OMNIS Studio Reference

OMNIS Software

September 1998

Table of Contents

About This Manual

This manual contains a description of all the following OMNIS objects

❑ *Functions* and external functions

❑ *Hash variables*

❑ *Event Codes* and event parameters

❑ *Methods*

❑ *Commands*

❑ *External commands*

To program in OMNIS you should be familiar with the method editor, and using the OMNIS commands and notation. All these topics are discussed in detail in the *OMNIS Programming* manual.

Note that object properties and the OMNIS constants are not listed here since the vast majority of them are self-explanatory. You can view the properties of any object using the Property Manager and Notation Inspector, and you can view the OMNIS constants in the Catalog. For your convenience, all objects including properties, methods, and constants are listed in the OMNIS Help.

Your Notes

Chapter 1—Functions

OMNIS provides a vast array of *functions* for manipulating numbers, strings, binarys, and dates, and for performing complex calculations and trigonometric operations. You can browse and access the OMNIS functions in the Catalog. You can use the functions in any calculation, and in any text string using square bracket notation. This chapter lists all the functions available in OMNIS in alphabetical order, and includes the FileOps and FontOps external functions at the end of the chapter.

Generally the functions accept one or more string or numeric values and return a value. They have no direct effect on the flag, although most of the functions have a true result for a successful operation (that is, any non-zero result) and false for failure (or no result), so you can test the result of a calculation containing a function.

To select a function

- Press F9/Cmnd-9 to display the **Catalog**

- Click on the **Functions** tab

- Click on the function group you require in the left-hand list

- Double-click on the function you require in the right-hand list

or you can

- Drag the function from the Catalog and drop it into a calculation field in the method editor

Syntax

All possible arguments to each function are in italic and parentheses. Square brackets further enclose optional arguments ([,*number2*]), for example; do not type the square brackets when you enter an optional argument.

Strings appear as *string*, numerical values as *number*, list names as *listname*, and so on. Where a function requires more than one argument of the same type, the description appends a number to the word: for example, *string1* is the first string, *string2* is the second string, and so on. Literal strings are always quoted.

Some functions take a list of values or arguments; for example the fld(*string1*[,*string2*]...) function. This means you must enter at least one string and any number of subsequent strings, separated by commas.

Functions

abs()

abs(*number*)

Returns the magnitude of a real *number* ignoring its positive or negative sign.

```
abs(1002)        ;; returns 1002
abs('-203.45')   ;; returns 203.45
abs('12ABC')     ;; returns 0
```

acos()

acos(*number*)

Returns the arc cosine of a *number* in the range 0 to 180 degrees (0 to pi radians if #RAD is true), or returns 0 if the number is not in the range -1 to 1.

```
acos(sqr(2)/2)   ;; returns 45
```

ann()

ann(*rate,nper,pmt,pv,fv*[,*prd*])

Evaluates an unknown for an annuity. The first five arguments are mandatory, but you can replace any one of them with '?' which is the unknown value returned by the function.

Parameters: *rate* is the interest rate per payment period; *nper* is the number of payment periods, which should be an integer greater than zero; *pmt* is the payment made to you (or by you) at the end of each period; *pv* is the amount paid to you (or by you) at the start of the first period; *fv* is the amount paid to you (or by you) at the end of the final period; *prd* is optional and represents a specific period which must be between 1 and *nper*; it is used for obtaining the split of interest and capital payments in a period.

The convention is that positive values for *pmt, pv* and *fv* denote a payment made *to you*, and negative amounts denote a payment made *by you*. It is important to ensure that *rate, nper* and *pmt* all refer to the same period length. The annuity is evaluated so that the sum of all payments made to you (or by you) when compounded at the interest rate evaluates to zero.

For example, a 25 year mortgage for $30000 at 11% pa interest payable monthly in arrears has monthly payments, paid by you, equal to:

```
rnd(ann(.11/12,25*12,'?',30000,0),2)    ;; returns -294.03
```

anna()

anna(*rate,nper,pmt,pv,fv*)

Evaluates an unknown for an annuity in advance. This function works in the same way as the *ann()* function except that the annuity is calculated in advance. The payments are assumed to be made at the start of each period instead of the end of each period. For example, using the same arguments as those used for the example for *ann(),* but using the *anna()* function, where the mortgage payments are assumed to be made in advance, the monthly payment is:

```
rnd(anna(.11/12,25*12,'?',30000,0),2)    ;; returns -291.36
```

ansichar()

ansichar(*code*)

Returns a string containing the ANSI symbol for the specified *code* (available under Windows only). You can display the result with an ANSI or TrueType font like Times New Roman or Arial.

ansicode()

ansicode(*string,index*)

Returns the ANSI character code for the specified *string* (available under Windows only).

asc()

asc(*string,number*)

Returns the ASCII value of a character in a *string.* The position of the character is specified by *number.* The value returned is between 0 and 255, or -1 if *number* is less than 1 or greater than the length of *string*.

```
asc('Quantity',1)
; returns 81, that is the ASCII value of the 1st character
asc('Car',3)
; returns 114, that is the ASCII value of the 3rd character
asc('Train',9)       ;; returns -1
```

asin()

asin(*number*)

Returns the arc sine of a *number* in the range -90 to 90 degrees (-pi/2 to pi/2 radians if #RAD is true), or returns 0 if the number is not in the range -1 to 1.

```
asin(sqr(3)/2)      ;; returns 60
```

atan()

atan(*number*)

Returns arc tangent of a *number* in range -90 to 90 degrees (-pi/2 to pi/2 radians if #RAD is true).

```
atan(1)        ;; returns 45
```

atan2()

atan2(*y,x*)

Returns the arc tangent of the point with *y,x* coordinates.

```
atan2(1,1)      ;; returns 45
```

avgc()

avgc(*listname,column*[,*ignore_nulls*])

Returns the average value for a list column specified by *listname* and *column.* If you set *ignore_nulls* to 1, null values are ignored and not counted. If you omit this parameter or it evaluates to zero, nulls are treated as zero values and are counted.

```
Calculate LVAR30 as avgc(PLIST,Age,1)
; returns the average for Age not including null or zero values
```

bdif()

bdif(*oldbinary,newbinary*)

Returns a binary representation of the differences between *oldbinary* and *newbinary*. It is useful for comparing different versions of the same file, whether it is an OMNIS library, external component, picture or text file, and so on.

binchecksum()

binchecksum(*binary*)

Calculates a checksum for a *binary* field. OMNIS generates the checksum by summing the bytes of the binary field, using a 32 bit number, ignoring overflow.

```
Calculate CHECKSUM as binchecksum(binary)
```

bincompare()

bincompare(*binary1, binary2*)

Compares two binary fields, *binary1* and *binary2*. Returns true if they are equal, and false if they are not. Fields of different length are *not* equal, meaning that the rule about extending the length of the shortest field does not apply in this case.

```
Calculate LVAR1 as bincompare(binary1, binary2)
```

binfromhex()

binfromhex(*string*)

Returns a binary field value generated from the specified character *string*. The character string encodes a hexadecimal value in ASCII. The *string* must not contain a leading 0x or 0X.

```
Calculate BINARY as binfromhex(string)
```

binfromlong()

binfromlong(*longint*)

Returns a binary field value containing the binary representation of a long integer *longint*. The binary value has a length of 4 bytes. Bit zero of the returned value is the most significant bit of the long, and bit 31 is the least significant bit. For example, the long value 0x12345678 is returned with byte 0 = 0x12, byte 1 = 0x34, byte 2 = 0x56 and byte 3 = 0x78.

```
Calculate BINARY as binfromlong(longint)
```

binlength()

binlength(*binary*)

Returns the length of a *binary* field, in bytes.

```
Calculate LENGTH as binlength(binary)
```

bintohex()

bintohex(*binary*)

Returns a character string representing the value of a *binary* field, in ASCII hexadecimal.

```
Calculate STRING as bintohex(binary)
```

bintolong()

bintolong(*binary*)

Returns a long value from the first 4 bytes of a *binary* field. For example, if the binary field contains 0x12345678, the returned long has the value 0x12345678.

```
Calculate LONG as bintolong(binary)
```

bitand()

bitand(*binary1, binary2*)

Performs an AND operation on *binary1* and *binary2*, and returns the result.

```
Calculate BINARY as bitand(binary1, binary2)
```

bitclear()

bitclear(*binary1, firstBitNumber, secondBitNumber*)

Clears a range of bits in a single argument *binary1* by setting them to zero, that is, *bitclear()* clears all bits with numbers >= *firstBitNumber* and <= *secondBitNumber*.

The function operates directly on the *binary1* argument, and returns 1 for success and 0 for failure. If the bit numbers identify some bits which are outside the current length of the binary field, OMNIS extends the field by appending bytes with value zero, and clears the bits.

```
Calculate STATUS as bitclear(binary1, firstBitNumber,
  secondBitNumber)
```

bitfirst()

bitfirst(*binary*)

Returns the number of the most significant bit with value 1 in a *binary* field.

```
Calculate NUMBER as bitfirst(binary)
```

sets NUMBER to the bit number of the first bit set to 1. If all bits are zero, *bitfirst()* returns -1.

bitmid()

bitmid(*binary1, firstBitNumber, secondBitNumber*)

Generates a binary field value identified as a range of bits of a *binary* field, that is, *bitmid()* extracts the bits with numbers >= *firstBitNumber* and <= *secondBitNumber*.

```
Calculate BINARY as bitmid(binary1, firstBitNumber, secondBitNumber)
```

Bit *firstBitNumber* of *binary1* becomes bit zero of BINARY, and so on.

bitnot()

bitnot(*binary1*)

Performs the 1's complement of a single argument. The function operates directly on the argument *binary1*, and returns 1 for success and 0 for failure.

```
Calculate STATUS as bitnot(binary1)
```

bitor()

bitor(*binary1, binary2*)

Performs an inclusive-OR on *binary1* and *binary2*, and returns the result.

```
Calculate BINARY as bitor(binary1, binary2)
```

bitrotatel()

bitrotatel(*binary, count*)

Rotates a *binary* field to the left, by a number of bits specified in *count*. The function operates directly on the argument, and returns 1 for success and 0 for failure. The vacated bits are replaced by the bits shifted off the left-hand end.

If the specified number of bits is greater than the bit-length of the field, OMNIS returns 0, and the field is unchanged.

```
Calculate STATUS as bitrotatel(binary, count)
```

bitrotater()

bitrotater(*binary, count*)

Rotates a *binary* field to the right, by a number of bits specified in *count*. The function operates directly on the argument, and returns 1 for success and 0 for failure. The vacated bits are replaced by the bits shifted off the right-hand end.

If the specified number of bits is greater than the bit-length of the field, OMNIS returns 0, and the field is unchanged.

```
Calculate STATUS as bitrotater(binary, count)
```

bitset()

bitset(*binary, firstBitNumber, secondBitNumber*)

Sets a range of bits in a single argument to 1, that is, *bitset()* sets all bits in a *binary* field with numbers >= *firstBitNumber* and <= *secondBitNumber*.

The function operates directly on the argument, and returns 1 for success and 0 for failure. If the bit numbers identify some bits which are outside the current length of the binary field, OMNIS extends the field by appending bytes with value zero, and sets the bits.

```
Calculate STATUS as bitset(binary, firstBitNumber, secondBitNumber)
```

bitshiftl()

bitshiftl(*binary, count*)

Shifts a *binary* field to the left, by a number of bits specified in *count*. The function operates directly on the argument, and returns 1 for success and 0 for failure. Vacated bits become zero. Bits shifted past bit 0 are lost.

```
Calculate STATUS as bitshiftl(binary, count)
```

bitshiftr()

bitshiftr(*binary, count*)

Shifts a *binary* field to the right, by a number of bits specified in *count*. The function operates directly on the argument, and returns 1 for success and 0 for failure. Vacated bits become zero. Bits shifted past the right-most bit are lost.

```
Calculate STATUS as bitshiftr(binary, count)
```

bittest()

bittest(*binary, firstBitNumber, secondBitNumber*)

Tests a range of bits in a single argument, that is, *bittest()* tests all bits in a *binary* field with numbers >= *firstBitNumber* and <= *secondBitNumber*. If any are 1, the function returns 1, otherwise it returns zero.

```
Calculate BOOL as bittest(binary, firstBitNumber, secondBitNumber)
```

bitxor()

bitxor(*binary1, binary2*)

Performs an exclusive-OR (XOR) on *binary1* and *binary2*, and returns the result.

```
Calculate BINARY as bitxor(binary1, binary2)
```

bundif()

bundif(*differences,binary*)

Restores an older version of a *binary* file using the *differences* created by the *bdif()* function. The differences must be passed to an older version of the *same* binary file.

bytecon()

bytecon(*binary1, binary2*)

Concatenates two binary fields *binary1* and *binary2*, and returns the result. Note that *bytecon()* concatenates *binary2* on to the end of *binary1*.

```
Calculate BINARY as bytecon(binary1, binary2)
```

bytemid()

bytemid(*binary1, firstByteNumber, secondByteNumber*)

Generates a binary field value identified as a range of bytes in a binary field, that is

```
Calculate BINARY as bytemid(binary1, firstByteNumber,
  secondByteNumber)
```

sets BINARY to the value generated by extracting bytes *firstByteNumber* to *secondByteNumber* inclusive of *binary1*. Thus byte 0 of BINARY becomes byte *firstByteNumber* of *binary1,* and so on.

byteset()

byteset(*binary1, byteNumber, value*)

Sets a byte in a binary field to a specified value, that is, *byteset()* sets the byte *byteNumber* of *binary1* to *value*. The function operates directly on the argument, and returns 1 for success and 0 for failure.

```
Calculate STATUS as byteset(binary1, byteNumber, value)
```

cap()

cap(*string*)

Returns the capitalized representation of a *string*, that is, the first letter of each and every word in the string is capitalized.

```
cap('gRaVeS, hutton, MONKS')  ;; returns 'Graves, Hutton, Monks'
cap('on the 8TH day')         ;; returns 'On The 8th Day'
```

cdif()

cdif(*class1,class2*)

Returns a list of differences between two classes *class1* and *class2*, the first parameter is the older class, and the second parameter is the newer class.

The *cdif()* function returns a binary representation of differences between two OMNIS library classes of the same type, for example, you can compare two versions of the same window class.

If an error occurs during execution, the flag is set to false. #ERRCODE will contain the error number, and #ERRTEXT will contain the error text. Examples of error text for *cdif()* are:

```
"Classes are of different types and cannot be compared."
"One of the classes to be compared has an invalid structure. (One of
  the classes may be corrupt.)"
```

An #ERRCODE value of 8095 means that the classes are identical. If no error occurs, the returned binary representation will contain data items of objects which have changed between the two classes.

```
; Define local vars DIF_LIST, OLD_CLASS, and
; NEW_CLASS with Binary type
Calculate OLD_CLASS as $windows.window1.$classdata
Calculate NEW_CLASS as $windows.window2.$classdata
Calculate DIF_LIST as cdif(OLD_CLASS,NEW_CLASS)
If (#ERRCODE)
   OK message (High position,Sound bell) {[#ERRTEXT]}
   Quit method kTrue
End If
```

chk()

chk(*string1,string2,string3*)

Returns true or false depending on a character-by-character comparison of *string1* with *string2* and *string3* using the ASCII value of each character for the basis of the comparison.

Firstly, each character of *string2* is compared with the corresponding character of *string1* to ensure that, for each character, *string2<=string1*. A character is said to be less than or greater than another character if its ASCII code is less than or greater than the ASCII code of the corresponding character. Secondly and provided *string2<=string1*, each character of *string1* is compared with the corresponding character of *string3* to ensure that, for each character, *string1<=string3*. If *both* conditions are true, that is *string2<=string1* and *string1<=string3* are both satisfied, the function returns true, otherwise it returns false.

```
chk('b','','c')
; the second string is a null
; returns true because 'b'>'' and 'b'<'c'
chk('B','B','C')
; returns true because 'B'='B' and 'B'<'C'
chk('SD04','AA00','ZZ99')
; returns true, since for each character of the respective
; strings, it is true that 'SD04'>'AA00' and 'SD04'<'ZZ99'
; That is, S>=A, D>=A, 0>=0, 4>=0
; and S<=Z, D<=Z, 0<=9, 4<=9
chk('SDA4','AA00','ZZ99')
; returns false, since in comparing the strings
; 'SDA4' and 'ZZ99', the character 'A'>'9'
chk('SDA4','AA00','ZZ99') + 1 = 0 + 1  ;; returns 1
```

chr()

chr(*number1*[,*number2*]...)

Returns a string by converting ASCII codes to characters. The first character of the resulting string has ASCII value *number1*, second character ASCII value *number2*, and so on. Any argument with a value less than zero or greater than 255 is ignored.

Only normal printable characters should be stored in Character or National fields. Also, since OMNIS uses the character with ASCII value 0 as the end of string marker, this means that if you use this character in any other way, the part of the string following the 0 value is ignored. Control characters in the data file may also cause problems when trying to import or export data. Records with index fields which contain characters with ASCII value 255 may not have the correct index order. It is safe, however, to have unprintable characters in the text for the *Transmit text* commands.

```
chr(66,111,111,107)         ;; returns 'Book'
chr(257,-1,66,111,111,107)  ;; returns 'Book' (first two ignored)
```

cmp()

cmp(*rate*,*periods*)

Returns the compound interest multiplier for a given interest *rate* over a given number of *periods*, that is, *cmp()* evaluates the expression (1+(*rate*/100))*periods*; the interest rate is given by the argument *rate*/100.

The following approximations are correct to 2 decimal places.

```
cmp(10,10) = (1+(10/100))10   ;; returns 2.59 approx
cmp(15,25) = (1+(15/100))25   ;; returns 32.92 approx
cmp(5,0.5) = (1+(5/100))0.5   ;; returns 1.02 approx
```

con()

con(*string1*,*string2*[,*string3*]...)

Returns a string by concatenating or combining two or more *string* values.

```
Calculate FirstName as 'Dick'
Calculate LastName as 'Rawkins'
con(FirstName,' ',LastName)
; returns 'Dick Rawkins'
con('OMNIS',' library')
; returns 'OMNIS library'
```

```
con('July ',5,'th 19',97)
; returns 'July 5th 1997'
; Note the use of spaces in the above examples
```

con() has a limit of 100 parameters. You can exceed this limit by using *Calculate CVAR1 as con(CVAR2,CVAR3)* where CVAR2 has 99 items and CVAR3 has 99 items, and so on.

cos()

cos(*angle*)

Returns the cosine of an *angle* where the *angle* is in degrees (or radians if #RAD is true).

```
cos(60)     ;; returns 0.5
```

createnames()

createnames(*file|field1*[,*file|field2*]...)

Returns the column specification to be used in a SQL Create statement.

The *createnames()* function produces a column specification clause suitable for inclusion in a SQL Create Table statement of the form

```
NAME1 CHAR(10), NAME2 NUMBER(16,2), NAME3 VARCHAR(n),....
```

This specification is based on an OMNIS file class, although you can omit individual fields. For example, this command creates a temporary table on Sybase based on the file class AUTHORS:

```
SQL: Create table TEMP (createnames(AUTHORS))
```

The file class AUTHORS supplies the data definition for the table. The advantage of using *createnames()* over explicit SQL is that the DAM interface does the work of finding suitable server data types for you. With the interface your "Create table" looks the same on Sybase, Oracle, ODBC, and so on, although the actual Create statements generated by OMNIS will vary with the server type.

By default, *createnames()*does not specify whether null values are permitted in each column created and uses the server default. A further complication is that some servers default to "not null", others to "null". You can add the options /N (for "null value permitted") or /NN (for "not null") to *createnames()* following a field name. For example:

```
SQL: createnames(field1/N,field2/NN)
; This line generates the function
; field1 Char Null, field2 int Not Null
```

The following example prompts the user with the SQL create function before sending:

```
; Define class var FCHAR with Character type
Begin SQL script
SQL: createnames(Felements)
End SQL script
Get SQL script {FCHAR}
Yes/No message {Send create statement?: [FCHAR]}
If flag true
   Execute SQL script
Else
   Reset cursor(s) (Current)
End If
```

Field name list

The general format of the field name list is to combine file and field names in a coma separated list:

```
createnames(File1,File2,field1,field3)
```

For all the fields in a file,

```
(filename)
```

You can remove particular fields from the values clause by inserting a minus sign before the field name. For example, to remove the sequence field FSEQ from the clause,

```
(File1,-FSEQ)
```

Field names from a List

If you have a list variable with field names in the first column, you can include these in the values clause using the ^*listname* notation, for example

```
Set current list LIST_NAMES
Define list {FileClass 1}
Build field names list {FileClass 1}
SQL: Create table TABLE createnames(^LIST_NAMES)
```

Qualified Field Names

If the Unique field names option is turned off, you can use the *file|fieldname* /Q notation to force OMNIS to qualify each field with the file name, that is, File1.Fieldname1, File1.Fieldname2, and so on.

```
(File1 /Q)
```

corresponds to the expression

```
(File1.FIELDNAME1,File1.FIELDNAME2,....)VALUES
   (@[File1.FIELDNAME1],@[File1.FIELDNAME2],..)
```

cundif()

cundif(*list*,*class*)

Restores an older version of a *class* using the *list* of differences created by *cdif().*

The cundif() function is used to roll back the changes made to a class after having compared two versions of the same class with *cdif().* The *list* of binary differences must be passed to an older version of the *same* class.

```
; having created DIF_LIST with cdif()...
Calculate OLD_CLASS as cundif(DIF_LIST,NEW_CLASS)
If #ERRCODE
   OK message (Sound bell) {[#ERRTEXT]}
   Quit method
End If
; now assign the binary field containing the
; class to a window class
Calculate LVAR1 as w1.$classdata.$assign(OLD_CLASS)
If LVAR1=0  ;; class in OLD_CLASS is not valid window
   OK message (Sound bell) {The assign has failed...}
   Quit method
End If
```

Note that you can store multiple sets of differences or "revisions" of a class and, at any time, "reconstruct" an earlier version by successively applying *cundif()* against that particular class.

dadd()

dadd(*datepart,number,date*)

Adds a *number* of date parts to a *date.* The *datepart* argument can be a number of days, months, or quarters depending on the constant you use. The *number* is interpreted as the number of date or time parts or units specified by a *datepart* constant. The *number* argument must be an integer when specifying *datepart* as kYear, kMonth, kWeek, kQuarter, or kCentiSecond (the fractional part of a number is ignored). You can use fractions for the other date parts.

The datepart constants that you can use are: kYear, kMonth, kWeek, kQuarter, kDay, kHour, kMinute, kSecond, kCentiSecond.

```
; examples assume #D is June 9, 1998
dadd(kDay,3,#D)
; returns June 12, 1998, that is, 3 days are added
```

```
dadd(kWeek,1.2,#D)      ;; returns June 16, 1998
; that is, one week is added, the fraction is ignored
```

dat()

dat(*datestring|number*[,*dateformat*])

Converts a *datestring* or *number* to a date value using an optional *dateformat* string. If you don't specify a *dateformat,* the first argument is converted using #FD. You can use the following symbols in the *dateformat* string:

Y	Year (89)	d	Day (12th)
y	Year (1989)	W	Day of week (5)
C	Century (19)	w	Day of week (Friday)
M	Month (06)	V	Short day of week (Fri)
m	Month (JUN)	E	Day of year (1–366)
n	Month (June)	G	Week of year (1–52)
D	Day (12)	F	Week of month (1–6)

```
dat('June 7th, 98')  ;; returns '7 JUN 98' if #FD = 'D m Y'
dat('July 8th, 1998','MDY')  ;; returns '070898'
dat(91,'w, d n, y')
; returns 'Monday, 1st April, 1901' i.e. the 91st day of 1901
```

ddiff()

ddiff(*datepart,date1,date2*)

Returns the difference between two dates, *date1* and *date2*, in the units specified by a *datepart* constant; the specified dates are included in the evaluation. When you specify one of the day of the week constants (kSunday thru kSaturday) as the *datepart* argument, you get the number of occurrences of that day between the two dates. When you specify kYear, kQuarter, kMonth, or kWeek as the *datepart* argument, the function counts the end of years, quarters, months, or weeks between the two dates.

The datepart constants that you can use are: kYear, kMonth, kWeek, kQuarter, kDay, kSunday thru kSaturday, kHour, kMinute, kSecond, kCentiSecond.

```
ddiff(kMonth,"1/31/98","3/1/98")    ;; returns 2
ddiff(kDay,"5/9/98",#D)
; returns 31, the number of days between the two dates
; assumes #D is June 9, 1998
```

dim()

dim(*datestring*,*number*)

Increments a *datestring* by a *number* of months. Months containing different numbers of days are accounted for. For example, Jan 31 96 increased by one month gives Feb 29 96, but beware, Feb 29 96 decreased by one month (using a negative value) gives Jan 29 96, not Jan 31 96.

```
dim(dat('4/23/97'),15)  ;; returns 'JUL 23 98' if #FD = 'm D Y'
dim(dat('6/23/98'),-1)  ;; returns 'MAY 23 98' if #FD = 'm D Y'
```

dname()

dname(*datepart*,*date*)

Returns the name of the day or month of a specified *date*, depending on a *datepart* constant which can be either kMonth or kDay.

```
dname(kMonth,#D)   ;; returns June; assumes #D is June 9, 1998
```

dpart()

dpart(*datepart*,*date*)

Returns a number that represents a part of the specified *date* depending on the *datepart* constant used. This is useful when you want to know the week number (that is, the week of the year; use kWeek), the day of the year or the day of the week, and so on.

The datepart constants that you can use are: kYear, kMonth, kWeek, kDayofYear, kQuarter, kMonthofQuarter, kWeekofQuarter, kDayofQuarter, kWeekofMonth, kDay, kDayofWeek, kHour, kMinute, kSecond, kCentiSecond.

When this function returns the week of the year (kWeek) the calculation is based on 1 Jan being the first day of the first week of the year, the last day of the year is week 53.

```
dpart(kWeek,#D)    ;; returns 23 (the week number)
dpart(kMonth,#D)   ;; returns 6
; the above assume #D is June 9, 1998
```

dtcy()

dtcy(*datestring*)

Returns the year and century of a *datestring* as a string.

```
dtcy(#D)            ;; returns '1998'
dtcy('12 06 98')    ;; returns '1998'
```

dtd()

dtd(*datestring*)

Returns the day part of a *datestring* as a string unless it is part of a calculation in which case it is returned as a number.

```
dtd(dat('Jul 21 98'))                ;; returns '21st'
con(dtd(dat('Jul 21 98')),' day')    ;; returns '21st day'
dtd(dat('Jul 21 98')) + 20           ;; returns 41
```

dtm()

dtm(*datestring*)

Returns the month part of a *datestring* as a string unless it is part of a calculation in which case it is returned as a number.

```
dtm(dat('Apr 16 1998'))              ;; returns 'April'
dtm(dat('Jul 21 98')) + 20           ;; returns 27
dtm(dat(dat('Jul 21 98') + 20))      ;; returns 'August'
```

dtw()

dtw(*datestring*)

Returns the day of the week of a *datestring* as a string unless it is part of a calculation in which case it is returned as a number.

```
dtw(dat('Jun 9 1951'))               ;; returns 'Saturday'
dtw(dat('Jul 21 98')) + 20           ;; returns 22
dtw(dat(dat('Jul 21 98') + 20))      ;; returns 'Monday'
```

dty()

dty(*datestring*)

Returns the year part of a *datestring* as a string unless it is part of a calculation in which case it is returned as a number. The string representation of the year part of a date is the set of numeric characters representing the year, that is, 00, 01, 02, 03, and so on, while the numeric representation is the number of years since the start of the century.

```
dty(dat('16 Apr 98'))            ;; returns '98'
dty(dat('Jul 12 98')) + 20       ;; returns 118
```

eval()

eval(*fieldname|variable*)

Evaluates a calculation held as a string in *fieldname* or *variable*. For example, if CVAR1 contains the string '3*4/2', *eval(CVAR1)* returns the result 6. You should use this function with extreme care because a runtime error will occur if the string is not a valid calculation. You can use the command *Test for valid calculation* to test a string before attempting to evaluate it.

```
Calculate CVAR1 as '3*LVAR1/15.5'
Test for valid calculation {eval(CVAR1)}
If flag true
   Calculate TAX as eval(CVAR1)
End If
```

evalf()

evalf(*fieldname|variable*)

Evaluates a calculation held as a string but stores the calculation in tokenized form back in *fieldname* or *variable*. The function *evalf()* is faster than the equivalent *eval()*; you should use it when your code will repeat an evaluation several times. You should use it in the *Set search as calculation* command and as the calculation for a window list field.

evalf() takes a single argument that must be a *fieldname* or *variable*. If the contents of this field or variable is the text for a valid calculation, *evalf()* returns the result of the calculation, else a runtime error occurs. At the same time, the tokenized form of the calculation is stored back in the field or variable, so that next time *evalf()* is called, there is no need to tokenize or check the string. Tokenizing a string is part of the interpretation process; once done, OMNIS can evaluate the calculation quickly. If you change the contents of the field or variable *evalf()* uses, OMNIS will recognize that the new string requires checking and tokenizing.

```
Test for valid calculation {evalf(SEARCH)}
If flag true
   Set search as calculation {evalf(SEARCH)}
End If
Find first on TOWN (Use search)

Calculate CVAR1 as 'TAX*100'
Calculate TOTAL as VALUE + evalf(CVAR1)
```

exp()

exp(*number*)

Returns e raised to the power of a given *number*, or 1e100 on overflow.

```
exp(0.5)      ;; returns 1.6487
```

fact()

fact(*number*)

Returns the factorial of a *number* rounded to an integer. If *number*<=0, 1 is returned, and if *number*>=70, 1e100 is returned.

```
fact(4)      ;; returns 24, that is 4*3*2*1
```

fday()

fday(*datepart,date*)

Returns the date of the first day of the year, month, week, or quarter in which the specified *date* falls.

The period is specified using one of the following *datepart* constants: kYear, kQuarter, kMonth, kWeek.

```
fday(kWeek,#D)
; returns June 8,  1998 if the start of the week is set to kMonday
fday(kQuarter,#D)
; returns April 1  1998, that is, the first day of the
; quarter in which today falls, #D is June 9, 1998
```

fld()

fld(*string1*[,*string2*]...)

Returns the value of the field name given by concatenating or combining one or more *string* values. For example, if the current values of the fields RATE1 and RATE2 are 10 and 15 respectively, then

```
fld('RATE','1')     ;; returns 10
fld('RATE','2')     ;; returns 15
```

fontlist()

fontlist(*listname*)

Returns a list of fonts currently installed in your system, including the font name and type. The *listname* parameter is any list field or variable. A return value of 0 indicates no fonts

found or some error, otherwise 1 is returned indicating a list was built. You should define the following columns in your list field or variable.

Column 1	Column 2 (Optional)	Column 3 (Optional)
String variable to hold name of the font	Numeric variable to hold value (0..7) for type of font	String variable to hold a name for the value in column 2. This will be combination of "Raster", "Vector", "TrueType" together with "Fixed" or "Proportional"

```
Set current list LIST1
Define list (FONT_NAME,FONT_TYPE,FONT_DESCRIPTION)
If fontlist(LIST1) <> 0
   Redraw lists
End If
```

getfye()

getfye()

Returns the current date of the fiscal year end (note no argument).

getseed()

getseed()

Returns the current content of the seed as an integer number (note no argument).

getws()

getws()

Returns the day of the week which is set as the beginning of the week (note no argument).

The day of the week is returned as one of the datepart constants: kSunday, kMonday, kTuesday, kWednesday, kThursday, kFriday, kSaturday.

insertnames()

insertnames(*file|field1*[,*file|field2*]...)

Returns a list of fields and values to be used in a SQL Insert statement.

The *insertnames()* function produces a field name list and a values list suitable for inclusion in a SQL Insert statement of the form

```
(NAME1,NAME2,.....) VALUES (@[NAME1],@[NAME2],....)
```

When inserting a complete row for which you have a corresponding set of OMNIS fields, use

```
SQL: Insert into FTEL insertnames(FTEL)
```

which OMNIS expands to the expression

```
Insert into FTEL (FTNAME, FTNUM) VALUES (@[FTNAME], @[FTNUM])
```

There are cases where you don't want to insert a value (or the default NULL) into certain columns. You can eliminate some columns from the insert statement like this:

```
SQL: Insert into FTEL insertnames(FTEL,-FTNUM)
Begin SQL script
SQL: Insert into CLIENTS
SQL: insertnames(CLIENTS,-C_TOWN,-C_CITY,-C_TEL)
```

Field name list

The general format of the field name list is to combine file and field names in a coma separated list:

```
createnames(File1,File2,field1,field3)
```

For all the fields in a file,

```
(filename)
```

You can remove particular fields from the values clause by inserting a minus sign before the field name. For example, to remove the sequence field FSEQ from the clause,

```
(File1,-FSEQ)
```

Field names from a List

If you have a list variable with field names in the first column, you can include these in the values clause using the ^*listname* notation, for example

```
Set current list LIST_NAMES
Define list {FileClass 1}
Build field names list {FileClass 1}
SQL: Create table TABLE createnames(^LIST_NAMES)
```

Qualified Field Names

If the Unique field names option is turned off, you can use the *file|fieldname* /Q notation to force OMNIS to qualify each field with the file name, that is, File1.Fieldname1, File1.Fieldname2, and so on.

```
(File1 /Q)
```

corresponds to the expression

```
(File1.FIELDNAME1,File1.FIELDNAME2,....)VALUES
   (@[File1.FIELDNAME1],@[File1.FIELDNAME2],..)
```

int()

int(*number*)

Returns the integer part of a *number*; it does not round to the nearest integer.

```
int(23.1056)        ;; returns 23
int('-2.66')        ;; returns -2
abs(int(-1.999))    ;; returns 1
```

isfontinstalled()

isfontinstalled(*fontname*)

Returns a true or false value indicating whether the named font has been fully installed into your system. The *fontname* argument can be a string literal, character field or variable with a maximum length of 255.

```
If not(isfontinstalled('O7Font'))
   Ok message { Cannot run library without 'O7Font' }
End If
```

isnull()

isnull(*fieldname*)

Returns true if *fieldname*, a field in the current record, is null. A null value is one where no value has been entered and the field definition is **Can be null** without **Insert as Empty.**

isnumber()

isnumber(*string[,decimal_char][,thousands_char]*)

Returns kTrue if the specified *string* can be evaluated as a number; kFalse otherwise. The optional parameters can be used to define the decimal and thousand separator If the optional parameters are not specfied the default separators are used, a '.' for the decimal and a ',' for the thousand.

```
Calculate STATUS as isnumber(STRING)
; STATUS is kTrue if STRING can be evaluated as a number
```

isoweek()

isoweek(*date*)

Returns the ISO 8601 standard week number for the week containing the specified *date*.

```
isoweek(#D)  ;; returns the current iso week number
```

jst()

jst(*string1*,*number1*[,*string2*,*number2*]...)

Returns a string containing the specified *string* left or right justified with sufficient spaces added to make a total length specified by *number*. The *jst()* function also includes concatenation. If *number* is negative the resulting *string* is right justified, if the *number* is positive the *string* is left justified; *number* must be in the range 1 to 999.

```
jst('This is left justified',30)
; gives
|This is left justified        |

jst('This is right justified',-30)
; gives
|       This is right justified|
```

When you define the columns for a list, *jst()* lets you fix the column width and using a non-proportional font the list columns will line up properly. For example, the calculation for a list containing the fields CODE and COMPANY could be

```
jst(CODE,7,COMPANY,25)
```

The *jst()* function also includes concatenation, for example

```
jst(p1,p2,p3,p4,p5,p6,...)
; is the same as
con(jst(p1,p2),jst(p3,p4),jst(p5,p6),...)
```

However it has a limit of 100 parameters, but you can exceed this limit by using *Calculate CVAR1 as jst(CVAR2,CVAR3)* where CVAR2 has 99 items and CVAR3 has 99 items, and so on.

Formatting Strings Using jst()

The *jst()* function can take a string for the second argument instead of a number, that is

jst(*string1*,*string1*[,*string2*,*string2*]...)

This form of *jst()* formats the first *string1* argument in a way controlled by the second *string2* argument. The second argument consists of a series of formatting options which you can use separately or together. Each option is represented by one or more characters. The order of the various formatting options is not important but the case is.

^n   (caret) causes the data to be centered in the field n characters wide.

```
jst('abc','^5')        ;; returns ' abc '
jst(FIELD,'^25')
; as a list calculation will center the FIELD values in
; a column 25 characters wide
```

£   places a £ sign in front of the data.

```
jst(TOTAL,'£')        ;; returns '£12.12' if TOTAL = 12.12
```

$ places a $ sign in front of the data.

```
jst(TOTAL,'$')        ;; returns ' $12.12' if TOTAL = 12.12
```

<   left justification, overriding the default set up in a report class. This is used by the Ad hoc report generator to control the justification of fields.

Pc   causes the part of the field not filled by the data to be filled by character c.

```
jst('abc','-5P*')     ;; returns '**abc'
```

X   causes the data to be truncated if its length exceeds the field length. The default is not to truncate.

```
jst('abcdef','4')     ;; returns 'abcdef'
jst('abcdef','4X')    ;; returns 'abcd'
```

U   causes the data to be converted to upper case.

```
jst('this IS it','U')     ;; returns 'THIS IS IT'
```

L   causes the data to be converted to lower case.

```
jst('THIS is IT','L')     ;; returns 'this is it'
```

C   causes the data to be capitalized.

```
jst('this is it','C')     ;; returns 'This Is It'
jst('THIS IS IT','C')     ;; returns 'This Is It'
jst('this IS IT','C')     ;; returns 'This Is It'
```

Nnn   causes the data to be treated as a fixed decimal number with nn decimal places. If nn is not specified, a suitable number of decimal places is used.

```
jst(0.235,'N')        ;; returns '0.235'
jst(0.235,'N2')       ;; returns '0.24'
```

Fnn   causes the data to be treated as a floating decimal number with format specified by nn. The nn argument can be a positive or negative number and has the same meaning as described for the #FDP variable. If nn is not specified, it defaults to the current value of #FDP.

```
jst(12.35,'-10F9')       ;; returns '      12.35'
jst(12.35,'-10F-9')      ;; returns '1.23500000e+01'
jst(12.35,'-10F-3U')     ;; returns '  1.24E+01'
```

D   causes the data to be treated as a date. The default formatting string is #FD, but you can specify a formatting string as described later.

```
jst('26/11/97','DC')
; returns '26 Nov 97' if #FD = 'D m Y'
```

DT   causes the data to be treated as a long date and time. The default formatting string is *#FDT* but you can specify a formatting string using the : (colon) argument as described below.

```
jst(#D,'DT')
; returns '26 Nov 97 15:30' if #FDT is 'D m Y H:N'
```

T   causes the data to be treated as a time using the formatting string *#FT*. You can include a format string using the : (colon) argument as described below.

```
jst('0620','T')      ;; returns '06:20' if #FT = 'H:N'
```

A   displays a null value as 'NULL'.

```
jst(Field1,'A')     ;; returns 'NULL' when Field1 is null
```

B   causes the data to be treated as Boolean.

```
jst(1,'LB')      ;; returns 'yes'
```

E   applies to numbers only and turns on the 'Zero shown empty' attribute.

```
jst(0,'N2')      ;; returns '0.00'
jst(0,'N2E')     ;; returns ''
```

,   (comma) applies to numbers only and turns on the 'Shown like 1,234' attribute.

```
jst(1234,'N2')       ;; returns '1234.00'
jst(1234,'N2,')      ;; returns '1,234.00'
```

((open bracket) applies to numbers only and turns on the 'Shown like (1234)' attribute.

```
jst(-1234,'N2')      ;; returns '-1234.00'
jst(-1234,'N2(')     ;; returns '(1234.00)'
jst(1234,'N2(')      ;; returns '1234.00'
```

)   (close bracket) applies to numbers only and turns on the 'Shown like 1234-' attribute.

```
jst(-1234,'N2)'      ;; returns '1234.00-'
jst(1234,'N2)')      ;; returns '1234.00 '
```

+   (plus sign) applies to numbers only and causes positive numbers to be shown with a "+" sign.

```
jst(1234,'N2+')      ;; returns '+1234.00'
jst(1234,'N2)+')     ;; returns '1234.00+'
```

:   (colon) characters following a colon are interpreted as a formatting string. This must be the last option since all characters following it become part of the formatting string. The meaning of the formatting string depends on the type of the data.

The formatting string has a similar format to #FDT if the data is a date/time, using the following characters:

Y	Year (99)	H	Hour (0..23)
y	Year (1999)	h	Hour (1..12)
C	Century (19)	N	Minutes
M	Month (06)	S	Seconds
m	Month (JUN)	s	Hundredths
n	Month (June)	A	AM/PM
D	Day (12)	V	Short day of week (Fri)
d	Day (12th)	E	Day of year (1–366)
W	Day of week (5)	G	Week of year (1–52)
w	Day of week (Friday)	F	Week of month (1–6)

For example:

```
jst(#D,'D:w, d n CY')
; returns 'Saturday, 29th November 1997'
cap(jst(#D,'D:V d n Y'))
; returns 'Sat 29th Nov 97'
```

The formatting string has a similar format to #FT if the data is a time. #FT is used as the formatting string if a formatting string is not specified for a time.

```
jst(#T,'T:H-N')  ;; returns '14-07'
```

If the data is neither a date nor a time, and if the formatting string contains an X, the data value is inserted at the position of the X to produce the data value.

```
jst(0,'BC:The answer is X!')
; returns 'The answer is No!'
```

The formatting string is concatenated to the left of the data value if the formatting string does *not* contain an X. The data value is left unchanged if a formatting string is not specified.

```
jst(12,'-7N2:$')     ;; returns ' $12.00'
jst(12,'-7N2:£')     ;; returns ' £12.00'
jst(12,'-8N2:DM')    ;; returns ' DM12.00'
```

lday()

lday(*datepart,date*)

Returns the date of the last day of the year, month, week, or quarter in which the specified *date* falls.

