keys/Indexes per File: n/a
Key Size : n/a
Memo fields per File: n/a
Memo Field Size : n/a
Open Data Files : Operating system dependent

DOS:Driver Strings and SEND functions

Driver strings (the second parameter of the DRIVER attribute) are all preceded by a forward slash character (/). SEND function commands can take two formatsone with an equal sign modifies a switch setting and return the value of the previous switch setting; the other format (without an equal sign) returns the value of the switch.

Driver strings are sent to the file driver when the file is opened. The SEND function sends a command to modify a setting after the file is open. Some driver strings have no effect after the file is open, so the SEND function syntax to modify the setting is not listed. However, the SEND function syntax to return the value of the switch is listed for all driver strings.

/FILEBUFFERS=n

Specifies a value for the number of buffers used to read and write to the file.

The DOS driver allocates internal buffers of 512 bytes, or the size of your record, whichever is larger, to store the retrieved file. The default number of buffers is 2 for files opened denying write access to other users, and 1 for all other open modes. Use the optional driver string to increase the buffers should you find access to records is slow.

SEND(file, 'FILEBUFFERS')

Returns the value of the number of buffers in STRING format.

/QUICKSCAN=on|off

SEND(file,'QUICKSCAN=on|off')

The DOS driver reads a buffer at a time (not a record), allowing for fast access. In a multi-user environment these buffers are not 100% trustworthy for subsequent access, because another user may change the database between accesses. As a safeguard, the driver rereads the buffers before each record access. To *disable* the reread, set QUICKSCAN to ON. The default is ON for files opened denying write access to other users, and OFF for all other open modes.

SEND(file,'QUICKSCAN')

Returns the Quickscan setting (ON or OFF) in the form of a STRING(3).

DOS:Unsupported Functions and Attributes

Memos:	**NOMEMO()**
Transaction Processing:	**COMMIT(), LOGOUT(), ROLLBACK()**
Key Processing:	**BUILD(key), BUILD(index)** **GET(file,key), GET(key,keypointer), RESET(key,string)** **SET(file,key), SET(key), SET(key,key), SET(key,keypointer), SET(key,key,filepointer), DUPLICATE()** **POINTER(key)** **POSITION(key)** **RECORDS(key)REGET(key)**
Record Locking:	**HOLD(), RELEASE()**
File Buffering:	**STREAM()**
File Information:	**RECORDS(file)**
Sequential Processing:	**PREVIOUS(), BOF(), SKIP()**
File Manipulation:	**BUILD(), DELETE(), PACK(), WATCH(), REGET()**

DOS:Miscellaneous

POSITION(file) returns a STRING(4).

FoxPro and FoxBase Files

The FoxPro file driver is compatible with FoxPro and FoxBase. The default data file extension is *.DBF.

The default index file extension is *.IDX. The default Memo file extension is .FBT. FoxPro also supports multiple index files, whose default extension is *.CDX. The miscellaneous section describes the procedures for using the .CDX files.

Files:	**CWFOX16.LIB**	Windows Export Library (16-bit)
	CWFOX32.LIB	Windows Export Library (32-bit)
	CLFOX16.LIB	Windows Static Link Library (16-bit)
	CLFOX32.LIB	Windows Static Link Library (32-bit)
	CWFOX16.DLL	Windows Dynamic Link Library (16-bit)
	CWFOX32.DLL	Windows Dynamic Link Library (32-bit)

Tip: **The FoxPro index file format is the backbone of its vaunted "Rushmore" technology. The old saying "There's no free lunch," however, applies. Adding and appending records to a large database is a slower process than in other xBase formats, due to the time required to update the index file.**

FoxPro and FoxBase:Data Types

The xBase file format stores all data as ASCII strings. You may either specify STRING types with declared pictures for each field, or specify native Clarion types, which the driver converts automatically.

FoxPro data type	Clarion data type	STRING w/ picture
Date	DATE	STRING(@D12)
*Numeric	REAL	STRING(@N-_p.d)
*Logical	BYTE	STRING(1)
Character	STRING	STRING
*Memo	MEMO	MEMO

If your application reads and writes to existing files, a pictured STRING will suffice. However, if your application *creates* a FoxPro or FoxBase file, you may require additional information for these FoxPro and FoxBase types:

To create a numeric field in the Data Dictionary, choose the REAL data type. In the External Name attribute, specify '*NumericFieldName*=N(*Precision*,*DecimalPlaces*)' where *NumericFieldName* is the name of the field, *Precision* is the precision of the field and *DecimalPlaces* is the number of decimal places.