The period is specified using one of the following *datepart* constants: kYear, kQuarter, kMonth, kWeek.

```
lday(kWeek,#D)
; returns June 14, 1998 if the start of the week is set to kMonday
lday(kMonth,#D)
; returns June 30 1998, that is the last day of the month
; the above assume #D is June 9, 1998
```

len()

len(*string*)

Returns the length of a *string*, that is, number of characters.

```
len('Hello there!')  ;; returns 12
len(abs(-10.25))     ;; is the same as len(10.25) which returns 5
len('OMNIS') + 20    ;; returns 25
```

list()

list(*row1*[,*row2*]...)

Returns a list from a number of *row* variables of identical structure, that is, the data type of each column in each row variable should match. One row in the list is created for each row variable passed. For example

```
Set current list myList
Define list {col1, col2, col3}
Calculate myList as list(row1, row2, row3)
; returns a list from row variables row1, row2, and row3
```

If the type of a particular column in the list does not match the type of a row variable column, *list()* tries to convert the row variable column to that of the list column, from number to string, for example. If the column type cannot be converted the column is left blank.

ln()

ln(*number*)

Returns the log to base e (the natural logarithm) of a *number*; or -1e100 if *number*<=0.

```
ln(exp(.5))  ;; returns 0.5
```

log()

log(*number*)

Returns the log to base 10 of a *number*; or -1e100 if *number*<=0.

```
log(100)     ;; returns 2
log(0.001)   ;; returns -3
```

lookup()

lookup([*refname*,]*searchvalue*[,*fieldnumber*])

Returns a field value from another data file *refname* (opened as a lookup file) using a *searchvalue*. The *fieldnumber* argument specifies the particular field in the lookup file to be returned by the function. Each lookup file is opened using the *Open lookup file* command at which time a reference label is assigned to that lookup.

If you omit the third argument, the value of the second field is returned by default. If one argument is given, the default lookup file is used, that is, the lookup file which was opened without a reference label, or the first lookup file opened if all have labels.

If no exact match is found, an empty value is returned. All field types are returned, including pictures and long text.

```
Open lookup file {City/Cities.df1/FCITY/1}
; Reference name is City, file class FCITY
; Uses first field in FCITY as the index
OK message {CNAME is [lookup('City','FOS',2)]}
```

low()

low(*string*)

Returns the lower case representation of a *string*. Any non-alphabetic characters in the strings are unaffected by *low().*

```
low('DAVID')              ;; returns 'david'
low('OrAcLe7')            ;; returns 'oracle7'
low(1017)                 ;; returns '1017'
mid(low(PERIPHERAL),3,3)  ;; returns 'rip'
```

lst()

lst([[*listname*,]*linenumber*,]*fieldname*)

Returns the value of *fieldname* from a line specified by *linenumber* in a list specified by *listname*. The *fieldname* argument must be a field stored in the list, not a constant or expression. If *listname* is not specified the current list is used. If only the *fieldname* argument is specified the current line of the current list is used.

```
lst(L3,23,PRICE)
; returns PRICE field value stored at line 23 of list L3
lst(23,PRICE)
; returns PRICE field value stored in line 23 of the current list

lst(PRICE)
; returns PRICE field value stored at current line of current list
lst(L3,0,LM)
; returns the maximum number of lines in list L3
```

max()

max(*value1*[,*value2*]...)

Returns the maximum value from a list of values. The values should all be numbers when numeric comparison is used or all strings when string comparison is used.

```
max(3,6,2,7)              ;; returns 7
max('dagger','dog','dig')    ;; returns 'dog'
```

maxc()

maxc(*listname,column*[,*ignore_nulls*])

Returns the maximum value for a list column specified by *listname* and *column.* If you set *ignore_nulls* to 1, null values are ignored and not counted. If you omit this parameter or it evaluates to zero, nulls are treated as zero values and are counted.

mid()

mid(*string,position,length*)

Returns a substring of a specified *length*, starting at a specified *position*, from a larger *string*. If *position* is less than 1 it is taken as 1, that is the first character; if it is greater than the length of the *string*, an empty string is returned. If *length* is greater than the maximum length of any substring of *string* starting at *position*, the returned substring will be the remainder of *string* starting at *position*.

```
mid('Information',6,3)    ;; returns 'mat'
int(mid(12.45,2,3))       ;; is the same as int('2.4'), returns 2
mid('interaction',6,24)   ;; returns 'action'
```

min()

min(*value1*[,*value2*]...)

Returns the minimum value from a list of values. The values should all be numbers when numeric comparison is used or all strings when string comparison is used.

```
min(3,6,2,7)              ;; returns 2
min('cat','dog','apple')  ;; returns 'apple'
```

minc()

minc(*listname,column*[,*ignore_nulls*])

Returns the minimum value for a list column specified by *listname* and *column.* If you set *ignore_nulls* to 1, null values are ignored and not counted. If you omit this parameter or it evaluates to zero, nulls are treated as zero values and are counted.

```
Calculate LVAR4 as minc(LIST1,Salary,1)
```

mod()

mod(*number1*,*number2*)

Returns the remainder of a number division, that is, when *number1* is divided by *number2* to produce a remainder; it is a true modulus function.

```
mod(6,4)     ;; returns 2
mod(4,6)     ;; returns 4
mod(-5,-2)   ;; returns -1
```

mousedn()

mousedn()

Returns true if the mouse button is held down, otherwise returns false (note no argument). This function returns a Boolean value describing the state of mouse button (the left-hand mouse button under Windows).

mouseover()

mouseover(*constant*)

Returns information about the mouse position defined by a predefined *constant* at the instant the function is evaluated. The function only works in an "open" user-defined window (not reports or searches). Moreover, it returns references only for fields and not background objects (text and graphic objects).

The mouse position is returned in a variety of ways depending on the constant you use. You can use the following constants:

kMItemref   returns a reference to the object under the mouse. The window instance itself can be returned as item 0 of the window.

kMCharpos   returns the nth character in an edit field.

kMLine   returns the line number for a list.

kMHorz   returns the horizontal position of the mouse relative to the topmost open user-defined window; if no user-defined window is open, returns the horizontal position of the mouse relative to the OMNIS application window

kMVert   returns the vertical position of the mouse relative to the topmost open user-defined window; if no user-defined window is open, returns the relative position to the OMNIS application window

mouseup()

mouseup()

Returns true if the mouse button is released after having been pressed, otherwise returns false (note no argument).

msgcancelled()

msgcancelled()

Returns true if the Cancel button is pressed on a message box. For example, you can use this to distinguish between No and Cancel on a *Yes/No message* which both clear the flag.

```
Yes/No message (Cancel button) {Do you want to proceed?}
If flag false
   If not(msgcancelled())
      ; user chose No
   End If
Else
   ; user chose Yes
End If
```

nam()

nam(*fieldname*)

Returns the name of a field as a string; *fieldname* must be a field name or variable, not a constant or an expression.

```
nam(CCODE)   ;; returns the string 'CCODE'
nam(#SUBFLD) ;; returns the name of the subtotal field
```

natcmp()

natcmp(*value1,value2*)

Returns the result of comparing two values using the national sort ordering. Returns 0 if the strings are equal, 1 if *value1 > value2*, and -1 if *value1 < value2*. Both values are converted to strings before the comparison is made. *natcmp()* uses the same rules for comparing the strings as it does for normal strings, except that it uses the national sort ordering.

```
natcmp(value1,value2)       ;; returns 0 if values are equal
```

nday()

nday(*datepart,date*)

Returns the date of the day *after* the specified *date* when the *datepart* constant is set to kDay. However, if one of the day of the week constants is used, this function returns the date of that day of the week following the specified *date*.

The datepart constants that you can use are: kDay, kSunday, kMonday, kTuesday, kWednesday, kThursday, kFriday, kSaturday.

```
nday(kDay,#D)          ;; returns June 10, 1998
nday(kMonday,#D)
; returns June 15, 1998 which is the next Monday after #D
; the above assume #D is June 9, 1998 which is a Tuesday
```

not()

not(*expression*)

Returns the Logical Not of an *expression*.

All expressions in OMNIS have a Boolean (truth) value. Firstly, non-zero numeric values (including negative values) are TRUE, zero values are FALSE. Secondly, string values are TRUE or FALSE depending on their numeric equivalent. String '1' has boolean value 1, therefore *not('1')* is 0. 'Bill' has boolean value 0, 'YES' has numeric value 0.

```
not(2501)              ;; returns 0
not('Hello there!')    ;; is the same as not(0) which returns 1
```

The numeric value of an expression that evaluates to true is 1, therefore *not(true)* is 0. Similarly, *not(false)* is 1.

```
not(31<45)          ;; is the same as not(true) which returns false
```

You can use *not()* to make method code more readable.

```
Do method ProcessList Returns Done
If not(Done)
   ..
```

oemchar()

oemchar(*code*)

Returns a string containing the PC symbol for the specified *code* (available under Windows only). You can display the result with the Windows Terminal font, since these characters are intended for DOS use only.

oemcode()

oemcode(*string,index*)

Returns the current PC character code for the specified *string* (available under Windows only). The result will depend on the nationality and code-page used, as installed by Windows' setup.

omnischar()

omnischar(*code*)

Returns a character for the specified *code* as defined by the OMNIS character set (available under Windows only). These characters are useful to display under MacOS, but most you can display with a full TrueType font. Some characters you can display only with an Accuware font, and some will have no representation at all. You can find the *code* for a character using *omniscode()*.

omniscode()

omniscode(*string,index*)

Returns the OMNIS code or character value/number (available under MacOS) for the specified character within a *string* (the function is available under Windows only).

```
Calculate LVAR1 as omniscode('ABC',1)
; returns the OMNIS code for the first character
```

pday()

pday(*datepart,date*)

Returns the date of the day *before* the specified *date* when the *datepart* constant is set to kDay. However, if one of the day of the week constants is used, this function returns the date of that day of the week preceding the specified *date*.

The datepart constants that you can use are: kDay, kSunday, kMonday, kTuesday, kWednesday, kThursday, kFriday, kSaturday.

```
pday(kDay,#D)            ;; returns June 8, 1998
pday(kThursday,#D)
; returns June 4, 1998 which is the Thursday before #D
; the above assume #D is June 9, 1998 a Tuesday
```

pick()

pick(*number,value0,value1*[*,value2*]...)

Selects an item from a list of *values* (strings or numbers) depending on the value or result of the *number* argument.

The *number* argument is rounded to an integer and used to select the item. *value0* is returned if the result is 0, *value1* if the result is 1, *value2* if the result is 2, and so on. If the *number* is less than zero or greater than the number of values in the list, an empty value is returned. Note the list of values can be a mixture of string and numeric values.

```
pick(LVAR2,123,'ABC','abc')    ;; returns 123 if LVAR2 evaluates to 0
pick(LVAR1,2200,4500,6800)     ;; returns 6800 if LVAR1 = 2
pick(LVAR30,'one',2,'three')   ;; returns null if LVAR30 = 5
```

pos()

pos(*substring,string*)

Returns the position of a *substring* within a larger *string*. The *substring* must be contained within *string* in its entirety for the returned value to be non-zero. Also, the comparison is case sensitive and only the first occurrence of substring is returned (see third example).

```
pos('Mouse','Mickey Mouse')      ;; returns 8
pos('mouse','Mickey Mouse')      ;; returns 0, note case
pos(' ','R S W Smith')
; returns 2, that is the position of the first space character
```

For example, you can strip the extension from a file name using *mid()* and *pos()* combined, as follows.

```
If pos('.',FileName)    ;; if FileName contains a dot
   Calculate FileName as mid(FileName,1,pos('.',IntName)-1)
End If
```

pwr()

pwr(*number,power*) raises a *number* to a *power*.

```
pwr(2,5)              ;; returns 32
pwr(21.37,0.831)      ;; returns 12.74 approx
int(pwr(1.105,5))     ;; is the same as int(1.65) which returns 1
```

rand()

rand()

Returns a real number in the range 0.0 < number > 1.0, inclusive (note no argument is required).

randintrng()

randintrng(*number1*,*number2*)

Returns an integer between *number1* and *number2,* inclusive.

randrealrng()

randrealrng(*number1*,*number2*)

Returns a real number between *number1* and *number2,* inclusive.

replace()

replace(*source-string, target-string, replacement-string*)

Replaces the first occurrence of the *target-string,* within the *source-string*, with the *replacement-string*. The *replace()* function returns the new string containing the changes, if any. If you replace part of a window field's contents, you should redraw it.

```
Calculate NEW_STRING as replace(STRING,chr(65),'a')
; replaces the first occurrence of upper case A in STRING
; with lower case a, and places the result in NEW_STRING
```

replaceall()

replaceall(*source-string, target-string, replacement-string*)

Replaces all occurrences of the *target-string,* within the *source-string*, with the *replacement-string*. The *replaceall()* function returns the new string containing the changes, if any. If you replace part of a window field's contents, you should redraw it.

```
Calculate NEW_STRING as replaceall(STRING,'_','-')
; replaces all occurrences of underscores in STRING with hyphens
; and places the result in NEW_STRING
```

rgb()

rgb(*red,green,blue*)

Sets the color of an object. The three arguments *red, green, blue* correspond to the RGB value of the desired color; each must be an integer in the range 0–255. For example, yellow has an RGB value of 255,255,0.

```
Do $cinst.$objs.FIELD1.$forecolor.$assign(rgb(0,178,178))
; changes the object forecolor to green
```

rmousedn()

rmousedn()

Returns true if the right mouse button is held down (under Windows), or the Ctrl key is held down while the mouse is clicked (under MacOS). If *rmousedn()* is true, *mousedn()* is also true but not vice versa. (Note no argument is required.)

rmouseup()

rmouseup()

Returns true if, after being pressed, the right mouse button is up (in Windows) or the mouse is up after it has been pressed with the Ctrl key held down (under MacOS). (Note no argument is required.)

rnd()

rnd(*number,dp*)

Rounds a *number* to a number of decimal places specified in *dp*.

```
rnd(2.105693,5)    ;; returns 2.10569
rnd(2.105693,3)    ;; returns 2.106
rnd(0.5,0)         ;; returns 1
```

It is often essential to use *rnd()* when comparingany two variables with field values to round the values to the same precision. For example

```
If rnd(LVAR1,2) = NUMBERFIELD  ;; if NUMBERFIELD is a Number 2 dp
   ; do what's expected
Else...
```

rolldice()

rolldice(*number*,*faces*)

Returns the result of a die roll. You specify the *number* of dice to roll and the number of *faces* on each die.

```
Calculate DICEROLL as rolldice(2,6)
; rolls two normal, six-sided dice and puts the result in DICEROLL
```

rollstring()

rollstring(*stringformula*)

Returns the result of a die roll from a *string* formula. The format of the *stringformula* is:

```
rollstring NdF [ + - * / offset ]
```

where N is the number of dice, d is a delimiter, and F is the number of faces for each die. In addition, you can add, subtract, multiply, or divide by an offset. For example

```
Calculate MY_NUM as rollstring('2d6')
; returns the result of rolling two standard six-faced dice
Calculate MY_NUM as rollstring('3d6+1')
Calculate MY_NUM as rollstring('12d4+6')
```

row()

row(*variable1*[,*variable2*]...)

Creates a row variable from a number of *variables*; it creates one column for each variable passed.

```
Calculate myRowVar as row(var1, var2, var3)
; creates a row variable with the columns var1, var2, and var3
```

selectnames()

selectnames(*file|field1*[,*file|field2*]...)

Returns a list of field names to be used in a SQL Select statement.

The *selectnames()* function produces a comma-separated list of field names suitable for inclusion in a SQL Select statement and elsewhere of the form

```
NAME1,NAME2,.....
```

The following examples use the file class FTEL which contains two fields FTNAME and FTNUM. The file FTEL is also a table on the server:

```
Perform SQL {Select selectnames(FTEL) from FTEL}
; sends the statement: Select FTNAME, FTNUM from FTEL
; and
Perform SQL {Select selectnames(FTEL,-FTNUM) from FTEL}
; sends the statement: Select FTNAME from FTEL
```

Field name list

The general format of the field name list is to combine file and field names in a coma separated list:

```
createnames(File1,File2,field1,field3)
```

For all the fields in a file,

```
(filename)
```

You can remove particular fields from the values clause by inserting a minus sign before the field name. For example, to remove the sequence field FSEQ from the clause,

```
(File1,-FSEQ)
```

Field names from a List

If you have a list variable with field names in the first column, you can include these in the values clause using the *^listname* notation, for example

```
Set current list LIST_NAMES
Define list {FileClass 1}
Build field names list {FileClass 1}
SQL: Create table TABLE createnames(^LIST_NAMES)
```

Qualified Field Names

If the Unique field names option is turned off, you can use the *file|fieldname* /Q notation to force OMNIS to qualify each field with the file name, that is, File1.Fieldname1, File1.Fieldname2, and so on.

```
(File1 /Q)
```

corresponds to the expression

```
(File1.FIELDNAME1,File1.FIELDNAME2,....)VALUES
   (@[File1.FIELDNAME1],@[File1.FIELDNAME2],..)
```

server()

server(*function*)

Sends a server *function* direct to the DAM in the current session.

The *server()* function takes an argument in which you specify a *function* to be carried out within the SQL interface. The result can be returned to OMNIS by including the function call in a *Calculate* command, for example

```
Calculate RESULT as server('Version')
; RESULT contains the version number of the active DAM
Calculate PATH as server('Path')
; PATH contains the directory path of the DAM
Calculate API as server('vendorAPI')
; API contains the directory path of server API if
; available, otherwise returns -1 if not available
Calculate DAM as server('DAM')
; DAM contains the name of the current DAM
Calculate FILE as server('File')
; FILE contains the file name of the current DAM
```

Other functions are specific to the server and are documented with the installation notes for the particular interface.

setfye()

setfye(*date*)

Sets the *date* of the fiscal year end. The *date* does not have to be in the current year, that is, the function ignores the year part of the date. The setting of the fiscal year end affects all other date functions that involve quarters. It returns the previous value so you can save it for later use.

```
setfye('MAR 31')      ;; sets the fye to March 31st
setfye('12 31 98')
; sets the fye to December 31st & ignores the year
```

setseed()

setseed(*seed*)

Sets the random number *seed* for the random functions, such as *rand()*. *setseed()* converts *seed* into a Long number. It returns the previous seed as an integer number.

setws()

setws(*datepart*)

Sets the beginning of the week to a particular day, using one of the day of the week *datepart* constants. It returns the previous value so you can save it for later use. If the *datepart* is invalid this function still returns the week start but does not change it.

The datepart constants that you can use are: kSunday, kMonday, kTuesday, kWednesday, kThursday, kFriday, kSaturday.

```
setws(kMonday)    ;; sets Monday as the week start
```

shufflelist()

shufflelist(*sourcelist, targetlist, number*)

Shuffles the items in *sourcelist,* the specified *number* of times, and puts the results in *targetlist*. A value of 2 or 3 for *number* provides a good shuffle. *shufflelist()* does not support Binary fields, List fields, and Picture fields stored in a list. An empty or null date converts to '31 DEC 00' in the *targetlist*.

sin()

sin(*angle*)

Returns the Sine of an *angle* where the *angle* is in degrees (or radians if #RAD is true).

```
sin(30)      ;; returns 0.5
```

sqr()

sqr(*number*)

Returns the square root of a *number*. OMNIS defines the square root of a negative number *X* as *sqr(abs(X))*.

```
sqr(100)                 ;; returns 10
sqr('-301.56')           ;; returns 17.37 approx
mid('OMNIS',sqr(16),2)   ;; returns 'is'
```

stddevc()

stddevc(*listname,column*[,*ignore_nulls*])

Returns the standard deviation for a list column specified by *listname* and *column*. If you set *ignore_nulls* to 1, null values are ignored and not counted. If you omit this parameter or it evaluates to zero, nulls are treated as zero values and are counted.

```
Calculate %RESULT as jst(stddevc(LIST,COL),'N2')
; returns the standard deviation rounded to 2 decimal places
```

strpbrk()

strpbrk(*string1, string2*)

Returns a substring of *string1* from the point where any of the characters in *string2* match *string1*.

```
Calculate CVAR1 as "this is a test"
Calculate CVAR2 as strpbrk(CVAR1, " absj")
OK Message { Result = [CVAR2] }
; displays the message "s is a test"!
```

strspn()

strspn(*string1, string2*)

Returns the index of the first character in *string1* that *does not match any* of the characters in the *string2*.

```
Calculate LENGTH as strspn(STRING, CONTAINEDCHARS)
; If STRING does not contain any of the characters in
; CONTAINEDCHARS, zero is returned in LENGTH
```

strtok()

strtok('*string1', string2*)

Tokenizes *string1*, using *string2* as the delimiter with which to tokenize. This function returns tokens which are a substring of *string1* until any character in *string2* matches a character in *string1*. When *strtok()* is called, the token found in *string1* is removed, so that the function looks for the next token the next time it is called.

```
Calculate CVAR1 as "The quick brown fox, jumped over the lazy dog"
Repeat
   Calculate CVAR2 as strtok( 'CVAR1', ",  " )
   OK Message { Token = [CVAR2] }
Until CVAR2 = ''
; returns each word in CVAR1 in an OK Message
```

style()

style(*style-character*[,*value*])

Inserts a *style-character* represented by an OMNIS constant into a calculation. Depending on the style character, you can also specify a *value*, which itself can be a constant. You can use this function to format the columns in a headed list box field. You can insert an icon by specifying its ID, a center tab, right tab, left tab, a color value, or text property such as italic. For example, to format the columns in a headed list box you could use the following calculation

```
con(Col1,style(kEscBmp,1756),chr(9),
    Col2,style(kEscColor,kRed),style(kEscRTab),chr(9),
        Col3,style(kEscStyle,kItalic))
; gives Col1 a blue spot icon, Col2 is red and
; right-justified, and Col3 italic
```

sys()

sys(*number*)

Returns information about the current system depending on a *number* argument. Using the *sys()* function, you can obtain system information such as the current printer name, the pathname of the current library, the screen width or height in pixels, and so on. The following example uses *sys(6)* to test the current OS and branches accordingly.

```
; declare lvListHD of List type, and lvFolder of Char type
Do lvListHD.$define(lvFolder)
If sys(6) = 'M'    ;; on MacOS
  Get folders (lvListHD,lvFolder,'Mac HD')
Else    ;; on other platforms
  Get folders (lvListHD,lvFolder,'C:\')
End If
Do lvListHD.$sort(lvFolder)
Redraw lists       ;; if it's a window list
```

You can use the following *number* values with the sys() function.

Sys(n)	Description
1	returns the OMNIS version number.
2	returns the OMNIS program type byte: bit 0 = full program (value 1), bit 1 = runtime (value 2), bit 2 = evaluation (value 4), bit 3 = integrated (value 8), bit 4 = unicode (value 16). For example, a runtime evaluation returns 6, that is 2+4. Note that the current version of OMNIS does not support the use of integrated versions
3	returns your company name entered on installation.
4	returns your name entered on installation.
5	returns your serial number entered on installation.
6	returns the platform code of the current executable: 'W' = Windows 3.x or Windows 95, 'N' = Windows NT, 'M' = Mac or PowerMac, 'S' = OS/2, 'U' = UNIX.
7	returns a string containing the version number of the current OS. For example, returns "3.11" under Windows for Workgroups version 3.11, "4.0" under Windows 95, and "7.5" under MacOS System Software 7.5.

Sys(n)	Description
8	returns the platform type of the current OMNIS program as a string: 'MAC68K', 'MAC600', 'WIN16', 'WIN32', 'OS2'
10	returns the pathname of the current open library file.
11,..,20	returns the pathname(s) of the current open data file segment(s) (empty if none are open).
21	returns the pathname of the current print file name (empty if not open).
22	returns the pathname of the current import file name (empty if not open).
23	returns the current port name (empty if no port open).
24	returns the current report device, for example, Printer, Screen, Preview, File (Screen is the default).
30,..,49	returns the name of the installed user-defined menu(s) starting from the left-most menu (empty if none are installed).
50,..,79	returns the name of the open user-defined window(s) starting with the top window (empty if none are open).
80	returns the current report name (empty if no report set).
81	returns the current search name (empty if no search set).
82	returns the main file name (empty if no main file set).
83	returns the number of records in main file.
85	returns the name of currently executing method in the form class name/method number.
86	returns a list of event parameters for the current event. The first parameter is always pEventCode containing an event code representing the event, for example, evClick for a click on a button: a second or third event parameter may be supplied which tells you more about the event
87	returns horizontal screen resolution in pixels per inch
88	returns vertical screen resolution in pixels per inch
89	returns the text for the current search calculation, or empty if no calculation is set.
91	returns the decimal separator
92	returns the thousand separator
93	returns the parameter separator for calculations
94	returns the file class field name separator
101	returns the current printer name, and network path (empty if not connected).
104	returns the screen width in pixels.

Sys(n)	Description
105	returns the screen height in pixels.
106	(MacOS only) returns the application heap size in bytes (empty on other platforms).
107	(MacOS only) returns the current free memory in bytes in the application heap after adding memory used for discardable objects (empty on other platforms).
108	(MacOS only) returns the current free memory in bytes in the application heap without adding memory used for discardable objects (empty on other platforms).
109	returns the unused memory in bytes. OMNIS attempts to use this for sorting and lists.
110	returns the CPU type for PCs, Macs, and compatibles. For PC: 3 = 80386, 4 = 80486, 5 = Pentium. For Mac: 3 = 68030, 4 = 68040. For PowerMac: 257 = PowerPC 601, 259 = PowerPC 603, 260 = PowerPC 604.
111	(MacOS only) returns the Apple ROM version: 121 = SI, 124 = IIsi, CI & FX.
112	(MacOS only) returns 1 (true) if balloon help is available, 0 otherwise.
113	(MacOS only) returns 1 (true) if Publish and Subscribe is available, 0 otherwise.
114	(MacOS only) returns 1 (true) if Apple events are available, 0 otherwise.
115	returns the pathname of the folder containing the OMNIS executable, including the terminating path separator.
120	(Windows 95 only) returns the width of the current dialog base-width unit based on the current system font; differs for Small and Large font mode.
121	(Windows 95 only) returns the height of the current dialog base-width unit based on the current system font; differs for Small and Large font mode.
130	returns the server name for the current session. For example, 'Oracle version [1.2 r0]' (empty if no server connected).
131	returns the SQL error code, or 0 for no error.
132	returns the SQL error text for the current error code (empty if not available).
133	returns the number of columns for the current Select table.
134	returns the number of rows processed by the previous Insert, Delete, or Update statement, returns 0 for most other statements.
135	returns the number of rows fetched from the Select table.
136	returns the name of the current cursor.

Sys(n)	Description
137	returns the name of the current session.
138	returns the number of Result sets to come back from the server following a Select. Returns 0 if no more results.

tan()

tan(*angle*)

Returns the Tangent of an *angle* where the *angle* is in degrees (or radians if #RAD is true).

```
tan(45)       ;; returns 1
```

textsize()

textsize(*string,fontname,pointsize,style,'width','depth'*)

Returns the width and depth in pixels of the specified text or string. The *string* parameter can be a literal string or character variable with a maximum length of 255; *fontname* is the name of the font; *pointsize* the point size of the font; *style* is represented by an integer, that is, 0 = Normal, 1 = Bold, 2 = Italic, 4 = Underline (or a combination of these: 3 would be bold-italic, for example). The *width* and *depth* parameters hold the returned values in pixels. The *width* and *depth* parameters are integer variables to hold the width and depth values returned; these must be in quotes. The function also returns a value of 0 to indicate the font does not exist or an error occurred, otherwise 1 is returned indicating the text was found.

```
; Declare class vars TWIDTH (Integer), TDEPTH (Integer),
; MYTEXT (Character 255), FONTNAME (Character),
; FONTSIZE (Integer), and STYLE (Integer)
Calculate STYLE as 6       ;; Italic Underline
Calculate FONTNAME as "Arial"
Calculate FONTSIZE as 16
If textsize(MYTEXT,FONTNAME,FONTSIZE,STYLE,'TWIDTH','TDEPTH')<>0
   ; do something depending on TWIDTH or TDEPTH
Else
   ; Font didn't exist or error occurred
End If
```

tim()

tim(*number*[,*timeformat*]) converts the specified *number* to a time determined by the *timeformat* argument; #FT is used as the time formatting string if the second argument is not specified. If the first argument is already a date/time, its format is changed to the format given by the second argument.

The #FT string defaults to 'H:N' and so *tim*(*number*) takes *number* to be a number of minutes. If you supply a format string such as 'N.S', *number* will be taken as a number of seconds.

```
tim(1)            ;; returns 00:01 when #FT is H:N (the default)
tim(950)          ;; returns 15:50
tim(950,'H:N.S')  ;; returns 00:15.50, that is 950 seconds
```

With a second argument, *tim()* is equivalent to *dat()* with two arguments. The format string determines how the conversion is carried out.

tot()

tot([*listname*,]*fieldname*)

Returns the total of the stored values of *fieldname* in a list specified by *listname*. The *fieldname* argument must be a field stored in the list, not a constant or expression. If the *listname* argument is not specified the current list is used. This function does not work for table based lists. If *fieldname* is not a numeric field, all values are converted to their numeric equivalents before being accumulated, for example

```
tot(LIST2,COST)
; returns total of all COST field values stored in list LIST2
tot(COST)
; returns total of all COST field values stored in current list
tot(#LSEL)
; returns total number of selected lines in the current list
```

totc()

totc([*listname*,]*expression*)

Returns the total of an *expression* evaluated for a list specified by *listname*. If the *listname* argument is not specified the current list is used. This function does not work for table based lists.

This is a more general version of the *tot()* function. The *expression* is totaled for the lines in the specified list. For example, if list *LIST1* contains field NUMBER, the sum of the squares of all the values of NUMBER in the list is:

```
totc(LIST1,NUMBER*NUMBER)
; note that totc(LIST1,NUMBER) is the same as tot(LIST1,NUMBER)
```

trim()

trim(*string*[,*leading=kTrue,trailing=kTrue,character*=space_char])

Removes the specified *leading* and/or *trailing* character from the *string*. You specify the character to be removed in the *character* argument. If this is omitted the space character is removed from the string by default.

```
trim(' ABCDE ') ;; returns 'ABCDE'
trim('*****ABCDE*****',kTrue,kFalse,'*') ;; returns 'ABCDE*****'
```

truergb()

truergb(*color*)

Converts the specified *color* into its true RGB value and returns the result.

```
truergb(kRed) ;; returns RGB value for red
truergb(rgb(255,0,0)) ;; returns RGB value for red
```

updatenames()

updatenames(*file|field1*[,*file|field2*]...)

Returns a list of files and/or fields to be used in a SQL Update statement.

The *updatenames()* function produces a "set" clause suitable for inclusion in a SQL Update statement of the form

```
SET NAME1=@[NAME1], NAME2=@[NAME2],....
```

If you do not want to update certain columns, you can eliminate some columns from the update statement like this:

```
SQL: Update FTEL updatenames(FTEL,-FTNUM)
; sends: Update FTEL SET FTNAME = @[FTNAME]
```

For example, to update a range of fields from the FCLIENTS file use

```
SQL: Update Table updatenames(CNAME..CCITY) WHERE wherenames(CKEY)
```

And, to update all the columns in FCLIENTS except C_SEQ use

```
SQL: Update Table updatenames(FCLIENTS,-C_SEQ) WHERE
  wherenames(CKEY)
```

Field name list

The general format of the field name list is to combine file and field names in a coma separated list:

```
createnames(File1,File2,field1,field3)
```

For all the fields in a file,

```
(filename)
```

You can remove particular fields from the values clause by inserting a minus sign before the field name. For example, to remove the sequence field FSEQ from the clause,

```
(File1,-FSEQ)
```

Field names from a List

If you have a list variable with field names in the first column, you can include these in the values clause using the *^listname* notation, for example

```
Set current list LIST_NAMES
Define list {FileClass 1}
Build field names list {FileClass 1}
SQL: Create table TABLE createnames(^LIST_NAMES)
```

Qualified Field Names

If the Unique field names option is turned off, you can use the *file|fieldname* /Q notation to force OMNIS to qualify each field with the file name, that is, File1.Fieldname1, File1.Fieldname2, and so on.

```
(File1 /Q)
```

corresponds to the expression

```
(File1.FIELDNAME1,File1.FIELDNAME2,....)VALUES
   (@[File1.FIELDNAME1],@[File1.FIELDNAME2],..)
```

upp()

upp(*string*)

Returns the upper case representation of a *string*. Any non-alphabetic characters in the *string* are ignored.

```
upp('Author')              ;; returns 'AUTHOR'
upp('oMnIs')               ;; returns 'OMNIS'
upp(mid('peripheral'),3,3)  ;; returns 'RIP'
```

wherenames()

wherenames([*comparison*][,*operator*,]*field1*[,*field2*]...)

Returns a constraining Where clause to be used in a SQL statement.

wherenames() is used as a shortcut when creating Select, Update or Delete statements which include the constraining Where clauses such as

```
Select * from Table WHERE Column = OMNIS_Value
; For example, WHERE KEY = [KEY]
```

wherenames() is most useful when there is a one-to-one correspondence between the name of the remote table key and the OMNIS field name that defines the row. Thus the command

```
Perform SQL {Select * from Table Where wherenames('=',KEY) }
```

is expanded by OMNIS to

```
Select * from Table where KEY = @[KEY]
```

When you create methods which are called with arguments such as the Table name and unique key, you can use square bracket notation to generalize the expression:

```
; Method 1
; Define parameter vars KEY and TABLE with Character type
; Call me with the name of the table and the key
SQL: Select * from [TABLE] WHERE wherenames('=', [KEY])
```

wherenames() takes three arguments: a *comparison* (=, >=, <=, >, <, <>,LIKE, NOT LIKE,!=) which defaults to = when omitted, a logical *operator* (AND, OR) defaulting to AND, and a *field* name list. Both the comparison and the logical operator should be enclosed in single quotes. The list of field names to be used in the Where clause can be from an OMNIS list as described earlier. See the following examples:

```
wherenames('>=',PCODE)
; becomes
PCODE >= @[PCODE]


wherenames('<=',PCODE,PTOWN)
; becomes
PCODE <= @[PCODE] AND PTOWN <= @[PTOWN]


wherenames('>','OR',PCODE,PTOWN)
; becomes
PCODE > @[PCODE] OR PTOWN > @[PTOWN]
```

FileOps External Functions

$changeworkingdir()

$changeworkingdir(*path*)

Changes the current working directory to the directory named in *path*. *$changeworkingdir()* only switches between folders on the same drive, not between drives. The function returns an error number, or zero if successful: see the FileOps function error codes at the end of this section.

```
Switch sys(6)
  Case 'M' ;; for MacOS
    Do FileOps.$changeworkingdir('HD:Omnis:Examples') Returns
  lvError
  Default ;; for other platforms
    Do FileOps.$changeworkingdir('c:\omnis\examples') Returns
  lvError
End switch
OK message {Working directory is now: [FileOps.$getworkingdir()]}
```

$copyfile()

$copyfile(*from-path* [,*to-path*])

Copies the file specified in *from-path* to the new location in *to-path*. You can use this function to copy and rename the specified file to the same folder or a different location. The file named in *to-path* should not already exist. The function returns an error number, or zero if successful: see the FileOps function error codes at the end of this section.

```
Do
  FileOps.$copyfile('c:\omnis\test.txt','c:\omnis\examples\test.txt
  ') Returns lvError
; copies 'test.txt' to the 'examples' folder
Do
  FileOps.$copyfile('c:\omnis\test.txt','c:\omnis\examples\test2.tx
  t') Returns lvError
; copies and renames 'test.txt' to 'test2.txt' in the 'examples'
  folder
Do FileOps.$copyfile('c:\omnis\test.txt','c:\omnis\test2.txt')
  Returns lvError
; copies and renames 'test.txt' to 'test2.txt' in the same folder
```

$createdir()

$createdir(*path*)

Creates the folder specified in *path*. The folder named in *path* must not already exist.
$createdir() does not create intervening folders, it only creates the last folder named in
path, therefore the intervening folders should already exist. The function returns an error
number, or zero if successful: see the FileOps function error codes at the end of this section.

```
Do FileOps.$createdir('c:\omnis\examples\extcomp\clock') Returns
  lvError
; creates the 'clock' folder assuming c:\omnis\examples\extcomp is a
  valid path
```

$deletefile()

$deletefile(*path*)

Deletes the file or folder named in *path*. Files deleted with *$deletefile()* are not moved into
the Recycled bin or Trash can, they are deleted irreversibly. You can delete a folder with
$deletefile(), but only if it is empty. The function returns an error number, or zero if
successful: see the FileOps function error codes at the end of this section.

```
Do FileOps.$deletefile('c:\omnis\examples\extcomp\test2.txt')
  Returns lvError
; deletes 'test2.txt' at 'c:\omnis\examples\extcomp'
Do FileOps.$changeworkingdir('c:\omnis') Returns lvError
Do FileOps.$deletefile('test3.txt') Returns lvError
; deletes 'test3.txt' at the current folder 'c:\omnis'
Do FileOps.$deletefile('c:\omnis\examples\extcomp\clock') Returns
  lvError
; deletes the 'clock' folder if empty
```

$doesfileexist()

$doesfileexist(*path*)

Returns true if the file or folder named in *path* exists. The function returns an error number, or zero if successful: see the FileOps function error codes at the end of this section.

```
Do
   FileOps.$doesfileexist('c:\omnis\examples\extcomp\clock\test2.txt
   ') Returns lvStatus
; lvStatus (Boolean) returns true if 'test2.txt' exists at
   'c:\omnis\examples\extcomp\clock'
Switch sys(6)
  Case 'M' ;; for MacOS
    Do FileOps.$changeworkingdir('HD:Omnis:Tutorial') Returns
   lvError
  Default ;; for other platforms
    Do FileOps.$changeworkingdir('c:\omnis\tutorial') Returns
   lvError
End switch
Do FileOps.$doesfileexist('mylib.lbs') Returns lvStatus
; returns true if 'mylib.lbs' exists at the current folder
   'c:\omnis\tutorial'
```

$filelist()

$filelist(*include* ,*path,* [*what-info, filter*]) Returns *list-name*

Returns a list containing a directory listing of the files, folders, and/or volumes in the folder specified in *path*. You specify what to *include* in the file list by specifying any one or a combination of the constants kFileOpsIncludeFiles, kFileOpsIncludeDirectories, and kFileOpsIncludeVolumes (you + multiple constants). You can specify *what-info* is returned by including any one or a combination of the kFileOpsInfo… constants, such as kFileOpsInfoSize, kFileOpsInfoCreated, otherwise the file name only is returned in the first column of the list. To return the long name under 32-bit Windows you must specify kFileOpsInfoFullName. The list returned by *$filelist()* can contain up to 11 columns always in the following order, regardless of the info requested or the order you specify the *what-info* constants.

Col	Col name	what-info constant	description
1	name	kFileOpsInfoName	name of the file
2	name83	kFileOpsInfoName83	DOS 8.3 name of the file
3	fullname	kFileOpsInfoFullName	32-bit Windows long name of the file
4	readonly	kFileOpsInfoReadOnly	file's read-only status
5	hidden	kFileOpsInfoHidden	file's hidden status
6	size	kFileOpsInfoSize	logical size of file
7	actualsize	kFileOpsInfoActualSize	physical size of file on disk; same as logical size under Windows
8	created	kFileOpsInfoCreated	date and time the file was created
9	modified	kFileOpsInfoModified	date and time the file was modified
10	creator	kFileOpsInfoCreatorCode	the file's creator under MacOS, blank under Windows
11	type	kFileOpsInfoTypeCode	the file's type under MacOS, the file extension under Windows

You can also apply a *filter* which specifies the file type or extension of the files to be included. For example, you can specify text files using '*.txt' under Windows, or 'TEXT' under MacOS. The function returns an error number, or zero if successful: see the FileOps function error codes at the end of this section.

```
Do FileOps.$filelist(kFileOpsIncludeFiles+
  kFileOpsIncludeDirectories,'c:\omnis') Returns lvList
; returns a list of the files and folders in the main OMNIS folder
Do FileOps.$filelist(kFileOpsIncludeFiles, 'c:\omnis\external',
  kFileOpsInfoName+kFileOpsInfoCreated+kFileOpsInfoSize, '*.dll')
  Returns lvList
; returns a list of DLLs in the OMNIS\EXTERNAL folder including the
  name, size, creation date and time of each file
Do FileOps.$filelist(kFileOpsIncludeFiles, 'c:\windows',
  kFileOpsInfoSize+kFileOpsInfoFullName+kFileOpsInfoReadOnly+
  kFileOpsInfoCreated) Returns lvList
; returns a list of files in the c:\windows folder
; including the fullname, read-only, size, creation date and time
; of each file
```

$getfileinfo()

$getfileinfo(*path*[,*what-info*]) Returns *list-name*

Returns the file information for the file named in *path*. You can specify *what-info* is
returned by including any one or a combination of the kFileOpsInfo… constants, such as
kFileOpsInfoSize, kFileOpsInfoCreated, otherwise the file name only is returned in the first
column of the list. To return the long name under 32-bit Windows you must specify
kFileOpsInfoFullName. The list returned by *$getfileinfo()* can contain up to 11 columns
always in the following order, regardless of the info requested or the order you specify the
what-info constants.

Col	Col name	what-info constant	description
1	name	kFileOpsInfoName	name of the file
2	name83	kFileOpsInfoName83	DOS 8.3 name of the file
3	fullname	kFileOpsInfoFullName	32-bit Windows long name of the file
4	readonly	kFileOpsInfoReadOnly	file's read-only status
5	hidden	kFileOpsInfoHidden	file's hidden status
6	size	kFileOpsInfoSize	logical size of file
7	actualsize	kFileOpsInfoActualSize	physical size of file on disk; same as logical size under Windows
8	created	kFileOpsInfoCreated	date and time the file was created
9	modified	kFileOpsInfoModified	date and time the file was modified
10	creator	kFileOpsInfoCreatorCode	the file's creator under MacOS, blank under Windows
11	type	kFileOpsInfoTypeCode	the file's type under MacOS, the file extension under Windows

The function returns an error number, or zero if successful: see the FileOps function error
codes at the end of this section.

```
; declare lvFileList (List), lvPath, lvFileName, lvSize, lvCreated
  all (Char)
Do sys(10) Returns lvPath ;; returns the name and path of the
  current library
Do FileOps.$getfileinfo(lvPath,kFileOpsInfoName+
  kFileOpsInfoCreated+kFileOpsInfoSize) Returns lvFileList
; returns the name, size, creation date and time of the current
  library
Do lvFileList.$redefine(lvFileName,lvSize,lvCreated)
Do lst(lvFileList,1,lvSize) Returns lvSize
; returns the value in the Size column
```

$getfilename()

$getfilename(*path* [,*prompt, filter, initial-directory*])

Opens the standard Open file dialog for the current OS. You can specify the dialog title in *prompt*, and limit the file type by specifying a *filter*. For example, you can specify text files using '*.txt' under Windows, or 'TEXT' under MacOS. You can also specify an *initial-directory* for the Open dialog. The name and full path of the file selected by the user is returned in the *path* parameter. Note the file is not opened as such, you must do something with the file name and path returned. The function returns an error number, or zero if successful: see the FileOps function error codes at the end of this section.

```
Do FileOps.$getfilename(lvPath,'Please locate the
   OMNIS help file','*.ohf','c:\omnis\help') Returns lvError
; returns the name and full path of the file
; selected, e.g. 'c:\omnis\help\omnis\omnis.ohf'
```

$getworkingdir()

$getworkingdir()

Returns the current working directory (no parameters required). The function returns an error number, or zero if successful: see the FileOps function error codes at the end of this section.

```
; declare lvWorkDir of Char type
Do FileOps.$changeworkingdir(sys(115)) Returns lvError
Do FileOps.$getworkingdir() Returns lvWorkDir ;; returns c:\omnis
```

$movefile()

$movefile(*from-path, to-path*)

Moves the file named in *from-path* to the new location in *to-path*. Use *$copyfile()* to copy a file to a new location. The function returns an error number, or zero if succe see the FileOps function error codes at the end of this section.

```
Do FileOps.$movefile('c:\omnis\examples\extcomp\extcomp.lbs',
   'c:\omnis\startup\extcomp.lbs') Returns lvError
; moves the library 'extcomp.lbs' to the OMNIS\Startup folder
```

$putfilename()

$putfilename(*path* [,*prompt, filter, initial-directory*])

Opens the Save as dialog for the current OS. You can specify the dialog title in *prompt*, and limit the file type by specifying a *filter*. For example, you can specify text files using '*.txt' under Windows, or 'TEXT' under MacOS. You can also specify an *initial-directory* for the

Save dialog. The name and full path of the file entered by the user is returned in *path*. Note the file is not saved as such, you must code a save method. The function returns an error number, or zero if successful: see the FileOps function error codes at the end of this section.

```
Do FileOps.$putfilename(lvPath,'Save print file','*.rep',
   'c:\omnis\examples') Returns lvError
; opens the Save dialog with the title 'Save print file'
; and returns the name and full path of file entered by the user
```

$rename()

$rename(*oldname, newname*)

Renames the file or folder named in *oldname* to the *newname*. The function returns an error number, or zero if successful: see the FileOps function error codes at the end of this section.

```
Do FileOps.$rename('c:\omnis\libs','c:\omnis\examples) Returns
   lvError
; renames the 'libs' folder to 'examples'
Do FileOps.$changeworkingdir('c:\omnis\datafile\odbc') Returns
   lvError
Do FileOps.$rename('odbc.txt','readme.txt') Returns lvError
; switches to the 'c:\omnis\datafile\odbc' folder and renames
   'odbc.txt'
```

$selectdirectory()

$selectdirectory(*path* [,*prompt, initial-directory*])

Opens the Select folder dialog for the current OS. You can specify the dialog title in *prompt*, and the *initial-directory*. The name and full path of the folder selected by the user is returned in *path*. The function returns an error number, or zero if successful: see the FileOps function error codes at the end of this section.

```
; declare lvPath, lvWorkDir both (Char)
Do FileOps.$changeworkingdir(sys(115)) Returns lvError
Do FileOps.$getworkingdir() Returns lvWorkDir
Do FileOps.$selectdirectory(lvPath,'Select a folder',lvWorkDir)
   Returns lvError
; switches the working dir to 'c:\omnis' and prompts the user to
   select a folder
```

$setfileinfo()

$setfileinfo(*path, what-info, info-setting, ...*)

Sets file information for the file named in *path*. You specify *what-info* is to be changed using one of the kFileOpsInfo… constants, although in practice you can change only the read/write and hidden status of a file (kFileOpsInfoReadOnly or kFileOpsInfoHidden), assuming you have permission. You specify kTrue or kFalse for the read-only or hidden status in the *info-setting* parameter. You can supply a list of file info settings, as shown below. The function returns an error number, or zero if successful: see the FileOps function error codes at the end of this section.

```
Do FileOps.$setfileinfo('c:\omnis\meths.txt',
   kFileOpsInfoReadOnly,kTrue,kFileOpsInfoHidden,kTrue) Returns
   lvError
; sets the file 'meths.txt' to read-only and hidden
```

$splitpathname()

$splitpathname(*path,drive-name,directory-name,file-name,file-extension*)

Splits the specified *path* into *drive-name, directory-name, file-name,* and *file-extension.* The function returns an error number, or zero if successful: see the FileOps function error codes at the end of this section.

```
; declare lvPath, lvDrive, lvDirName, lvFileName, lvFileExtn all
   (Char) type
Do sys(10) Returns lvPath ;; returns the name and path of current
   library
Do
   FileOps.$splitpathname(lvPath,lvDrive,lvDirName,lvFileName,lvFile
   Extn) Returns lvError
; under Windows, when lvPath='C:\OMNIS\EXAMPLES\EXTCOMP.LBS'
; lvDrive returns      C:
; lvDirName returns     \OMNIS\EXAMPLES\
; lvFileName returns    EXTCOMP
; lvFileExtn returns    .LBS
Do
   FileOps.$splitpathname('c:\omnis',lvDrive,lvDirName,lvFileName,lv
   FileExtn) Returns lvError
; under Windows
; lvDrive returns      C:
; lvDirName returns     \
; lvFileName returns    OMNIS
; lvFileExtn returns    (Empty)
```

FileOps External function Error Codes

The following errors are returned from the FileOps functions.

kFileOpsNoOperation	999	Operation not supported on this platform
kFileOpsUnknownError	998	Unknown error
kFileOpsOutOfMemory	12	Out of memory
kFileOpsParamError	1	Too few parameters passed
kFileOpsOK	0	No Error
kFileOpsDirFull	-33	File/Directory full
kFileOpsDiskFull	-34	Disk full
kFileOpsVolumeNotFound	-35	Specified volume doesn't exist
kFileOpsDiskIOError	-36	Disk I/O error
kFileOpsBadName	-37	Bad file name or volume name (perhaps zero-length)
kFileOpsFileNotOpen	-38	File not open
kFileOpsEndOfFile	-39	Logical end-of-file reached during read operation
kFileOpsPositionBeforeStart	-40	Attempt to position before the start of the file
kFileOpsTooManyFilesOpen	-42	Too many files open
kFileOpsFileNotFound	-43	File not found
kFileOpsHardwareVolumeLock	-44	Volume is locked by a hardware setting
kFileOpsFileLocked	-45	File is locked
kFileOpsSoftwareVolumeLock	-46	Volume is locked by a software flag
kFileOpsMoreFilesOpen	-47	One or more files are open
kFileOpsAlreadyExists	-48	A file with the specified name already exists
kFileOpsAlreadyWriteOpen	-49	Only one access path a file can allow writing
kFileOpsNoDefaultVolume	-50	No default volume
kFileOpsVolumeNotOnline	-53	Volume not on-line
kFileOpsPermissionDenied	-54	Permission denied.
kFileOpsReadOnlyFile	-54	Read only file
kFileOpsVolumeAlreadyMounted	-55	Specified volume is already mounted and on-line

kFileOpsBadDrive	-56	No such drive number
kFileOpsInvalidFormat	-57	Volume lacks Macintosh-format directory
kFileOpsExternalSystemError	-58	External file system error
kFileOpsProblemDuringRename	-59	Problem during rename
kFileOpsBadMasterBlock	-60	Master directory block is bad; must re-initialize volume
kFileOpsCantOpenLockedFile	-61	Cannot open a locked file
kFileOpsDirectoryNotFound	-120	Directory not found
kFileOpsTooManyDirOpen	-121	Too many working directories open
kFileOpsCantMoveToOffspring	-122	Attempted to move into offspring
kFileOpsNonHFSOperation	-123	Attempt to do HFS operation on a non-HFS volume
kFileOpsInternalSystemError	-127	Internal file system error

FontOps External Functions

$replistfonts()

$replistfonts(*list*)

Populates the specified *list* with the report fonts installed on your system, and indicates whether or not they are truetype. The list must contain two columns, the first character type, the second boolean. The function returns zero for success, less than zero for failure. Having built the list you can search and manipulate the list using the standard list functions and methods.

```
; declare lvfonlist (List), lvfont (Char), lvTrueType (Boolean)
Do lvfonlist.$define(lvfont,lvTrueType)
Do FontOps.$replistfonts(lvfonlist)
; returns a list something like...

Arial             True
Bookman Old Style True
Courier           False
Garamond          True
etc...            ...
```

```
; declare lvNumOfFonts, lvTrueFonts, lvNotTrueType all
; Number type, using lvfonlist from above...
Do lvfonlist.$linecount() Returns lvNumOfFonts ;; returns 64
Do tot(lvfonlist,lvTrueType) Returns lvTrueFonts ;; returns 52
Do totc(lvfonlist,lvTrueType=kFalse) Returns lvNotTrueType ;;
  returns 12
```

$reptextheight()

$reptextheight(*font-name|font-table-index*,*point-size*[,*font-style*,*extra-points*])

Returns the height in inches/cms (depending on $usecms preference) of the specified report font. You specify the font using either the *font-name* or *font-table-index*. When called with a font table index, $reptextheight() uses the report font system table of the current library which can contain up to 15 fonts numbered 1 to 15. You specify the *point-size* of the font, and you can include a *font-style* constant and a number of *extra-points*.

```
Do FontOps.$reptextheight('Times',144) Returns lvHeight
; returns 2.24 ins / 5.69 cms
Do FontOps.$reptextheight('Times',144,,24) Returns lvHeight
; returns 2.57 ins / 6.54 cms
```

$reptextwidth()

$reptextwidth(*string*, *font-name|font-table-index*,*point-size*[,*font-style*])

Returns the width in inches/cms (depending on $usecms preference) required to draw the *string* using the specified report font. You specify the font using either the *font-name* or *font-table-index*. When called with a font table index, $reptextwidth() uses the report font system table of the current library which can contain up to 15 fonts numbered 1 to 15. You can include a *font-style* constant, or combination of styles.

```
Do FontOps.$reptextwidth('Hello WWW','Arial',24) Returns lvWidth
; returns 1.83 ins / 4.66 cms
Do FontOps.$reptextwidth('Hello WWW','Arial',24,kBold+kItalic)
  Returns lvWidth
; returns 1.85 ins / 4.71 cms
Do FontOps.$reptextwidth('Hello WWW',2,72,kBold+kItalic) Returns
  lvWidth
; returns 5.40 ins / 13.72 cms ;; note Courier is at position 2 in
  #WIRFONTS
```

$winlistfonts()

$winlistfonts(*list*)

Populates the list with the window fonts installed on your system, and indicates whether or not they are truetype. The list must contain two columns, the first character type, the second boolean. The function returns zero for success, less than zero for failure. Having built the list you can search and manipulate the list using the standard list functions and methods.

```
; declare lvfonlist (List), lvfont (Char), lvTrueType (Boolean)
Do lvfonlist.$define(lvfont,lvTrueType)
Do FontOps.$winlistfonts(lvfonlist)
; returns a list something like...

Arial               True
Bookman Old Style   True
Chicago             False
Courier             False
Garamond            True
etc...              ...

; declare lvFirstFont, lvLastFont of Char type
Do lst(lvfonlist,1,lvfont) Returns lvFirstFont ;; returns Arial
Do lst(lvfonlist,lvfonlist.$linecount,lvfont) Returns lvLastFont ;;
   returns Wingdings
```

$wintextheight()

$wintextheight(*font-name|font-table-index*,*point-size*[,*font-style*,*extra-points*])

Returns the height in screen units of the specified window font. You specify the font using either the *font-name* or *font-table-index*. When called with a font table index, $wintextheight() uses the window font system table of the current library which can contain up to 15 fonts numbered 1 to 15. You specify the *point-size* of the font, and you can include a *font-style* constant and a number of *extra-points*.

```
Do FontOps.$wintextheight('Courier',72) Returns lvHeight
; returns 96 under Windows
Do FontOps.$wintextheight('Courier',72,,24) Returns lvHeight
; returns 128 under Windows
Do FontOps.$wintextheight(2,36) Returns lvHeight
; returns 48 under Windows ;; note Courier is at position 2 in
   #WIWFONTS
```

$wintextwidth()

$wintextwidth(*string*, *font-name|font-table-index*,*point-size*[,*font-style*])

Returns the width in screen units required to display the *string* using the specified window font. You specify the font using either the *font-name* or *font-table-index*. When called with a font table index, $wintextwidth() uses the window font system table of the current library which can contain up to 15 fonts numbered 1 to 15. You can include a *font-style* constant, or combination of styles.

```
Do FontOps.$wintextwidth('Hello WWW','Courier',36) Returns lvWidth
; returns 243
Do FontOps.$wintextwidth('Hello WWW','Courier',36,kBold+kItalic)
  Returns lvWidth
; returns 276
Do FontOps.$wintextwidth('Hello WWW',5,36) Returns lvWidth
; returns 240 ;; note System font is at position 5 in #WIWFONTS
```

Chapter 2—Hash Variables

This chapter describes the hash variables available in OMNIS. They are arranged in alphabetical order.

About the Hash Variables

A *hash variable* is an OMNIS variable with global scope that you can use to temporarily store data. The name comes from the fact that all hash variables begin with the "#"character, the hash sign.

To select a hash variable

- Press F9/Cmnd-9 to display the **Catalog,** and select the **Hash** tab

- In the left-hand list, click on the appropriate group of hash variables

- In the right-hand list, double-click on the hash variable you require

Hash Variables

#???

#??? is displayed by OMNIS when a reference is made to a field which does not exist. This can occur when you delete a field name from a file class or when a window class is copied into your library that references non-existing fields (for example, the corresponding file class has not been copied). This is a "special" variable with zero length, hence it *cannot* hold data.

#1, #2,..,#60

Numeric variables numbered from #1 to #60 for storing positive and negative values. Initialized to zero, their values are retained when you close your library or open a new one. You can reinitialize them in your library using *Clear range of fields #1 to #60*.

You can use the notation #*nn*D*x* with numeric fields to limit the number of decimal places displayed. The value of *nn* can be from 1 to 60 corresponding to the sixty numeric variables

available. The value of *x* can be in the range 0 to 15, that is, values are displayed with zero to 14 decimal places. For example, #3D2 will display the number held in the variable #3 to 2 decimal places. The value of a numeric variable is always stored as a real number. Changing the decimal places only affects the way the number is displayed and not its stored value. All numeric variables are set to zero decimal places when OMNIS starts, and the number of decimal places for a particular variable is unaffected by changing to a different library from within OMNIS.

You can use the notation #nnF to display #nn as a floating decimal; again, this does not affect how the value is stored.

When comparing hash variables with file class fields, you must use the *rnd()* function, for example, *If rnd(#2,2) = FIELD* will ensure an exact comparison, assuming FIELD is a Number 2 dp type.

#ALT

True if the Alt key (or Option key under MacOS) is held down.

#CLIST

Read-only string variable which stores the name of the current list. The eight built-in lists are held as the strings "#L1" to "#L8".

#COMMAND

True if the Cmnd key (or Ctrl key under Windows) is held down.

#CT

Read-only numeric variable which stores the current tick count since the system was booted; the tick count is incremented 60 times per second.

#CTRL

True if the Ctrl key (or Cmnd key under MacOS) is held down.

#D

Read-only string variable which is set to the operating system date when OMNIS starts, and changed during the operation of OMNIS only if the system date is changed using the Control Panel. #D is actually a Date and Time data type but is made to look like a Short date using #FD. Thus, the calculation *dat(#D,#FDT)* returns the full date *and* time.

For example, you can place the date on a report using the following text object (the square brackets will force the date to print).

```
Date: [#D]
```

The following example uses #D to initialize month and year library variables.

```
Calculate LV_CurrentMonth as dtm(#D)
Calculate LV_CurrentYear as dtcy(#D)
```

#ENTER

True if the Enter key is pressed when an evOK event is reported to a control method.

#ERRCODE

Numeric variable which reports the error number generated by a method. For example, running the *Set main file* command without a valid file name causes an error with #ERRCODE set to 108139. Warning error codes are between 1 and 99,999 while fatal errors are greater than 100,000.

Warning error codes are also represented by constants; see the *Constants* chapter in this manual.

#ERRTEXT

String variable which reports the error text generated by a method. For example, running the *Set main file* command without a valid file name causes an error with #ERRTEXT set to "Set main file command with no valid file name."

The following method attempts to start the session named in LV_SESSION. If it cannot be found, #ERRCODE is set and the user is alerted.

```
Start session {[LV_SESSION]}
If #ERRCODE
   OK message {Error: [#ERRCODE]//[#ERRTEXT]}
   ; says "Error: 8739"
   ;      "The SQL DAM specified cannot be found"
   ; do prompt for valid session name
End If
```

#F

Numeric variable which stores the status of the OMNIS flag; it can be *true* (numeric value of 1) or *false* (numeric value of 0). There are several commands that test the value of the flag, such as *If flag true, If flag false, Until flag true, While flag true,* and so on.

#FD

String variable used to specify the display format of a Short date field value. The default value of #FD depends on the language version of OMNIS you are using. For example, European versions of OMNIS set #FD to 'D m Y', but you can assign it a new value using the following date formatting symbols.

Y	Year (89)	d	Day (12th)
y	Year (1989)	W	Day of week (5)
C	Century (19)	w	Day of week (Friday)
M	Month (06)	V	Short day of week (Fri)
m	Month (JUN)	E	Day of year (1–366)
n	Month (June)	G	Week of year (1–52)
D	Day (12)	F	Week of month (1–6)

You can see the effect of different values of #FD in the following table, which shows the display of a fixed date, JUN 12 97, for various values of #FD.

#FD	Date display
m D CY	JUN 12 1997
M/D/Y	06/12/97
MDY	061297
D-m-y	12-JUN-1997

#FDP

Numeric variable which specifies the format used for display or string conversion of a floating point number; it does not affect how the values are stored internally by OMNIS. #FDP defaults to 12 for a newly selected library file.

If #FDP is positive, then floating numbers are displayed with #FDP digits in decimal format if possible or otherwise in 'e' format. For example, if #FDP equals +2, then 45.456 is displayed as 45, and 999 is displayed as 1.0e3. In the case of 999, there are too many whole numbers to be accommodated by the decimal format, and the figure is rounded and displayed in 'e' format. Note that 9.9×10^2 becomes 1.0×10^3.

If #FDP is negative, then floating numbers are always displayed with exactly *abs(#FDP)* digits in 'e' format. For example, if #FDP equals -2, then 45.456 is displayed as 4.5e+01, that is, 4.5×10^1.

#FDT

String variable used to specify the display format of a Long date data type, that is, a Date and time field value.

The default value of #FDT depends on the language version of OMNIS you are using. For example, European versions of OMNIS set #FDT to 'D m Y H:N:S', but you can assign it a new value using the following date formatting symbols.

Y	Year (89)	H	Hour (0..23)
y	Year (1989)	h	Hour (1..12)
C	Century (19)	N	Minutes
M	Month (06)	S	Seconds
m	Month (JUN)	s	Hundredths
n	Month (June)	A	AM/PM
D	Day (12)	V	Short day of week (Fri)
d	Day (12th)	E	Day of year (1–366)
W	Day of week (5)	G	Week of year (1–52)
w	Day of week (Friday)	F	Week of month (1–6)

For example, you can use the following commands in a method to set the format of #FDT temporarily.

```
; #FDT is curently 'D m Y  H:N:S'
Begin reversible block
   Calculate #FDT as 'm D Y'
End reversible block
; do something with dates...
; when method ends #FDT is reverted
```

The **Catalog** (F9/Cmnd) contains a list of the codes as given above.

You can see the effect of different values of #FDT in the following table, which shows the display of a fixed date JUN 12 97, at a fixed time 15:45.

#FDT	Date and Time display
m D CY H:N.S.s	JUN 12 1997 15:45.00.00
D m Y H:N A	12 JUN 97 3:45 PM
M/D/Y H.N	06/12/97 15.45
D-m-19Y	12-JUN-1997

You can create up to 30 date field subtypes with preset formatting strings using the **File>>Preferences>>Change Date Formats** menu item. The date types are stored in the library format #DFORMS.

#FT

String variable which specifies the display format of a Short time field value. The default value of #FT is 'H:N', but you can assign it a new value using the following date formatting symbols.

H   Hour (0..23)

h   Hour (1..12)

N   Minutes

S   Seconds

s   Hundredths

A   AM/PM

You can see the effect of different values of #FT in the following table, which shows the display of a fixed time, 15:45, for various values of #FT.

#FT	Time display
H:N	15:45
h N A	3 45 PM
H:N.S.s	15:45.00.00

#L

Numeric variable which stores the line number of the current line in the current list. If there is no current line or the current list is empty #L is set to zero. The value of #L is updated when a different line in the list is made current. #L is unchanged by list commands such as *Merge lists, Sort lists* and *Add line to list*. A *Calculate #L as...* command is the normal way to change #L within a method. In addition, *Search list* searches the list and sets #L to the first line number which matches the search condition and loads the values from the selected line into the CRB. If you wish, you can set #L to a value greater than #LN or less than 0.Note #L is equivalent to the notation Listname.$line.

For example, you can implement a *Repeat* loop that steps through the contents of the list: the loop repeats until #L, the current line, reaches the end of the list, #LN.

```
; Define and build the list
Set current list LISTNAME
Calculate #L as 1
Repeat
   Load from list
   ; do something with each list line
   Calculate #L as #L + 1
Until #L > #LN
```

In a *For each line in list* loop, #L is automatically incremented.

#L1,..,#L8

Global list variables that let you create list structures available to all libraires. To use one of these lists, you must make it the current list using the command *Set current list* and define the fields for the columns using *Define list*.

For example, the following method uses #L2 to set the columns for a graph.

```
Set current list #L2
Define list {#S2}
Add line to list {'Sales'}
Add line to list {'Expenses'}
Add line to list {'Projections'}
Set graph attribute ('W_Graph',1,'$a2d_labellist_column',#L2)
```

Note #L1 to #L8 retain their definitions and values between libraries. You can clear them using the *Clear range of fields #L1 to #L8* command.

#LM

Numeric variable which stores the maximum number of lines to be stored in a list. Each list stores its own #LM value which defaults to 100,000,000. By changing the value of #LM, you can limit the number of lines held in a list. The number of lines you can store is also limited by the available memory. When lines are added to a list you can test value of the flag; a flag false indicates that either #LM or your memory has been exceeded.

#LN

Numeric variable which stores the current number of lines in the current list. Each list stores its own value of #LN. Its value is changed as lines are added to or deleted from the list. You can specify its value with *Set final line number*. You can use this command to truncate the list or add blank lines to the end. You cannot give #LN a negative value or set it greater than #LM. Note #LN is equivalent to the notation Listname.$linecount.

For example, you can use #LN to set the End value of a list For loop

```
; Define and build the list
Set current list LISTNAME
For each line in list from 1 to #LN step 1
   Load from list
   ; do something with each list line
End For
```

#LSEL

Read/write boolean variable which stores the selection status of current line in the current list. #LSEL=1 if the current line (#L) of the current list (#CLIST) is selected. Its value is changed by the *Select list line(s), Deselect list line(s)* commands and by the user clicking on the list field.

#MU

Read-only numeric variable which stores the current workstation number (#MU=0 when OMNIS is running in single-user mode). When in multi-user mode, OMNIS automatically assigns a number to each workstation. The numbers are not necessarily contiguous and often jump in multiples of 255 depending on the serial numbers of the workstations currently logged on. For data files on PC hard disks which are non-sharable , #MU is set equal to 0. When a user quits using a data file, the #MU number is made available for a new user.

The *Test for only one user* command is used to check if anyone else is accessing a data file.

#NULL

Returns a NULL value. You can use it to assign a null value to field or variable. A null value is not the same as zero (for numeric and Boolean data types) or "empty" (for non-numeric or Boolean data types). When a field has a value of null, it is completely unknown as to what that value is, and there is therefore no way to operate on that field value.

#OPTION

True if the Option key (or Alt key under Windows) is held down.

#P

Numeric variable which stores the current page number during the printing of a report instance. #P will return the number of pages in a subtotal and/or totals report section.

You can place #P on a report either as a field or within square brackets in text strings, for example, "Page [#P]".

#PI

Read-only numeric variable which stores the value of pi.

#R

Numeric variable which stores the current number of records printed during the printing of a report instance. You can place #R on a report either as a field or within square brackets in text strings, for example, "Record number [#R]". The value of #R in a report subtotal section is set to the number of records printed in that particular subtotal section.

#RAD

Local boolean variable which controls whether the angles for trigonometric functions are in degrees or radians; the default is degrees since #RAD is initialized to false. However, if you set #RAD to true, angles are in radians.

#RATE

Numeric variable which stores the initial guess of interest rate with which an annuity is calculated. #RATE defaults to 0.05 for a newly selected library file.

#RETURN

True if the Return key is pressed and an evOK event is reported to a control method terminating enter data.

#S1,..,#S5

Global string variables that let you store string values up to 10 million (10,000,000) characters long. Initialized as "empty", their values are retained when you close your library or open a new one. You can initialize them using *Clear range of fields #S1 to #S5*.

#SHIFT

True if the Shift key is held down.

#SUBFLD

String variable which stores the name of the report subtotal field which triggered that subtotal. Thus *nam(#SUBFLD)* returns a string containing the name of the subtotal field and [#SUBFLD] placed in the subtotal section returns the subtotal value.

#T

Read-only string variable which is set to the operating system time, that is, the value of #T is updated from the system clock each time it is used in a format. #T is actually a Date and Time data type but is made to look like a Short time data type using #FT. Thus *dat(#T,#FDT)* returns the full time *and* date.

#UL

Read-only numeric variable which stores the current user level in the password security system. In a range 0 to 8, a value of 0 represents the master user level.

Chapter 3—Events

This chapter describes the standard event messages reported in OMNIS and their parameters. In this chapter the event messages are arranged in groups according to the object that generates or receives the event.

About the Event Codes

Almost all user actions in OMNIS generate an *event*. When the event occurs an *event message* is sent to the object in which the event occurred. These messages are intercepted by your event handling methods. A message may contain one or more *event parameters,* and first parameter always contains an *event code* representing the event. All the event parameters are prefixed with the letter "p", and all event codes are prefixed with the letters "ev". For example, a standard mouse click on a window field generates a message with a pEventCode event parameter containing an evClick.

An event message may contain a second or a third event parameter. These parameters tell you more about the event. For example, a click on a list field generates a message with pEventCode containing evClick, and a second event parameter pRow containing the number of the row clicked on. You can use the event codes and parameters in your event handling methods. For example

```
On evClick         ;; method behind a list field
  If pRow = 1      ;; if row 1 was clicked on
    ; Do this...
  End If
  If pRow = 2      ;; if row 2 was clicked on
    ; Do that...
```

Also you can test pEventCode in your event handling methods.

```
On evAfter,evBefore       ;; method behind field
  ; Do this code for both event messages
  If pEventCode = evAfter
    ; Do this for evAfter events only
  End If
  If pEventCode = evBefore
    ; Do this for evBefore events only
  End If
```

To select an event code in an event handling method

- Enter the On command in the method editor and click on the required event code from the list provided in the method editor, e.g. On evClick

or to enter an event in your code, or multiple events

- Press F9/Cmnd-9 to display the **Catalog,** and select the **Events** tab

- In the left-hand list, click on the appropriate group of events

- In the right-hand list, double-click on the event code you require

To select an event parameter for the current event

- Enter the On command in the method editor plus the required event code, e.g. On evClick

- Press F9/Cmnd-9 to display the **Catalog,** and select the **Variables** tab

- In the left-hand list, click on the **Event parameters** group of events

The right-hand list now contains event parameters relevant to the current event code. To enter a parameter into your code

- Double-click on the event parameter

Event Parameters

The following event parameters are available.

Event Parameter	Description
pCellData	the data in the grid cell
pChannelNumber	the DDE channel number
pClickedField	a reference to the field clicked on
pClickedWindow	a reference to the window clicked on
pCommandNumber	internal number of the menu option selected
pContextMenu	a reference to the menu instance
pDdeValue	the new value received using DDE
pDdeItemName	the DDE data item name used to address the received value
pDragField	a reference to the dragged field
pDragType	the field type of the dragged field
pDragValue	the actual value of the data being dragged
pDropField	a reference to the field being dropped on
pEventCode	the type of event, contains an event constant
pHorzCell	the column selected in a grid field
pIsVertScroll	true if the scrolltip is for the vertical scroll bar
pKey	the letter key pressed
pLineNumber	the line number of a list field
pMenuLine	line number of option selected in a custom menu
pNextCode	the event code to follow an evAfter
pNodeItem	a reference to the tree list node clicked on
pRow	the number of the selected row in a grid
pScrollPos	the new scroll position following a scroll
pScrollTip	the string in the scrolltip
pSelectionCount	the number of selected objects in a modify report field
pSystemKey	the system key pressed
pTabNumber	the tab number selected for a tab pane
pVertCell	the row selected in a grid field

Field Events

The following event messages are sent to the current *target* field ($ctarget).

Event Code	Generated when...	Event Parameters
evAfter	the cursor is about to leave the current field; a field oftens gets evAfter, but remains the focus field and many more evAfter than evBefore events are generated; the second event parameter is the reason the field is about to loose the focus (e.g. evWindowClick, evMouseDown, evCloseBox, evOk, evTab, evShiftTab), and for evWindowClick, the third event parameter is a reference to the field or window being clicked. Discarding this event causes the current field to remain the target field (however, this may not prevent a window or library from closing)	pEventCode, pClickedField, pClickedWindow, pMenuLine, pNextCode, pCommandNumber, pRow
evBefore	the cursor is about to enter the current field. Discarding this event has no effect since the event has already ocurred	pEventCode, pRow
evClick	a click on buttons, lists, other controls or the window background (but not entry fields); generated when an evMouseDown occurs, no drag operation occurs and an evMouseUp is still within the field's boundary	pEventCode, pRow
evDoubleClick	a double-click on lists, other controls and the window background; generated in response to an evMouseDouble	pEventCode, pRow
evOpenContextMenu	a context menu has been opened over the field, also reported for windows; the second parameter is an item reference to the context menu instance	pEventCode, pContextMenu, pClickedField
evSent	the contents of a field gets updated by a DDE or AppleEvent message; the second and third parameters are for DDE only and contain the new value received using DDE, and the DDE data item name used to address the received value	pEventCode, pDdeValue, pDdeItemName, pChannelNumber

Grid Events

The following event messages are generated when a grid field is changed in some way by the user.

Event Code	Generated when...	Event Parameters
evCellChanged	the cell has changed; perhaps the user has tabbed. The second and third parameters give you the position of the cell, and the fourth parameter contains the data in the cell	pEventCode, pHorzCell, pVertCell
evCellChanging	the cell that is about to change. The second and third parameters give you the position of the cell, and the fourth parameter contains the data in the cell	pEventCode, pHorzCell, pVertCell, pCellData
evExtend	the complex grid has been expanded, that is, extra lines have been added	pEventCode, pLineNumber, pRow
evRowChange	the row in the complex grid has changed	pEventCode, pLineNumber, pRow
evScrollTip	the grid field is being scrolled	pEventCode, pIsVertScroll, pScrollPos, pScrollTip

Headed List Box Events

The following event messages are generated for headed list box fields only.

Event Code	Generated when...	Event Parameters
evHeadedListEditFinished	a cell in a headed list box has been edited; the second and third parameters are the line and column numbers of the selected cell	pEventCode, pLineNumber, pColumnNumber
evHeadedListEditFinishing	the user has entered a new value and pressed return or clicked away from the edit field; discarding this event leaves the field in edit mode; the second and third parameters are the line and column numbers of the selected cell; the fourth parameter is the new text entered, which you can transfer to the list	pEventCode, pLineNumber, pColumnNumber, pNewText
evHeadedListEditStarting	sent on the first click in the selected cell which puts the cell into edit mode; discarding this event prevents editing; the second and third parameters are the line and column numbers of the selected cell	pEventCode, pLineNumber, pColumnNumber
evHeaderClick	a header button has been clicked on; the second parameter contains the column number	pEventCode, pColumnNumber

Icon Array Events

The following event messages are generated for icon array fields only.

Event Code	Generated when...	Event Parameters
evIconDeleteFinished	the delete has occurred, after all selected lines in the list have been deleted	pEventCode
evIconDeleteStarting	the delete is pressed; discarding this event prevents the delete occurring	pEventCode
evIconEditFinished	the user has finished editing; the second parameter contains the line number of the list that has been edited	pEventCode, pLineNumber
evIconEditFinishing	the user enters a new value by hitting return or clicking away from the edit field; discarding the event leaves the field in edit mode; the second parameter contains the line number of the list being edited; the third is the new text entered	pEventCode, pLineNumber, pNewText
evIconEditStarting	the first click in the selected cell which puts the cell into edit mode; discarding the event prevents editing; the second parameter contains the line number of the list that is to be edited	pEventCode, pLineNumber

Key Events

The following event messages are generated when the user presses a key. Key events are generated only if the $keyevents property is enabled. Discarding these events prevents the key being handled by the field.

Event Code	Generated when...	Event Parameters
evKey	any key is pressed; the second event parameter holds the letter of the key being pressed or zero if a system key is pressed; the third event parameter holds a constant containing the system key being pressed (see the Keyboard constants) or zero if a normal key is pressed. This event occurs before any processing of the key has been carried out	pEventCode, pKey, pSystemKey
evShiftTab	the shift-tab keys is pressed	pEventCode
evTab	the tab key is pressed	pEventCode

Modify Report Field Events

The following event messages are generated for modify report fields only.

Event Code	Generated when...	Event Parameters
evSelectionChanged	the user selects another object in the field; the second parameter is the number of objects selected	pEventCode, pSelectionCount

Mouse Events

The following mouse event messages are sent to a field or window background. Mouse and right-button mouse events are generated only if the $mouseevents and $rmouseevents properties are enabled. Discarding any of these events (except evMouseDouble and evMouseDown) has no effect since the event has already occurred.

Event Code	Generated when...	Event Parameters
evCanDrop	a drag operation is started to test whether the field or window containing the mouse can accept a drop. Discarding this event prevents a drop onto this field or window	pEventCode, pDragType, pDragValue, pDragField
evDrag	the mouse is held down in a field and a drag operation is about to start. Discarding this event cancels the drag	pEventCode, pDragType, pDragValue
evDrop	the mouse is released over the destination field or window at the end of a drag operation; the second event parameter is a reference to the object being dropped	pEventCode, pDragType, pDragValue, pDragField
evMouseDouble	the mouse is double-clicked in a field or window. Discarding this event ensures that no double-click is generated	pEventCode
evMouseDown	the mouse is pressed and held down in a field or window. Discarding this event ensures that no drag action happens and no click is generated (but evMouseUp is still reported)	pEventCode
evMouseEnter	the mouse enters a field or leaves a field and enters the window background	pEventCode
evMouseLeave	the mouse leaves a field or enters a field from the window background	pEventCode
evMouseUp	the mouse is released over the field or window which had the evMouseDown	pEventCode
evRMouseDouble	the right-button is pressed	pEventCode
evRMouseDown	the right-button is pressed	pEventCode
evRMouseUp	the right-button is released	pEventCode
evWillDrop	the mouse is released at the end of a drag operation. Discarding this event prevents the evDrop message from being generated	pEventCode, pDragType, pDragValue, pDropField

Scroll Events

The following event messages can occur for a field or window provided they have a vertical or horizontal scroll bar as appropriate.

Event Code	Generated when...	Event Parameters
evHScrolled	the field or window is scrolled horizontally	pEventCode
evVScrolled	the field or window is scrolled vertically	pEventCode

Status Events

The following event messages are reported for fields only. They reflect the current status of a field, and are generated only if the $statusevents property is enabled. Discarding any of these events has no effect since they report the status of a field.

Event Code	Generated when...	Event Parameters
evDisabled	a field is disabled either by notation or by OMNIS	pEventCode
evEnabled	a field is enabled either by notation or by OMNIS	pEventCode
evHidden	a field is hidden either by notation or by OMNIS	pEventCode
evShown	a field is made visible either by notation or by OMNIS	pEventCode

Tab Pane and Tab Strip Events

A tab pane or tab strip can have a number of tabs. The following event message is generated when the user selects one of the tabs.

Event Code	Generated when...	Event Parameters
evTabSelected	a tab has been selected; the second event parameter is the number of the tab selected	pEventCode, pTabNumber

Tree List Events

A tree list object can have a number expandable and collapsable nodes which the user clicks on. The following event messages are generated when the user expands or collapses a node, or clicks on a node, or edits a node name.

Event Code	Generated when...	Event Parameters
evTreeCollapse	a node is about to be collapsed; the second event parameter is a reference to the node	pEventCode, pNodeItem
evTreeExpand	a node is about to be expanded; the second event parameter is a reference to the node	pEventCode, pNodeItem
evTreeExpandCollapseFinished	a node has expanded or collapsed; sent after an evTreeCollapse or evTreeExpand message	pEventCode
evTreeNodeIconClicked	a node has been clicked; the second parameter is a reference to the node	pEventCode, pNodeItem
evTreeNodeNameFinished	a node name change has finished; the second parameter is a reference to the node; the third parameter contains the new text	pEventCode, pNodeItem, pNewText
evTreeNodeNameFinishing	a node name is about to change; the second parameter is a reference to the node; the third parameter contains the new text	pEventCode, pNodeItem, pNewText

Window Events

The following event messages are sent to the current top window ($cwind). Discarding the majority of these events has no effect since the event has already ocurred. For example, you can detect an evMinimized for a window, but discarding it has no effect since the user will have already minimized the window.