If you're hand coding a native Clarion data type, add the NAME attribute using the same syntax. If you're hand coding a STRING with picture, STRING(@N-_9.2), NAME('*Number*'), where *Number* is the field name.

To create a logical field, using the data dictionary, choose the BYTE data type. There are no special steps; however, see the miscellaneous section for tips on reading the data from the field.

If you're hand coding a STRING with picture, add the NAME attribute: STRING(1), NAME('*LogFld* = L').

To create a date field, using the data dictionary, choose the DATE data type, rather than LONG, which you usually use for the TopSpeed or Clarion file formats.

MEMO field declarations require the a pointer field in the file's record structure. Declare the pointer field as a STRING(10) or a LONG. This field will be stored in the .DBF file containing the offset of the memo in the .DBT file. The MEMO declaration must have a NAME() attribute naming the pointer field. An example file declaration follows:

```
File FILE, DRIVER('FoxPro')
Memo1    MEMO(200),NAME('Notes')
Memo2    MEMO(200),NAME('Text')
Rec    RECORD
Mem1Ptr    LONG,NAME('Notes')
Mem2Ptr    STRING(10),NAME('Text')
             END
     END
```

FoxPro and FoxBase:File Specifications/Maximums

File Size: **2,000,000,000 bytes**

```
Records per File:       1,000,000,000 bytes
Record Size:          4,000 bytes
Field Size
     Character:         254 bytes
     Date:              8 bytes
     Logical:         1 byte
     Numeric:         20 bytes including decimal point
     Float:           20 bytes including decimal point
     Memo:              65,520 bytes (see note)
Fields per Record:    255
Keys/Indexes per File: No Limit
Key Sizes
     Character:         100 bytes (.IDX)
              254 bytes (.CDX)
     Numeric, Date:     8 bytes
Memo fields per File:  Dependent on available memory
Open Files:           Operating system dependent
```

FoxPro and FoxBase:Driver Strings and SEND functions

Driver strings (the second parameter of the DRIVER attribute) are all preceded by a forward slash character (/). SEND function commands can take two formatsone with an equal sign modifies a switch setting and return the value of the previous switch setting; the other format (without an equal sign) returns the value of the switch.

Driver strings are sent to the file driver when the file is opened. The SEND function sends a command to modify a setting after the file is open. Some driver strings have no effect after the file is open, so the SEND function syntax to modify the setting is not listed. However, the SEND function syntax to return the value of the switch is listed for all driver strings.

/BUFFERS=n

Specify a value for the number of buffers used to read and write to the file.

The FoxPro driver utilizes DOS buffering. The default is three buffers of 1024 bytes each. Increasing the number of buffers will not increase performance when a file is shared by multiple users.

SEND(file,'BUFFERS')

Returns the number of buffers in the form of a STRING.

/RECOVER

SEND(file,'RECOVER')

Equivalent to the Xbase RECALL command, which recovers records marked for deletion. When using the FoxPro driver, the DELETE statement flags a record as "inactive." The driver does not remove the record until the PACK command is executed.

/RECOVER is evaluated each time you open the file if you add the driver string to the data dictionary. When the driver recovers the records previously marked for deletion, you must manually rebuild keys and indexes with the BUILD statement.

/IGNORESTATUS=on|off

When set *on*, the driver does *not* skip deleted records when accessing the file with GET, NEXT, and PREVIOUS in file order. It also enables a PUT on a deleted or held record. /IGNORESTATUS requires opening the file in exclusive mode.

SEND(file,'IGNORESTATUS')

Returns the IGNORESTATUS setting (ON or OFF) in the form of a STRING(3).

SEND(file,'DELETED')

For use only with the SEND command, when IGNORESTATUS is on. Reports the status of the current record. If deleted, the return string is "ON;" if not, "OFF."

FoxPro and FoxBase:Unsupported/Modified Functions & Attributes

Memos: **BINARY**

FoxPro and FoxBase support only text memos.

Keys: **DUP, NOCASE, OPT, ascending|descending**

File: **ENCRYPT, OWNER, RECLAIM**

The FoxPro driver cannot read encrypted FoxPro or FoxBase files. To reclaim space from deleted records, call PACK(file).

Transaction Processing: **COMMIT(), LOGOUT(), ROLLBACK()**

The FoxPro driver does not support any transaction logging.