Event Code	Generated when...	Event Parameters
evCancel	the user has clicked the Cancel button or equivalent keys; $ctarget gets evCancel then $cwind (evAfter is not reported with evCancel). Discarding this event prevents enter data being terminated	pEventCode
evClose	the top window is closed, sent to the window just after $canclose message has been sent (so evClose is an alternative to $canclose); $ctarget gets an evAfter (if it's on the window being closed) then $cwind gets evClose. Discarding this event prevents the window from closing, but forcing OMNIS to quit closes everything	pEventCode
evCloseBox	the user has clicked the close box of the top window; $ctarget gets evAfter then $cwind gets evCloseBox. Discarding this event stops the window closing	pEventCode
evCustomMenu	the user has selected a line in a custom menu; $ctarget gets evAfter then $cwind gets evClick; the second event parameter is the line number of the option selected. Discarding this event prevents the method for the selected line from being executed	pEventCode, pMenuLine
evMaximized	the window is maximized	pEventCode
evMinimized	the window is minimized	pEventCode
evMoved	the window is moved	pEventCode
evOK	the user has clicked the OK button or has pressed the equivalent key; $ctarget gets evAfter then $cwind gets evOK. Discarding this event prevents enter data being terminated	pEventCode

Event Code	Generated when...	Event Parameters
evOpenContextMenu	a context menu has been opened over the field, also reported for windows; the second event parameter is an item reference to the context menu instance	pEventCode, pContextMenu
evResized	the window is resized	pEventCode
evRestored	the window is restored to its normal size	pEventCode
evStandardMenu	the user has selected a line in a standard menu or clicks one of the standard buttons (Next, Edit, etc.); $ctarget gets evAfter then $cwind gets evClick; the second event parameter is an internal number representing the menu option selected. Discarding this event prevents the standard menu action from being performed	pEventCode, pCommandNumber
evToTop	the window has come to the top. Discarding this event has no effect since the event has already occurred	pEventCode
evWindowClick	the mouse is clicked on another window; $ctarget gets evAfter, then $cwind gets evWindowClick, then the new window is brought to the top and (if it keeps bring to front clicks) may get an evMouseDown; the second event parameter is a reference to the window clicked on. Discarding this event prevents the clicked window from coming to the top	pEventCode, pClickedWindow

Chapter 4—Methods

Every object in OMNIS has certain characteristics that determine exactly how it looks and behaves. These characteristics are defined by the object's *properties* and *methods.* Properties are things like color, size, type, and visibility, while methods are pieces of code contained in the object that perform some action when you send the object the appropriate message. You can manipulate the properties and methods of an object or OMNIS itself using the *notation*, OMNIS' own hierarchical programming language. Before using the OMNIS notation you should read the *Programming Methods* chapter in the *OMNIS Programming* manual.

Object properties are not listed in this chapter since the vast majority of them are self-explanatory. You can view the properties of any object using the Property Manager, and the notation as a whole using the Notation Inspector. You can turn on Help Tips or short descriptions for the properties or methods by Right-clicking in the Property Manager and Notation Inspector. Using the same context menu in the Property Manager or Notation Inspector you can show Runtime properties and Methods for the current object.

Graph objects, properties, and methods are described in the *OMNIS Graphs* manual available in PDF on the OMNIS CD.

$canomit() and $canassign()

Notation strings are often long, but you can shorten them by omitting certain intermediate objects. The $canomit() method for the intermediate object returns true if you can leave the property out of an expression. In practice however $canomit() is true for the following objects only: $root, $extobjects, $constants, $clib, $hashvars, $libs, $tvars, $datas, $cvars, $files, $lvars, $vals. Therefore in the following expression

```
Do $root.$clib.$windows.MyWindow.$closebox.$assign(kTrue)
```

you can omit $root and $clib leaving

```
Do $windows.MyWindow.$closebox.$assign(kTrue)
```

In addition, the $canassign() method tells you whether you can use the $assign() method to assign a value to an object. For example

```
Do $cclass.$forecolor.$canassign() Returns #F
```

returns true if $cclass is a window class, which means you can assign a value to the forecolor, otherwise for some other classes this example would return false.

Common

Notation   [notation.]OBJECT.**method**

All objects have common properties, such as $name, and some objects have the following methods.

Method	Description
$assign()	NOTATION.PROPERTY.$assign(value) assigns the specified value to the property; the value and syntax depends on the object you are assigning to
$att()	NOTATION.OBJECT.$att(n) returns the nth attribute for the object (*does not* include custom methods)
$canassign()	NOTATION.PROPERTY.$canassign() returns true if the $assign() method is implemented for the property, that is, you can change the value of the property; $canassign() is true for most properties, but depending on the current context it may be false for certain properties
$canomit()	NOTATION.PROPERTY.$canomit() returns true if you can omit the property from the notation; the properties with $canomit set to true are: $root, $clib, $libs, $datas, $constants, $hashvars, $tvars, $cvars, $lvars, $vals, and $files
$chain()	$chain(n) returns the nth object in the reference chain for a reference variable

$root

Notation $root.**method**

The $root object is the object at the base of the OMNIS object tree. It has the following methods.

Method	Description
$redraw()	$redraw(bSetcontents=kTrue,bRefresh=kFalse) redraws the contents and/or refreshes *all* window instances in the context of $root; note window instances and window objects also contain the $redraw() method and redraw the object depending on the current context
$exechelp()	$exechelp([cInstName], [cWindowTitle], [cHelpFolder], [cDocumentName], [cTopic]) opens the OMNIS help system, where cInstName specifies the optional instance name of the help window; cWindowTitle specifies the optional window title; cHelpFolder specifies a help folder name, overriding the folder named in $helpfoldername; cDocumentName specifies the name and partial path of the help topic to be displayed, if empty help searches on the topic specified in cTopic; cTopic specifies the title or beginning of a topic title. If cDocumentName is empty and a topic title is specified, help attempts to locate the topic. If no topic is found, the help window enters the given text into the word search entry field and displays any topics found. If both cDocumentName and cTopic are empty, the contents list is displayed

Group

Notation   [notation.]ANYGROUP.**Method**

A group is a special type of object that contains a number of related objects. Some groups are static, while others change dynamically at runtime, such as the $iwindows group which contains all the window instances currently open. The group methods perform some action on the group as a whole, or return some information about the group or an individual object in the group.

Method	Description
$count()	$count() returns the number of objects in a group, for example, Do $components.$count() Returns lv_xcompnum returns the number of external components currently available in your system
$makelist()	$makelist(col1[,col2]…) lists the members of a group, for example, Do $components.$makelist($ref.$name) Returns lv_xcomplist builds a list of the external components currently available in your system, or Do $iwindows.$makelist($ref.$name) Returns lv_winlist builds a list of window instances currently open in the order that they appear on the screen
$appendlist()	$appendlist(list,col1[,col2]…) appends the members or contents of a group to the specified list
$insertlist()	$insertlist(list,line,col1[,col2]…) inserts the members or contents of a group into the specified list at the specified line
$sendall()	$sendall(message[,condition]) sends a message to all objects in a group; for example, $cwind.$objs.$sendall($ref.$visible.$assign(kFalse)) hides all the fields on the current window; $sendall() returns the number of objects which received the message; the return value of the message sent to an individual object is discarded
$first()	$first() returns a reference to the first object in a group, for example, Do $iwindows.$first() Returns lv_topwin returns a reference to the top window
$next()	$next(object) returns the next object in a group
$findname()	$findname(object) returns a reference to the specified object in a group
$findident()	$findident(ident) returns the object with the specified unique numeric identifier from within a group. Not applicable for all groups

Method	Description
$add()	$add(param1[,param2]…) inserts a new object into a group and returns an item reference to the new object created; you can use $add() to create libraries, classes, window objects, list columns, and so on; its parameters depend on the group and object being created; usually the Property Manager indicates the parameters required, or you can use the notation [Notation.]ANYGROUP.$add.$desc to return the parameters. For example, you can create a new window class in the current library using Do $clib.$windows.$add('MyWin') Returns iWinRef; this creates a window class called MyWin and returns a reference to it in the variable iWinRef. Having created the new class you can add objects to it using $add(), such as Do iWinRef.$objs.$add(kEntry,20,20,25,200) Returns iObjRef; this creates an entry field and returns a reference to it in the variable iObjRef; note you specify the size for window objects in pixels, and report object in inches or cms depending on the current units. You can also create new objects in the current window instance using $cinst, such as Do $cinst.$objs.$add(objtype[,params…]). For example, you can create an external component using Do $cinst.objs.$add(kComponent,'OmnisIcn Library','OmnisIcn Control',lvTop,lvLeft,lvHeight,lvWidth) Returns lvSubRef; this creates an OMNIS icon control with the size and position specified in lvTop, lvLeft, lvHeight, lvWidth.
$remove()	$remove(object) removes the object from a group (it does not delete the object from the disk)
$addafter()	$addafter(object) inserts a new object after the specified object. Not applicable for all groups, e.g. doesn't work for $cols in a list
$addbefore()	$addbefore(object) inserts a new object before the specified object. Not applicable to all groups, e.g. doesn't work for $cols in a list

OMNIS Modes

Notation   [$root.]$modes.**Method**

The OMNIS modes control the overall behavior or mode of your system. You can view the OMNIS modes group in the Notation Inspector under $root. The $modes group contains the following methods.

Method	Description
$welcome()	$welcome(library-mode) controls the opening and running of the Welcome library, a constant: kWelcomeNewLibrary, kWelcomeLastLibrary, kWelcomeToggleStop
$dotoolmethod()	$dotoolmethod(tool-constant,method-name[,parameters]) executes a public method within one of the OMNIS tools; tool-constant can be: kEnvToolAdhoc, kEnvToolCms, kEnvToolMethods, kEnvToolSql, kEnvToolVcs

OMNIS Preferences

The $root.$prefs group contains the following method(s).

Method	Description
$serialize()	$serialize([bGenericLogo=kTrue,cTitle,iBitmapID]) opens the OMNIS serialization dialog; you can pass your own title and bitmap, otherwise if bGenericLogo is kFalse (the default) the OMNIS logo is displayed, or if kTrue a generic Serialize logo is displayed; if you pass a title it replaces the one in the dialog window title

Printing Devices

The following methods are available for some printing devices only. $cando() returns true if the device supports the method.

Method	Description
$open()	opens the device ready for printing or transmitting text or data
$close()	closes the device if the device is open
$canclose()	returns true if the device can be closed; if the device was opened by calling $open(), it usually returns true, but if the device was opened via a print job, and the job is still in progress it returns false
$sendtext()	$sendtext(cText, bNewLine, bFormFeed) sends the text in cText to the device. All normal character conversion takes place. If bNewLine is true, the device will advance to a new line or an end of line character is sent. If bFormFeed is true, a new page is started or a form feed character is sent. Data is sent in the same order as the parameters
$senddata()	$senddata(xData[,xData1]…) sends cData in binary format to the device. No character conversion takes place unless the data is of type kCharacter. If you specify more than one parameter the data is sent in individual packets
$flush()	flushes the device. In the case of the File device, you can call $flush() to make sure all data is written to disk. $cando() returns true for all devices that support either $senddata() or $sendtext()
$getparam()	$getparam(nParamNumber) returns the value of the specified parameter for a custom device
$setparam()	$setparam(nParamNumber,Value[,nParamNumber,Value]...) sets the value(s) of the specified parameter(s) for a custom device

Window Class

Notation   [$root.$libs.LIBRARYNAME.]$windows.WINDOWNAME.**Method**

All classes that you can instantiate and those that support inheritance, such as window classes, contain the following methods. For example, the $open() method opens the specified window class.

```
Do $windows.WindowName.$open('InstanceName',kWindowCenter) Returns
  WinRef
; returns an item reference to the instance created
```

Method	Description
$makesubclass()	Classname.$makesubclass([cLibraryname.]cNewclassname) creates a new subclass of the class in the current or specified library
$isa()	Class|instancename.$isa(rClass) returns true if the class or instance *is a subclass of* the specified class
$open()	Classname.$open([cInstancename][,iLocation][,parameters]) creates an instance of the specified window class and returns a reference to the instance; you can specify the instance name, or $open('*') will assign a unique instance name in the form ClassName_Number, otherwise the simple class name is used; you can send parameters to the $construct() method of the instance, and for window classes you can specify the initial location or position of the window instance
$openonce()	Classname.$openonce([cInstancename][,iLocation][,parameters]) creates an instance of the specified window class, but only if one does not already exist, excluding subwindows; in the case of a window, this method brings the window instance to the top if it already exists; $openonce() returns a reference to the instance either newly created or already open, just like the $open() method

Menu Class

Notation   [$root.$libs.LIBRARYNAME.]$menus.MENUNAME.**Method**

All classes that you can instantiate and those that support inheritance, such as menu classes, contain the following methods. For example, the $open() method opens or installs the specified menu class.

```
Do $menus.MenuName.$open('InstanceName') Returns WinRef
; returns an item reference to the instance created
```

Method	Description
$makesubclass()	Classname.$makesubclass([cLibraryname.]cNewclassname) creates a new subclass of the class in the current or specified library
$isa()	Class|instancename.$isa(rClass) returns true if the class or instance *is a subclass of* the specified class
$open()	Classname.$open([cInstancename][,iPosition][,parameters]) creates an instance of the specified menu class and returns a reference to the instance; you can specify the instance name, or $open('*') will assign a unique instance name in the form ClassName_Number, otherwise the simple class name is used; you can send parameters to the $construct() method of the instance, and for menu classes you can specify the initial position of the menu instance on the menubar, the default position is the right-most position on the main menu bar
$openonce()	Classname.$openonce([cInstancename][,iLocation][,parameters]) creates an instance of the specified menu class, but only if one does not already exist, excluding instances that are submenus; $openonce() returns a reference to the instance either newly created or already open, just like the $open() method

Toolbar Class

Notation   [$root.$libs.LIBRARYNAME.]$toolbars.TOOLBARNAME.**Method**

All classes that you can instantiate and those that support inheritance, such as toolbar classes, contain the following methods. For example, the $open() method opens or installs the specified toolbar class.

```
Do $toolbars.ToolbarName.$open('InstanceName') Returns WinRef
; returns an item reference to the instance created
```

Method	Description
$makesubclass()	Classname.$makesubclass([cLibraryname.]cNewclassname) creates a new subclass of the class in the current or specified library
$isa()	Class|instancename.$isa(rClass) returns true if the class or instance *is a subclass of* the specified class
$open()	Classname.$open([cInstancename][,DockingArea][,parameters]) creates an instance of the specified toolbar class and returns a reference to the instance; you can specify the instance name, or $open('*') will assign a unique instance name in the form ClassName_Number, otherwise the simple class name is used; you can send parameters to the $construct() method of the instance, and for toolbar classes you can specify the initial docking area for the toolbar instance, the default being the top toolbar
$openonce()	Classname.$openonce([cInstancename][,DockingArea][,parameters]) creates an instance of the specified toolbar class, but only if one does not already exist, excluding window toolbars; $openonce() returns a reference to the instance either newly created or already open, just like the $open() method

Report Class

Notation   [$root.$libs.LIBRARYNAME.]$reports.REPORTNAME.**Method**

All classes that you can instantiate and those that support inheritance, such as report classes, contain the following methods. For example, the $open() method instantiates the specified report class.

```
Do $reports.ReportName.$open('InstanceName') Returns WinRef
; returns an item reference to the instance created
```

Method	Description
$makesubclass()	Classname.$makesubclass([cLibraryname.]cNewclassname) creates a new subclass of the class in the current or specified library
$isa()	Class|instancename.$isa(rClass) returns true if the class or instance *is a subclass of* the specified class
$open()	Classname.$open([cInstancename][,parameters]) creates an instance of the specified report class and returns a reference to the instance; you can specify the instance name, or $open('*') will assign a unique instance name in the form ClassName_Number, otherwise the simple class name is used; you can send parameters to the $construct() method of the instance
$openonce()	Classname.$openonce([cInstancename][,parameters]) creates an instance of the specified report class, but only if one does not already exist, and returns a reference to the instance either newly created or already open, just like the $open() method

Task Class

Notation   [$root.$libs.LIBRARYNAME.]$tasks.TASKNAME.**Method**

All classes that you can instantiate and those that support inheritance, such as task classes, contain the following methods. For example, the $open() method opens the specified task class.

```
Do $tasks.TaskName.$open('InstanceName') Returns WinRef
; returns an item reference to the instance created
```

Method	Description
$makesubclass()	Classname.$makesubclass([cLibraryname.]cNewclassname) creates a new subclass of the class in the current or specified library
$isa()	Class|instancename.$isa(rClass) returns true if the class or instance *is a subclass of* the specified class
$open()	Classname.$open([cInstancename][,parameters]) creates an instance of the specified task class and returns a reference to the instance; you can specify the instance name, or $open('*') will assign a unique instance name in the form ClassName_Number, otherwise the simple class name is used; you can send parameters to the $construct() method of the instance
$openonce()	Classname.$openonce([cInstancename][,parameters]) creates an instance of the specified task class, but only if one does not already exist, and returns a reference to the instance either newly created or already open, just like the $open() method

Table Class

Classes that support inheritance, such as table classes, contain the following methods. Note that table classes do not contain the $open() or $openonce() methods, rather a table instance is created automatically when you create a list based on a schema, query, or table class.

Method	Description
$makesubclass()	Classname.$makesubclass([cLibraryname.]cNewclassname) creates a new subclass of the table class in the current or specified library
$isa()	Class|instancename.$isa(rClass) returns true if the class or instance is a subclass of the specified table class

Object Class

Object classes let you define your own structured data objects containing your own
variables and methods. All classes that you can instantiate and those that support
inheritance, such as object classes, contain the following methods.

Method	Description
$makesubclass()	Classname.$makesubclass([cLibraryname.]cNewclassname) creates a new subclass of the class in the current or specified library
$isa()	Class|instancename.$isa(rClass) returns true if the class or instance is a subclass of the specified class

$new()	$new(parm1[,parm2]..) creates an object instance dynamically; the parameters are passed to the object instance's $construct() method; when the new instance is assigned any existing instances of the class are replaced; for example
	
	``Do
	  $clib.$objects.objectclass.$new(parm1,parm2,...)
	  Returns objectvar```
	
	where parameters parm1 and parm2 are the $construct() parameters for the object instance

List Variable

Notation   [notation.]LISTNAME.**Method**

A list variable contains multiple values of fields and variables. OMNIS lets you define and build as many lists of data as memory allows. A list defined from a schema, query, or table class has the properties and methods of a table instance.

A list variable with the smart list property enabled contains two lists: the *normal list*, containing the list data, and the *history list* containing the change tracking and filtering information. The history list has one row for each row in the normal list, together with a row for each row that has been deleted. Defining a list by any mechanism, or adding columns to a list, discards the history list, and turns off change tracking.

You can use the following methods against any type of OMNIS list. The history list $savelist.. and $revertlist.. methods are only available for smart lists. Note also that list variables have some group methods, such as $add() and $remove(); these are listed below and are described in the context of manipulating lists.

In addition, row variables behave in exactly the same way as list variables, except that some methods do not apply to row variables since they have only one line.

Method	Description
$define()	$define(var1[,var2]...) clears the current list definition and defines the list with the specified variables or fields as columns; you can use fields in a no-data file, but they must be quoted in the parameter list
$definefromsqlclass()	$definefromsqlclass(sqlclass[,cSchemaCol1,cSchemaCol2,...] [,,constructor params]) defines a list or row variable from a query, schema, or table class and instantiates a table instance; for lists based on a schema class, or a table class referencing a schema class, all columns are used to define the list unless you pass a list of schema columns as a subset of those in the schema class; the constructor parameters are passed to the $construct() method of the table instance (note the empty parameter before the constructor params)
$copydefinition()	$copydefinition(list or row variable[,constructor params) clears the list and copies the definition but not the data from another list or row variable; if the source list has a file class the new list will also have a file class; the parameters are passed to the $construct method
$redefine()	$redefine(var1[,var2]…) redefines the names and data types of the list columns, but does not change or delete existing data in the list

Method	Description
$clear()	$clear() clears the list data; the list definition is unchanged
$search()	$search(calculation[,bFromStart=kTrue, bOnlySelected=kFalse, bSelectMatches=kTrue, bDeselectNonMatches=kTrue]) searches a list using the specified calculation; this method has the same function as the Search list command. The search calculation can use $ref.colname or list_name.colname to refer to a list column. With bSelectMatches or bDeselectNonMatches the first line number whose selection state is changed is returned (or 0 if no selection states are changed), otherwise the first line number which matches the selection is returned (or 0 if no line is found). This method does not change any CRB values, the current row is changed if neither bSelectMatches or bDeselectNonMatches is used
$sort()	$sort(column1,bDescending=kFalse[,column2, bDescending=kFalse]...) sorts the list on the specified columns sort fields; you can specify up to 9 columns including the bDescending flag for each (which defaults to kFalse meaning cols are sorted ascending). The columns can be column names or calculations using $ref.colname or list_name.colname. For calculated sorts, the calculation is evaluated for line 1 of the list to determine the comparison type (Character, Number or Date)
$merge()	$merge(list or row,bByName,bSelectedOnly) merges the two lists; if you specify a row it is treated as a single row list. If you specify bByName, columns are matched by name rather than by number; if you specify bOnlySelected only selected lines in the source list are merged
$savelistdeletes()	$savelistdeletes() removes all kRowDeleted rows from the history list, and also from the normal list if $rowpresent is kTrue
$savelistinserts()	$savelistinserts() changes all kRowInserted rows to kRowUnchanged, and sets the old contents of those rows to the current contents. It does not change $rowpresent
$savelistupdates()	$savelistupdates() changes all kRowUpdated rows to kRowUnchanged and, for all rows, sets the old contents to the current contents; this does not change $rowpresent
$savelistwork()	$savelistwork() executes the $savelist... methods
$revertlistdeletes()	$revertlistdeletes() changes all kRowDeleted rows to kRowUnchanged or kRowUpdated (depending on whether the contents have been changed); for these rows $rowpresent is set to kTrue

Method	Description
$revertlistinserts()	$revertlistinserts() removes any inserted rows from both the normal and history list
$revertlistupdates()	$revertlistupdates() changes all kRowUpdated rows to kRowUnchanged and, for all rows, the current contents are set to the old contents; this does not change $rowpresent
$revertlistwork()	$revertlistwork() quick and easy way to execute the $revertlist... methods
$includelines()	$includelines(row status) includes rows of a given status, represented by the sum of the status values of the rows to be included. Thus 0 means no rows, kRowUnchanged + kRowDeleted means unchanged and deleted rows, and kRowAll means all rows, irrespective of status
$filter()	$filter(search-calculation) applies a filter to a smart list; this method restricts the list to only those rows which match the search calculation; for example, Do LIST.$filter (COL1 = '10') will only display lines where COL1 is 10
$unfilter()	$unfilter(level) removes a filter or filters from a smart list
$refilter()	$refilter() reapplies all current filters to a smart list
$remove()	$remove(rLine|iLineNumber|kListDeleteSelected|kListKeepSelected) deletes the specified line or lines from the list; note you can specify a reference to a list line, a line number, or you can remove all selected or non-selected lines; if you specify 0, the current line is removed (kTrue is returned if successful)
$first()	$first(bSelectedOnly=kFalse, bBackwards=kFalse) sets the current row of the list to the first row or first selected row and returns a reference to that row; if there are no further rows the current row is set to zero
$next()	$next(list row or row number, bSelectedOnly=kFalse, bBackwards=kFalse) sets the current row of the list to the next row or next selected row and returns a reference to that row; if there are no further rows the current row is set to zero and nothing is returned. If you specify 0 for the list row it is taken as the current row
$add()	$add(col1value[,col2value]...) inserts a row at the end of the list with the specified column values. If you use $add() without parameters any columns which correspond to CRB fields are loaded with the current CRB values
$addbefore()	$addbefore(list row or row number,col1value[,col2value]...)

Method	Description
	inserts a row before the specified row with the specified column values.. If you specify 0, $row is used
$addafter()	$addafter(list row or row number,col1value[,col2value]...) inserts a row after the specified row with the specified column values. If you specify 0, $row is used

List Column

Notation   [notation.]LISTNAME.$cols.COLNUMBER.**Method**

The columns of a list are contained in the $cols group. You can access a column using its column number.

Method	Description
$clear()	clears the data for the column for every row in the list, the column definition is left unchanged, for example, LIST.$cols.col1.$clear() clears col1
$removeduplicates()	$removeduplicates(bSortnow,bIgnorecase) removes lines with duplicate values in the column. If bSortnow is true the list is sorted on that column, otherwise you must sort the list using $sort() before applying this method. If bIgnorecase is true the case of character values is ignored when making the comparison. The number of rows removed is returned
$count()	returns the number of rows for the column, including rows that have empty or null values
$total()	returns the total for a column containing numeric data
$average()	returns the average for a column containing numeric data
$minimum()	returns the minimum value in a column containing numeric data
$maximum()	returns the maximum value in a column containing numeric data

List Row

Method	Description
$assigncols()	$assigncols(col1value[,col2value]...) replaces the column values for the row with the specified values; if you use $assigncols() the values of any columns which correspond to CRB fields are loaded with the current CRB values
$assignrow()	$assignrow(row, by name) assigns the column values from the row specified by the first parameter into the row on a column by column basis; if you specify 'by name' the columns are matched by name, otherwise by column number
$clear()	$clear() clears the value of all the columns for the row
$loadcols()	$loadcols(variable1[,variable2]...) loads the column values for the row into the specified variables; if you use $loadcols() the values of any columns which correspond to CRB fields are loaded into the CRB fields

External Components

Notation   [$root.]$components.LIBNAME.**Method**

You can access automation objects using the $cmd() method via the $components group. If an error occurs, for example the construction of an automation object fails, #ERRCODE and #ERRTEXT can be inspected to determine the error.

Method	Description
$cmd()	$cmd(parm1[,parm2]...) issues a component-specific command to the component; it lets you interact with components such as the Automation component, without needing a component object in a window or report instance; not all components support this method.
	For example, you can control the JavaBean component using $cmd() and one or more parameters, such as
	`Do $components.JavaBean.$cmd("GetPaths", List)`
	populates the specified list with the Java Bean search paths.
	Using the Automation component and the $cmd() method you can launch a browser window, for example

	`Set reference iRef to`
	`    $components.Automation Library`
	`Do iRef.$cmd("$createobject",`
	`    "InternetExplorer.Application.1")`
	`    Returns pDISPapp`
	`Do iRef.$cmd(pDISPapp,"Navigate()","www.MyWebSite")`
	`Do iRef.$cmd(pDISPapp,"Visible") Returns #1`
	`If #1=0`
	`    Do iRef.$cmd(pDISPapp,"Visible",kTrue)`
	`End If`

	This method sets an item reference to the automation component, constructs an instance of InternetExplorer and returns a unique descriptor to the new object; the descriptor is a character string of 15 chars and a unique pointer to an Automation dispatch interface, which you pass to other calls related to the object; the method then invokes the Navigate() method and gets the current value of the Visible property; if the object is not visible, the method sets the visible property to true

Method Lines

You can manipulate method lines using the following method(s).

Method	Description
$modify()	$modify([iLine=1]) opens the method editor at the specified line in the method; the default is line 1; this method is not available in the runtime version of OMNIS

Instance

Notation   [notation.]INSTANCENAME.**Method**

All class instances, except table instances, have the following methods.

Method	Description
$canclose()	$canclose(bIsquit) is sent to an instance just before any action which may cause it to be destructed; bIsquit is passed as kTrue when the $canclose() message is sent as a result of a Quit OMNIS event
$close()	$close(instancename) closes or destructs the instance (if it can be closed) and returns true if the instance is closed. $close() calls $canclose() and if it returns true the instance is closed

Report Instance

Notation   [$root.]$ireports.REPORTINST.**Method**

Report instances have the following methods.

Method	Description
$openjobsetup()	$openjobsetup() opens the job setup dialog, and can be called immediately after $open() for a report; if it returns kFalse (the user has canceled), the report instance should be closed without printing any data
$printrecord()	$printrecord() is sent to the report instance by the *Print record* command. The default handler prints the record section
$printtotals()	$printtotals(section) is sent to the report instance when a subtotal break has been triggered or the report is about to be terminated; section is the highest level subtotal to be printed (a constant such as kSubtotal5 or kTotals), if section is not a subtotal or totals section only the subtotal header sections are printed. The default handler prints the correct subtotal sections followed by the corresponding subtotal header sections
$printsection()	$printsection(section) is sent when a section is printed; section is one of the constants (kRecord, kTotals, etc.) or a reference to a section field on the report instance. The default handler prints the section positioned according to $sectionstart, $sectionend and the positioning mode for the section . For a subtotal or total section the current field values are temporarily reset to those which were current when $printsection for a detail section was previously called
$accumulate()	$accumulate(kSection) accumulates the subtotals and totals; it is sent to the report instance during the printing of a record section and the current field values into the most rapidly changing subtotals. $accumulate(section) is sent from $printtotals(section) to accumulate the current level subtotals into the next level of subtotals; you can use $accumulate() instead of $accumulate(kSection)
$checkbreak()	checks if a subtotal break is required by comparing the current field values with those when it was last called; returns a constant: kSubtotal1 to kSubtotal9 or kNone if no subtotal break is required
$skipsection()	causes any further processing of the current section to be skipped; if you call this during $print() for a field, no further fields will be printed for that section, so positioning sections count as new sections and are not skipped if the previous section was skipped

Method	Description
$startpage()	$startpage(page number) is sent to a report instance when another page is started. The default handler adds the page header section to the page: for the first page, the default handler also adds the report header section to the page. Calls to $startpage() for a large number of pages will result in large memory use. Starting a page always ends the previous page (only one page is started and not ended)
$endpage()	$endpage(page number) is sent to a report instance just before a page is ended (the next page is to be started or the page is about to be ejected). The default handler adds the footer section to the page. Calls to $endpage() for a large number of pages will result in large memory use. $endpage() without a parameter ends all pages which have been started
$ejectpage()	$ejectpage(page number) is sent to a report instance just before a page is ejected. You cannot add to a page once it is ejected; the default handler ejects the page; pages are ejected in order so ejecting a page also ejects all earlier pages. Calling $ejectpage(page number) will start and end all pages before they are ejected. $ejectpage() without a parameter ejects all pages which have been ended and not ejected
$endprint()	$endprint() is sent to the report instance by the *End print* command and in other circumstances when the report is terminated. The default handler prints the final subtotals and totals sections and ejects all the remaining pages

Report Instance Object

Notation   [$root.]$ireports.REPORTINST.$objs.REPORTOBJ.**Method**

A field or object in a report instance has the following methods.

Method	Description
$print()	$print(position,value) is sent to the field or section when it is to be printed. If you specify *value* this data is printed, otherwise the normal field value is printed (when the default processing calls $print() the value parameter is set up): *position* is the starting position for the field or section, if no position is specified the field is printed at $sectionend. When $print() is called for a section the position has already taken account of the positioning mode of the section, $sectionstart and $sectionend

Table Instance

Notation   [notation.]SQLLIST.**Method**

A table instance is created when you create a list or row variable based on a schema, query, or table class. The following methods let you populate and change a list based on a sql class, and apply changes to the data on the server. Note that some of these methods execute SQL in the context of the current OMNIS session.

Method	Description
$select()	$select([cText,...]) issues a SELECT statement to the server; it can take one or more arguments, either literals or variable values which are concatenated into one text string and appended to the SELECT
$selectdisticnt()	$selectdistinct([cText,...]) issues a SELECT DISTINCT statement to the server; it can take one or more arguments, either literals or variable values which are concatenated into one text string and appended to the select statement
$fetch()	$fetch(iFetchcap[,bAppend=kFalse]) fetches the next iFetchcap number of rows from the server; if bAppend is kTrue the fetched data is appended to the list, otherwise if kFalse or omitted the list is cleared before the fetch; parameters do not apply for row variables, since the current data in the row is always replaced
$sqlerror()	$sqlerror(iErrortype,iErrorcode,cErrortext) called when an error occurs while executing $select(), $fetch(), $update(), $delete(), $insert(), or $do... methods; performs default processing for the error unless you override $sqlerror() with your own method to handle SQL errors
$createnames()	$createnames() returns a text string suitable for using with a CREATE TABLE statement
$selectnames()	$selectnames() returns a text string, containing a comma separated list of column names, suitable for using with a SELECT statement
$insertnames()	$insertnames([cRowName]) returns a text string suitable for using with an INSERT statement, in the form (col1,...,colN) VALUES (@[cRowName.col1],...,@[cRowName.colN]) where col1...colN are the names of the columns in the row variable; if cRowName is omitted $cinst is used in bind variables

Method	Description
$updatenames()	$updatenames([cOldrowName][,cRowName]) returns text suitable for using with an UPDATE statement, in the form SET col1=@[cRowName.col1], ...,colN=@[cRowName.colN] where col1...colN are the names of the columns in the row variable; if cRowName is omitted $cinst is used in bind variables
$wherenames()	$wherenames([cOperator][,cRowName]) returns text suitable for using as a WHERE clause, in the form WHERE col1=@[cRowName.col1] AND ,..., AND colN=@[cRowName.colN] where col1...colN are the names of the columns in the row variable; if cRowName is omitted $cinst is used in bind variables
$doinserts()	$doinserts() inserts list rows with status kRowInserted into the server database
$doupdates()	$doupdates([bDisableWhere=kFalse]) updates list rows with status kRowUpdated in the server database; when bDisableWhere is true it prevents $doupdates from appending a WHERE clause to the UPDATE statement
$dodeletes()	$dodeletes([bDisableWhere=kFalse]) deletes list rows with status kRowDeleted from the server database; when bDisableWhere is true it prevents $dodeletes from appending a WHERE clause to the DELETE statement
$dowork()	$dowork([bDisableWhere=kFalse]) executes the $dodeletes(), $doupdates(), $doinserts() methods in that order; it passes the value of bDisableWhere to the $dodeletes() and $doupdates() methods
$undoinserts()	$undoinserts() removes any inserted rows from the list
$undoupdates()	$undoupdates() restores any updated rows to their original value, and resets their status to kRowUnchanged
$undodeletes()	$undodeletes() restores any deleted rows to the list, and resets their status to kRowUnchanged
$undowork()	$undowork() executes the three $undo... methods in the order insert, update, delete, that is, the reverse order to the $dowork method
$doinsert()	$doinsert(wRow) inserts a row into the server database; it is called by $doinserts() for each row to be inserted
$doupdate()	$doupdate(wRow,wOldrow) updates a row in the server database; it is called by $doupdates() for each row to be updated

Method	Description
$dodelete()	$dodelete(wRow) deletes a row from the server database; it is called by $dodeletes() for each row to be deleted
$insert()	$insert() inserts a row into the server database
$update()	$update(wOldrow[,bDisableWhere=kFalse]) updates a row in the server database; when bDisableWhere is true it prevents $update from appending a WHERE clause to the UPDATE statement. You would typically use this when $extraquerytext is not empty. $update() appends $extraquerytext after the WHERE clause
$delete()	$delete([bDisableWhere=kFalse]) deletes a row from the server database; when bDisableWhere is true it prevents $delete from appending a WHERE clause to the DELETE statement. You would typically use this when $extraquerytext is not empty. $delete() appends $extraquerytext after the WHERE clause

Window Instance

Notation   [$root.]$iwindows.WINDOWINST.**Method**

A window instance contains the methods of an instance together with the following.

Method	Description
$bringtofront()	brings the window instance to the front, returns true if successful
$minimize()	minimizes the window instance, returns true if successful
$maximize()	maximizes the window instance, returns true if successful
$redraw()	$redraw(bSetcontents=kTrue,bRefresh=kFalse) redraws the contents and refreshes the window

Window Instance Object

Notation   [$root.]$iwindows.WINDOWINST.$objs.OBJNAME.**Method**

All window objects have the $redraw() method. Methods for specific field types are listed separately.

Method	Description
$redraw()	$redraw(bSetcontents=kTrue,bRefreshWindow=kFalse) resets the contents of the field and/or refreshes the window instance containing the field

Methods for Tree Lists

Method	Description
$getnodelist()	$getnodelist(Listmode,rNodeRef,lListname) returns the list data under the current node or for the entire tree; Listmode can be kRelationalList or kFlatList; rNoderef can be a reference to a node or NULL to retrieve the entire tree, lListname is the name of a list variable to receive the list data
$setnodelist()	$setnodelist(Listmode,rNodeRef,lListname) lets you populate the current node or whole tree with the data in lListname; Listmode can be kRelationalList or kFlatList; rNoderef can be a reference to a node or NULL to populate the whole tree
$currentnode()	returns an item reference to the current node in the tree
$count()	returns the number of nodes under the current node, or all root nodes in whole tree
$clearallnodes()	clears all nodes under the current node, or all the nodes in the entire tree
$findnodename()	$findnodename(rNodeRef,cName,bRecursive) returns a reference to a found node using the node $name property, or NULL if nothing is found: rNodeRef is the starting node, a NULL value searches the whole tree; cName is the name to search for; if bRecursive is kTrue, any child nodes are also searched
$findnodeident()	$findnodeident(rNodeRef,iIdent,bRecursive) returns a reference to a found node using the node $ident property, or NULL if nothing is found: rNodeRef is the starting node, a NULL value searches the whole tree; iIdent is the ident value to search for; if bRecursive is kTrue, any child nodes are also searched
$first()	returns a reference to the first root node
$add()	$add(cName[,iIdent]) adds a new root node or node after the specified iIdent

Method	Description
$remove()	$remove(rItem) deletes the specified child node
$setcurrentnode()	$setcurrentnode(rNodeRef) sets the current node to the node in rNodeRef
$nextnode()	$nextnode(rItem,bRecursive) returns the next node in the tree after the node in rItem, or the first root node if rItem is NULL; if bRecursive is kTrue the method steps into any child nodes
$prevnode()	$prevnode(rItem,bRecursive) returns the previous node in the tree before the node in rItem; if bRecursive is kTrue, the method steps back into node parents
$expand()	opens all child nodes under the current node, or all nodes in the entire tree list
$collapse()	closes all child nodes under the current node, or all nodes in the entire tree list
$getvisiblenode()	$getvisiblenode(iVisLine) returns a reference to the node for a visible line
$findname()	$findname(cName) returns a reference to the node named in cName
$findident()	$findident(iIdent) returns a reference to the node specified by iIdent
$edittext()	lets the user edit the text for the current node

Methods for Icon Arrays

Method	Description
$edittext()	$edittext() lets the user edit the text for the current icon

Methods for Headed List Boxes

Method	Description
$edittext()	$edittext(iColumnNumber) lets the user edit the cell for the current line of the specified column
$getcolumnalign()	$getcolumnalign(iColumnNumber) returns the alignment of the specified column
$setcolumnalign()	$setcolumnalign(iColumnNumber[,Alignment]) sets the alignment of the specified column; you can specify Alignment as kLeft, kRightJst, or kCenterJst, otherwise if it is omitted the method uses the current value of $columnalignmode for the field

Methods for Tab panes

Method	Description
$showpane()	$showpane(iPaneNumber,bShow=kTrue) shows the specified pane; if bShow is kFalse the pane is hidden
$ispaneshown()	$ispaneshown(iPaneNumber) returns true if the specified pane is visible
$enablepane()	$enablepane(iPaneNumber,bEnable=kTrue) enables the specified pane; if bEnable is kFalse the pane is disabled or grayed out and the user cannot select it
$ispaneenabled()	$ispaneenabled(iPaneNumber) returns true if the specified pane is enabled

Methods for Screen Report Fields

Method	Description
$redirect()	$redirect(bPrompt=kTrue) redirects the current report by prompting for a different print device, rather than the device specified in default preferences
$print()	prints the current report in the field
$printpage()	prints the current page of the report in the field
$zoom()	$zoom(bZoomOn=kTrue) sets the zoom mode when the screen report field is in page preview mode
$clear()	clears the field of the current report

Methods for Modify Report Fields

Method	Description
$sortfields()	opens the sort fields dialog for the current report
$pagesetup()	opens the page setup dialog for the current report

Chapter 5—Commands

This chapter describes the OMNIS commands, including the fifty or so external commands. In this chapter they are arranged in alphabetical order. The external commands are listed in a separate section at the end of this chapter. Each entry includes a short description of the command, its reversibility, its effect on the flag, and its parameters and syntax.

To learn how to use the commands you should read the *Using OMNIS Studio* manual. Also you should be familiar with the method editor and OMNIS debugger before using the commands.

About the Commands

There are over 500 OMNIS commands that provide a powerful interpreted programming language with which you can build client/server applications. With them you can monitor events in the client user interface, control SQL objects and transaction management, manipulate classes and data in your libraries and data files, and so on.

You can enter all the commands under all operating systems, but some commands only run under a particular OS indicated by the appropriate OS icon.

External Commands

External commands add functionality to OMNIS. They are implemented as Dynamic Link Libraries (DLLs) under Windows, or external code resources under MacOS. External command packages are placed in the EXTERNAL folder and appear in the *External commands...* group in the method editor. The external commands are described in the next chapter in this manual.

Commands

Accept advise requests

Reversible: YES **Flag affected:** NO

Parameters: ☐ Accept

Syntax: Accept advise requests [*(Accept)*]

DDE command, OMNIS as server. This command enables or disables responses to a request Advise message from a client. With the **Accept** check box selected, OMNIS will respond to an Advise request message specifying a valid field name by repeatedly sending the field value to the client at appropriate times. If the **Accept** option is unchecked, all conversations with Advises in force will be terminated unless the command is part of a reversible block.

```
Accept advise requests (Accept)      ;; Check the Accept option
```

Accept commands

Reversible: YES **Flag affected:** NO

Parameters: ☐ Accept

Syntax: Accept commands [*(Accept)*]

DDE command, OMNIS as server. This command determines whether OMNIS will accept commands from the client program. When *Accept commands* is in force, OMNIS will respond to a DDE EXECUTE message by attempting to execute a command string sent by the client program. All conversations are terminated when you close your OMNIS library.

```
Accept advise requests (Accept)
Accept commands (Accept)         ;; Check the Accept option
```

Accept field requests

Reversible: YES   **Flag affected:**   NO

Parameters: ☐ Accept

Syntax: Accept field requests [*(Accept)*]

DDE command, OMNIS as server. This command enables or disables responses to a request for field values issued by a client application. With the **Accept** option selected, OMNIS will respond to a Request message specifying a valid field name by sending the field value to the client program. Values are taken from the current record buffer. Values are only sent when OMNIS is in enter data mode or when no methods are running.

```
Accept advise requests (Accept)
Accept commands (Accept)
Accept field requests (Accept)        ;; Check the Accept option
```

Accept field values

Reversible: YES   **Flag affected:**   NO

Parameters: ☐ Accept

Syntax: Accept field values [*(Accept)*]

DDE command, OMNIS as server. This command determines whether OMNIS is able to receive data from a client via a DDE POKE message. With the **Accept** option selected, OMNIS will respond to a Poke message specifying a valid field or variable name, by setting the value of that field to the value transmitted by the client program. Values are stored in the current record buffer and, if the relevant field is on the top window, that window is redrawn.

Field values are only accepted when OMNIS is in enter data mode, Prompted find, or when no methods are running. All conversations are terminated when you close your OMNIS library.

```
Accept advise requests (Accept)
Accept field values (Accept)          ;; Check the Accept option
```

Add line to list

Reversible: NO   **Flag affected:**   YES

Parameters: Line number (default is end of list)
               List of values

Syntax: Add line to list [*{*[*line-number*] [(*value1*[,*value2*]...)]*}*]

This command adds a new line to the current list using the current field values in the CRB or values you specify in the list of values. Any conversions required between data types are carried out automatically. The flag is cleared if the line cannot be added, either because the maximum number of lines in the list or the memory limits have been exceeded.

You can specify the line number at which the new line is inserted, otherwise the line is added to the end of the list. If the line number you specify in the command line is empty or evaluates to zero, the new line is added to the end of the list.

You can specify a comma-separated list of values (enclosed in parentheses) to be added to the list. For example

Add line to list {$line ('abc',,VAR1+3)}

stores 'abc' into the first column of the current line of the current list, leaves the value of the second column empty, and loads the result of VAR1+3 into the third column. If too few values are specified, the other columns are left empty; if too many values are specified, the extra values are ignored. When you supply a comma-separated list of values, the values in the CRB are ignored.

The following example sets the current list to MYLIST, defines and builds the list and adds the values in S3 and LVAR1 at line 4 (note the first column of line 4 is left empty).

```
Set current list MYLIST
Define list {CODE,NAME,CREDIT}
Build list from file on CLIENTS
Calculate S3 as 'New string'
Calculate LVAR1 as 23
Add line to list {4(,S3,LVAR1)}
OK message (Icon) {New value in list is [lst(4,S3)]}
; lst() defaults to current list when name not specified
```

You can use *Add line to list* to create fixed lists of string and numeric data. For example

```
Set current list DROPDATALIST
Define list { Name, Sales, Expenses }
Add line to list {('Fred',100,20)}
Add line to list {('Sam',81,15)}
Add line to list {('George',92,34)}
Add line to list {('Niles',45,15)}
```


Alternatively, you can use the $add() method to add lines to your list. The following method defines the list and adds three rows

```
Do LIST1.$define(Name,Sales,Expenses)
Do LIST1.$add('Henry',231,154)
Do LIST1.$add('Moses',342,132)
Do LIST1.$add('Cynthia',423,231)
```

You can also use the $addbefore() and $addafter() methods to add lines at a specific position in the list.

Advise on find/next/previous

Reversible: YES   **Flag affected:**   NO

Parameters: ☐ Accept

Syntax: Advise on find/next/previous [*(Accept)*]

DDE command, OMNIS as server. This command determines when OMNIS is permitted to send requested Advise messages to the client program. When Advise requests have been received from a client, the *Set server mode* command determines when OMNIS is permitted to send field values that have changed. In addition to the *Set server mode* options, the three commands *Advise on Find/next/previous, Advise on OK,* and Adv*ise on Redraw* let you toggle individual options on or off. *Advise on Find/next/previous* lets you control this particular option without affecting the other two.

```
Advise on Find/next/previous (Accept)   ;; Check the Accept option
```

Advise on OK

Reversible: YES   **Flag affected:**   NO

Parameters: ☐ Accept

Syntax: Advise on OK [*(Accept)*]

DDE command, OMNIS as server. This command determines when OMNIS is permitted to send requested Advise messages to the client program. When Advise requests have been received from a client, the *Set server mode* command determines when OMNIS is permitted to send field values that have changed. In addition to the *Set server mode* options, the three commands *Advise on Find/next/previous, Advise on OK,* and *Advise on Redraw* let you toggle individual options on or off. The *Advise on OK* command lets you control this particular option without affecting the other two.

```
Advise on OK (Accept)        ;; Enables advise on OK
Advise on OK                 ;; Disables advise on OK
```

Advise on redraw

Reversible: YES   **Flag affected:**   NO

Parameters: ☐ Accept

Syntax: Advise on redraw [*(Accept)*]

DDE command, OMNIS as server. This command determines when OMNIS is permitted to send requested Advise messages to the client program. When Advise requests have been received from a client, the *Set server mode* command determines when OMNIS is permitted to send field values that have changed. In addition to the *Set server mode* options, the three commands *Advise on Find/next/previous, Advise on OK,* and *Advise on redraw* let you toggle individual options on or off. The *Advise on redraw* command lets you control this particular option without affecting the other two.

```
Advise on redraw (Accept)       ;; Enables advise on redraw
Advise on redraw                ;; Disable advise on redraw
```

AND selected and saved

Reversible: NO   **Flag affected:**   YES

Parameters: Line number (can be a calculation, default is current line)
                  ☐ All lines

Syntax: AND selected and saved [*(All lines)*] [*{line-number}*]

This command performs a logical AND of the Saved selection with the Current selection. You can specify a particular line in the list by entering either a number or a calculation. The **All lines** option performs the AND for all lines of the current list.

To allow sophisticated manipulation of data via lists, a list can store two selection states for each line; the "Current" and the "Saved" selection. The Current and Saved selections have nothing to do with saving data on the disk; they are no more than labels for two sets of selections. The lists may be held in memory and never saved to disk: they will still have a Current and Saved selection state for each line but they will be lost if not saved. When a list is stored in the data file, both sets of selections are stored.

The list data structure contains the column definitions, the field values for each line of the list, the current selected status and saved selected status for each line, *LIST.$line*, *LIST.$linecount* and *LIST.linemax*.

The *AND selected and saved* command performs a logical AND on the saved and current state, and puts the result into the Current selection. Hence, for a particular line, if both the Current and Saved states are selected, the Current state remains selected, but if either or both states are deselected, the resulting Current state will become deselected.

Saved State	Current State	Resulting Current State
Selected	Selected	Selected
Deselected	Selected	Deselected
Selected	Deselected	Deselected
Deselected	Deselected	Deselected

The following example selects all but the middle line of the list:

```
Set current list MYLIST
Define list {LVAR1}
Calculate LVAR1 as 1
Repeat
   Add line to list
   Calculate LVAR1 as LVAR1+1
Until LVAR1=6
Select list line(s) (All lines)
Save selection for line(s) (All lines)
Invert selection for line(s) {3}
AND selected and saved (All lines)
Redraw lists
```

Autocommit

Reversible: NO   **Flag affected:** YES

Parameters: On or Off mode (On is the default)

Syntax: Autocommit *(On|Off)*

This command turns on or off the automatic commit or rollback; on being the default. It lets you turn off the default behavior of OMNIS whereby statements between *Begin SQL script* and *End SQL script* commands that are completed without error are automatically committed at the next *Begin SQL script*, *Reset session* or *Logoff from host*. After each *Execute SQL script*, an error causes OMNIS to roll back the transaction. The default for a session which has not issued an *Autocommit* is automatic commit on. SQL statements sent to a remote database using *Perform SQL* are committed at the next *Begin SQL script*, *Reset session* or *Logoff from host*.

When *Autocommit* is off, you can use *Commit current session* and *Rollback current session* to commit or rollback uncommitted statements at any time; with automatic commit off, OMNIS will only issue explicit commits and rollbacks when it encounters these commands.

(Under some circumstances, the external database may commit or rollback as a consequence of some other action.) You can use *Commit current session* and *Rollback current session* with automatic commit switched on but there will not, usually, be anything to commit or rollback.

In some situations you should use *Set transaction mode* instead of *Autocommit.* This is described more fully in the server-specific programming section of the *OMNIS Studio Data Access Manager* manual.

```
Autocommit (Off)
Begin SQL script
SQL: Update TABLE set (column='value') where CODE='IDI'
End SQL script
Execute SQL script
If flag true
   Commit current session
End If
```

Begin print job

Reversible:	NO	**Flag affected:**	YES
Parameters:	None		
Syntax:	Begin print job		

This command defines the beginning of an OMNIS print job. To create a print job, you use the following sequence of commands.

```
Begin print job
   ; Print the various reports, using either
   ; Print report, or Prepare for print etc.
End print job
```

Only one print job can be started at any time: you cannot nest *Begin print job* commands.

If printing is already in progress, *Begin print job* returns an error and sets the flag to false. It also returns an error if it cannot set up the printer, or open the printer document; again, it sets the flag to false in this case.

Begin print job sets the flag to true if it succeeds. It automatically sets the report destination to the printer and closes the report destination selection window if it is open.

Each report is printed in the same way as if it were in an individual document. If you print two reports in a job, then page numbering starts at 1 for each report.

You cannot change the page setup while a print job is in progress, although OMNIS does not try to enforce this, as it will probably cause an OS error (and abnormal termination of printing) if you do.

Under MacOS, there is a spool file limit of 128 pages, imposed by the operating system. If a job exceeds this limit then the job will be printed as multiple documents, and this may not result in the desired interleaving.

The *Begin* and *End print job* commands only apply to reports sent to a printer, via the printer report destination.

Begin reversible block

Reversible: NO   **Flag affected:**   NO

Parameters: None

Syntax: Begin reversible block

This command begins a reversible block of commands. All *reversible* commands enclosed within the commands *Begin reversible block/End reversible block* are reversed *when the method containing this block finishes.* However, a reversible block in the $construct() method of a window class reverses *when the window is closed*—not when the method is terminated as is normally the case. OMNIS always steps backwards through a reversible block of commands, thus the first command is reversed last.

Reversible blocks let you create subroutines that restore the values of variables, the current record buffer, and so on, to their previous state when the method terminates. Most commands are reversible: those that are not usually involve an irreversible action such as changing the data in an OMNIS data file or running another program.

A method can contain more than one block of reversible commands. In this case, commands contained within *all* the blocks are reversed when the method terminates.

```
; all the commands in the following example are reversed
; when the method containing the block is finished
Begin reversible block
   Disable menu line {MMENU/5}
   Set current list LVAR1
   Build open window list (Clear list)
   Calculate LVAR1 as 0
   Open window instance WEDIT
End reversible block
; more commands...
```

When this block is reversed:

1. The window instance WEDIT is closed

2. *LVAR1* returns to its former value

3. MYLIST is restored to its former contents and definition

4. The current list is set to the former value

5. Menu line 5 is enabled

Methods called from within a reversible block are *not* reversed. For example

```
; FirstMethod
Begin reversible block
   A...
   Do method SecondMethod
   B...
   C...
   D...
   E...
End reversible block

; SecondMethod
   M...
   N...
   O...
   P...
```

In this example, commands A... to E... within the reversible block are reversed (if they are commands that can be reversed), while commands M... to P... within the called method are *not* reversed.

Further examples will show how reversible blocks are used. The following method hides fields Entry1 and Entry2 and installs the menu MCUSTOMERS.

```
Begin reversible block
   Hide fields Entry1,Entry2
   Install menu MCUSTOMERS
End reversible block
OK message (Icon) {MCUSTOMERS is now visible}
```

When this method ends, first MCUSTOMERS is removed, then the fields are shown.

In the following example, the current list is LIST1.

```
Begin reversible block
   Set current list LIST2
   Define list {AMOUNT,TOS}
   Set main file {FACCOUNTS}
   Build list from file on ACCNUM
   Enter data
End reversible block
```

When this method terminates and the command block is reversed, the Main file is reset, the former list definition is restored and the current list is restored to LIST1.

Begin SQL script

Reversible: NO   **Flag affected:** YES
Parameters: None
Syntax: Begin SQL script

This command defines the start of a block of SQL statements and text to be stored in the SQL buffer. The SQL text buffer is cleared when you execute this command. The *End SQL script* command defines the end of the block. The lines are not checked by OMNIS in any way and must be valid SQL in order for the server to be able to operate on them. When an *Execute SQL script* command is issued, the text in the buffer is sent to the remote server. The *Begin SQL script* and *End SQL script* markers usually denote a transaction which OMNIS will automatically commit if no errors occur.

The commands *Begin SQL script*, *Execute SQL script, Perform SQL,* and *Reset cursor(s)* all empty the SQL statement buffer for the current session.

```
; method to select all customers
Begin SQL script
SQL: Select * from CUSTOMERS
End SQL script
Execute SQL script
```

Begin text block

Reversible: NO   **Flag affected:** NO
Parameters: ☐ Keep current contents
Syntax: Begin text block [*(Keep current contents)*]

This command defines the start of a block of text to be stored in the global text buffer. The *Begin text block* command clears the text buffer by default, and adds the text in subsequent *Text:* commands to the text buffer. However, you can keep the current contents of the buffer by checking the **Keep current contents** option, in which case text is appended to current text in the buffer. You build the text block using the *Text:* command, which supports leading and trailing spaces and can contain square bracket notation. The *End text block* command defines the end of the text block, and you can return the contents of the text buffer using the *Get text block* command.

```
; Declare var cTEXT of Character type
Begin text block
Text: If a train station is where the
Text: train stops, what is a work station?
End text block
Get text block cTEXT
```

Break to end of loop

Reversible: NO   **Flag affected:**   NO

Parameters: None

Syntax: Break to end of loop

This command terminates a *Repeat, While* or *For* loop, passing control to the command following the *Until, End While* or *End For* command. An *If* command is usually placed before the *Break to end of loop* to determine the condition under which a break occurs.

```
Open window instance WClient
Set main file {FCLIENT}
Find first on SEQ
While SEQ<201
   Prepare for edit
   Enter data
   If flag false
     Break to end of loop
   End If
   Update files
   Next
End While
; Control breaks to here if Enter data is canceled
```

Break to end of switch

Reversible: NO   **Flag affected:**   NO

Parameters: None

Syntax: Break to end of Switch

This command causes OMNIS to jump out of the current *Case* statement (i.e. terminate the Case before the end of Case is reached), and resume method execution after the *End Switch* command. You use it in conjunction with the *Switch* and *Case* commands.

```
Switch LVAR1
  Case 16
     OK message {Got a 16}
     Break to end of switch
     OK message {I never run}
  Case 4
     OK message {Got a 4}
     Break to end of switch
     OK message {I never run}
  Default
     OK message {didn't get a 4 or 16}
End Switch
```

Breakpoint

Reversible: NO   **Flag affected:**   NO

Parameters: Message (text)

Syntax: Breakpoint [*{message}*]

This command places a breakpoint at a command line in a method where you want to stop execution, to check your coding for example. You can include a message with the command which is displayed in the debug window when the break occurs. The command does nothing at runtime.

When OMNIS encounters a breakpoint the debugger is opened with the current method loaded and the *Breakpoint* command line highlighted. You can examine the value of fields and variables by right button/Ctrl-clicking on the field or variable name.

Following a breakpoint you can continue method execution by clicking the Go button or by using Step or Trace mode.

```
Open window instance WCONTROL
Calculate LVAR1 as sqr(MASS/2)
```
Breakpoint {Check MASS and LIMIT}
```
If LVAR1 >> LIMIT
   Do method SetLimit
End If
```

Bring window instance to front

Reversible: NO   **Flag affected:**   NO

Parameters: Window instance name

Syntax: Bring window instance to front *window-instance-name*

This command brings the specified window instance to the front. If the window is already in front, the command does nothing. If the specified window instance does not exist (that is, the window is not open) this command will cause an error.

```
Test for window open {winst1}
If flag true
```
   Bring window instance to front winst1
```
Else
   Open window instance Mywin/winst1
End If
```

Build export format list

Reversible: YES   **Flag affected:**   YES

Parameters: ☐ Clear list

Syntax: Build export format list [*(Clear list)*]

This command builds a list containing the name of each export format. The list is built in the current list for which you must define a single column to contain the export format.

The **Clear list** option clears the current list and redefines it to include only the S4 field. With this option, the command becomes reversible.

```
Set current list EXPORTLIST
```
Build export format list (Clear list)
```
; Defines the list as containing S4
```

Build externals list

Reversible: YES   **Flag affected:** YES

Parameters: ☐ Clear list

Syntax: Build externals list [*(Clear list)*]

This command builds a list of the externals in the **EXTERNAL** folder. The list of extensions is placed in the current list for which you must define the following columns

	Col 1 (Character)	Col 2 (Character)	Col 3 (Number)	Col 4 (Character)
Windows	File name	Routine name	Routine index	File extension
MacOS	File name	Routine name	Routine ID	Routine type

The *Clear list* option clears the current list. The command becomes reversible with this option.

The following method builds a list of extensions.

```
; declare Local vars NAME, ROUTINE, IND, TYPE
; declare Local var EXTERNALLIST of List type
Set current list EXTERNALLIST
Define list (NAME,ROUTINE,IND,TYPE)
```
Build externals list

When an external routine is called, the internal list of routines is always searched before the current resource path. If a full pathname for a file *and* a routine name is specified, only that path is searched.

Build field names list

Reversible: YES   **Flag affected:**   YES

Parameters: ☐ Clear list
    ☐ Full names
    File name

Syntax: Build field names list [*([Clear list][,Full names])] {file-name}*

This command builds a list of field names for the specified file class in the current list. You must specify the following columns in the current list.

Column 1 (Character)	Column 2 (Character)	Column 3 (Character)
Field name	Field type and length	Description; for index fields only

When you use the **Clear list** option you get column 1 only defined as #S5. With this option the command becomes reversible. The flag is cleared if the value of *LIST.$linemax* prevents a complete list from being built.

The **Full names** option creates a list in which the fields are prefixed with the file class name, for example, PO_DATE becomes FPORDERS.PO_DATE.

```
Set current list FIELDLIST
Build field names list (Clear list) {FILENAME}
; Clear list option defines the list as containing #S5
```

or you can do it like this

```
Do $files.filename.$makelist($ref.$name)
```

Build file list

Reversible: YES   **Flag affected:**   YES

Parameters: ☐ Clear list

Syntax: Build file list [*(Clear list)*]

This command builds a list containing the name of each file class in the current library. The list is built in the current list for which you must specify the following columns.

Column 1 (Character)	**Column 2 (Character)**
File name	Description for file (if you have entered one)

When you use the **Clear list** option you get column 1 only defined as #S5. With this option the command becomes reversible, that is, the original contents of the list are restored. The flag is cleared if the number of lines in the list exceeds *LIST.$linemax*.

```
Set current list FILELIST
Build file list (Clear list)
; Clear list option defines the list column as #S5
```

or you can do it like this

```
Do $files.$makelist($ref.$name)
```

Build indexes

Reversible: NO   **Flag affected:**   YES

Parameters: File name

Syntax: Build indexes *{file-name}*

This command rebuilds all the indexes for the specified file which have been dropped with the *Drop indexes* command. *Drop indexes* deletes all the indexes for the specified file apart from the sequence number index. *Build indexes* checks that all the indexes defined in the file class actually exist in the data file and builds those which are not there. This command does not build any indexes which already exist even if they are in a damaged state.

If the specified file name does not include a data file name as part of the notation, the default data file for that file is assumed. If the file is closed or memory-only, the command does not execute and returns flag false.

If you are not running in single user mode, this command automatically tests that only one user is using the data file (the command fails with the flag false if this is not true), and further users are prevented from logging onto the data until the command completes.

If a working message with a count is open while the command is executing, the count will be incremented at regular intervals. The command may take a long time to execute, and it is not possible to cancel execution even if a working message with cancel box is open.

The flag is set if at least one index is successfully rebuilt. Note that the command is not reversible.

```
Do not flush data
Drop indexes {Pictures}
Repeat
   Working message {Building indexes...}
   Build indexes {Pictures}
Until flag true
```

Build installed menu list

Reversible: YES   **Flag affected:** YES

Parameters: ☐ Clear list

Syntax: Build installed menu list [*(Clear list)*]

This command builds a list containing the name of all menu instances on the main OMNIS menu bar, starting from the left. All the standard OMNIS menus such as **File** and **Edit** are ignored. The list is built in the current list for which you must define the following columns:

Column 1 (Character)	Column 2 (Character)
Menu instance name	Description for menu class (if one has been entered)

When you use the **Clear list** option you get column 1 only defined as #S5 with a 15 character column width. With this option, the command becomes reversible.

Menu instances from libraries other than the current library are prefixed with their library names. The flag is cleared if the command fails due to a shortage of memory.

```
Set current list MENULIST
Build installed menu list (Clear list)
; clear list option defines list as #S5
```

or you can do it like this

```
Do $imenus.$makelist($ref.$name,$ref.$desc)
```

Build list columns list

Reversible: YES   **Flag affected:** YES

Parameters: List or row name (default is the current list)
☐ Clear list

Syntax: Build list columns list [*list-name*] [*(Clear list)*]

This command builds a list containing the column names and data types of the current or specified list. This information is placed in the current list. If the current list contains one column, it contains the column names only. The current list column headings are ignored, but to obtain all the available information, you define the list with two columns as follows:

Col 1 (Character)	Col 2 (Character)
List Column name	List Column data type

The **Clear list** option clears and defines the current list to contain one column, #S5, so the column data types are not returned. With this option, the command becomes reversible.

The flag is cleared if the value of *LIST.$linemax* prevents a complete list from being built. The following method and the list of data it loads into the list illustrate the typical values produced:

```
Set current list COLSLIST
Define list {PO_DATE,PO_NUMBER,PO_BATCHED,SU_CONTACT,IT_UNITPRICE}
Set current list LIST1
Define list {CVAR2,CVAR3}
Build list columns list COLSLIST
; Here are the values for LIST1:
```

#S2	#S3
PO_DATE	Short date 2000..2099
PO_NUMBER	Character 15
PO_BATCHED	Boolean
SU_CONTACT	Character 30
IT_UNITPRICE	Number 2 dp

or you can do it like this

```
Do LIST.$cols.$makelist($ref.$name, $ref.$coltype)
```

Build list from file

Reversible: NO   **Flag affected:**   YES

Parameters: Field name (must be indexed)
            ☐ Exact match
            ☐ Use search
            ☐ Use sort

Syntax: Build list from file on *field-name* [*(*[*Exact match*]
        [*,Use search*] [*,Use sort*]*)*]

This command builds a list of data from the main file using a specified index field. The records are selected and corresponding field values added to the list in the order of the specified index field. You must set the main file before using the command.

If the **Exact match** option is specified, only records matching the current value of the specified field are added to the list. Similarly, if the **Use search** check box is selected, only records matching the current search class are added. In both cases, an error occurs if neither a field nor a search class is specified.

When large files are involved, that is, those that may require more than the maximum number of available lines (the value of *LIST.$linemax*), you can use the flag false condition to detect when an incomplete list is built.

Building a list using this command does not affect the current record buffer and does not clear 'Prepare for update' mode.

The **Use sort** option lets you use the database records in sorted order without first having to load them into a list. You use *Set sort field* to specify a sort field after which *Build list from file (Use sort)* creates a sorted table of records in memory before loading them into the list. The main advantage of this method is that the sort fields do not have to be read into the list at all. The Sort field order overrides the index field order but if the sort field is non-indexed, the index is used as the order in which to gather up records before sorting. Multi-level sorts are possible by using repeated *Set sort field* commands to accumulate the required sorting order. Since sort levels are cumulative you should first clear any existing ones with *Clear sort fields*.

The following method compiles a list of all records where CODE equals the current value of CODE in the CRB.

```
Set current list LIST1
Build list from file on CODE (Exact match)
```

The following method compiles a list of all records sorted in order of descending PO_NETTOTAL values and within each value, in increasing PO_NUMBER order.

```
Set current list LIST1
Set main file {FPORDERS}
Define list {PO_DATE, PO_NETTOTAL}
Clear sort fields
Set sort field PO_NETTOTAL (Descending)
Set sort field PO_NUMBER
Build list from file on PO_SEQ (Use sort)
;  Note PO_NUMBER is not in the list
```

Build list from select table

Reversible: NO   **Flag affected:**   NO

Parameters: Cursor name (default is the current)
     ☐ Add CRB fields
     ☐ Clear list
     List name (default is the current)

Syntax: Build list [*list-name*] from select table [for cursor *cursor-name*]
     [*([Add CRB fields] [,Clear list])*]

This command copies the select table for the current or specified cursor into the current or specified list. Each row in the select table corresponds to one line in the OMNIS list. A *Define list* command should have already been executed to ensure that suitable list field types correspond to the correct table columns. *Build list from select table* appends lines to the current or specified list. You can set *LIST.$linemax* to limit the size of the resulting list. A *Build list from select table* occurring after a sequence of *Fetch next row* commands stores only the part of the table which has not already been fetched.

The flag is not a reliable indicator of whether the list build was successful, although it is cleared if an error occurred. Otherwise *sys(138)* should be used to check if there are more rows to fetch.

Picture and Binary field types are not supported by *Build list from select table*.

The **Add CRB fields** option adds values taken from the current record buffer to the list. You can use this option for columns with no available SQL data but this slows down the command by about 20%.

The **Clear list** option clears the current list before building the new list otherwise the command appends the data to the current list.

```
; declare class variable CLIST with List type
Set current list FLIST
Define list (COL1,COL2,COL3,COL4)
Describe database (Tables)
Build list from select table (Clear list)
```

For potentially large tables, setting the maximum number of lines in the list allows users to control the retrieval of the rows, for example

```
Calculate LIST.$linemax as 0
Clear list
Repeat
   Calculate LIST.$linemax as LIST.$linemax + 50
   Build list from select table
   If sys(138)
      Yes/No message {Load next 50?}
   End If
Until not(sys(138))
```

Alternatively, you can do it like this

```
Do TableBasedList.$select()
Do TableBasedList.$fetch(nRows)
```

Build list of event recipients

Reversible:	NO	**Flag affected:**	NO
Parameters:	None		
Syntax:	Build list of event recipients		

This command builds a list of Apple event recipients. The list is built in the current list for which the columns must have been defined. The columns are

Column 1 (Character)	**Column 2 (Character)**
Recipient tag	Application name

At any one time, you may have multiple recipients. *Build list of event recipients* uses the current list to build a list of recipient tags and application names that are currently known to OMNIS; one recipient tag per row of the list.

```
Begin reversible block
   Set current list LIST1
End reversible block
Define list {S2,S3}
Build list of event recipients
Redraw lists
```

Build menu list

Reversible: YES   **Flag affected:** YES

Parameters: ☐ Clear list

Syntax: Build menu list [*(Clear list)*]

This command builds a list containing the name of each menu class in the current library. The list is built in the current list for which the columns must have been defined. The columns are

Column 1 (Character)	Column 2 (Character)
Menu class name	Description for menu (if one has been entered)

The **Clear list** option clears the current list and redefines it to include only the #S5 field. With this option, the command becomes reversible but you get column 1 only.

```
Set current list MENULIST
Build menu list (Clear list)
; defines the list as containing #S5
```

or you can do it like this

```
Do $menus.$makelist($ref.$name)
```

Build open window list

Reversible: YES   **Flag affected:** YES

Parameters: ☐ Clear list

Syntax: Build open window list [*(Clear list)*]

This command builds a list containing the name of each window instance, starting with the topmost window instance. The window instance names are stored in the first column of the list. You can also return the position and size coordinates of each window instance in the second to fifth columns. The list is built in the current list for which you must define the following columns:

Col 1 (Character)	Col 2 (Long Int)	Col 3 (Long Int)	Col 4 (Long Int)	Col 5 (Long Int)
Window instance name	/left window coord	/top window coord	/right window coord	/bottom window coord

If you use the **Clear list** option, the list will contain one column only defined as #S5, so the window coordinates are not returned. Also, with the **Clear list** option selected, the

command is reversible, that is, the list definition and contents are restored when the method terminates.

```
Set current list WINSLIST
Build open window list (Clear list)   ;; list uses #S5
```

or you can do it like this

```
Do $iwindows.$makelist($ref.$name)
```

Build report list

Reversible: YES   **Flag affected:**   YES

Parameters: ☐ Clear list

Syntax: Build report list [*(Clear list)*]

This command builds a list containing the name of each report class in the current library. The list is built in the current list for which the columns must have been defined. The columns are

Column 1 (Character)	Column 2 (Character)
Report class name	Description for report (if one has been entered)

You get column 1 only when you use the **Clear list** option.

The **Clear list** option clears the current list and redefines it to include only the #S5 field. With this option the command becomes reversible.

```
Set current list REPLIST
Build report list (Clear list)   ;; list use #S5
```

or you can do it like this

```
Do $clib.$reports.$makelist($ref.$name)
```

Build search list

Reversible: YES **Flag affected:** YES

Parameters: ☐ Clear list

Syntax: Build search list [*(Clear list)*]

This command builds a list containing the name of each search class in the current library. The list is built in the current list for which the columns must have been defined. The columns are

Column 1 (Character)	Column 2 (Character)
Search class name	Description for search (if one has been entered)

You get column 1 only when you use the **Clear list** option.

The **Clear list** option clears the current list and redefines it to include only the #S5 field. With the **Clear list** option, the command is reversible. The flag is cleared if the value of *LIST.$linemax* prevents a complete list from being built.

This example displays the names of the search classes.

```
Set current list SEARCHLIST
Build search list (Clear list)   ;; list uses #S5
```

or you can do it like this

```
Do $clib.$searches.$makelist($ref.$name)
```

Build window list

Reversible: YES   **Flag affected:** YES

Parameters: □ Clear list

Syntax: Build window list [*(Clear list)*]

This command builds a list containing the name of each window class in the current library. The list is built in the current list for which you must define the following columns

Column 1 (Character)	Column 2 (Character)
Window class name	Description for window (if one has been entered)

You get column 1 only when you use the **Clear list** option, but the command becomes reversible.

The **Clear list** option clears the current list and redefines it to include only the #S5 field. With the **Clear list** option, the command becomes reversible.

```
Set current list WINDOWLIST
Build window list (Clear list)   ;; list uses #S5
```

or you can do it like this

```
Do $clib.$windows.$makelist($ref.$name)
```

Calculate

Reversible: YES   **Flag affected:** NO

Parameters: Field name
             Calculation (leave blank for null values)

Syntax: Calculate *field-name* as [*calculation*]

This command assigns a new value to a data field or variable. The form of the command is "Calculate X as Y", where X is a valid data field or variable name and Y is either a valid data field or variable name, value, calculation, or notation. When *Calculate* is executed the state of the flag is unchanged, unless #F is recalculated by this command.

You can use *Calculate* in a reversible block. The data field returns to its initial value when the method containing the block of reversible commands finishes.

WARNING The *Calculate* command does not redraw a calculated field so if your field is on a window you must use the *Redraw* command or the $redraw() method after the *Calculate* command to reflect the change.

The following examples show how you can use the *Calculate* command.

```
Calculate LVAR1 as LVAR2
; sets field LVAR 1 equal to the contents of LVAR 2
Calculate PRICE as 100.50
; sets PRICE equal to 100.50
Calculate PRICE as COST*(1+MARKUP/100)
; calculates the value of PRICE from the current
; values of COST and MARKUP
```

You can also use notation in the field or calculation, for example

```
Calculate $cwind.$objs.Field1.$top as 9999
; recalculates the position of Field 1
Calculate $clib.$prefs.$mouseevents as kFalse
; turns off mouse events
```

You can operate on variables with the *Calculate* command, for example

```
; Declare local variable L_FILES of List type
Set current list L_FILES
Calculate L_FILES as $libs.LIBNAME.$files.$makelist($ref.$name)
; builds a list of files in the library
```

Note that certain operations executed via the notation are better performed using the *Do* command, rather than *Calculate*, for example

```
Do $iwindows.win1.$bringtofront()
; brings the window instance to the front, but is simpler than
Calculate #F as $iwindows.win1.$bringtofront()
```

Operator Precedence

Mathematical expressions are evaluated using the operator precedence so that in the absence of brackets, * and / operations are evaluated before + and -. The full ordering from highest to lowest precedence is:

```
unary minus
* and /
+ and -
>, <, >=, <=, <>, =
& and |
```

For example, if you execute the command

```
Calculate LVAR1 as 10-2*3
```

the calculation part is evaluated as

```
10-(2*3) which equals 4
```

Call external routine

Reversible:	NO	**Flag affected:**	YES

Parameters: Routine name or library name/routine name
Parameters list
Return field

Syntax: Call external routine [*library name*/]***routine-name***
[*(parameter1*[,*parameter2*]*...)]* returns ***return-field***

This command calls an external routine with mode ext_call and returns a value from the external in the specified *return-field*. The return value is placed in the specified field by the external code using the predefined field reference Ref_returnval with the functions SetFldVal or SetFldNval. The flag is set if the external routine is found and the call is made but this does not necessarily mean that the external code has executed correctly. The flag is cleared if the routine is not found. Note that the routine cannot use the flag to pass information back to the method.

You can pass parameters to the external code by enclosing a comma-separated list of fields and calculations. If you pass a field name, for example, *Call external routine Maths1 (Num1,Num2),* the external can directly alter the field value. Enclosing the field in brackets, for example, *Call external routine Maths1 ((Num1),(Num2)),* converts the field to a value and protects the field from alteration.

In the routine itself, the parameters are read using the usual GetFldVal or GetFldNval with the predefined references Ref_parm1, Ref_parm2, and so on, Ref_parmcnt gives the number of parameters passed. If the field name is passed as a parameter, you can use SetFldVal or SetFldNval with Ref_parm1, and so on, to change the field's value.

```
Call external routine Mathslib/sqroot (Num1) returns Num2
```

Cancel advises

Reversible:	NO	**Flag affected:**	NO

Parameters: Field name
☐ All channels

Syntax: Cancel advises [*field-name*] [*(All channels)*]

DDE command, OMNIS as client. This command cancels one or more *Request advises* from the current channel. If you omit the field name, all Request advises to the current channel are canceled. If you specify a field name, all Request advises to the current channel which refer to that field name are canceled.

The command is addressed to the current channel only, and if the current channel is not open, an error occurs. No error occurs, however, if there are no *Request advises* commands to cancel.

If you use the **All channels** option, all channels are canceled. There is no need to use a *Cancel advises* command before a *Close DDE channel* command.

When OMNIS issues a *Request advises* to a DDE server, OMNIS is in effect saying "Hey, tell me if this value changes and send me an update". The *Enter data* command must be running to allow the incoming data to get through.

```
Yes/No message {Do you want updates?}
If flag false
   Cancel advises (All channels)     ;; clears all advises
   Quit method
Else
   Request advises C_COMPANY {C_COMPANY}
   Request advises C_ADDRESS {C_ADDRESS}
End If
Prepare for insert
Enter data
Update files if flag set
```

Cancel event recipient

Reversible: NO   **Flag affected:**   YES
Parameters: Recipient tag
Syntax: Cancel event recipient *{recipient-tag}*

This command cancels the specified Apple event recipient.

```
Set event recipient {Microsoft Excel}
; do something...
Yes/No message {Do you want to keep Excel?}
If flag false
   Cancel event recipient {Microsoft Excel}
Else
; continue...
```

Cancel prepare for update

Reversible: NO   **Flag affected:**   NO

Parameters: None

Syntax: Cancel prepare for update

This command cancels the Prepare for update mode and releases any semaphores which may have been set. You use the *Prepare for edit/insert* command to prepare OMNIS for editing or insertion of records. It is usually followed by *Update files* which is the usual way of terminating the Prepare for... state but you can also terminate this state with *Cancel prepare for update*. It must be followed by commands which prevent an *Update files* command from being encountered.

When you execute a *Prepare for...* command in multi-user mode, semaphores are used to implement record locking. *Cancel prepare for update* neutralizes the effect of a *Prepare for...* command and releases all semaphores.

You can use this command within a timer method to implement a timed record release.

```
Set timer method 600 sec  {Timer method}
Prepare for edit
Enter data
Update files if flag set
Clear timer method


; Timer method
Yes/No message {Time's up, cancel edit?}
If flag true
   Cancel prepare for update
   Queue cancel
End If
```

Cancel publisher

Reversible: NO   **Flag affected:**   YES

Parameters: File or field list

Syntax: Cancel publisher [*{file|field1*[,*file|field2*]*...}*]

This command cancels the publication of the fields specified. The field list can take a file name (for all fields in a file) or a range of fields, which includes a range of fields in the order listed in the Field names window. If no list is given, all publications for the library are canceled. There are no errors if the list includes unpublished fields.

The flag is set if the command cancels the publication for one or more fields.

```
Publish field CNAME {HD80:Public:Sales-Name}
Publish field CTOTAL {HD80:Public:Sales-Total}
Find first on CNAME
Publish now {CNAME,CTOTAL}
Cancel publisher {CNAME,CTOTAL}
```

Cancel subscriber

Reversible: NO   **Flag affected:**   YES

Parameters: File or field list

Syntax: Cancel subscriber [*{file|field1*[,*file|field2*]*...}*]

This command cancels the subscription of the fields specified. The field list can take a file name (for all fields in a file) or a range of fields, which includes a range of fields in the order listed in the Field names window. If no list of field names is given, all subscriptions for the library are canceled. There are no errors if the list includes nonsubscribed fields.

The flag is set if the command cancels the subscription for one or more fields.

```
Subscribe field CNAME {Freds Mac: Public:Sales-Name}
Subscribe field CTOTAL {Freds Mac: Public:Sales-Total}
Enter data
Subscribe now {CNAME,CTOTAL}
Cancel subscriber {CNAME,CTOTAL}
```

Case

Reversible: NO   **Flag affected:** NO

Parameters: Constant value, field name or expression

Syntax: Case *expression*

The Case statement is part of a *Switch* construct that chooses one of an alternative set of options. The options in a Switch construct are defined by the subsequent *Case* commands. The *Case* command takes either a constant, field name, single calculation, or a comma-separated series of calculations. You must enclose string literals in quotes. Date values must match the date format in *#FDT*.

```
; a value between 1 and 4 is passed to Group
; declare parameter Group (Short integer (0 to 255))
Switch Group
   Case 1      ;; North
      Set search as calculation {Div='N'}
   Case 2      ;; East
      Set search as calculation {Div='E'}
   Case 3      ;; South
      Set search as calculation {Div='S'}
   Case 4      ;; West
      Set search as calculation {Div='W'}
End Switch
; now use search on a list perhaps
```

You can add multiple conditions in a comma-separated list to one Case statement (see below). Use *Default* to specify commands that should run if the value is not one of those specified in the Case statements. For example

```
Switch CVAR1
   Case 'A'
      ; do this, if CVAR1 is A
   Case 'B','C','D'
      ; or do this, if CVAR1 is B, C or D
   Default
      ; otherwise do this, if CVAR1 is none of the above
End Switch
```

Change user password

Reversible: NO   **Flag affected:**   NO

Parameters: None

Syntax: Change user password

This command opens the Password dialog in which the user can change passwords. The menus are redrawn and lists and variable values (apart from *#UL*) are unaffected.

If the current user is the master user, passwords can be changed. In addition, the command gives the user the choice of using another password to re-enter the current library at a different user level, thus gaining access to different areas of the library. If a user re-enters at a different level, the value of #UL will change (within the range 0–8) to reflect that new user level.

```
Test for menu installed {Options}
If flag false
   Yes/No message {You must install the Options menu to continue
   with this operation. You must re-enter as master user. Re-enter?}
   If flag true
      Change user password
   End If
   Quit all methods
End If
```

Check data

Reversible: NO   **Flag affected:**   YES

Parameters: ☐ Perform repairs
               ☐ Check data file structure
               ☐ Check records
               ☐ Check indexes
               File or list of file names (the default is all files)

Syntax: Check data [*([Perform repairs][,Check data file structure]*
               [,Check records][,Check indexes])] [{file1[,file2]...}]

This command checks the data for the specified file or list of files, and works only when one user is logged onto the data file. If you omit a file name or list of files, *all* the files with slots in the current data file are checked. If the specified file name does not include a data file name as part of the notation, the default data file for that file is assumed. If the file is closed or memory-only, the command does not execute and returns with the flag false.

There are **Check data file structure**, **Check records**, and **Check indexes** checkbox options. If none of these is specified, the command does nothing; if only **Check data file structure** is specified, the list of files is ignored. If **Perform repairs** is specified, any

repairs required are automatically carried out, otherwise the results of the check are added to the check data log. The check data log is not opened by this command but is updated if already open.

If you are not running in single user mode, this command automatically checks that only one user is using the data file (the command fails with flag false if this is not true), and further users are prevented from logging onto the data until the command completes.

If a working message with a count is open while the command is executing, the count will be incremented at regular intervals. The command may take a long time to execute and it is not possible to cancel execution even if a working message with cancel box is open.

The command sets the flag if it completes successfully and clears the flag otherwise. It is not reversible.

```
Check data (Check records) {MYDATA.ACCOUNT}
If flag true
   Yes/No message {View log}
   If flag true
      Open check data log
   End If
Else
   OK message (Icon,Sound bell) {The check data file command could
   not be carried out//Please make sure that only one user
   is logged onto the data file}
End If
```

Check menu line

Reversible:	YES	**Flag affected:**	NO

Parameters: Menu instance name
Menu line number

Syntax: Check menu line *menu-instance-name*/*menu-line-number*

This command places a check mark on the specified line of a menu instance to show that the option has been selected. You specify the menu instance name and the number of the menu line you want to check.