Record Access: **GET(file, fileptr, len), ADD(file, len)**

FoxPro and FoxBase do not support variable length records

File updates: **PUT(file, fileptr, len)**

FoxPro and FoxBase do not support variable length records

EOF(file), BOF(file)

Although the driver supports these functions, we do not recommend their use. They must physically access the files and add overhead. Instead, test the value returned by ERRORCODE() after each sequential access. NEXT or PREVIOUS post Error 33 (Record Not Available) if an attempt is made to access a record beyond the end or beginning of the file.

ADD(file) VS. APPEND(file)

The ADD statement tests for duplicate keys before modifying the data file or its associated KEY files. Consequently it is slower than APPEND which performs no checks and does not update KEYs. When adding large amounts of data to a database use APPEND...BUILD in preference to ADD.

BUILD(key, str)

When building dynamic indexes, the *str* component may take one of two forms:

BUILD(DynNdx, '+Pre:FLD1, -Pre:FLD2')

This form specifies the names of the fields on which to build the index. The field names must appear as specified in the fields' NAME() attribute if supplied, or must be the label name. A prefix may be used for compatibility with the Clarion conventions but is ignored.

BUILD(DynNdx, 'T[Expression]')

This form specifies the type and Expression used to build an index, see

the miscellaneous section, below.

COPY(file, newname)

The COPY() command copies data and memo files using *newname*, which may specify a new file name or directory. Key or index files are copied if the *newname* is a subdirectory specification. To copy an index file to a new file, use a special form of the copy command:

COPY(file,'<index>|<newname>')

This returns *File Not Found* if an invalid index is passed. The COPY command assumes a default extension of ".IDX" for both the source and the target file names if none is specified. If you require a file name without an extension, terminate the name with a period. Given the file structure:

```
Clar2    FILE,CREATE,DRIVER('FOXPRO')
NumKey   KEY(Num),DUP
StrKey   KEY(Str1)
StrKey2    KEY(Str2)
AMemo      MEMO(100), NAME('mem')
Record   RECORD
Num        STRING(@n-_9.2)
STR1       STRING(2)
STR2       STRING(2)
Mem        STRING(10)
           .   .
```

The following commands copy this file definition to A:

```
COPY(Clar2,'A:\CLAR2')
COPY(Clar2,'StrKey|A:\STRKEY')
COPY(Clar2,'StrKey2|A:\STRKEY2')
COPY(Clar2,'NumKey|A:\NUMKEY')
```

After these calls, the following files would exist on drive A: CLAR2.DBF, CLAR2.DBT, STRKEY.IDX, STRKEY2.IDX, and NUMKEY.IDX.

DELETE(file)

When the driver deletes a record from a FoxPro or FoxBase database, the record is not physically removed, instead the driver marks it inactive. Memo fields are not physically removed from the memo file, however they cannot be retrieved if they refer to an inactive record. Key values *are* removed from the index files. To remove records and memo fields permanently, execute a PACK(file).

Tip: To those programmers familiar with FoxPro, this driver processes deleted records consistent with the way FoxPro processes them after the SET DELETED ON command is issued. Records marked for deletion are ignored from processing by executable code statements, but remain in the data file.

HOLD(file), HOLD(file, timeout)

FoxPro and FoxBase perform record locking by locking the entire record within the data file. This prevents read access to other processes. Therefore we recommend minimizing the amount of time for which a record is held.

POINTER(file), POINTER(key)

There is no distinction between file pointers and key pointers; they both contain the same value for any given record.

RECORDS(file), RECORDS(key)

Under FoxPro and FoxBase the RECORDS() function reports the same number of records for the data file and its keys and indexes. Usually there will be no difference in the number of records *unless* the INDEX is out of date. Because the DELETE statement does not physically remove records, *the number of records reported by the* RECORDS() *function includes inactive records*. Exercise care when using this function.

RENAME(file, newname)

The RENAME command copies the data and memo files using *newname*, which may specify a new file name or directory path. Key and index files must be renamed using the same syntax as the COPY command, above.


POSITION(file) returns a STRING(12).


POSITION(key) returns a STRING the size of the key fields + 4 bytes.


BUILD(DynamicIndex, expression,filter) is not supported.