You can remove the check mark with *Uncheck menu line*. If you use this command in a reversible block, the check mark is removed when the method terminates. Nothing happens if the menu instance is not installed on the menu bar.

The following method tests whether a line in the menu instance is checked and either checks or unchecks it accordingly.

```
Install menu mBookings/MINST1
Test for menu line checked MINST1/3
If flag true
   Uncheck menu line MINST1/3
Else
   Check menu line MINST1/3
End If
```

Clear all files

Reversible:	YES	**Flag affected:**	NO

Parameters: None

Syntax: Clear all files

This command clears the current record buffer of all file variables for all open libraries and all open data files, including any memory-only files. However, it does not clear the hash variables. Window instances are not automatically redrawn so you must follow it by *Redraw* if you want the screen to reflect the current state of the buffer.

```
Clear all files
Redraw wInvoices
; clears CRB fields from fInvoices, fCustomers, fStock
```

Clear check data log

Reversible: NO   **Flag affected:**   NO

Parameters: None

Syntax: Clear check data log

This command clears the check data log, which stores all the results of a check data operation. To clear the log, there is no need for the log to be open.

```
Check data (Check records) {MYDATA.ACCOUNT}
If flag true
   Yes/No message {View log}
      If flag true
         Open check data log
         ; After looking at the data log
         Yes/No message {Clear the log?}
         If flag true
            Clear check data log
         End If
      End If
Else
   OK message {Check data file could not be carried out//Please
   ensure that only one user is logged onto the data file}
End If
```

Clear class variables

Reversible: NO   **Flag affected:**   NO

Parameters: None

Syntax: Clear class variables

This command clears any class variables used within the class and clears the memory used for the class variables. *Clear class variables* is placed in a method within the class where you want to clear variables.

A class variable is initialized to empty or its initial value the first time it is referenced. It remains allocated until the class variables for its class are cleared. The class variables for all classes are cleared when the library file is closed.

```
; declare class variables with initial values
; transfer values to instance variables
Clear class variables  ;; all variables for class are now clear
Redraw (Refresh now) {Entry1,Entry2,Entry3}
```

Clear data

Reversible:	NO	**Flag affected:**	YES

Parameters: Field name
    ☐ Redraw field
    ☐ All windows

Syntax: Clear data [*field-name*] [*([Redraw field][,All windows])*]

This command clears the data from the specified field or current selection. The data is lost and is not placed on the clipboard. If you do not specify a field, the current field's data is cleared (assuming there is a selection).

In the case of a null selection when the cursor is merely flashing in a field and no characters are selected, *Clear data* will literally clear "nothing".

The following method is placed behind the TO_PRICE field and checks if the value is over 5000; if it is, the value entered into the field is cleared and the cursor remains in the field.

```
On evAfter
   If TO_PRICE > 5000
   Yes/No message {Is this price correct?}
   If flag false
      Clear data TO_PRICE (Redraw field)
      Queue set current field {ePrice}
      Quit event handler (Discard event)
   End If
```

Clear DDE channel item names

Reversible:	YES	**Flag affected:**	NO

Parameters: None

Syntax: Clear DDE channel item names

DDE command, OMNIS as client. This command clears all server data item names selected for use with a print-to-channel report. You use this command when exporting data via a DDE channel to another Windows application. The channel item names become the item names into which the server places the fields printed in the OMNIS report.

Clear DDE channel item names clears all the item names set up with *Set DDE channel item name*.

```
Set DDE channel number {2}
Open DDE channel {EXCEL|SHEET1}
Send to DDE channel
Set report name RDDEXPORT
Clear DDE channel item names
Send command {[[TakeControl]]}
; Double first [['s so OMNIS accepts text
If flag true
   Set DDE channel item name {R1,C1}
   Set DDE channel item name {R2,C1}
   ...
   Set DDE channel item name {R50,C1}
   Print report
End If
```

Clear find table

Reversible: NO   **Flag affected:**   NO

Parameters: None

Syntax: Clear find table

This command clears the find table for the current main file and releases the memory it used.

When a *Find* or *Next*/*Previous* command is encountered, OMNIS uses the Index, Search and Sort field parameters to create a table of records (similar to a SQL Select table). This may simply be an existing index in which case no further processing takes place or, if there is a search and/or sort condition, a file may be scanned and a selection of records sorted in memory. If a *Next* or *Previous* returns an unexpected record or no record, this is probably because there is still a find table in existence from another Find operation.

For a large file, a substantial amount of RAM may be used.

```
Set main file {FCLIENT}
Set sort field TOWN
Set sort field COUNTRY
Find first on CCODE (Use sort)
Do method ProcessTable
Clear find table
```

Clear line in list

Reversible: NO   **Flag affected:**   YES

Parameters: Line number (can be a calculation, default is current line)

Syntax: Clear line in list [*{line-number}*]

This command clears the values stored in the specified line of the current list. You can specify the line number in a calculation, otherwise the current line (*LIST.$line*) is used. The flag is cleared if the list is empty or if the line is beyond the current end of the list.

This method deletes the current line from the list if CREDIT value is zero:

```
Set current list LIST2
For each line in list from 1 to LIST2.$linecount step 1
   If lst(CREDIT)=0
      Clear line in list
   End If
End For
Redraw lists
```

or you can do it like this

```
Do LIST.rownumber.$clear()
```

Clear list

Reversible:	YES	**Flag affected:**	NO
Parameters:	☐ All lists		
Syntax:	Clear list [*(All lists)*]		

This command clears all the lines in the current list and frees the memory they occupy. It does not alter the definition of the list. If you use *Clear list* as part of a reversible block, the list lines will be reloaded when the method containing the reversible block finishes. The list is only reloaded if it occupies 50,000 bytes of storage or less.

The **All Lists** option only clears the hash variable lists #L1 to #L8: all other lists including task, class, instance and local variable lists, are *not* cleared by this command.

The following method builds a list of data formats depending on the type of graph selected by the user. Before the method is built the list is cleared using the *Clear list* command; this ensures the list is initialized and completely empty of data.

```
Set current list List1
Clear list
Add line to list (1,'Bloggs',pSal)
```

or you can do it like this

```
Do LIST.$clear()
```

Clear main & connected

Reversible:	YES	**Flag affected:**	NO
Parameters:	None		
Syntax:	Clear main & connected		

This command clears the memory of current records from the main file and any files connected to the main file. The windows are not automatically redrawn so you must follow it with a *Redraw window-name* command if you want the screen to reflect the current state of the buffer.

You can use *Clear main & connected* to release locked records to other users.

In the following example, the memory is cleared after Insert is canceled.

```
Prepare for insert
Enter data
If flag false
   Clear main & connected
   Quit method
End If
```

Clear main file

Reversible:	YES	**Flag affected:**	NO
Parameters:	None		
Syntax:	Clear main file		

This command clears the main file record from the current record buffer. The command does not clear the values taken from the other files.

The *Clear main file* command does not redraw the window so remember to include an explicit *Redraw window-name* command if you want the screen to reflect the contents of the buffer.

The Prepare for update mode is unaffected.

```
Clear main file
Redraw DataEntryWin
```

Clear method stack

Reversible:	NO	**Flag affected:**	NO
Parameters:	None		
Syntax:	Clear method stack		

This command cancels all currently executing methods and clears the method stack. A *Clear method stack* at the beginning of a method terminates all the methods in the chain which called the current method but without quitting the current method. $control() methods are not cleared.

As each method calls another, a return point is stored so that control can pass to the command following *Do method* or *Do code method* as the called method terminates. When the current method terminates, control returns to the method which was running before it was called.

The *Clear method stack* command clears all the return points and is used if the method commences a completely new operation. This command followed by a *Quit method* is the same as *Quit all methods*.

WARNING It is unwise to clear the method stack if local variables have been passed as fieldname parameters and you continue executing the current method. This will break all local variables on the stack.

```
; Calling method
Calculate CVAR1 as 1
Do method Message
; the following message never gets displayed
Do CVAR1+1
OK message {CVAR1=[CVAR1]}

; Message
```
Clear method stack
```
Do CVAR1+1
; This message prints CVAR1=2
OK message {CVAR1=[CVAR1]}
Quit method
```

Clear range of fields

Reversible: YES   **Flag affected:**   NO

Parameters: First data name
               Final data name

Syntax: Clear range of fields *first-data-name* to *final-data-name*

This command clears the specified range of fields from the current record buffer.

When used in a reversible block, the fields cleared are restored when the method terminates.

```
; declare local vars Char1, Char2, Char3 of Character type
Call procedure (Char1,Char2,Char3) {Initialize}
Clear range of fields Char1 to Char3
```

Clear search class

Reversible: YES   **Flag affected:**   NO

Parameters: None

Syntax: Clear search class

This command clears the current search class so you can print a report using all records. This also frees the memory required by the search class.

If you use *Clear search class* in a reversible block, the search class reverts to its former setting when the method terminates.

```
Set report name REPORT1
Set search name SITEMS_OS
Yes/No message {Do you want to use the search?}
If flag false
   Clear search class
End If
Print report (Use search)
```

Clear selected files

Reversible: YES   **Flag affected:** NO

Parameters: List of files

Syntax: Clear selected files [*{file1*[,*file2*]...*}*]

This command clears the current record buffer of records from the specified files. The command is particularly useful in a multi-user system where it may be necessary to remove only certain files so that they are not locked.

In the method editor, a list of files is displayed. You can Ctrl/Cmnd-click on the file names to select multiple names. If no file name or file list is specified, the command does nothing.

```
Clear selected files {fInvoices, fCustomers}
Redraw window instance wInvoices ;; clear the window
```

Clear sort fields

Reversible: YES   **Flag affected:** NO

Parameters: None

Syntax: Clear sort fields

This command removes the sort fields that are currently active. This enables the data to be printed without any sorting taking place. Alternatively, the command removes the current sort fields so you can specify new sort levels with *Set sort field*.

If you use *Clear sort fields* in a reversible block, the original sort values are restored when the method terminates.

```
Clear sort fields
Set sort field TITLE (Upper case)
Send to screen
Print report
```

Clear timer method

Reversible: YES   **Flag affected:**   NO

Parameters: None

Syntax: Clear timer method

This command clears or cancels the current timer method. Usually a timer method remains in operation until the library is closed or an error occurs. In a reversible block, the current timer method is restored when the method terminates.

```
; Set Timer
Set timer method (60 sec) MENU/Timer
OK message {Now play the minute waltz!}

; Timer
OK message {Timer method triggered once only}
Clear timer method
```

Close all designs

Reversible: NO   **Flag affected:**   YES

Parameters: None

Syntax: Close all designs

This command closes all the design windows currently open, including all Browser and instances of the method editor.

Close all windows

Reversible: NO   **Flag affected:**   NO

Parameters: None

Syntax: Close all windows

This command closes all open window instances in all open libraries, and automatically cancels any working message. The *Close all windows* command does not close private instances which do not belong to the current task.

```
Update files
Yes/No message {Have you deleted all for now?}
If flag true
   Close all windows
End If
```

or you can do it like this

```
Do $iwindows.$sendall($close)
```

Close check data log

Reversible: NO   **Flag affected:**   NO

Parameters: None

Syntax: Close check data log

This command closes the check data log if it is open. The command is not reversible and the
flag is not affected.

```
; Check Data
Check data (Check records) {MYDATA.ACCOUNT}
If flag true
   Yes/No message {View log?}
      If flag true
         Open check data log (Do not wait for user)
      End If
      ; leaves log window open
Else
   OK message {The check data file command could not be carried out,
   please make sure that only one user is logged onto the data file}
End If
; Close Log
Close check data log
```

Close client import file

Reversible: NO   **Flag affected:**   NO

Parameters: None

Syntax: Close client import file

This command closes the current client import file. After finishing with the current import file, be sure to close it using this command. If the file is already closed, nothing will happen. If it is open, OMNIS will ensure that all current data is flushed to the file and the file is closed. You *must* close the import file before you import the data into an OMNIS data file.

```
; Get file from VAX
Set client import file name {xprImportFile}
Open client import file
Perform SQL {select * from customers}
Retrieve rows to file
Close client import file
If flag true
   OK message {Comms with VAX successful}
Else
   OK message {Comms with VAX unsuccessful}
   Quit method
End If
; next, you import the fields from the file using a While loop with
; Import field from file. Not a good way of downloading, but the
; file could be imported by another program such as a spreadsheet.
```

Close cursor

Reversible: NO   **Flag affected:**   YES

Parameters: Cursor name

Syntax: Close cursor [*{cursor-name}*]

This command closes the named cursor. If *cursor-name* is not given the current cursor is closed. If there is only one remaining cursor in the session this command quits the session. It is the same as

```
Set current cursor {SQL_1}
Quit cursor(s) (Current)
```

Close data file

Reversible: NO **Flag affected:** YES

Parameters: Internal name (of open data file)

Syntax: Close data file [*{internal-name}*]

This command closes the open data file with the specified internal name, or closes all the open data files if no name is specified. It sets the flag if at least one data file is closed. It clears the flag and does nothing (that is, does not generate a runtime error) if the specified internal name does not correspond to an open data file.

```
If #UL > 4
   Close data file {Data1}
   Open data file {Data2}
   Set main file {fPictures}
```

Close DDE channel

Reversible: NO **Flag affected:** NO

Parameters: ☐ All channels

Syntax: Close DDE channel [*(All channels)*]

DDE command, OMNIS as client. This command closes the current channel. If you use the **All channels** option, all open DDE channels are closed. No error occurs if the current channel is not open.

```
Set DDE channel number {2}
Open DDE channel {OMNIS|COUNTRY}
If flag false
   OK message {Country library not running}
Else
   Do method TransferData
   Close DDE channel
   OK message {Update finished}
End If
```

Close design

Reversible: NO   **Flag affected:** YES

Parameters: Class name

Syntax: Close design *{class-name}*

This command closes the specified design window. Trying to close a class which is not open simply clears the flag.

```
Close design {MYMENU}
```

Close import file

Reversible: NO   **Flag affected:** NO

Parameters: None

Syntax: Close import file

This command closes the current import file. You should use it once the data has been read in.

```
Set import file name {Data}
Prepare for import from file {Delimited (commas)}
Import data {list1}
End import
Close import file
```

Close library

Reversible: NO   **Flag affected:** YES

Parameters: Internal name (default is all libraries)

Syntax: Close library [*{internal-name}*]

This command closes the open library file with the specified internal name, or closes all the open library files if no name is specified. It sets the flag if at least one library file is closed. It clears the flag and does nothing if the specified internal name does not correspond to an open library.

Note that the internal name for a library defaults to its physical file name from which the path and DOS extension has been removed. The *Open library* command also lets you specify the internal name (see the example below).

Closing a library closes all windows, reports, and menus belonging to that library which are open or installed. It also disposes of the CRBs for the file classes and class variables belonging to that library, closes all lookup files opened by that library, and if there is a running method from that library on the stack, clears the method stack. If the method stack is cleared, the command following the current executing command will not execute, and it is not possible to test the flag value returned from the command.

```
Open library MYLIB.LBR
Open library YOURLIB.LBR/ALIAS
Close library MYLIB
Close library ALIAS
```

Close lookup file

Reversible: NO   **Flag affected:** YES

Parameters: Lookup name

Syntax: Close lookup file [*{lookup-name}*]

This command closes the lookup file which matches the reference name given in the parameters. Each lookup file is given a reference label when it is opened. For example, "City" in:

```
Open lookup file {City/LOOKUP.DF1/FCITIES}
OK message {The city you require is [lookup('City',S1)]}
Close lookup file {City}
```

If the reference label given in the *Open lookup file* command is omitted, you can omit the lookup name in the *Close lookup file* command. If the specified lookup file is closed, the flag is set; if the lookup file doesn't exist, the flag is cleared.

Close other windows

Reversible: NO   **Flag affected:** NO

Parameters: None

Syntax: Close other windows

This command closes all but the top window instance. As window instances are not automatically closed in OMNIS, you can use this command to close all window instances except the top window instance. The *Close other windows* command does not close private instances which do not belong to the current task.

```
If sys(51)              ;; more than 1 window open?
   Close other windows  ;; close the others
End If
```

Close port

Reversible: NO   **Flag affected:**   NO

Parameters: None

Syntax: Close port

This command closes the current port. You should use it after the data has been transferred.

```
Set port name {1 (Modem port)}
Set port parameters {1200,n,7,2}
Repeat
   Import field from port into CVAR1
Until CVAR1='start data'
Do method ImportData
Close port
```

Close print or export file

Reversible: NO   **Flag affected:**   NO

Parameters: None

Syntax: Close print or export file

This command closes the current print or export file. You use it after the data has been written to the file. If the file is left open, subsequent data printed to the file is added to the end of the earlier data.

```
Send to file
Set print file name {MyText}
Set report name MyReport
Print report
Close print or export file
```

Close task instance

Reversible: NO   **Flag affected:**   NO

Parameters: Task instance name

Syntax: Close task instance *task-instance-name*

This command closes the specified task instance. Alternatively you can use the $close() method to close a task instance.

Close top window

Reversible: NO   **Flag affected:**   YES

Parameters: None

Syntax: Close top window

This command closes the top window instance. As window instances are not automatically closed in OMNIS, you can use this command to close the top window. No error occurs if there is no window open. This command clears the flag and does nothing if the top window is a private instance not belonging to the current task.

```
If sys(50) = wCustomerEntry  ;; check the top window
   Close top window
End If
```

or do it like this

```
Do $topwind.$close()
```

Close window

Reversible: NO   **Flag affected:**   YES

Parameters: Window instance name

Syntax: Close window *window-instance-name*

This command closes the specified window instance. *Close window* clears the flag and does nothing if the window is a private instance belonging to the current task. Alternatively you can use the $close() method to close a window instance.

```
Open window instance WEXPORT/winst1
Do method ExportData
Close window winst1
```

or you can do it like this

```
Do $iwindows.WINDOWINST.$close()
```

Close working message

Reversible: NO   **Flag affected:** NO

Parameters: None

Syntax: Close working message

This command closes the current working message. No error occurs if there is no working message displayed. Working messages close themselves when methods stop running and control returns to the user.

Once a working message is displayed, a call to another method leaves the message on the window. The message is not cleared automatically until the first method ends.

```
Working message {[LVAR1]}    ;; show the message
Do method PrintReport        ;; Working message still there
Close working message        ;; Working message gone
```

; Comment

Reversible: NO   **Flag affected:** NO

Parameters: Message (comment text)

Syntax: ; [*comment-text*]

This command adds a comment to a method. When entering a method, you can select the *Comment* command from the command list by typing a semicolon. OMNIS prefixes comments in your code with a semicolon ";". Also, you can add "in-line" comments to all commands at the bottom of the method editor screen. These are prefixed by two semicolons ";;". Comments have no effect in your code, but do slow down method execution. Therefore you should avoid placing comments inside for and repeat loops, or any code that is called repeatedly.

You can turn lines of code into comments by selecting them and using the **Comment Selected Lines** menu item in the debugger. Alternatively, you can press Ctrl/Cmnd-; (semicolon) to comment selected lines of code, or Ctrl/Cmnd-' (apostrophe) to uncomment code. Code will uncomment only if it has valid syntax, otherwise it will remain commented out. When you uncomment existing comments that also contain in-line comments, the in-line comments will be lost.

```
; here are some comments
; variable delay set by LVAR2
; adjust Until calculation to suit computer speed if required
Calculate LVAR1 as 1
Repeat         ;; this is an in-line comment
   Calculate LVAR1 as LVAR1+1
Until LVAR1 >= LVAR2*10
```

Commit current session

Reversible: NO   **Flag affected:**   YES

Parameters: None

Syntax: Commit current session

This command commits any changes made to the server tables following an *Execute SQL script*, that is, it issues an explicit instruction to make permanent any changes made to the server. It allows a finer control of transaction management than the default *Autocommit* action. With *Autocommit (On)*, OMNIS will commit all uncommitted statements after a successful *Execute SQL script* at the next *Begin SQL script*, *Logoff from host* or *Reset session* and rollback all unsuccessful statements after an unsuccessful *Execute SQL script*. You can use *Commit current session* only when *Autocommit* is off.

```
Autocommit (Off)
; must be off for Commit to work
Begin SQL script
SQL: Delete Elements where ATNO > 50
End SQL script
Execute SQL script
If flag true
   Commit current session
Else
   Rollback current session
End If
```

Autocommit waits for the next *Begin SQL script* before committing. Some servers close cursors on committing and dispose of the select table by waiting for the next *Begin SQL script*.

Context help

Reversible: NO   **Flag affected:**   YES

Parameters: Command mode
                 Help file name
                 Context id

Syntax: Context help *{**command-mode** [('help-file-name'[,context-id])]}*

This command provides context help to the user. You specify a command mode option, and depending on the mode you can specify the help file name and context id. The command mode options are constants listed in the Catalog.

kHelpContextMode
initiates context help mode, showing a '?' cursor.

kHelpContext ('helpfile name', context id)
opens a general help window for the topic specified.

kHelpContextPopup ('helpfile name', context id)
opens a popup help window for the topic specified.

kHelpContents ('helpfile name')
opens the help file at the contents page.

kHelpQuit ('helpfile name')
closes window mode help.

Some options do not work on all platforms.

To implement context help for an object or area, you set the help id as a decimal value in the $helpid property of a class or object, including windows, menus, and toolbars. You can make your custom help file which must be placed in the Help folder and the name entered in the library preference property $clib.$prefs.$helpfilename.

When the user clicks on an object with the help cursor or presses the F1/Help key, OMNIS looks for the help id. If it finds none for a window object, menu line, or toolbar control, it then looks in the next higher containing object.

```
Context help {kHelpContext ('MyHelp.hlp',56789)}
; shows help topic 56789 from MyHelp.hlp in a general help window
Context help {kHelpContextPopup ('MyHelp.hlp',56789)}
; shows help topic 56789 from MyHelp.hlp in a popup help window
Context help {kHelpContextMode}  ;; shows ? cursor and awaits click:
; when user clicks, shows a popup window with topic $cobj.$helpid
; from $clib.$prefs.$helpfilename located in the Help folder
```

Copy list definition

Reversible: NO **Flag affected:** YES

Parameters: List or row name
☐ Clear list

Syntax: Copy list definition *list-name* [*(Clear list)*]

This command redefines the column headings of the current list by copying the columns and data structure from the specified list. If the current list contains data and you do not clear the list, no change is made to the internal structure of the list; in this case, columns are neither added nor removed, merely renamed and the command is similar to *Redefine list*.

When the current list is empty or the **Clear list** option chosen, the command is the equivalent to 'Define the list so that it matches the specified list'.

```
Set current list LIST1
Define list {Field1Date, Field2Num, Field3Char}
Add line to list
Set current list LIST2
Define list {Field4Date, Field5Num, Field6Char}
Add line to list
; Now change list LIST2 definition to that of LIST1
Copy list definition LIST1 (Clear list)
```

or you can do it like this

```
Do LIST.$copydefinition(other LIST)
```

Copy to clipboard

Reversible: NO **Flag affected:** YES

Parameters: Field name

Syntax: Copy to clipboard [*field-name*]

This command copies the contents of the specified field or current selection and places it on the clipboard. In the case of a null selection when the cursor is merely flashing in a field and no characters are selected, the *Copy to clipboard* command will literally copy "nothing".

```
; copies one field to another then clears the first field
Copy to clipboard C_NAME
Paste from clipboard C_COMPANY (Redraw field)
Clear data C_NAME (Redraw field)
```

Create data file

Reversible:	NO	**Flag affected:**	YES

Parameters: ☐ Do not close other data
Data file name
Internal name (of new data file)

Syntax: Create data file [*(Do not close other data)*]
   {data-file-name [/*internal-name*]*}*

This command creates and opens a new and empty, single segment data file, which becomes the "current" data file. You can specify the path name of the file to be created and the internal name for the open data file.

The **Do not close other data** option lets you have multiple open data files. If you uncheck this option ,all open data files are closed even if the command fails.

If the disk file with the specified path name cannot be created (and opened), the flag is cleared. Otherwise, the flag is set if the data file is successfully created and opened.

WARNING If the file and path name is the same as an existing data file, all segments for that data file are deleted before the new file is created. If the data file was open, it is closed and deleted; a new and empty data file is then reopened.

```
Yes/No message {Do you wish to add a new company}
If flag true
   Do method InsertCompany
   Create data file (Do not close other data)[CoCode].df1/[CoCode]
   Open data file (Do not close other data)[CoCode].df1/[CoCode]
End If
```

or do it like this

```
Do $datas.$add(path,create,name)
```

Create library

Reversible: NO   **Flag affected:**   YES

Parameters: ☐ Do not close others
                   Library file name
                   Internal name

Syntax: Create library [*(Do not close others)*] *{library-name[/internal-name]}*

This command creates and opens a new library file. You specify the file name (and pathname if you wish) and internal name of the library. The internal name is an alias that you supply and use in your methods to refer to that library file.

If no internal name is specified, the default internal name is the disk name of the file with the path name and suffix removed. For example, under Windows the internal name for 'C:\MYFILES\TESTLIB.LBR' is TESTLIB. Similarly, under MacOS the internal name for 'hd:myfiles:testlib.lbr' is 'testlib'.

A **Do not close others** option can also be specified so that you can open multiple libraries. If the disk file with the specified path name cannot be created (and opened), the flag is cleared and no libraries are closed. Otherwise, if the option is not specified, all other open libraries are closed (see *Close library* for the consequences of closing a library).

WARNING If the path name is the same as an existing library, the existing library is overwritten. If the existing library is open, it is closed and deleted and a new, empty library is opened.

```
Switch sys(6)='M'
   Case kTrue
      Create library {HD200:OMNIS:MyLib/MyAlias}
   Default
      Create library {C:\OMNIS\MYLIB.LBR/MyAlias}
End Switch
; or do it like this
Do $libs.$add(path,create,name)
```

Cut to clipboard

Reversible: NO   **Flag affected:**   YES

Parameters: Field name
                  ☐ Redraw field
                  ☐ All windows

Syntax: Cut to clipboard [*field-name*] [*(*[*Redraw field*][*,All windows*]*)*]

This command cuts the contents of the specified field or current selection and places it on the clipboard. In the case of a null selection when the cursor is merely flashing in a field and no characters are selected, *Cut to clipboard* will literally cut "nothing".

```
Cut to clipboard FIELD1 (Redraw field)
Paste from clipboard FIELD2 (Redraw field)
```

Declare cursor

Reversible: NO   **Flag affected:**   YES

Parameters: Cursor name
                  SQL script

Syntax: Declare cursor *cursor-name* for *sql-script*

This command defines a cursor for the current session and lets you send a SQL statement. The *Declare cursor* command takes a new *cursor-name* and a *SQL-script* for that cursor that is to be executed for the current session. This command opens a session for the named cursor if one does not exist.

```
Declare cursor EMP_CURSOR for SELECT * FROM EMP FOR UPDATE
Open cursor { EMP_CURSOR }
```

This command is the same as:

```
Set current cursor
Begin SQL script
SQL: SELECT * from EMP FOR UPDATE
End SQL script
```

Default

Reversible: NO **Flag affected:** NO

Parameters: None

Syntax: Default

This command marks the block of commands to be run when there is no matching case in a *Switch* statement. When a *Switch–Case* construct is used, the *Default* command marks the start of a block of commands that are executed if none of the preceding *Case* statements are executed.

```
Switch cTEXT
   Case 'Fred'
      OK message {Fred}
      Break to end of switch
      OK message {I never execute}
   Case 'Jim'
      OK message {Got Jim}
      Break to end of switch
      OK message {I never execute}
   Default
      OK message {Neither Fred nor Jim}
End Switch
```

Define list

Reversible:	YES	**Flag affected:**	NO

Parameters: List of variables, file class field names, or file class name

Syntax: Define list *{variable|field1*[,*variable|field2*]*...}*

This command defines the variables or file class field names to be used as the column definitions for the current list; it should follow *Set current list*. The variables or fields used in the definition also describe the data type and length for each column of data held. This command clears the definition and data in the current list. When reversed, the contents and definition of the current list are restored to their former values. Duplicate names are ignored in your list of variables or fields.

```
; declare variable LIST1 of List type
Set current list LIST1
Define list {Field1,Field2,Field3}
; defines columns Field1, Field2, Field3 for the current list
Define list {Field1,Field2,Field3,Field1}
; same as above, ignores duplicate reference to Field1
Define list {FileName}
; includes all the fields in the file class
```

Alternatively you can use the $define() method to define a list; in this case you don't need to set the current list before executing this method

```
Do LIST1.$define(var1,var2,var3)
; defines columns var1, var2, var3 for LIST1

Do LIST1.$definefromtable(tablename)
; defines a list from a table class
```

Fixed Length Columns

Normally the length of a column is set by the type or length of the variable or field defined for the column, therefore the column length for a default Character variable would be 10 million. However, when you define the list you can truncate the data stored in a column using the notation VariableName/N. For example, to use the first 10 characters of the variable CVAR1 in column 1 use

```
Define list {CVAR1/10,CVAR2,CVAR3}
```

Define list from SQL class

Reversible:	YES	**Flag affected:**	NO

Parameters: Table name
Parameters list

Syntax: Define list from SQL class *sql-class-name*
[*(parameter1*[,*parameter2*]...*)*]

This command defines the column names and data types for the current list based on the specified schema, query, or table class. You can use it to redefine the format of the current list, but usually it should follow a *Set current list* command. When reversed, the contents and definition of the list are restored to their former values.

```
Set current list cList
Define list from SQL class {MySchema}
```

or do it this way

```
Do LIST.$definefromsqlclass(MySchema)
```

Delete

Reversible:	NO	**Flag affected:**	YES

Parameters: None

Syntax: Delete

This command deletes the current record in the main file without prompting the user to confirm the command, so you should use it with caution. The flag is set if the record is deleted, or cleared if there is no main file record. The flag is also cleared if the **Do not wait for semaphores** option is on and the record is locked.

The following example deletes records selected by a search class.

```
Set main file {f_clients}
Set search name S_no_money
Find first on SURNAME (Exact match)
Repeat
   Delete
   Next
Until flag false
```

This example checks the semaphore and tells the user if the record is locked:

```
Do not wait for semaphores
Delete
If flag false
   OK message (Sound bell) {Record in use and can't be deleted}
End If
```

Delete class

Reversible:　NO　　　　**Flag affected:**　　YES

Parameters:　Class name

Syntax:　　　Delete class *{class-name}*

This command deletes the specified library class. It is not possible to delete a file class, an installed menu or an open window. It is also not possible to delete a class if one of its methods is currently executing, that is, if it is somewhere on the method stack. Deleting a class does not reduce the library file size. It does, however, create free library file blocks so that creation of another class may be possible without further increase in library size. Errors, such as attempting to delete a name that does not exist, simply clear the flag and display an error message.

Delete class {S_User}

Delete client import file

Reversible: NO   **Flag affected:**   NO
Parameters: Client import file name
Syntax: Delete client import file *{file-name}*

This command deletes the current import file that was named with *Set client import file name*. It removes the file if possible but does not warn you if the file is open or if it doesn't exist. You are responsible for deciding if the client import file name set previously is the correct one.

```
Set client import file name {xprImportFile}
Open client import file
Begin SQL script
SQL: select cust_name, cust_city, credit_line from customer
End SQL script
Execute SQL script
Retrieve rows to file
Close client import file
Do method UseModem
If flag true
   Delete client import file {xprimportFile}
End If
```

Delete data

Reversible: NO   **Flag affected:**   YES
Parameters: File class name (that is, slot name)
Syntax: Delete data *{file-name}*

This command deletes all the data and indexes for a specified file in a data file. The data and indexes for a file class are called a "slot". You can delete a slot only if and when one user is logged onto the data file.

If a specified file name does not include a data file name as part of the notation, the default data file for that file is assumed. If the file is closed or memory-only, the command does not execute and returns flag false. If you are not running in single user mode, the command automatically tests that only one user is using the data file (the command fails with the flag false if this is not true), and further users are prevented from logging onto the data until the command completes.

If a working message with a count is open while the command is executing, the count will be incremented at regular intervals. The command may take a long time to execute, and it is not possible to cancel execution even if a working message with cancel box is open. The

command sets the flag if it completes successfully and clears the flag otherwise. It is not reversible.

```
Delete data {MYDATA.FILE1}
If flag true
   OK message {Data for FILE1 deleted}
Else
   OK message {Data could not be deleted; too many users}
End If
```

or do it like this

```
Do $datas.DATAFILE.$slots.SLOTNAME.$delete()
```

Delete line in list

Reversible: NO **Flag affected:** YES

Parameters: Line number (can be a calculation, default is current line)

Syntax: Delete line in list [*{line-number}*]

This command deletes the specified line of the current list by moving all the lines below the specified line up one line. If the line number is not specified or if it evaluates to 0, the current line *LIST.$line* is deleted. The line in a list selected by the user can determine the value of LIST.$line and is the line deleted if no parameters are specified. LIST.$line is unchanged by the command unless it was the final line and that line is deleted; in this case LIST.$line is set to the new final line number. The command never releases any of the memory used by the list.

The flag is cleared if the list is empty or if the line is beyond the current end of the list; otherwise, the flag is set.

This example deletes the first five lines of the current list. *LVAR1* is used as a counter; each time through the loop, the first line is deleted and all the following lines move up one line

```
Calculate LVAR1 as 5  ;; LVAR1 is the loop counter
Calculate LIST.$line as 1
Repeat
   Delete line in list  ;; deletes line number LIST.$line
   Do LVAR1-1
Until LVAR1=0
; LIST.$line is still equal to 1
```

or do it like this

```
Do LIST.$remove(row number)
```

Delete selected lines

Reversible: NO   **Flag affected:**   YES

Parameters: None

Syntax: Delete selected lines

This command deletes all the selected lines from the current list. This is carried out in memory and has no effect on the lists stored in the data file unless a *Prepare for Edit/Update* command is performed. *LIST.$line* is unaffected unless it is left at a value beyond the end of the list, in which case it is set to *LIST.$linecount*. The following example results in a list of one line, with a value of 3 stored in it:

```
Set current list LIST1
Define list {LVAR1}
Calculate LVAR1 as 1
Repeat
   Add line to list
   ;  Adds lines to end of list
   Calculate LVAR1 as LVAR1+1
Until LVAR1=6
Select list line(s) (All lines) ;;  selects all the lines
Invert selection for line(s) {3}
Delete selected lines             ;;  deletes all but line 3
Redraw lists
```

Delete with confirmation

Reversible: NO   **Flag affected:**   YES

Parameters: Message (text)

Syntax: Delete with confirmation [*{message}*]

This command displays a message asking the user to confirm or cancel the deletion and, if confirmation is granted, deletes the current record in the main file. An error is reported if there is no main file.

If a message is not specified, OMNIS uses a default message. The message can contain square-bracket notation which is evaluated when the command is executed. If the current record is deleted, the flag is set, otherwise it is cleared. If the **Do not wait for semaphores** option is on, the flag is cleared if the record is locked.

This example allows selected records in the main file to be deleted:

```
Set main file {f_clients}
Set search name S_no_money
Open window instance W_show_balance
Find first on SURNAME (Use search)
While flag true
   Redraw W_show_balance
   Delete with confirmation {Delete [SURNAME]'s record?}
   Next (Use search)
End While
```

Describe cursors

Reversible:	NO	**Flag affected:**	NO

Parameters: None

Syntax: Describe cursors

This command creates a select table that lists all the available cursors within the current session. There is one row for each open cursor. The resulting list has one column only:

Column 1

Cursor name

You can read the select table into a list using *Build list from select table*. When you create a session with multiple cursors, you can build a list of the cursors as follows:

```
; declare class variable CLIST of List type
; declare class variable COL of Character type
Set current list CLIST
Define list {COL}
Describe cursors
Build list from select table
```

Describe database

Reversible: NO   **Flag affected:**   YES

Parameters: Tables or Views option

Syntax: Describe database *(Tables|Views)*

This command creates a select table for either Tables or Views available to the current session.

Tables

When the **Tables** option is specified, the *Describe database* command creates a select table with one row for each Table available to the current session.

Column 1

Table name

A data dictionary query is sent to the server and you can read the select table into a list using *Build list from select table*. This example builds a select table of available Tables and reads it into the current list:

```
; declare class variable TLIST of List type
Set current list TLIST
Define list {#S5}
Describe database (Tables)
Build list from select table
```

Views

When the **Views** option is specified, *Describe database* creates a select table with one row for each View available to the current session.

Column 1

View name

This example builds a list of available views and creates special file classes within OMNIS so that server data can be mapped to them:

```
Set current list LVIEWS ;; LVIEWS contains a column called VIEWNAME
Describe database (Views)
Build list from select table
For each line in list from 1 to $linecount step 1
   Describe server table (Columns) {[lst(VIEWNAME)]}
   Make schema from server table {[lst(VIEWNAME)]}
End For
```

Describe results

Reversible: NO   **Flag affected:** YES
Parameters: None
Syntax: Describe results

This command creates a list which describes the columns in the select table. The data returned by *Describe results* is placed automatically into the current list.

When processing a select table, you can use *Describe results* at any point to create a list which describes the columns in the select table. This is done without disturbing the select table and no fetches are done. Also, no calls to *Build list from select table* or *Fetch next row* are required.

The information returned by *Describe results* is as follows:

Col   **Column description**

  1   Column name

  2   OMNIS data type for each column in the select table

  3   Defined or maximum data length (for Character columns only)

  4   Number of dp (for numeric columns); empty for floating numbers

  5   NULL or NOTNULL (for SQL Server only.)

This example builds a simple select table and uses *Describe results* to make a list of columns and their types:

```
Set current list LIST_RESULTS
Define list {COL1..COL7}
Begin SQL script
SQL: Select Name, Town, Tel from Clients
SQL: Where Town = 'London'
End SQL script
Execute SQL script
Describe results
```

Describe server table

Reversible: NO   **Flag affected:**   YES

Parameters: Columns or Indexes option
                    Server table name

Syntax: Describe server table *(Columns|Indexes) {server-table-name}*

This command creates a select table which describes the Columns or Indexes for the specified remote server table.

Columns

When you specify the **Columns** option, the *Describe server table* command creates a select table with one row for each Column of the specified remote server table. The information about the Columns of a remote server table is listed as follows:

Col   Description

1   Column name

2   Standard SQL data type for each column

3   Column length (for Char columns)

4   Number of decimal places (for numeric cols only), empty for floating Nos.

5   NULL or NOTNULL; where available

6   Empty; reserved for index info

7   Description for the column; where available

You can obtain this information by issuing a query to the server data dictionary and converting the base data types to OMNIS data types.

```
; declare class variable CLIST of List type
; declare class vars COL1, COL2, COL3 of Character type
Set current list CLIST
Define list {COL1,COL2,COL3}
Describe server table (Columns) { MyTable }
Build list from select table
OK message {There are [CLIST.$linecount] columns in the table}
```

You can make a schema class based on the select table created by the *Describe server table (Columns)* command using the *Make schema from server table* command, as follows

```
Describe server table (Columns) {TableName}
Make schema class from server table {SchemaName}
```

Indexes

When you specify the **Indexes** option, the *Describe server table* command creates a select table that lists the unique indexes for the specified remote server table.

Col	Description
1	Unique indexed column name
2	Name of the index used for the column (in column 1 of the list)
3	Numeric position of the column within a composite index (defaults to 1 for non-composite indexes)

You can obtain a list of non-unique indexes by adding the /N switch to the command, for example

Describe server table (Indexes) {MyTable/N}

You can obtain a list of all indexes by adding the /A switch to the command. The default switch /U lists the unique indexes, and can be left in or out of the command. This command lets you write general purpose data handling methods, for example

```
; declare class variable KEY_LIST of List type
; declare class variable KEY_NAME of Character type
; declare parameter variable TABLE of Character type
; Pass name of table to this method
Set current list KEY_LIST
Define list {KEY_NAME}
Describe server table (Indexes) {[TABLE]}
Build list from select table
Begin SQL script
SQL: Delete from CUSTOMERS where wherenames(^KEY_LIST)
End SQL script
Execute SQL script
If flag false
   OK message {Can't delete row for [TABLE]}
Else
   OK message {Row deleted}
End If
; Assumes that you have set up a specification
; for the record to delete in the OMNIS field
```

Describe sessions

Reversible: NO   **Flag affected:** NO

Parameters: None

Syntax: Describe sessions

This command creates a select table that lists all the available sessions. There is one row for each open session, that is, one row per cursor/session combination. You can read the select table into a list using *Build list from select table*. The columns in the select table are:

Column 1	**Column 2**	**Column 3**
Cursor name	Session name	Remote database

When you create multiple sessions, you can build a list of them as follows:

```
; declare class variable CLIST of List type
; declare class vars COL1, COL2, COL3 of Character type
Set current list CLIST
Define list {COL1,COL2,COL3}
Describe sessions
Build list from select table
```

Deselect list line(s)

Reversible: NO   **Flag affected:**   YES

Parameters: Line number (can be a calculation, default is current line)
           ☐ All lines

Syntax: Deselect list line(s) [*(All lines)*] [*{line-number}*]

This command deselects the specified list line. The specified line of the current list is deselected and is shown without highlight on a window list field when redrawn. You can specify the line number as a calculation. The **All lines** option deselects all lines of the current list. When a list is saved in the data file, the line selection state is stored. The following example selects all but the middle line of the list:

```
Set current list LIST1
Define list {LVAR1}
Calculate LVAR1 as 1
Repeat
   Add line to list
   Calculate LVAR1 as LVAR1+1
Until LVAR1=6
Select list line(s) (All lines)
Deselect list line(s) {LIST1.$linecount/2} ;; rounds to 3
; Or we could use Deselect list line(s) 3
Redraw lists
```

or do it like this

```
Do LIST.$selected.$assign(kfalse)
```

Disable all menus and toolbars

Reversible: YES   **Flag affected:**   NO

Parameters: None

Syntax: Disable all menus and toolbars

This command disables all built-in OMNIS menus currently installed on the menu bar, except **Edit,** and all toolbars including floating toolbars. OMNIS also disables any menus and toolbars installed after the *Disable* command is executed. The menu lines and toolbar controls are grayed out and cannot be selected. The Apple, Help, and Application menus under MacOS are unaffected.