FoxPro and FoxBase:Miscellaneous

FoxPro and FoxBase allow a logical field to accept one of 11 possible values (0,1,y,Y,n,N,t,T,f,F or a space character). The space character is neither true nor false. When using a logical field from a preexisting database in a logical expression, account for all these possibilities. Remember that when a STRING field is used as an expression, it is true if it contains any data and false if it is equal to zero or blank. Therefore, to evaluate a Logical field's truth, the expression should be true if the field contains any of the "true" characters (1,T,t,Y, or y). For example, if a Logical field were used to specify a product as taxable or nontaxable, the expression to evaluate its truth would be:

(If Condition):

Taxable='1' OR Taxable='T' OR Taxable='t' OR Taxable='Y' OR Taxable='y'

Clarion for Windows supports MEMO fields up to a maximum of 64K. If you have an existing file which includes a memo greater than 64K, you can use the file but not modify the large MEMOs.

You can determine when your application encounters a large MEMO by detecting when the memo pointer variable is non-blank, but the memo appears to be blank. Error 47 (Bad Record Declaration) is posted, and any modification to the MEMO field is ignored.

FoxPro and FoxBase support a maximum of 10 characters in a field name. If you require more, use an External Name with 10 characters or less.

FoxPro and FoxBase support the use of expressions to define keys. Within the Dictionary Editor, you can place the expression in the external name field in the *Key Properties* dialog. The general format of the external name is :

'FileName=T[Expression]'

Where *FileName* represents the name of the index file (which can contain a path and file extension), and *T* represents the type of the index. Valid types are: C = character, D = date, and N = numeric. If the type is D or N then *Expression* can name only one field.

Multiple-index (.CDX) files require the NAME() attribute on a KEY or INDEX to specify the storage type of the key and any expression used to generate the key values. The general format of the NAME() attribute on a KEY or INDEX is:

NAME('TagName|FileName[PageSize]=T[Expression],COMPRESSED')

The following documents the parameters for the NAME() attribute:

TagName	Names an index tag within a multiple index file. If the TagName is omitted the driver creates an .IDX file with the name specified in FileName.
FileName	Names the index file, and optionally contains a path and extension.
PageSize	May only be specified when creating a .CDX file (if a TagName is specified). It is a number in the range 2-32 specifying the number of 512-byte blocks in each index page. This value is only used when creating the file. If multiple values are specified via declarations for different tags in the same .MDX file, the largest value will be selected. The default value is 2.
T	Specifies the type of the index; legal types are C = character, D = date, N = numeric. If the type is D or N then *Expression* may name only one field.
Expression	Specifies the expression used to generate the index. The expression may refer to multiple fields, and invoke multiple of xBase functions. The functions currently supported are listed below. Square brackets must enclose the expression.

COMPRESSED When specified, the FoxPro Driver creates a FoxPro 2 compatible compressed .IDX file.

Elements of the NAME() attribute may be omitted from the right. When specifying an Expression, the type and name must also be specified. If the Expression is omitted, the driver determines the Expression from the key fields when the file is created, or from the index file when opened.

If the type is omitted, the driver determines the index type from the first key component when the file is created, or from the index file when opened.

If the NAME() attribute is omitted altogether, the index file name is determined from the key label. The path defaults to the same location as the .DBF.

Tag names are limited to 9 characters in length; if the supplied name is too long it is automatically truncated.

All field names in the NAME() attribute must be specified without a prefix.

 FoxPro additionally supports the use of the Xbase FOR statement in expressions to define keys. The expressions supported in the FOR condition must be a simple condition of the form:

expression comparison_op expression

comparison_op may be one of the following: <, <=, =<, <>, =, =>, >= or >.

The expression may refer to multiple fields in the record, and contain xBase functions. Square brackets must enclose the expression. The currently supported functions appear below. If the driver encounters an unsupported Xbase function in a preexisting file, it posts error 76 'Invalid Index String' when the file is opened for keys and static indexes.

String expressions may use the '+' operator to concatenate multiple string arguments. Numeric expressions use the '+' or '-' operators with their conventional meanings. The maximum length of a FoxPro or FoxBase expression is 250 characters.

Supported xBase Commands

ALLTRIN(string) Removes leading and trailing spaces.

CTOD(string) Converts a string key to a date. The *string* format mm/dd/yy; the result takes the form 'yyyymmdd'. The yyyy element of the date defaults to the twentieth century. An invalid date results in a key containing blanks.

DELETED() Returns TRUE if the record is deleted.

DTOC(date) Converts a date key to string format 'mm/dd/yy.'

DTOS(date) Converts a date key to string format 'yyyymmdd.'

FIXED(float) Converts a float key to a numeric.

FLOAT(numeric) Converts a numeric key to a float.

IIF(bool,val1,val2) Returns val1 if the first parameter is TRUE, otherwise returns val2.

LEFT(string, n) Returns the leftmost *n* characters of the string key as a string of length *n*.

RIGHT(string, n) Returns the rightmost *n* characters of the string key as a string of length *n*.

RTRIM(string) Removes spaces from the right of a string.

STR(numeric [,length[, decimal places]])
converts a numeric to a string. The length of the string and the number of decimal places are optional. The default string length is 10, and the number of decimal places is 0.

SUBSTR(string,offset,n) Returns a substring of the *string* key starting at *offset* and of *n* characters in length.

TRIM(string) Removes spaces from the right of a string (identical to RTRIM).

UPPER(string) Converts a string key to upper case.

VAL(string) Converts a string key to a numeric.

TopSpeed Database Files

The TopSpeed Database file system is a high-performance, high-security, proprietary file driver for Clarion development tools. It is *not* compatible with Clarion 2.1 and 3.0 files.

Data tables, keys, indexes and memos can all be stored together in a single DOS file. The default file extension is *.TPS. A separate "Transaction Control File" takes the *.TCF extension.

The TopSpeed driver can optionally store multiple tables in a single DOS file. This allows you to open as many data tables, keys and indexes as necessary using *a single DOS file handle*. This feature may be especially useful when there are a large number of small tables, or when a group of related files are normally accessed together. All keys, indexes, and Memos are always stored internally.

In addition, the TopSpeed file system supports the **BLOB** data type (Binary Large OBject), a string field which is completely variable-length and may be greater than 64K in size (in both 16 and 32-bit applications). A BLOB must be declared before the RECORD structure. Memory for a BLOB is dynamically allocated and de-allocated as necessary. For more information, see *BLOB* in Chapter 10 of the *Language Reference.*

Files:	**CWTPS16.LIB**	Windows Export Library (16-bit)
	CWTPS32.LIB	Windows Export Library (32-bit)
	CLTPS16.LIB	Windows Static Link Library (16-bit)
	CLTPS32.LIB	Windows Static Link Library (32-bit)
	CWTPS16.DLL	Windows Dynamic Link Library (16-bit)
	CWTPS32.DLL	Windows Dynamic Link Library (32-bit)

Tip: This new driver offers speed, security, and takes up fewer resources on the end users system.

See Also:

TopSpeed:Storing multiple Tables (data files) in a single DOS file.

TopSpeed Database Recovery Utility

TopSpeed:Data Types

BYTE DECIMAL
SHORT STRING
USHORT CSTRING
LONG PSTRING
ULONG MEMO
SREAL GROUP
REAL BLOB

TopSpeed:Maximum File Specifications

File Size : Limited only by disk space
Records per File : Unsigned Long (4,294,967,295)
Record Size : 64K
Field Size : 64K
Fields per Record : 64K
Keys/Indexes per File: 240
Key Size : 64K
Memo fields per File: 255
Memo Field Size : 64K
BLOB fields per File: 255
BLOB Size : Hardware dependent
Open Data Files : Operating system dependent

TopSpeed:Driver Strings and SEND functions

Driver strings (the second parameter of the DRIVER attribute) are all preceded by a forward slash character (/). SEND function commands can take two formatsone with an equal sign modifies a switch setting and return the value of the previous switch setting; the other format (without an equal sign) returns the value of the switch.

Driver strings are sent to the file driver when the file is opened. The SEND function sends a command to modify a setting after the file is open. Some driver strings have no effect after the file is open, so the SEND function syntax to modify the setting is not listed. However, the SEND function syntax to return the value of the switch is listed for all driver strings.

SEND(file,'TCF=file name')

/TCF=file name Specifies a transaction control file other than the default \TOPSPEED.TCF. The file holds all multi-file commits until the program terminates or a SEND(TCF=file name) executes.

SEND(file,'PNM=name')

/PNM=name Retrieves the names of the tables in a group (a single DOS file).

To retrieve the first name, issue this command: SEND(file,'PNM='). This returns the name of the first table. Subsequent calls pass the name received and return the next name.

For example, given a file with three tablesSupp, Part, and Ship, the example below displays an alphabetical listing:

CODE

```
name = ' '
  LOOP
  name = (SEND(Supp,'PNM=' & name)
    If name
      DISPLAY name
    ELSE
      BREAK
    END
  END
```

TopSpeed:Unsupported Functions and Attributes

- Record Access: **GET(file, fileptr, len), ADD(file, len), APPEND(file, len)**

  The TopSpeed Driver does not support variable length records

  GET(file,1)

  This relies on a valid pointer returned from the POINTER() function. You cannot use GET(file,1) to retrieve the first record because 1 is not a valid pointer.