WARNING You should use *Disable all menus and toolbars* in a reversible block. Otherwise, if you disable all menus using this command and a *Quit all methods* command is

executed or an error occurs, the computer may have to be switched off and the program restarted to reinstate the menus.

You can reverse this command with the *Enable all menus and toolbars* command.

```
Begin reversible block
   Disable all menus and toolbars
End reversible block
; do something with standard menus disabled
; menus are enabled when method ends
```

or do it like this

```
Do $imenus.$sendall($disable)
```

Disable automatic publications

Reversible:	YES	**Flag affected:**	YES
Parameters:	None		
Syntax:	Disable automatic publications		

This command turns off the automatic publication of all published fields. It affects only those fields which have been published automatically, that is, whose publisher options have been set up. The command can be reversed by using *Enable automatic publications* and if used within a reversible block, the *Disable automatic publications* command is reversed, restoring the automatic publications to their former state when the method terminates.

When a library is launched, automatic publications are enabled. The command clears the flag and does nothing if System 7 is not running. If System 7 is running, the command sets the flag.

```
Publish field CNAME {HD80:Public:Sales-Name}
Publish field CTOTAL {HD80:Public:Sales-Total}
Set publisher options (Publish on save) {CNAME,CTOTAL}
..
Disable automatic publications
Prepare for edit
Enter data
Update files if flag set
Enable automatic publications
```

Disable automatic subscriptions

Reversible: YES   **Flag affected:**   YES

Parameters: None

Syntax: Disable automatic subscriptions

This command turns off the automatic update of all subscribed fields. It affects only those fields which have been subscribed automatically, that is, whose subscriber options have been set up. The command can be reversed by using *Enable automatic subscriptions* and if used within a reversible block, the *Disable automatic subscriptions* command is reversed, restoring the automatic publications to their former state, when the method terminates. The command clears the flag and does nothing if System 7 is not running. If System 7 is running, the command sets the flag.

When a library is launched, automatic subscriptions are enabled.

```
Subscribe field CNAME {HD80:Public:Sales-Name}
Subscribe field CTOTAL {HD80:Public:Sales-Total}
Set subscriber options (Subscribe automatically) {CNAME,CTOTAL}
..
Disable automatic subscriptions
Prepare for edit
Enter data
Update files if flag set
Enable automatic subscriptions
```

Disable cancel test at loops

Reversible: YES   **Flag affected:**   NO

Parameters: None

Syntax: Disable cancel test at loops

This command prevents OMNIS from quitting a loop when the user presses Ctrl-Break under Windows or Cmnd-period under MacOS. Normally, when a *Repeat* or *While* command is executed, OMNIS tests for a break command. Also, periodic Cancel tests are performed during lengthy commands such as searches and *Build lists*. You use *Disable cancel test at loops* to turn off the test when updating files, for example.

Cancel keys and clicks on a Cancel pushbutton are ignored even if a working message with a Cancel box is included in the method but you can use *If canceled* to include an explicit check for Cancel within the loop. The command is reversed with *Enable cancel test at loops*, whenever a new library is selected, or if placed in a reversible block.

```
; This deletes all records where code = 'ABC'
Calculate CODE as 'ABC'
Find on CODE
Disable cancel test at loops
While flag true
   Working message (Repeat count)
   Delete
   Next on CODE (Exact match)
End While
```

Disable enter & escape keys

Reversible: YES   **Flag affected:**   NO
Parameters: None
Syntax: Disable enter & escape keys

This command disables the Enter and Escape keys or Cmnd-period under MacOS, that is, it disables the keyboard equivalents of the OK and Cancel pushbuttons. You can use it during enter data mode to prevent the user from prematurely updating records by hitting the Enter key, to attempt to start a new line, for example. The option will remain set until either it is reversed with an *Enable* command, a new library is selected, or it is reversed as part of a reversible block.

Before using this command in a method that initiates an *Enter data* command, ensure that the user has some way of ending data entry, that is, by installing an OK and a Cancel pushbutton, or by using a $control() method that detects the end of data entry.

```
Begin reversible block
   Disable enter & escape keys
End reversible block
Set main file {FCLIENTS}
Prepare for edit
Enter data
Update files if flag set
```

Disable fields

Reversible: YES   **Flag affected:**   NO
Parameters: Field name or list of field names
Syntax: Disable fields *{field1*[,*field2,...*]*}*

This command disables the specified field or list of fields, making them inactive during *Enter data* and *Prompted find*. Thus the data entry cursor skips a disabled entry field when in data entry mode, find, and so on, and disabled pushbuttons cannot be clicked. If an entry

field with scroll bar is disabled, you can tab to it but not change the data. You can reverse *Disable fields* or enable a display field using *Enable fields*.

```
Begin reversible block
   Disable fields {Entry1,Entry2}
End reversible block
Do method CheckCredit
; method ends and fields are enabled
```

or to disable all the fields on the current window

```
Do $cwind.$objs.$sendall($ref.$enabled.$assign(kFalse))
```

Disable menu line

Reversible: YES   **Flag affected:**   NO

Parameters: Menu instance name
                  Menu line number

Syntax: Disable menu line *menu-instance-name/menu-line-number*

This command disables the specified line of a menu instance, that is, the menu line becomes grayed out and cannot be selected. You specify the *menu-instance-name* and the number of the menu line you want to disable. You can disable a complete menu instance by disabling line zero, that is the menu title.

You can reverse *Disable menu line* with the *Enable menu line* command or, you can use it in a reversible block. Nothing happens if the specified menu instance is not installed on the menu bar.

```
Install menu STARTUP/minst1
Begin reversible block
   Test for menu installed {minst1}
   If flag true
      Disable menu line minst1/1
   End If
   Do method ProcessData
End reversible block
; now menu line is enabled
```

or do it like this

```
Do $menus.MENU.$obj.LINE.$enabled(kfalse)
```

Disable receiving of Apple events

Reversible: YES   **Flag affected:** NO

Parameters: □ Disable compulsory events

Syntax: Disable receiving of Apple events [*(Disable compulsory events)*]

This command prevents OMNIS libraries from being sent Apple events. When you launch an OMNIS library, receiving of Apple events is disabled by default: you use this command to reverse the *Enable receiving of Apple events* command. All Apple events are disabled except the four compulsory events (*Open application*, *Quit application*, *Open documents* and *Print documents)*, unless you check the **Disable compulsory events** option.

When *received by* OMNIS, the compulsory events do the following:

❑ *Open application* launches OMNIS,

❑ *Quit application* quits OMNIS,

❑ *Open document* loads a library or report,

❑ *Print document* opens a library, and prompts the user for a report to print.

```
Disable receiving of Apple events
Prepare for edit
Enter data
Update files if flag set
```

Disable relational finds

Reversible: YES   **Flag affected:** NO
Parameters: None
Syntax: Disable relational finds

This command reverses the action of *Enable relational finds*. The default situation is reinstated, that is, the main file and its connected parent files are joined using the OMNIS connection.

```
Set main file {INVOICES}
Enable relational finds {CUSTOMERS, INVOICES}
Set search as calculation {C_CODE = INVC_CODE}
Set sort field INVNUMBER
Find first (Use search, use sort)
While flag true
   ; process invoices
   Next
End While
Disable relational finds
```

Do

Reversible: NO   **Flag affected:** NO
Parameters: Calculation
              Return field
Syntax: Do *calculation* [returns *return-field*]

This command executes the specified *calculation,* which is typically some notation that operates on a particular object or part of your library. It returns a value if you specify a *return-field*, which can be a variable of any type.

```
Do $clib.$windows.win1.$open('win1',kWindowMaximize)
; opens a window instance maximized
Do $winds.winst1.$bringtofront()
; brings a window instance to the front
Do $topwind.$objs.EntryField1.$redraw()
; redraws EntryField1 on the top window
Do $winds.$sendall($ref.$objs.EntryField1.$redraw())
; redraws EntryField1 on all window instances
```

The optional return field can be used to check whether the operation succeeded.

You also use *Do* to assign a property

```
Do $cobj.textcolor.$assign(kRed) Returns myFlag
```

or to return an operation on a variable

```
Do iNum+5 Returns iNum
Do $clib.$windows.$makelist($ref.$name) Returns cWindowList
```

Note that where the return field is an item reference, the command sets the reference but does not assign to it: you must do this with *Calculate* or *Do Itemref.$assign(value)*.

Do code method

Reversible:	NO	**Flag affected:**	NO
Parameters:	Code class name		
	Method name		
	Parameters list		
	Return field		
Syntax:	Do code method ***code-class-name***/***method-name***		
	[(*parameter1*[,*parameter2*]...)] [returns *return-field*]		

This command runs the specified code class method, and accepts a value back from the called method. The specified *method-name* must be in the code class *code-class-name*. The command accepts a value back from the called method if you specify a *return-field*. The return field can be a variable of any type.

When a code class method is executed using this command, control is passed to the called method but the value of $cinst is unchanged, therefore the code in the code class method can refer to $cinst. When the code class method has executed, control passes back to the original executing method. The current task is not affected by execution moving to the code class.

For example, the following method is placed behind a toolbar button and runs a general purpose method PrintInvoice in a code class called PrintMethods.

```
On evClick  ;; toolbar button method
   Do code method PrintMethods/PrintInvoice
```

You could use the same code class method from a menu, such as

```
; line method for menu class
Do code method PrintMethods/PrintInvoice
```

Passing Parameters

You can include a list of parameters with the *Do code method* command which are passed to the called method. For example, the following command calls the method named EndOfMonth in the CINVOICE code class, and passes the current values in InvDate, InvTotal, and the result of the calculation InvNet*15/100. The values are received by the parameter variables in the order they appear in the variable pane of the called method. If the called method has fewer parameters than values passed to it, the extra values are ignored.

```
Do code method CINVOICE/EndOfMonth (InvDate,InvTotal,InvNet*15/100)
; EndOfMMonth method
; Declare Parameter vars P1, P2, and P3 to receive values
```

Note that where the return field is an item reference, the command sets the reference but does not assign to it: you must do this with *Calculate* or *Do Itemref.$assign(value).*

Do default

Reversible: NO   **Flag affected:** YES

Parameters: Return field

Syntax: Do default [returns *return-field*]

This command is used within the code for a custom attribute, and performs the default behavior for the built-in attribute with the same name as a custom attribute. *Do default* sets the flag if some built-in processing for the attribute exists.

For example, you could define a custom attribute, called $horzscroll.$assign, that assigns a horizontal scroll bar. If the window is over 20 pixels wide the default behavior for $horzscroll.$assign is called, that is, a scroll bar is added, otherwise a scroll bar is not allowed.

```
; $horzscroll.$assign
; Declare parameter Switchon of type Boolean
If Switchon & $cinst.$width < 20
   Quit method   ;; window too narrow for a scroll bar
Else
   Do default    ;; assign a horz scroll bar
End If
```

Note that where the return field is an item reference, the command sets the reference but does not assign to it: you must do this with *Calculate* or *Do Itemref.$assign(value).*

Do inherited

Reversible: NO   **Flag affected:**   YES

Parameters: None

Syntax: Do inherited

This command runs the superclass method with the same name as the currently executing method in the current subclass. For example, you can use *Do inherited* in the $construct() method of a subclass to execute the $construct() method of its superclass. Similarly you can run the $destruct() method in a superclass from a subclass.

```
; $construct method
Do inherited      ;; do superclass construct
Do ......         ;; make your own settings

$destruct
Do ......         ;; reverse your own settings
Do inherited      ;; do superclass destruct
```

The flag is set if a method with the name of the current method is found in one of the superclasses.

Normally a method in the current class takes precedence, but the inherited version of the method can be executed using the Do inherited command. Alternatively you can use the $inherited property with a method name

```
; $init method
Do inherited   ;; this is the same as Do $inherited.$init
```

Do method

Reversible:	NO	**Flag affected:**	NO

Parameters: Method name
Parameters list
Return field

Syntax: Do method ***method-name*** [*(parameter1*[,*parameter2*]*...)*]
                [returns *return-field*]

This command runs the specified method in the current class, and accepts a value back from the called method. If you use the *Do method* command in a field or line method, OMNIS searches for the specified method in the field or line methods for the class, and then searches in the class methods. If the specified method is not found there is an error.

The command accepts a value back from the recipient or receiving method if you specify a *return-field*, which can be a variable of any type. Note that where the return field is an item reference, the command sets the reference but does not assign to it: you must do this with *Calculate* or *Do Itemref.$assign(value)*.

When another method is executed using this command, control is passed to the called method. When the called method has executed, control passes back to the original executing method. Note that you should use *Do code method* if you want to run a method in a code class, that is, a method outside the current class.

```
Do method ProcessData
; OMNIS calls the method named 'ProcessData', then returns
; here and continues execution in this method
```

You can use the notation for the called method, for example

```
Do method $cclass.$methods.//ProcessData//
```

You can use $cinst, $cfield, and $ctask to specify a method in the current instance, field, or task. For example

```
Do method $cinst.methodname
Do method $cfield.methodname
Do method $ctask.methodname
```

Passing Parameters

You can include a list of parameters with *Do method* which are passed to the called method. For example, the following command calls the method named EndOfMonth and passes the current values in INV_DATE, INV_TOTAL, and INV_NET*15/100. The parameters are taken in the order they appear in the parameter list and placed in the parameter variables in the called method.

```
Do method EndOfMonth (INV_DATE,INV_TOTAL,INV_NET*15/100)

; EndOfMMonth
; Declare Parameter vars P1 and P2
; now do something with these values...
; note that in this case the third parameter is ignored
```

Passing by Reference

You can pass a reference to a field by using the special parameter variable type Field reference. This means that the called method can make changes to the field passed to it. For example

```
Do method SetParameters (NUMBER1)

; SetParameters method
; Declare parameter var P1 with type Field reference
Calculate P1 as 25
; NUMBER1 and P1 are now changed to 25
```

Recursion

OMNIS allows a method to call itself, but will eventually run out of memory. For example

```
; Loop method
; command lines
Do method Loop
```

Do not flush data

Reversible: YES   **Flag affected:** YES

Parameters: None

Syntax: Do not flush data

This command causes all data file operations to be carried out without writing the changed data to disk at each *Update files* or *Delete*. The command is designed to speed up data file operations when the user is prepared to take the extra risk of data loss.

The command operates best when there is a single user logged into the data file. It is unlikely to cause speed increase if the data is on a network volume (that is, shared by several users).

If you use *Test for only one user* at the beginning of the method, further users are prevented from opening the data file until the method terminates.

The command sets the flag if the state of the 'Do not flush data' mode is changed. When placed in a reversible block, the command restores the previous state of the 'Do not flush' flag upon the termination of the method.

```
; fast import via window
Test for only one user
If flag true
   Do not flush data
   Drop indexes
End If
Open window instance W_IMPORT
Set current list List1
Prompt for import file
Prepare for import from file {Delimited(tabs)}
Import data {List1}
End import
Close import file
For each line in list from 1 to $linecount
   Prepare for insert   ;; transfer list to file
   Load from list
   Update files
End For
Flush data now  ;; writes the data immediately to disk
Rebuild indexes
Flush data      ;; Changes mode back to 'Flush data'
```

Do not wait for semaphores

Reversible: YES   **Flag affected:**   NO

Parameters: None

Syntax: Do not wait for semaphores

This command causes *all* commands which set semaphores to return with a flag clear if the semaphore is not available.

If *Do not wait for semaphores* is run first in a method, it will ensure that any subsequent commands that lock records, such as *Prepare for...*, *Update* commands, do not wait for records to be released. It causes the command to return a flag false and control to return immediately to the method, if a record is locked.

Semaphores

Semaphores are internal flags or indicators set in the data file to show other users that the record has been required elsewhere for editing. Semaphores are only set when running in multi-user mode, that is, the data file is located on a networked server, a Mac volume or on a DOS machine on which SHARE has been run.

The commands which set semaphores are *Prepare for edit*, *Prepare for insert*, *Update files* and *Delete*, and also, if prepare for update mode is on and the file acted upon is Read/Write, *Single file find*, *Load connected records*, *Set read/write files*, all types of *Find*, *Next*, and *Previous*. *Update files* commands lock the whole data file while indexes are re-sorted.

The *Edit/Insert* commands always wait for a semaphore, as do automatic find entry fields.

```
Do not wait for semaphores
Prepare for edit
If flag true
   Set read-only files {FLOOKUP}
   Single file find on LO_CODE (Exact match)
   If flag false
      OK message {Can't find record [LO_CODE]}
      Cancel prepare for update
      Quit method kFalse
   End If
   Repeat
      Working message (Cancel) {Locking record [LO_CODE]}
      Set read/write files {FLOOKUP}
   Until flag true
   Repeat
      Update files
   Until flag true
End If
```

This method illustrates how any command which causes a change in record locking requirements can fail (returning flag false). If, when in 'Prepare for' mode, a *Single file find* cannot lock the new record, it returns a flag false. This could mean either that the record could not be found, or that it was in use by another workstation. For this reason, it was made read-only before the *Single file find* and then changed to read/write. Note also that *Update files* can fail if the file cannot be locked while the indexes are re-sorted, that is:

```
Repeat
   Update files
Until flag true
```

Do redirect

Reversible: NO   **Flag affected:**   YES

Parameters: Notation for the object
                Return field

Syntax: Do redirect *notation* [returns *return-field*]

This command redirects execution from a custom attribute to any other method. You specify the notation (or a calculation which evaluates to a reference to an object) for the recipient. The recipient of the attribute being processed is $crecipient. The flag is set if the recipient exists and handles the attribute with a built-in or custom attribute. For example

```
$method1
Do $cwind.$setup  ;; the call to $setup in current instance ..


$setup   ;; for current instance
Do redirect $cwind.$objs.1005              ;; .. is diverted ..


$setup  ;; for object 1005               ;; .. to here
```

Drop indexes

Reversible: NO   **Flag affected:**   YES

Parameters: File class name

Syntax: Drop indexes *{file-name}*

This command deletes all the indexes for the specified file apart from the record sequence number index. This enables intensive operations such as data import to proceed without the overhead of updating all the indexes. You can use *Build indexes* to rebuild the indexes which were dropped.

If the specified file name does not include a data file name as part of the notation, the default data file for that file is assumed. If the file is closed or memory-only, the command does not execute and returns with the flag false.

If you are running on a shareable volume, OMNIS automatically tests that only one user is logged onto the data file (the command fails with flag false if this is not true) and further users are prevented from logging onto the data until the command completes.

If a working message with a count is open while the command is executing, the count will be incremented at regular intervals. The command may take a long time to execute, and it is not possible to cancel execution even if a working message with cancel box is open.

The command is not reversible: it sets the flag if it completes successfully and clears it otherwise, for example if there is more than one user logged onto the data file.

```
; Fast import via a window
Do not flush data
Drop indexes MFILE
Open window instance WIMPORT/winst1
Do method ImportData
Close window instance winst1
```

Duplicate class

Reversible:	NO	**Flag affected:**	YES

Parameters: Class name/New name

Syntax: Duplicate class *{class-name/new-name}*

This command creates a new library class by duplicating an existing one. The name for the new class is specified in addition to the class you want to duplicate. Errors, such as attempting to use a name that is already in use, simply clear the flag and display an error message.

Typical uses of this command are to allow users to make changes to reports and searches.

```
Duplicate class {S_Area/S_USER}
If flag true
   Modify class {S_USER}
   Set search name S_USER
   Print report (Use search)
End If
```

Else

Reversible: NO   **Flag affected:**   NO

Parameters: None

Syntax: Else

This command is used after an *If* command to mark the beginning of some commands that are carried out if the condition in the preceding *If* command is false.

In the example below, the value of SEX is tested against the condition specified in the *If* statement. If the condition fails, control branches to the first *Else If* statement in the method. If the condition again fails, control branches to the *Else* command.

```
If SEX='M'
   OK Message SEX {Record is MALE}
Else If SEX='F'
   OK Message SEX {Record is FEMALE}
Else
   OK Message SEX (Sound bell) {Unknown for this record}
End If
; is the same as...
Switch SEX
   Case 'M'
      OK Message SEX {Record is MALE}
   Case 'F'
      OK Message SEX {Record is FEMALE}
   Default
      OK Message SEX (Sound bell) {Unknown for this record}
End Switch
```

Else If calculation

Reversible: NO   **Flag affected:**   NO

Parameters: Calculation

Syntax: Else If *calculation*

This command is used after an *If* command to mark the beginning of some commands that are carried out if the condition in the preceding *If* command is false, or the calculation in the *Else If* command is true.

In the example below, the value of SEX is tested against the condition specified in the *If* statement. If the condition fails, control branches to the first *Else If* statement in the method. If the condition fails again, control branches to the *Else* command.

```
If SEX='M'
   OK Message {Record is MALE}
Else If SEX='F'
   OK Message {Record is FEMALE}
Else
   OK Message {Sex unknown for this record}
End If
```

Else If flag false

Reversible: NO **Flag affected:** NO
Parameters: None
Syntax: Else If flag false

This command is used after an *If* statement and provides a marker before a series of commands that have to be carried out if the flag is false.

```
Open window instance WCHOOSE
Enter data
If VALUE >= 100
   Print record
Else If flag false   ;; User canceled in Enter data mode
   Close window WCHOOSE
End If
```

Else If flag true

Reversible: NO **Flag affected:** NO
Parameters: None
Syntax: Else If flag true

This command follows an *If* statement and provides a marker before a series of commands that have to be carried out if the flag is true and if the value does not meet the condition specified in the *If* statement.

```
; you use the Yes/No message to set or clear the flag
Yes/No message {Set flag with Yes or No}
If flag false
   OK message {flag is 0}
Else If flag true
   OK message {flag is 1}
End If
```

Enable all menus and toolbars

Reversible: YES　　**Flag affected:** NO

Parameters: None

Syntax: Enable all menus and toolbars

This command enables all menus and toolbars. It reverses the action of *Disable all menus and toolbars*. This command will not enable a menu which has been disabled by disabling line zero. This menu can only be enabled by enabling line zero with *Enable menu line*.

User-defined windows with the property **enablemenubarandtoolbars** turned off use a call to *Disable all menus and toolbars* to prevent menu bar access.

```
Disable all menus and toolbars
Prepare for edit
Enter data
Update files if flag true
Enable all menus and toolbars
```

or do it like this

```
Do $imenus.$sendall($ref.$enable)
Do $itoolbars.$sendall($ref.$enable)
```

Enable automatic publications

Reversible: YES   **Flag affected:** YES

Parameters: None

Syntax: Enable automatic publications

This command turns on the automatic publication of all published fields. It affects only those fields which have been published automatically, that is, whose publisher options have been set up as **Publish on save**. The command can be reversed by using *Disable automatic publications* and if used within a reversible block, the *Enable automatic publications* command is reversed, restoring the automatic publications to their former state when the method terminates.

When a library is launched, automatic publications are enabled. If System 7 is not running, the command clears the flag and does nothing.

```
Publish field CNAME {HD80:Public:Sales-Name}
Publish field CTOTAL {HD80:Public:Sales-Total}
Set publish options (Publish on save) {CNAME,CTOTAL}
..
Enable automatic publications
Prepare for edit
Enter data
Update files if flag set
Disable automatic publications
```

Enable automatic subscriptions

Reversible: YES   **Flag affected:** YES

Parameters: None

Syntax: Enable automatic subscriptions

This command turns on the automatic update of all subscribed fields. It affects only those fields which have been subscribed automatically. The command can be reversed by using *Disable automatic subscriptions* and if used within a reversible block, the *Enable automatic subscriptions* command is reversed, restoring the automatic publications to their former state when the method terminates.

When a library is launched, automatic subscriptions are enabled. If System 7 is not running, the command clears the flag and does nothing.

```
Subscribe field CNAME {HD80:Public:Sales-Name}
Subscribe field CTOTAL {HD80:Public:Sales-Total}
Set subscriber options (Subscribe automatically) {CNAME,CTOTAL}
.
Enable automatic subscriptions
Prepare for edit
Enter data
Update files if flag set
Disable automatic subscriptions
```

Enable cancel test at loops

Reversible: YES **Flag affected:** NO

Parameters: None

Syntax: Enable cancel test at loops

This command causes OMNIS to test for the break key at the end of each loop in the method. It reverses the *Disable cancel test at loops* command. Unless OMNIS has executed a *Disable cancel test at loops*, this test is carried out automatically. The break key is when the user presses Ctrl-Break under Windows or Cmnd-period under MacOS.

```
; this method deletes all records where code = 'ABC':
Calculate CODE as 'ABC'
Find on CODE
Disable cancel test at loops
While flag true
   Working message  (Repeat count)
   Delete
   Next on CODE (Exact match)
End While
Enable cancel test at loops
```

Enable enter & escape keys

Reversible: YES **Flag affected:** NO

Parameters: None

Syntax: Enable enter & escape keys

This command enables the Enter and the Escape keys (or Cmnd-period under MacOS). It reverses the action of the *Disable enter & escape keys* command.

In some libraries where the user may accidentally press Enter and terminate enter data mode, it is useful to disable the Enter key.

Enable fields

Reversible: YES   **Flag affected:** NO

Parameters: Field name or list of field names

Syntax: Enable fields *{field1*[,*field2,...*]*}*

This command enables the specified field or list of fields. You can use it to reverse the *Disable fields* command, or turn Display fields into Entry fields temporarily.

```
Begin reversible block
   Enable fields {Entry1,Entry2}
End reversible block
Prepare for insert
Enter data
Update files if flag set
; method ends and fields are now disabled
```

or to enable all the fields on the current window

```
Do $cwind.$objs.$sendall($ref.$enabled.$assign(kTrue))
```

Enable menu line

Reversible: YES   **Flag affected:** NO

Parameters: Menu instance name
Menu line number

Syntax: Enable menu line *menu-instance-name*/*menu-line-number*

This command enables the specified line of a menu instance. It reverses the *Disable menu line* command. However, you cannot enable a line using this command if you have no access to it, or if there is no current record. You specify the *menu-instance-name* and the number of the menu line you want to enable. The command clears the flag if the menu instance is not installed or if the line cannot be enabled.

```
Install menu STARTUP/minst1
Test for menu installed {minst1}
If flag true
   Enable menu line minst1/3      ;; enables menu line 3
End If
```

or do it like this

```
Do $imenus.MENU.$objs.LINE.$enable.$assign(kTrue)
```

Enable receiving of Apple events

Reversible: YES   **Flag affected:**   YES

Parameters: None

Syntax: Enable receiving of Apple events

This command enables the receiving of Apple events. When you launch an OMNIS library, receiving of Apple events is disabled by default, apart from the compulsory events. You can disable the compulsory events using the *Disable receiving of Apple events* command with the **Disable compulsory events** option checked.

The Apple events OMNIS receives may have been created by itself, another OMNIS library, or by another System 7 application. A library in which *Enable receiving of Apple events* has been executed is able to receive the full range of Apple events implemented under OMNIS.

When *received by* OMNIS, the compulsory events do the following:

❑ *Open application* launches OMNIS,

❑ *Quit application* quits OMNIS,

❑ *Open document* loads a library or report,

❑ *Print document* opens a library, and prompts the user for a report to print.

When enabled, OMNIS accepts events from the Core event suite, the Database event suite, and the Finder event suite. The Core events received include *Send data, Get data* and *Do script*; see the *Send core event* command. The Database events are described with the *Send database event* command. The Finder events are described with the *Send Finder event* command.

Include the following line in your STARTUP menu if you want to enable Apple Events:

```
Yes/no message {Do you want to accept Apple Events?}
If flag true
   Enable receiving of Apple events
End if
```

Enable relational finds

Reversible: YES   **Flag affected:**   NO

Parameters: ☐ Use connections
                      List of files

Syntax: Enable relational finds [*(Use connections)*] *{file1,file2*[*,file3*]*...}*

This command causes all find tables to be built relationally, ignoring the main file. The file list is a list of files to be joined and, if **Use connections** is checked, all connections between the joined files are made when building the table. In effect, the connections provide the relational joins, that is, "sequence number = sequence number". So if there are three files: F1, F2 and F3, with F1 (child file) connected to F2 (parent file) and F2 connected to F3:

```
Enable relational finds (Use connections) {F1,F2,F3}
Print report
```

will result in a child/parent/grandparent report. If you provide no other search conditions, you would generate the SQL where clause for this report from the internal OMNIS connections which would look like this: "...where F1.connection_number = F2.sequence_number and F2.connection_number = F3.sequence_ number". When you use key fields to join records, you use a search to set up the "Where" condition, for example:

```
Set search as calculation {(F1.city=F2.city) & (F1.date>=#D)}
Enable relational finds {F1,F2}
Build list from file (Use search)
```

This will generate a list containing fields from records from F1 and F2 which have the same values in the "city" fields and with F1.date greater than today's date (and ignoring the connection between F1 and F2 as well as the main file).

When relational finds are enabled, the index field specified for find and build list commands is ignored. It is necessary to use a sort to determine the order of the table.

The *Disable relational finds* command causes a reversion to the default situation where the main file and its connected parent files are joined using the connections. The *Enable relational finds* and *Disable relational finds* commands are both reversible and do not affect the flag.

Enclose exported text in quotes

Reversible: NO   **Flag affected:**   NO

Parameters: ☐ Enable

Syntax: Enclose exported text in quotes [*(Enable)*]

This command specifies that all text exported in tab-delimited and comma-delimited format is enclosed in quotes; to enable this option you must run the command with the Enable option checked. This command sets the $exportedquotes library preference which is enabled (set to kTrue) by default. Exported literals that are already quoted will be further enclosed in quotes, for example, "hello" becomes ""hello"". You can turn off this option by executing the command with the check box unchecked, or using the notation.

```
Set report name R_EXPORT1
Send to file
Prompt for print file
Enclose exported text in quotes (Enable)
Print report
```

or to disable the option with the notation

```
Do $clib.$prefs.$exportedquotes.$assign(kFalse)   ;; turns it off
```

End export

Reversible: NO   **Flag affected:**   NO

Parameters: None

Syntax: End export

This command ends the export of data from an OMNIS list or row variable.

```
Set print or export file name {Export.txt}
Prepare for export to file {Delimited (commas)}
Export data LIST1
End export
```

End For

Reversible: NO   **Flag affected:** NO
Parameters: None
Syntax: End For

This command ends a For loop. The two For loops *For field value* and *For each line in list* perform looping type operations. The *End For* command terminates both these commands.

```
For LVAR1 from 1 to 10 step 2
; do something
End For
For each line in list from 1 to LIST.$linecount step 2
; do something
End For
```

End If

Reversible: NO   **Flag affected:** NO
Parameters: None
Syntax: End If

This command terminates an *If* statement once OMNIS has executed the commands inside the *If* statement; it also marks the end of the commands to be executed as part of the *If...Else If* block. Once the commands associated with the *If...Else If* block have been executed, control passes to the next command after *End If.* For every *If* command, you should have a corresponding *End If* command.

```
Calculate Count as Count + 1
If Count = 25
   OK message {Halfway through now}
Else If Count = 50
   Calculate Count as 1
End If
```

End import

Reversible: NO   **Flag affected:** NO

Parameters: None

Syntax: End import

This command ends the import of data without closing the port, DDE channel, or file through which data is being imported.

```
Prompt for import file
Prepare for import from file {Delimited (commas)}
Import data {list1}
End import
Close import file
```

End print

Reversible: NO   **Flag affected:** YES

Parameters: Report instance name

Syntax: End print [*{report-instance-name}*]

This command terminates the specified report and prints the totals section. If you omit the report instance name the *End print* command terminates the most recently started report instance. The flag is cleared if no report instances exist.

End print cancels the Prepare for print mode. You must include it after a *Prepare for print* command even if a totals section is not required.

You can print running totals of fields in the Record section by including the same fields in the Totals section of the report. Provided you choose the Totaled property for the field in the Record section, OMNIS automatically maintains a running total.

```
Set main file {f_client}
Set report name r_letters
Send to screen
Prepare for print
While flag true
   Print record
   Next
End While
End print
```

or do it like this

```
Do $ireports.REPORT.$endprint()
```

End print job

Reversible: NO   **Flag affected:** YES

Parameters: None

Syntax: End print job

This command terminates a print job initiated with *Begin print job* and sends it to the printer.

End print job clears the flag and returns an error if a job has not been started. It sets the flag if it succeeds: in this case, the document is now available for the operating system to print.

Once a print job is started, any attempt to set the report destination fails, that is, you cannot select a new destination until you have issued an *End print job*.

Issuing *End print job* immediately after *Begin print job* may result in an empty document being printed.

OMNIS automatically issues *End print job* at shutdown; it does not do this at any other time.

End reversible block

Reversible: NO   **Flag affected:** NO

Parameters: None

Syntax: End reversible block

This command defines the end of a reversible block of commands. All *reversible* commands enclosed within the commands *Begin reversible block/End reversible block* are reversed *when the method containing this block finishes.* However, a reversible block in the $construct() method of a window class reverses *when the window is closed*—not when the method is terminated as is normally the case.

See *Begin reversible block* for more information on reversible blocks.

End SQL script

Reversible: NO   **Flag affected:** NO

Parameters: None

Syntax: End SQL script

This command defines the end of a block of SQL statements and text which are placed in the SQL buffer before being sent with the *Execute SQL script* command. The marker for the start of the block is the *Begin SQL script* command. When *Autocommit* is on, the statements between the *Begin SQL script* and *End SQL script* commands are committed or rolled back automatically.

The *Perform SQL* command is an alternative to the *Begin–End–Execute SQL script* sequence, and allows SQL statements to be executed while bypassing the SQL buffer.

```
Begin SQL script
Describe server table (Columns)
Build list from select table
SQL: Insert table_name Col1, Col2 values
SQL: ([lst(1,Col1)], [lst(1,Col2)])
End SQL script
Execute SQL script
; If flag is true, there were no errors and the transaction is
; committed at the next Begin SQL script
```

End text block

Reversible: NO   **Flag affected:** NO

Parameters: None

Syntax: End text block

This command marks the end of a block of text which is placed in the global text buffer. You build up the text block using the *Begin text block* and *Text:* commands. Following an *End text block*, you can return the contents of the text buffer using the *Get text block* command.

```
; Declare var cTEXT of Character type
Begin text block
Text: To be, or not to be,
Text: those are the parameters.
End text block
Get text block cTEXT
```

End Switch

Reversible: NO   **Flag affected:**   NO

Parameters: None

Syntax: End Switch

This command terminates a *Switch* statement and defines the point where method execution continues after each *Case* statement.

For example, the following method selects the correct graph window depending on the graph type selected in the GraphType parameter.

```
; Graph Options
; Declare Parameter GraphType (Short integer (0 to 255))
Switch GraphType
  Case kGraphPie
    Do method GraphPieWindow/Open Window
    ; calls method and jumps to End switch
  Case kGraphBars,kGraphArea,kGraphLines
    Do method Graph2DWindow/Open Window
    ; calls method and jumps to End switch
  Case kGraph3D
    Do method Graph3DWindow/Open Window
    ; calls method and ends switch
End Switch
```

End While

Reversible: NO   **Flag affected:**   NO

Parameters: None

Syntax: End While

This command marks the end of a *While* loop. When the condition specified at the start of the loop is not fulfilled (testing the flag or calculation) the command after the *End While* command is executed. Each loop that begins with a *While* command must terminate with an *End While* command, otherwise an error occurs.

```
Calculate Count as 1
Repeat                    ;; Repeat loop
  Calculate Count as Count+1
Until Count >= 3
OK message {Count=[Count]}  ;; prints 'Count=3'
```

```
Calculate Count as 1
While Count <= 3              ;; While loop
   Calculate Count as Count+1
End While
OK message {Count=[Count]}   ;; prints 'Count=4'
```

Enter data

Reversible: NO   **Flag affected:**   YES

Parameters: Termination condition

Syntax: Enter data [*condition*]

This command puts OMNIS into enter data mode which allows data to be entered via the current window. An error is generated if there is no open window. It initiates an internal control loop which does the following:

1. Places the cursor in the first entry field,

2. lets the user enter data from the keyboard,

3. Detects the use of Tab, Shift-Tab and other cursor movements such as click and moves the cursor to the appropriate field,

4. Waits for an OK, setting flag true before allowing control to pass to the command following *Enter data* in the method,

5. Detects a Cancel which aborts data entry with a false flag.

```
Open window instance W1
Enter data
If flag true
   OK message {User has pressed Return}
Else
   OK message {User has canceled}
End If
```

By default, the *Enter data* command waits for an evOK or evCancel event. When these events are triggered enter data mode is terminated (assuming the window is not in modeless enter data mode). However you can include a termination condition with *Enter data* and, in this case, the command waits until the expression becomes true. For example

```
Calculate instvar as 0
Enter data until instvar>0
```

causes enter data mode to continue until the variable becomes greater than zero. In this case, the evOK or evCancel events do not cause the enter data to terminate, but they are reported to the window's $event() method in the usual way.

Execute SQL script

Reversible: NO **Flag affected:** YES

Parameters: None

Syntax: Execute SQL script

This command executes the contents of the SQL buffer, that is, the SQL statement or transaction contained within the previously specified *Begin SQL script* and *End SQL script* commands.

After the evaluation of the square bracket notation and indirect square bracket notation, the content of the SQL buffer is sent to the remote database as a series of SQL statements.

Syntax errors in commands generate errors and return a false flag. Non-fatal errors during a command do not prevent further commands from executing. This means that it is important to test the flag after an *Execute SQL script*. The function *sys(131)* returns the error code reported by the server and *sys(132)* returns the error text supplied by the server.

```
Begin SQL script
SQL: Create table TABLE1 createnames(file1)
End SQL script
Execute SQL script
If flag false
   OK Message {Create failed: [sys(132)]}
   Reset cursor(s)
   Quit all methods
End If
```

The default action for a session is to commit all uncommitted statements after a successful *Execute SQL script* and to rollback all uncommitted statements after an unsuccessful *Execute SQL script* at the next *Begin SQL script*, *Reset cursor(s)* or *Logoff from host*. The *Autocommit (Off)* command allows the automatic commit and rollback to be disabled so that you can use *Commit current session* and *Rollback current session* commands to control transaction management.

The commands *Begin SQL script, Execute SQL script, Perform SQL,* and *Reset cursor(s)* all empty the SQL statement buffer for the current session.

Export data

Reversible:	NO	**Flag affected:**	NO

Parameters: List or row variable name

Syntax: Export data *list/row-name*

This command exports data from an OMNIS list or row variable.

```
Set print or export file name {Export.txt}
Prepare for export to file {Delimited (commas)}
Export data LIST1
End export
```

Fetch current row

Reversible:	NO	**Flag affected:**	YES

Parameters: Cursor name (Current is the default)
List of files and/or fields

Syntax: Fetch current row [from *cursor-name*] [into
{file|field1[,*file|field2*]...*}*]

This command fetches or reads the current row of the select table. The flag is set if a row is fetched. If a list of fields is added to the Fetch, the current map is overwritten and the columns are mapped into the fields listed.

See the *Fetch next row* command for more information about fetching data.

Fetch first row

Reversible:	NO	**Flag affected:**	YES

Parameters: Cursor name (Current is the default)
List of files and/or fields

Syntax: Fetch first row [from *cursor-name*] [into *{file|field1*[,*file|field2*]...*}*]

This command fetches or reads the first row of the select table. The flag is set if a row is fetched. If a list of fields is added to the Fetch, the current map is overwritten and the columns are mapped into the fields listed.

A *Fetch first row* command followed by a series of *Fetch next row* commands enables the select table to be processed on a row-by-row basis in a descending order.

See the *Fetch next row* command for more information about fetching data.

Fetch last row

Reversible: NO   **Flag affected:**   YES

Parameters: Cursor name (Current is the default)
                  List of files and/or fields

Syntax: Fetch last row [from *cursor-name*] [into *{file|field1*[,*file|field2*]*...}*]

This command fetches or reads the last row of the select table. The flag is set if a row is fetched. If a list of fields is added to the Fetch, the current map is overwritten and the columns are mapped into the fields listed.

A *Fetch last row* command followed by a series of *Fetch previous row* commands enables the select table to be processed on a row-by-row basis in an ascending order.

See the *Fetch next row* command for more information about fetching data.