- File updates: **PUT(file, fileptr, len)**

  The TopSpeed Driver does not support variable length records

- Keys: **NAME()**

  The TopSpeed Driver does not support external names for keys. All keys are stored internally.

TopSpeed:Miscellaneous

SHARE and open access modes:

The following open access modes are supported *Share required*

34 (12h) Read/Write, deny write (default for OPEN) Yes

```
66 (42h) Read/Write, deny none (default for SHARE)  Yes
64 (40h) Read Only,  deny none  Yes
18 (12h) Read/Write, deny all No
16 (10h) Read Only,  deny all No
32 (20h) Read Only,  deny write No
```

For the modes indicated, SHARE.EXE (which implements DOS record locking) must be loaded in AUTOEXEC.BAT or CONFIG.SYS. The following example loads SHARE in AUTOEXEC.BAT, providing 500 maximum file locks, and the default 2048 bytes for the storage area.

C:\DOS\SHARE.EXE /L:500

If SHARE.EXE is required but not loaded, the program generates a runtime error when OPEN or SHARE is called (deny none modes), or when an update is attempted (deny write modes).

APPEND()

APPEND() is recommended over ADD() if the total size of the keys exceeds the amount of RAM available, if there is more than one key, or when adding a large number of records. The size of a key (for this purpose) is the number of entries times (the sum of key fields + 10 bytes). If the records being added are already in an approximate key order, then you can discount that key for the purposes of the above calculation.

As an example, if a file has two 40 byte keys and 2 Megabytes of RAM are available, then ADD() becomes (relatively) slow when the database size exceeds about 2,000,000 / (40 + 10 + 40 + 10) = 20,000 records.

BUILD(file), BUILD(key)

The TopSpeed driver implements incremental building; this means that building a key only reads records starting from the first record appended since the key was last built. The driver merges the new keys with the existing key. Thus building a large key where only a few recently added records have been modified should be *fast*. Building an index is similar, but must start at the minimum physical record whose position in the index has changed since the index was last built.

Dynamic indexes are not retained, so cannot be built incrementally.

LOCK(file)

LOCK() only affects other LOCK() calls. The only effect of a successful call to LOCK() is that other processes will get an error FLALLK when they call LOCK().

LOGOUT(), COMMIT(), ROLLBACK()

A transaction control file is used to ensure that transactions which update more than one DOS file are committed atomically. By default the transaction control file (.TCF) has the name "\TOPSPEED.TCF." A SEND() command allows you to change this.

The .TCF file must be accessible when any files controlled by it are accessed. If a transaction involves updating more than one shared network file, you should specify a transaction control file on the network. It is not necessary to use the same TCF file for all transactions; however, it must reside where it can be read by everyone accessing the file. If not, after a crash/power-fail during a COMMIT(), some files may be updated, and others not. (The files will not be corrupted - they may just not be consistent with one another).

A .TCF file can be deleted only if all files controlled by it may have been opened (for writing) since a crash/power-fail.

POINTER(key)

The value returned by POINTER(key) corresponds to a physical data record. Consequently when that record is removed by a call to DELETE() the pointer becomes invalid. Any subsequent access using the pointer fails. If you require fuzzy matching whereby the nearest record is returned, use the POSITION() function and appropriate access functions.

STREAM(), FLUSH()

When reading a large number of records, use STREAM() or open the file in a deny write mode e.g. OPEN(f) rather than SHARE(f). After the records have been read, call FLUSH() to allow other users access.

> It is very important to use STREAM() when adding/appending/putting a large number of records. STREAM() will typically make processing about 20 times faster. For example, adding 1000 records might take nearly 2 minutes without STREAM(), but *only 5 seconds* with STREAM. It is not necessary to use STREAM() or FLUSH() on a logged out file (performance on logged out files is always good).

Tip: **When utilizing STREAM() to update a large number of records, the driver stores uncommitted or unflushed pages in memory, and it is possible to run out of memory. Calling COMMIT(), FLUSH(), or LOGOUT() periodically prevents this. To calculate the maximum "updates" between each COMMIT(), divide the available memory by the update size. When appending, the update size is approximately the size of the record in bytes. When adding, the update size is approximately the size of the records and key component fields in bytes. When updating records using PUT(), it's theoretically possible for the update size to reach 7K. In practice, we recommend committing data every 100 or so updates.**

- POSITION(file) returns a STRING the size of the key fields + 4 bytes.
- POSITION(key) returns a STRING the size of the key fields + 4 bytes.

TopSpeed:Storing multiple Tables (data files) in a single DOS file.

By using the special escape sequence '\!' in the NAME() attribute of a TopSpeed file declaration, you can specify that a single DOS file will store more than one table. For example, to declare a single DOS file 's&p.tps' which is to contain 3 logical tables, called *supp*, *part* and *ship*:

```
Supp  FILE,DRIVER('TopSpeed'),PRE(Supp),CREATE,NAME('S&P\!Supp')
      ...
Part  FILE,DRIVER('TopSpeed'),PRE(Part),CREATE,NAME('S&P\!Part')
      ...
Ship  FILE,DRIVER('TopSpeed'),PRE(Ship),CREATE,NAME('S&P\!Ship')
      ...
```

The data files share a single DOS file handle, opened when the first file is opened, and closed when the last file is closed. The first open mode determines the open mode for *all* the other files. If the first open mode is read-only, then no updates of any kind can be performed successfully (ACCDNID will be returned).

If one file in a group is logged out, then all the files in the group are effectively logged out. If one file in a group is flushed, then all files in the group are flushed.

This feature is especially useful when there are a large number of small tables, or when the application must normally access group of related files together.

If no escape sequence is specified, then a default table name 'unnamed' is supplied, so that the following are all equivalent:

```
foo     FILE,DRIVER('TopSpeed')
foo       FILE,DRIVER('TopSpeed'),NAME('foo')
foo       FILE,DRIVER('TopSpeed'),NAME('foo\!unnamed')
```

A SEND() command allows the programmer to determine the names of the files within a group. Files can be renamed within a group; for example, given the above declarations the following command will rename the file called Supp to Old_Supp:

```
RENAME(Supp,'S&P\!Old_Supp')
```

Renaming to another existing group normally involves copying/removal, so is less efficient.

If your are using the OWNER attribute on multiple tables in a TopSpeed database file, all tables *must* have the same OWNER attribute.

TopSpeed Database Recovery Utility

The TopSpeed file system is designed to automatically repair most errors. If the data file is physically damaged during a system malfunction, the TopSpeed Database Recovery Utility can recover the undamaged portions of your data.

Note: The TopSpeed Database Recovery Utility is an emergency repair tool and should not be used on a regular basis. Use it only when a file has been damaged.

The TopSpeed Database Recovery Utility reads the damaged file and writes the recovered records to a new file. It uses the information stored in the file's header or scans the file recovering undamaged portions. Optionally, you can provide an example file containing table (individual file) and key layout.

The TopSpeed Database Recovery Utility is a freely distributable utility designed to enable your end users to recover damaged files.

The recovery utility is designed to work interactively or transparently via command line parameters. Interactively, you can use the utility to recover damaged files and provide the parameters via two wizard dialogs. Using the command line parameters, you can incorporate it in your application using a RUN() statement or create a shortcut (in Windows 95) or Program Manager Icon (in Windows 3.1x) with the parameters to enable end users to recover data files.

See Also:

Using the TopSpeed Database Recovery Utility Interactively

TopSpeed Database Recovery Utility:Command Line Parameters

Using the Utility in your Application

Running the TopSpeed Database Recovery Utility

Using the TopSpeed Database Recovery Utility Interactively

1. Start the utility by double-clicking on the TopSpeed Database Recovery Utility Icon In the Clarion for Windows 1.5 Program Group.

   The **TopSpeed Database Recovery Utility** dialog appears. The utility consists of two wizard dialogs.

2. In the **Source** (file to recover) section, specify the file name or press the **Browse** button to select it from a standard file open dialog.

3. If the file has a password, type it in the **Password** entry box.

   If the database file contains multiple tables (data files), each table *must* have the same password.

4. Optionally, in the **Destination** (result file) section, specify the file name for the target file or press the **Browse** button to select it from a standard file open dialog.

   By default the .TPR extension is added to the source file name. This parameter is optional. If omitted, the original (source file) is overwritten and a backup file is created. The source file is renamed to *filename.*TPx*,* where x is automatically incremented from 1 to 9 each time a new file is created. If all nine numbers are used, any subsequent files created are given the extension .TP$ and are overwritten.

5. If the result file is to have a different password, type it in the **Password** entry box. If omitted, the password is removed.

6. Press the **Next** button.

   The second wizard dialog for the TopSpeed Database Recovery Utility appears.

7. Optionally, specify the **Example File** file name or press the **Browse** button to select it from a standard file open dialog.

   The utility uses the Example File to determine table layouts and key definitions in the event those areas of the source file are damaged. The default extension is .TPE, but if you choose, you may use any valid DOS extension

Tip: We recommend shipping an example file when you deploy your application. This improves data recovery from a damaged file.

8. If the example file has a password, type it in the **Password** entry box.

9. If you want the utility to rebuild Keys, check the **Build Keys** box.

   If omitted, the keys are rebuilt by the original application when it attempts to open it.

10. If you want to use the Header Information in the source file, check the **Use Header** box.

    Utilizing Header Information optimizes the utility's performance, but should not be used if the file header is corrupt. If omitted, the utility searches the entire data file and restores all undamaged pages.

11. If the application uses a Locale (.ENV) File for an alternate collating sequence, specify the .ENV file or press the **Browse** button to select it from a standard file open dialog.

12. If the file is using the OEM attribute to control the collating sequence, Check the **Use OEM** box.

This enables the OEMTOANSI and ANSITOOEM conversion.

13. Press the **Start** button to begin the recovery process.

If the utility does not find any errors, a message appears informing you that "No Errors Detected in <fliename.ext>" and asks if you want to continue with recovery.

TopSpeed Database Recovery Utility:Command Line Parameters

The utility can also accept command line parameters which enables you to execute it from an application or Program Manager Icon (or Shortcut in Windows 95).

TPSFIX *sourcepath*[*?password*] [*destpath*[*?password*]]
[*/E:examplepath*[*?password*]] [*/L:localepath*] [*/H*] [*/K*] [*/P*] [/O]

sourcepath	The file name and path of the source (damaged) database file.
[?password]	The database file's password.
destpath	The file name and path of the recovered database file.
[?password]	The recovered database file's password.
/E:*examplepath*	The file name and path of the example database file.
[?password]	The example database file's password
/L:*localepath*	The Locale (.ENV) file used to specify an alternate collating sequence.
/H	If specified, the utility uses the header information in the source file.
/K	If specified, the utility rebuilds all keys for the database.
/P	If specified, the user is prompted for each parameter even if they are supplied on the command line.
/O	If specified, the file uses OEMTOANSI and ANSITOOEM to determine the collating sequence. See *Internationalization* in Chapter 10 of the *Language Reference.*

Using the Utility in your Application

There are some issues to consider before allowing the utility to run.

- The database file should NOT be open when running the utility. Ensure that the file is closed before allowing the user to start the utility.
- To prevent access during the recovery process is completed, the utility locks the file automatically.
- It is more efficient and safer to allow the application to rebuild the KEYs (by omitting the /K parameter in the recovery). It is also a good way to check the status of a recovery.

Running the TopSpeed Database Recovery Utility

There are basically two methods you can use from a RUN() statement: Using the first method, you omit the *destpath* parameter so the original (source) file is overwritten. This requires an Example file.

In the Application Generator:

1. In the **Actions** dialog for a button or menu item, choose *Run a Program* from the drop down list.

2. In the **Program Name** entry box, specify *TPSFIX.EXE.*

3. In the parameters entry box, specify the parameters (see *Command Line Parameters* above).

For Example:

TPSFIX.EXE Datafile.TPS /E:Example.TPE /H

In Embedded Source Code:

RUN('TPSFIX.EXE Datafile.TPS /E:Example.TPE /H')

This recovers the "datafile.TPS" file using the "Example.TPE" file as an example for the table and key layouts, does not rebuild the keys, and uses the header information in the original file. The original file is saved to a backup file with an extension of TP1 through TP9. Each time the utility is executed, the numeric portion of the extension is incremented.

The second method requires two lines of embedded source code but gives you control over the renaming process. You insert the source code in the Accepted Embed point for the Menu Item or button.

For example:

```
COPY(datafilelabel, 'Datafile.OLD')        !  copies the original file
                                           !  to Datafile.OLD
RUN(TPSFIX Datafile.OLD Datafile.tps /H)  !  Runs the utility using the
                                           !  renamed file as
                                           !  the source and the original
                                           !  name as the target
```

This copies the datafilelabel file to DATAFILE.OLD, recovers the file and writes it to DATAFILE.TPS using the header information in the original file.