Fetch next row

Reversible: NO   **Flag affected:**   YES

Parameters: Cursor name (Current is the default)
                  List of files and/or fields

Syntax: Fetch next row [from *cursor-name*] [into *{file|field1*[,*file|field2*]*...}*]

This command fetches or reads the next row of the select table. The flag is set if a row is fetched. If you add a list of fields to the Fetch, the current map is overwritten and the columns are mapped into the fields listed. A series of *Fetch next row* commands enables the select table to be processed on a row-by-row basis in descending order.

You can fetch the previous, first, last, or current row using one of the other *Fetch...* commands. Their behavior is the same as *Fetch next row* except that they fetch a different row.

```
Set report name R_SQL1
Prepare for print
Fetch next row
While flag true
   Print record
   Fetch next row
End While
; Last fetch found empty select table
End print
```

The following example prints an OMNIS report using data from the select table:

```
Set report name REL
Begin SQL script
SQL: Select * from FELEMENTS where ATNO < '50';
End SQL script
Execute SQL script
If flag false
   OK message SQL Error (Icon) {Select error//[sys(132)]}
   Reset cursor(s) (Current)
   Quit all methods
End If

Fetch next row      ;; gets the first row of the select table
If flag true
   Prepare for print
   Repeat
      Print record
      Fetch next row
   Until flag false  ;; this indicates end of select table
   End print
Else
   OK message {No rows were selected to print}
End If
```

or do it like this

```
Do TableBasedList.$fetch(1)
```

Fetch previous row

Reversible:	NO	**Flag affected:**	YES
Parameters:	Cursor name (Current is the default)		
	List of files and/or fields		
Syntax:	Fetch previous row [from *cursor-name*]		
	[into *{file|field1*[*,file|field2*]*...}*]		

This command fetches or reads the next row of the select table. The flag is set if a row is fetched. If a list of fields is added to the Fetch, the current map is overwritten and the columns are mapped into the fields listed.

A series of *Fetch previous row* commands enables the select table to be processed on a row-by-row basis in an ascending order.

See the *Fetch next row* command for more information about fetching data.

Find

Reversible: YES   **Flag affected:**   YES

Parameters: Field name (must be indexed)
                    Calculation
                    ☐ Exact match
                    ☐ Use search

Syntax: Find on **field-name** [([*Exact match*][,*Use search*])] [*{calculation}*]

This command builds a find table and locates the first record in the table, that is, it loads the main and connected files into the current record buffer. The flag is false and the buffer is cleared if no record is found.

You use the *Find* command to locate records within a file. If you don't use a search, the file is searched in the order specified by the indexed field until the value given in the calculation line is matched. In this case, the current find table is the same as the chosen Index.

When the closest match is found, the main and connected files are read into the current record buffer and the flag is set true. If the indexed field is from a connected file, the search is repeated automatically until the record having a connected entry in the main file is found.

A blank calculation indicates that the *Find* is to be performed using the current value of the selected index field. Thus, if you precede the command with a *Clear main file,* it is the same as a *Find first.*

OMNIS can perform a *Find* with an **Exact match** requirement. In this case, the value in the "field found" record must correspond in every detail (for example, upper or lower case characters) to the current value of the indexed field in the current record buffer. A flag true indicates a successful Find, otherwise a flag false results, and the main and its connected files are cleared.

You use the exact match option to locate child records connected to a current parent record.

Clearing the find table

The find table is cleared if:

1. A *Clear find table* command is executed with the same main file setting.

2. A new *Find* is carried out on the same file.

3. A *Next/Previous* command with a new (non-blank) index or a **Use Search** or **Exact match** option where the original *Find* had none, is used.

The following example illustrates a find table used to print and process records:

```
Set main file {F_CLIENTS}
Set search as calculation {C_CREDIT>=1000}
Clear main file
Find on C_NAME (Use search)
If flag true
   Prepare for print
   Repeat
      Print record
      Do method LogPrintOut
      Next
   Until flag false
   End print
End If
```

Reversibility

If you use a *Find* command in a reversible block, the records modified by the Find are restored when the method containing the reversible block finishes. Although the main and connected records are recovered, the data within the record may not be recovered if it has been deleted or changed. The current index is not reversed.

Examples using Find

```
; Delete
; Deletes records with confirmation using a search
Set main file {LCUST}
Set search name SRCH001
Clear main file
Find on LNAME (Use search)
While flag true
   Working message (Repeat count)
   Delete with confirmation {Delete record [LNAME]?}
   Next (Use search)
End While
```

```
; Find children
; Finds all connected children for current parent
Begin reversible block
   Single file find on P_CODE (Exact match)
End reversible block
; The reversible block ensures that the parent
; record is restored when the method ends
Set main file {F_CHILD}
Clear main file
Find on P_CODE (Exact match)
While flag true
   OK message {Found child [C_CODE]}
   Next on P_CODE (Exact match)
End While
; Note that parent has been lost by the last Next command
; but it is restored when the reversible block reverses
```

Find first

Reversible:	YES	**Flag affected:**	YES

Parameters: Field name (must be indexed)
☐ Use search
☐ Use sort

Syntax: Find first on *field-name* [*([Use search][,Use sort])*]

This command automatically locates the first record in a file using the index for the specified field. If no field is given, the record sequence number is used. The main and connected files are read into the CRB if a valid first record is found. The flag is set false if no record is found.

You use the **Use search** option in conjunction with the specified indexed field to select the *first* record which fulfills the search specification. If the search is a calculation, the optimizer will choose the best index if the index field is left blank.

You use the **Use Sort** option in conjunction with the current sort fields (see *Set sort field*) to create a table of entries from the data file which are sorted into an order set by up to nine sort fields.

The find table is cleared if:

1. A *Clear find table* command is executed with the same main file setting.

2. A new *Find* is carried out on the same file.

3. A *Next/Previous* command with a new (non-blank) index or a **Use Search** or **Exact match** option where the original *Find* had none, is used.

If you use the *Find first* command within a reversible block, it is reversed when the method finishes, that is, the main and connected records are restored. However, if the data within the original record has been deleted or changed, it will not be possible to completely restore the buffer.

```
Begin reversible block
   Clear sort fields
   Set sort field NAME
   Set sort field TOWN
   Set main file {FINVOICES}
   Find first on INV_NUMBER (Use sort)
End reversible block
While flag true
   Enter data
   Next
End While
```

Find last

Reversible: YES   **Flag affected:**   YES

Parameters: Field name (must be indexed)
                    ☐ Use search
                    ☐ Use sort

Syntax: Find last on ***field-name*** [([*Use search*][,*Use sort*])]

This command automatically locates and displays the last record in a file using a specified indexed field. You can use the *Find last* command to locate the last record added to a file by using the record sequencing number as the index. The flag is set false if no record is found.

You use the **Use search** option in conjunction with the specified indexed field to select the *last* record which fulfills the search specification. If the search is a calculation, the optimizer will choose the best index if the index field is left blank.

Whenever you use a *Find* command, a find table is created which determines the order in which records are displayed using subsequent *Next* and *Previous* commands. Once a find table has been created, subsequent *Next* or *Previous* commands will use the table provided the commands have an empty or the same Index, and the same (or empty) **Search** and **Exact match** conditions. A *Clear find table*, a new *Find* on the same file or *Next/Previous* commands with a new (non-blank) index or a Search or Exact match where the original *Find* had none, will clear the find table.

The **Use Sort** option works in conjunction with the current sort fields (see *Set sort field*) to create a table of entries from the data file which are sorted into an order set by up to 9 sort fields. Refer to the *Find* command for details of the find table and its use.

```
Begin reversible block
   Set main file {FINVOICES}
   Find last on INV_NUMBER
End reversible block
OK message {Last invoice record inserted was RSN [I_SEQ]}
```

Floating default data file

Reversible: YES   **Flag affected:** NO

Parameters: File or list of files

Syntax: Floating default data file *{file1*[,*file2*]*...}*

This command sets the default data file as the current data file and changes whenever the current data file changes. You use *Floating default data file* in libraries which open more than one data file at once. The default behavior in OMNIS is that, as each new data file is opened, it becomes the "current" data file. The concept of a current data file is important when your commands refer to file classes without specifying a data file. So, for example, the command

```
Set main file {FCUSTOMERS}
```

is ambiguous if more than one data file is open at the same time. To specify the data file to be used, you can use *Set default data file* to associate a file class with the current data file. For example, to associate FCUSTOMERS with DATA1.DF1, you can use:

```
Set current data file {DATA1}
Set default data file {FCUSTOMERS}
```

References to FCUSTOMERS are now equivalent to references to DATA1.FCUSTOMERS. The association between FCUSTOMERS and DATA1 remains in effect even if the current data file is set to a different data file. To return to the default state where the default data file "floats" to whatever the current data file is, you can use:

Floating default data file {FCUSTOMERS}

The *Floating default data file* command sets the default data file, for the specified list of files, to be equal to the current data file and allows it to change (float) whenever the current data file changes.

The command does not change the flag but is reversible, that is, the previous default data files are restored when the method containing the command in a reversible block terminates.

Flush data

Reversible: YES   **Flag affected:**   YES

Parameters: None

Syntax: Flush data

This command reverses *Do not flush data* and reverts to the default mode where the changed data is immediately written to disk after each *Update files* or *Delete* command.

The command sets the flag if the state of the 'Do not flush data' mode is changed and is reversible, restoring the previous state of the 'Do not flush' flag when reversed. If the previous mode was 'Do not flush data', *Flush data* will cause any modified data which has not been written to disk, to be written on the next *Update files* or *Delete*.

```
; fast import via window
Test for only one user
If flag true
   Do not flush data
   Drop indexes
End If
Open window instance W_IMPORT
Set current list List1
Prompt for import file
Prepare for import from file {Delimited(tabs)}
Import data {List1}
End import
Close import file
For each line in list from 1 to $linecount
   Prepare for insert   ;; transfer list to file
   Load from list
   Update files
End For
Flush data now  ;; writes the data immediately to disk
Rebuild indexes
Flush data       ;; Changes mode back to 'Flush data'
```

Flush data now

Reversible: NO   **Flag affected:**   NO

Parameters: None

Syntax: Flush data now

This command causes any modified data which has not been written to disk to be immediately written to disk. This command will only do something if a *Do not flush data* command has been executed.

This command leaves the flag unaffected and is not reversible. See *Flush data* for an example.

For each line in list

Reversible: NO   **Flag affected:**   NO

Parameters: □ Selected lines only
                □ Descending
                Start value (line number of list)
                End value (line number of list)
                Step value (default is 1)

Syntax: For each line in list [*([Selected lines only][,Descending])*] from
                    start-value to **end-value** step **step-value**

This command marks the beginning of a loop that processes the lines of the current list. You must specify the current list before executing the For loop. The For loop is a convenient way to write *While/End While* loops to step through each line of a list. With the **Selected lines only** option, the loop will skip over any lines encountered that are not selected.

The **Start value** specifies the line in the list at which method execution of the For loop starts. The loop continues until the processed line exceeds or is equal to the **End value**. If the **Step value** is not specified, the default value of 1 is used. The values involved must all be integers. The **Descending** option tells OMNIS to step through the list from a high line number to a low line number. The Start and End values are swapped if the End value is less than the Start value.

You can use *Jump to start of loop* within the loop to continue the next iteration of the loop. Similarly, *Break to end of loop* will exit the loop prematurely.

For each line in list operates on the current list. The matching *End For* will also operate on the current list. Unpredictable behavior will result if the current list is changed and not restored within the *For/End For* construct.

```
Prepare for print
Set current list F_LIST
For each line in list from 1 to LIST.$linecount step 1
   Load from list
   Print record
End For
End print
; this is equivalent to the method below
Prepare for print
Set current list F_LIST
Calculate LIST.$line as 1
While LIST.$line<=LIST.$linecount
   Load from list
   Print record
   Calculate LIST.$line as LIST.$line+1
End While
End print
```

For field value

Reversible: NO   **Flag affected:**   NO

Parameters: Field name or variable
                  Start value
                  End value
                  Step value (default is 1)

Syntax: For *field-name* from *start-value* to *end-value* step *step-value*

This command marks the beginning of a For loop which defines a series of commands to be repeated a number of times. You use *field-name* as a counter that is automatically incremented by the *step-value* each time the *End For* statement is reached.

The values involved must all be numbers, preferably integers. If *start-value* is greater than *end-value*, and *step-value* is positive, the command will perform no loops. Similarly, no loops are performed if *start-value* is less than *end-value*, and *step-value* is negative.

You can use *Jump to start of loop* within the loop to continue the next iteration of the loop. Similarly, *Break to end of loop* will exit the loop prematurely.

The following example builds a list containing the sales totals for four regions.

```
; declare local vars LV_Sales, LV_Expenses of type Short number 0 dp
Set current list GRAPHLIST
Define list {Division,NetSales}
For LVARI from 1 to 4 step 1
   Do method PopulateDrilldownList (LVAR1)
   Set current list GRAPHLIST2
   Calculate LV_Sales as tot(Sales)
   Calculate LV_Expenses as tot(Expenses)
   Calculate NetSales as LV_Sales - LV_Expenses
   Calculate Division as pick(LVARI-1,'North','East','South','West')
   Set current list GRAPHLIST
   Add line to list
End For
Calculate TotNetSales as totc(NetSales)
; etc.
```

Get SQL script

Reversible: NO **Flag affected:** YES
Parameters: Field name or variable
Syntax: Get SQL script *{field-name|variable}*

This command loads the contents of the SQL buffer for the current session into a specified field or variable. It provides direct access to the SQL statement buffer. The field name parameter can be any OMNIS character field or variable. The SQL buffer holds all SQL statements and text entered since the last *Begin SQL script* which have not yet been executed. The square brackets and SQL functions will have been evaluated but the values of indirect @[] square bracket notation will not be available.

The commands *Begin SQL script*, *Execute SQL script, Perform SQL* and *Reset cursor(s)* all empty the SQL statement buffer for the current session. Therefore, using *Get SQL script* after *Perform SQL* will do nothing.

```
Begin SQL script
SQL: Insert into [TABLE] insertnames(FTABLE)
End SQL script
Get SQL script {S1}
Yes/No message {Do you want to send '[S1]'}
If flag true
   Execute SQL script
Else
   Reset cursor(s)
End If
```

Get text block

Reversible: NO   **Flag affected:**   NO

Parameters: Field name or variable

Syntax: Get text block *{field-name/variable}*

This command loads the current contents of the global text buffer into the specified field or variable. You build up the text block using the *Begin text block* and *Text:* commands. Following an *End text block*, you can return the contents of the text buffer using the *Get text block* command.

```
; Declare var cTEXT of Character type
Begin text block
Text: To be, or not to be,
Text: those are the parameters.
End text block
Get text block cTEXT
```

Go to next selected line

Reversible: NO   **Flag affected:**   YES

Parameters: ☐ From start
                  ☐ Backwards

Syntax: Go to next selected line [*([From start][,Backwards])*]

This command scans a list for selected lines and goes to the first one it finds. It sets the current line (*LIST.$line*) for the current list (*#CLIST*) equal to the next selected line in that list.

The *Go to next selected line* command steps through the list starting at the current line (if no options are selected) until a selected line is found. When a selected line is located, LIST.$line is set equal to that line number. If a selected line is not found, the flag is cleared and LIST.$line is unchanged.

The **Backwards** option causes the list to be searched in descending order; the **From start** option causes the list to be searched from the start. If both options **Backwards** and **From start** are selected, the list is searched from the end. The following example loads the list with values 1 to 5 and ends with values: 3, 2, 3, 4, 3:

```
Set current list LIST1
Define list {LVAR1}
Calculate LVAR1 as 1
Repeat
   Add line to list
   Calculate LVAR1 as LVAR1+1
Until LVAR1=6
Calculate LIST.$line as 3
Load from list  ;; transfers value 3 from list to LVAR1 in CRB
Select list line(s) {1}
Select list line(s) {5}
Go to next selected line (From start)   ;; selects line 1
Replace line in list
; takes value of LVAR1 (that is, 3) and uses it
; to replace the value in line 1 of the list
Go to next selected line                ;; selects line 5
Replace line in list
Redraw lists
```

Hide docking area

Reversible: NO   **Flag affected:** NO

Parameters: Docking area (a constant)

Syntax: Hide docking area {*docking-area-name*}

This command closes either the top, bottom, left, or right docking area. The docking area is specified using one of the docking area constants: kDockingAreaTop, kDockingAreaBottom, kDockingAreaLeft, or kDockingAreaRight.

When you close a library, OMNIS does not automatically close any docking areas that are open. You must explicitly hide each docking area using *Hide docking area*. Leaving docking areas open and closing the library containing those docking areas can cause problems in your application.

```
Show Docking Area { kDockingAreaLeft }
Install Toolbar {TDESK}   ;; toolbar installed on Left Docking Area
; When the library closes...
Hide docking area { kDockingAreaLeft } ;; hides current docking area
```

Alternatively you can use

```
Do $root.$prefs.$dockingarea.$assign(kDockingAreaNone)
```

Hide fields

Reversible: YES **Flag affected:** NO
Parameters: Field name or list of field names
Syntax: Hide fields *{field1*[,field2,...]*}*

This command hides the specified field or list of fields. You can display hidden fields with
Show fields.

```
Yes/No message {Do you want to hide fields?}
If flag true
   Begin reversible block
   Hide fields {Field1,Field2,Field3]
   End reversible block
End If
For Count from 1 to 20 step 1    ;; delay loop
End For
OK message {Fields will now reappear after method has run}
```

To hide a single field on the current window you can use

```
Do $cwind.$objs.FIELD.$visible.$assign(kfalse)
```

or to hide all fields on the current window

```
Do $cwind.$objs.$sendall($ref.$visible.$assign(kFalse))
```

If calculation

Reversible: NO **Flag affected:** NO
Parameters: Calculation
Syntax: If *calculation*

This command tests the result of the calculation and branches if zero. If the result of the
calculation is non-zero, the result of the test will be true; a result of zero is interpreted as
false. As with all *If* commands, control passes to the next command in the method if the
result is true, otherwise to the next *End If*, *Else* or *Else If* in the method.

```
If SECURITY > 4
   Disable menu line MREPORTS/4
End If
```

If canceled

Reversible: NO   **Flag affected:** NO
Parameters: None
Syntax: If canceled

This command tests whether a Cancel function has been selected and branches if false. The condition is true if either a working message Cancel button is clicked, or the Escape key (under Windows) or Cmnd-period (under MacOS) is pressed. The condition is false if none of these events happens. If *Enable cancel test at loops* is switched on, a loop may detect a Cancel and quit all methods before it is detected by an *If canceled* command.

```
Disable cancel test at loops
Working message (Cancel box)
Repeat
   Redraw working message
   If canceled
      Sound bell
      OK message (Icon) {Method Terminated.}
      Quit method
   End If
Until flag false
```

If flag false

Reversible: NO   **Flag affected:** NO
Parameters: None
Syntax: If flag false

This command lets you implement a branch or change of processing order within a method depending on the result of the previous command. It tests the flag and if it is false, the commands following the *If flag false* are executed. However, if the flag is true, control branches to the next *Else*, *Else If* or *End If* in the method.

```
Test for window open {w_calendar_date}
; If the window is closed, flag will be false
If flag false
   Set main file {f_constant}
   Clear main file
   Next
End If
```

If flag true

Reversible: NO   **Flag affected:** NO
Parameters: None
Syntax: If flag true

This command lets you implement a branch or change of processing order within a method depending on the result of the previous command. It tests the flag and if it is true, the commands following the *If flag true* are executed. However, if the flag is false, control branches to the next *Else*, *Else If* or *End If* in the method.

```
Open window instance w_calendar
Enter data
If flag true
   Open window instance w_schedule
   Enter data
   Close window w_schedule
End If
```

Import data

Reversible: NO   **Flag affected:** YES
Parameters: List or row name
Syntax: Import data *list/row-name*

This command reads the next data item into the the specified list or row variable. You use the *Import data* command to import data from a file or port. Once you select an import file or port, and issue a *Prepare for import* command, *Import data* adds the data to the specified list or row variable.

If a record is successfully read from the file or port, OMNIS sets the flag. An error occurs if the import file or port is closed or if the specified list or row variable does not exist. The flag is set after reading a record successfully.

After the import is complete, you should follow *Import data* with an *End import* and the appropriate *Close import file* or *Close port*.

There is a one-to-one mapping between the columns or fields in the import file and the columns in the list or row variable. Therefore, if there are fewer columns or fields in the import file than in the list or row, the excess import columns or fields are ignored. Likewise, if there are more columns in the list or row than in the import file, the excess columns are left blank.

```
Set port name {2 (Printer port)}
Set port parameters {1200,n,7,2}
Prepare for import from port {Delimited (tabs)}
Import data IMPORTLIST
End import
Close port
```

Import field from file

Reversible:	NO	**Flag affected:**	YES

Parameters: Field name
    ☐ Single character
    ☐ Leave in buffer

Syntax: Import field from file into *field-name*
      [([*Single character*][,*Leave in buffer*])]

This command reads a line of characters from the current import file to the specified field. It lets you read fields from a file without using a window and *Import data*. Usually the command reads a whole line at a time but there are options which modify this.

The **Single character** option tells OMNIS to read a single character at a time. If the field is a Character or a National field, it is set to have a length of one, containing the single character imported from the file. If the field is a Number field, the field value is set to the ASCII code of the single character imported from the file.

The **Leave in buffer** option tells OMNIS to read the string or single character but not remove it from the buffer. Therefore, the next *Import field from file* will read exactly the same value.

An error will occur if the import file has not been opened; OMNIS clears the flag on reaching the end of the file. Do not mix *Import data* and *Import field from file* because they use the input buffer in different ways.

```
Set import file name {Data.TXT}
Prepare for import from file {Delimited (tabs)}
Repeat
   Import field from file into CVAR1
Until CVAR1='start data'
Do method ImportData
Close import file
```

Import field from port

Reversible: NO   **Flag affected:**   YES

Parameters: Field name
                □ Single character
                □ Leave in buffer
                □ Clear buffer
                □ Do not wait

Syntax: Import field from port into *field-name* [([*Single character*]
          [,*Leave in buffer*][,*Clear buffer*][,*Do not wait*])]

This command reads a line of characters from the current port to the specified field. *Import field from port* lets you read fields from a port without using a window and *Import data*. Usually the command reads a whole line at a time but there are options which modify this:

Single character tells OMNIS to read a single character at a time. If the field is a Character or a National field, it is set to have a length of one, containing the single character imported from the port. If the field is a Number field, the field value is set to the ASCII code of the single character imported from the port.

Leave in buffer tells OMNIS to read the string or single character but not remove it from the buffer. Therefore, the next *Import field from port* command will read exactly the same value.

Clear buffer clears the import buffer so that previously received values are ignored.

Do not wait prevents OMNIS from waiting until a string or character is available.

An error will occur if the import port has not been opened; OMNIS clears the flag if nothing has been read. Do not mix the *Import data* and *Import field from port* commands because they use the input buffer in different ways.

```
Set port name {1 (Modem port)}
Prepare for import from port {One field per line}
Repeat
   Import field from port into CVAR1
Until CVAR1='start data'
Do method Importdata
Close port
```

Insert line in list

Reversible: NO   **Flag affected:** YES

Parameters: Line number (can be a calculation, default is current line)
                  Field values

Syntax: Insert line in list [*{*[*line-number*] [*(value1*[,*value2*]...)*]}*]

This command takes the current field values and inserts them at a particular line in the list. The new line is inserted before the specified line and all the lines below the specified line are moved down one place.

If a set of comma-separated values is included as a parameter, these values are read (in order) into the columns of the new line. In this case, the field names for the columns are not used to specify the data for the new line, for example

```
Define list {BOOL,NUM,CHAR}
Insert line in list {('Yes',LVAR1,'Good')}
```

You can specify the line number using a calculation. However, if the parameter for the command is empty or evaluates to zero, the current line is used, that is, the field values are inserted at the current line and all other lines are moved down one place.

If there is no current line (*LIST.$line* = 0), the field values are added at the end of the list. If the line is beyond the current end of the list (for example, the LIST.$line given is greater than *LIST.$linecount*), *Insert line in list* is equivalent to *Add line to list*. The flag is cleared if the list is already at its maximum size (*LIST.$linemax*).

```
; this example inserts 50 calculated lines into the list
Set current list LIST2
Define list {LVAR1,S4}
For LVAR1 from 1 to 50 step 1
   Calculate S4 as rnd(1/LVAR1,6)
   Insert line in list
End For
Redraw lists (All windows)


; in this example two values are added to the list
; and a third is inserted between them
Set current list cList
Define list {NAME}
Insert line in list {('John')}
Insert line in list {('Mary')}
Calculate cList.$line as 2   ;; sets current line as line 2
Insert line in list {('Piggy')}
Redraw lists (All windows)
```

Alternatively you can use the $addbefore() and $addafter() methods to add lines to the current list.

Install menu

Reversible: YES   **Flag affected:**   YES

Parameters: Menu class name
                  Instance name
                  Parameters list

Syntax: Install menu ***menu-name***[*/instance-name*]
                  [*(parameter1*[*,parameter2*]*...)*]

This command installs an instance of the specified menu class on the main menu bar and assigns an instance name. The default instance name is the name of the menu class. The flag is set if the menu is installed.

You can choose the menu class from a list containing your own menus in the current library, and the standard menus ***File**, ***Edit**, and so on. When the menu instance is installed its $construct() method is called.

Passing parameters

You can send parameters to the menu's $construct() method. In the following example, three values are passed as parameters and used to set up the conditions required by the menu options.

```
Install menu MREPORTS/rep1 (CVAR1,LVAR1,CO_NAME)

; the $construct() method for MREPORTS
; declare parameter variable MODE of Character type
; declare parameter variable SECURITY of type Number 0 dp
; declare parameter variable COMPANY of type Field reference
If SECURITY > 4
   Disable menu line MREPORTS/4
End If
```

Three values are passed to the method. This allows the menu to perform different functions depending on the parameters passed to it when installed.

If you use the *Install menu* command in a reversible block, the menu instance is removed from the menu bar when the method terminates. However, the order of the menus on the menu bar may not necessarily be the same as before.

You can install a menu using the $open() method.

```
Do $clib.$menus.MENU.$open()
```

Install toolbar

Reversible: NO **Flag affected:** NO

Parameters: Toolbar class name
   Instance name (default is class name)
   Docking area (a constant)
   Parameters list

Syntax: Install Toolbar *{class-name*[/*instance-name*][/docking-
   area][(parameter1/[,parameter2]...)]*}*

This command installs the specified toolbar class into the named docking area. You specify
the docking area using one of the toolbar constants: *kDockingAreaTop*,
kDockingAreaBottom, *kDockingAreaLeft*, *kDockingAreaRight, or kDockingAreaFloating*.
If you omit the docking area name the toolgroup is installed into the docking area specified
in the class. You can install multiple toolbars onto the same docking area.

```
Show docking area {kDockingAreaTop}
Show docking area {kDockingAreaLeft}
Install Toolbar {T_Format}/Top
Install Toolbar {T_Style}/Left
Install Toolbar {T_Utils/kDockingAreaTop}
```

You can install a toolbar using the $open() method.

```
Do $clib.$toolbars.TOOLBAR.$open()
```

Invert selection for line(s)

Reversible: NO **Flag affected:** YES

Parameters: Line number (can be a calculation, default is current line)
   ☐ All lines

Syntax: Invert selection for line(s) [*(All lines)*] [*{line-number}*]

This command inverts the selection state of a line, that is, from selected to deselected or
vice-versa. You can specify a particular line in the list by entering either a number or a
calculation. You can show the selection state on the window by invoking the *Redraw lists
(Selection only)* command.

The **All lines** option inverts the selection states of all lines of the current list. If no line number is given, the current line selection is inverted. When a list is saved in the data file, the selection state of each line is stored. The following example selects all but the middle line of the list:

```
Set current list LIST1
Define list {LVAR1}
For LVAR1 from 1 to 6 step 1
   Add line to list
End For
Select list line(s) (All lines)
Invert selection for line(s) {LIST1.$linecount/2}
; Or use Invert selection for line(s) {3}
Redraw lists
```

Jump to start of loop

Reversible: NO   **Flag affected:**   NO

Parameters: None

Syntax: Jump to start of loop

This command jumps to the *Until* or *While* command at the beginning of the current loop, missing out all commands after the jump. When used in a *While–End While* loop, *Jump to start of loop* jumps to the start of the loop so that OMNIS can make the While test; the loop continues or terminates depending on the result of this test, whereas, *Break to end of loop* automatically terminates the loop regardless of the value of the condition. Placing a *Jump* outside a loop causes an error.

```
Repeat
   Next
   Prepare for edit
   Enter data
   If flag false
      Jump to start of loop
   End If
   Calculate TOTAL as CREDIT
Until TOTAL > 10000
```

Launch program

Reversible:	NO	**Flag affected:**	YES

Parameters: ☐ Do not quit OMNIS
MacOS program name
Document or file name (full pathname of document or file)

Syntax: Launch program [*(Do not quit OMNIS)*]
{program-name[/*document-name*]*}*

This command launches the specified MacOS program. If you include a file name, the application is launched with the file name as a document. If the specified file name represents a document which the program cannot understand, it will be ignored. You must specify pathnames for the program and document, as shown in the example below.

The default action is to quit OMNIS, but the **Do not quit OMNIS** option lets you keep OMNIS open. If you choose this option, OMNIS will continue to run in the background, concurrently with the new program. A new program launched by OMNIS will always be opened on top, even if OMNIS is already in the background. The flag is set false if an error is detected, for example, if a program or file name cannot be found. Once you attempt a *Launch program*, control passes from your application to the operating system and there is no automatic way of returning to OMNIS.

```
Launch program {Word 6/mac HD:Admin:Memo}
; Note full path for document or file name
If flag false
   OK message (Icon,Sound bell) {Couldn't find Word 6}
End If
```

Load connected records

Reversible: YES   **Flag affected:** YES

Parameters: File class name

Syntax: Load connected records *{file-name}*

This command loads the connected records for the specified file. The *Load connected records* command ensures that the identity of the current connected records for the current record is correct. As OMNIS automatically loads connected records of the main file into the current record buffer, this command is not usually required. However, in multi-user systems, this command ensures that, if any other workstation makes changes to the way in which records are connected, these changes will be reflected at the current workstation.

The flag is cleared if there is no current record for the specified file class, and in the event that no file class is specified, OMNIS uses the main file. This command does not clear the *Prepare for update* mode but does cause multi-user semaphores to be set and should be avoided when in *Prepare for...* mode.

If a parent record requires locking, another user is editing it, and the *Wait for semaphores* command is on, the lock cursor will be displayed. If the user cancels the lock, the flag is cleared and the parent record is not loaded. The *Do not wait for semaphores* command prevents the user from having to wait for the record and returns a flag false if the parent record is not available.

If placed in a reversible block, the parent record reverts to its former value when the method terminates. If you need to read in grandparent records, you can add this command to the usual *Next* command:

```
Next
Load connected records {FPARENT}
Redraw MyWindow
```

Load error handler

Reversible: YES   **Flag affected:**   NO

Parameters: ☐ All libraries
                  Number or name/number (of custom menu method)
                  First error code number
                  Last error code number

Syntax: Load error handler [*(All libraries)*] [*class-name*/]**number** [*(first-error-number*[,*last-error-number*])] [*{method-name}*]

This command loads a specified method which handles errors which may occur within a library. You can specify a range of error codes to be handled by the handler by giving the first and last error number. If no range is specified, the handler is called for all errors. Errors are either *Fatal* or *Warning*.

Error codes such as *kerrUnqindex*, *kerrBadnotation*, *kerrSQL*, can also be used as parameters. The **Catalog** window lists all the constants available in OMNIS.

Fatal errors

A *fatal* error is one that normally stops method execution and drops into the debugger if available. The error code *#ERRCODE* is displayed on the status line in the debugger and is *greater than* 100,000.

Warning errors

A *warning* error is one that does not normally quit the method nor report an error description. The error code *#ERRCODE* is displayed on the status line in the debugger, if invoked, and is *less than* 100,000.

The check box option **All libraries** is provided. If this is not checked, the handler is called only for errors encountered in the library which loaded the error handler. This command leaves the flag unaffected and is reversible; that is, the handler is unloaded when the command is reversed. An error handler remains loaded until it is unloaded or the library containing the handler method is closed. Error handlers loaded within an error handler always unload when that error handler terminates.

Here is a typical error handler:

```
; declare local variable LCODE of Long integer type
; declare local variable LTEXT of Character type
Calculate LCODE as #ERRCODE
Calculate LTEXT as #ERRTEXT
If LCODE = kerrBadnotation
   ; handle error
End If
```

An alternative to assigning *#ERRCODE* and *#ERRTEXT* to local variables is to pass them as parameters to the error handler. You must define LCODE and LTEXT as parameter variables (with the same types) in the error handling method.

An error handler can use one of the *Set error action* commands (SEA) to set what it requires the next action to be. If the error handler quits without making a *Set error action* and there is another handler capable of accepting the error, the second handler is called. Otherwise, the default action for the error is carried out, depending on whether it is a fatal error or warning.

If an error occurs within an error handler, that error is handled in the usual way except that the original error handler will not be used (even if it could handle that error). It is possible to load error handlers within an error handler; these are meant to deal with errors within the handler and are unloaded automatically when the error handler completes execution. The following example handles the error returned by the data manager when an attempt to duplicate a unique index occurs on update:

```
Load error handler Code1/ErrorHnd(kerrUnqindex)
Prepare for edit
Enter data
Update files if flag set

ErrorHnd   ;; Error handler
If #ERRCODE = kerrUnqindex
   OK message (Icon) {You have entered a duplicate field value. I am
   appending 'X' to your entry}
   Calculate INDVAL as con(INDVAL,'X')
   Enter data
   If flag true
      SEA repeat command
   Else
      SEA continue execution
   End If
End If
```

Load event handler

Reversible:	YES	**Flag affected:**	YES

Parameters: Library name
Routine name
Parameters list

Syntax: Load event handler [*library-name*/]**routine-name**
[*(parameter1*[,*parameter2*]...)]

This command makes the specified external routine an event handler, enabling the routine to show its own windows, put its own menus on the menu bar, act as its own event filter, and so on.

Event handlers are modules of code which, when loaded, form part of the OMNIS event-processing loop. Events are passed to the external before being handled by OMNIS. As each call to the external takes place, it can identify whether to take appropriate action. If the event handler returns a flag false, OMNIS knows that the event was meant for OMNIS and the external has ignored it.

You can enter the routine name as the parameter. If the library/resource is not in the EXTERNAL folder, the name of the file containing the library/resource and the name of the library/resource within that file are given as parameters. If no file name is given, the current dynamic link library/resource is searched for the specified routine name.

When the method is called, any existing event handler is not unloaded but continues to be called along with the new handler. The flag is cleared if the routine cannot be loaded.

If you use *Load event handler* in a reversible block, the event handler is unloaded when the method containing the reversible block terminates.

You can pass parameters to the external code by enclosing a comma-separated list of fields and calculations. If you pass a field name, for example, *Call external Maths1 (LVAR1,LVAR2),* the external can directly alter the field value. Enclosing the field in brackets, for example, *Call external Maths1 ((LVAR1),(LVAR2)),* converts the field to a value and protects the field from alteration.

In the routine itself, the parameters are read using the usual GetFldVal or GetFldNval with the predefined references Ref_parm1, Ref_parm2, and so on, Ref_parmcnt gives the number of parameters passed. If the field name is passed as a parameter, you can use SetFldVal or SetFldNval with Ref_parm1, and so on, to change the field's value.

```
Load event handler EventHand
```

Load external routine

Reversible: YES   **Flag affected:** YES

Parameters: Routine name or
Library name/routine name
Parameters list

Syntax: Load external routine [*file-name*/]**routine-name**
[*(parameter1*[,*parameter2*]*...)*]

This command loads the specified external code into memory. You can enter the routine name as the parameter. If the library/resource is not in the EXTERNAL folder, the name of the file containing the library/resource and the library/resource name within that file are given as parameters.

If the library/resource is already loaded or is not found, the flag is cleared and no action is taken. If this command is included in a reversible block, the library/resource is unloaded when the method terminates. If the library/resource is loaded in, it is called with the mode set at ext_load.

You can pass parameters to the external code by enclosing a comma-separated list of fields and calculations. If you pass a field name, for example, *Call external Maths1 (LVAR1,LVAR2),* the external can directly alter the field value. Enclosing the field in brackets, for example, *Call external Maths1 ((LVAR1),(LVAR2)),* converts the field to a value and protects the field from alteration.

In the routine itself, the parameters are read using the usual GetFldVal or GetFldNval with the predefined references Ref_parm1, Ref_parm2, and so on, Ref_parmcnt gives the number of parameters passed. If the field name is passed as a parameter, you can use SetFldVal or SetFldNval with Ref_parm1, and so on, to change the field's value.

```
Load external routine {mathslib/sqroot} (value,CVAR1)
```

Load from list

Reversible: NO   **Flag affected:**   YES

Parameters: Line number (can be a calculation, default is current line)
                    List of field or variables

Syntax: Load from list [*{[line-number]* [(*field1*[,*field2*]...)]*}*]

This command transfers field values from the current list to the corresponding fields in the current record buffer. However, if you include a list of fields, the values in the current list are transferred to the specified fields (see example). Each column value, taken in the order it was defined, is copied to the corresponding field in the field list.

Field names parameter list

The command *Load from list* with '0 (CVAR1,,CVAR12)' specified will load the first column of the current line of the list into CVAR1, ignore the second column, and load the third column into LVAR12. If too few field names are specified, the other columns are not loaded. If too many field names are specified, the extra fields are cleared. Any conversions required between data types are carried out.

If the line number specified in the command line is empty, or if it evaluates to zero, the values are loaded from the current line. If the list is empty or if the line evaluates to a value greater than the total number of lines in the list, the flag is cleared and the fields in the parameter list or in the list definition are cleared.

```
Set current list LIST2
Define list {CODE,NAME,CREDIT}
Build list from file on CLIENTS
Load from list {4(,CVAR3,LVAR1)}
If flag false
   OK message (Sound bell) {Line 4 is beyond the end of list}
Else
   OK message {CVAR3=[CVAR3], LVAR1 is [LVAR1]}
   ;  CVAR3 is lst(4,NAME), LVAR1 is lst(4,CREDIT)
End If
```

Logoff from host

Reversible: No   **Flag affected:** YES

Parameters: None

Syntax: Logoff from host

This command causes a logoff from the current session without disconnecting from the remote database. A Commit is carried out on any uncommitted transactions. You can use another *Set hostname*, *Set username*, *Set password* and *Logon to host* sequence to log the session onto another database, or the same database as another user, for the same remote database. Alternatively, you can issue another *Start session* to disconnect from the current remote database and set up communication with another remote database. *Logoff from host* places OMNIS in an "off-line" state, in which case you must execute another *Logon to host* before proceeding with the next SQL transaction.

```
Set current session {ORACLE2}
Logoff from host
Set current session {SYBASE2}
Logoff from host
```

Logon to host

Reversible: NO   **Flag affected:** YES

Parameters: None

Syntax: Logon to host

This command issues a logon to the current session. If you have correctly supplied the *Set hostname*, *Set username* and *Set password* commands, OMNIS will log onto the remote computer and will initialize communications with the server.

```
Set username {SA}
Set password {Lion}
Set hostname {Serve300}
Logon to host
If flag false
   OK message (Icon,Sound bell) {Logon failed}
Else
   OK message {Logon was successful}
End If
```

There are some important differences in the way you specify the logon parameters for different servers. Refer to *OMNIS Studio Data Access Manager* manual for details.

Make schema from server table

Reversible: NO   **Flag affected:** YES
Parameters: Schema class name
Syntax: Make schema from server table *{schema-name}*

This command makes a schema class from a select table of column definitions. It is typically used after *Describe server table (Columns)* which creates a select table defining a table on your server.

Make schema from server table{schema-name} creates or redefines an OMNIS schema class using the current select table. The select table should have the same structure as that created by *Describe server table (Columns).* One column in the schema class is defined for each row in the select table. The command will generate an OMNIS schema class with the same column names as the server table column names, provided that the column names are valid OMNIS column names.

The *Make schema class from server table* command tries to convert column names and data types to the OMNIS schema class and does not usually generate errors. In some cases, however, it may be necessary to modify the schema class to produce the desired result. If the schema name already exists, the old class will be overwritten by the new one thus redefining the schema class. Any references throughout the library to columns from the old schema, either as field names or in calculations, will become references to the columns in the same positions in the new schema class. This does not apply to literals containing the field names such as parameters to the *fld()* function.

```
Describe server table (Columns) {TableName}
Make schema class from server table {SchemaName}
```

Describe server table (Columns) {table-name} creates a select table with one row for each column of the specified server database table. The following example creates a set of OMNIS schemas for each available table on the server:

```
; declare var LTABLES of List type and give it column TABLE
Set current list LTABLES
Describe database (Tables)
Build list from select table
If LTABLES.$linecount      ;; if the list has data
   For each line in list from 1 to LTABLES.$linecount
     Describe server table (Columns) {[lst(TABLE)]}
     Make schema class from server table {[lst(TABLE)]}
   End For
End If
```

Making a schema class from a list

You can use *Make schema class from server table* to create a schema class from a schema definition held in a table-based list using the ^ notation, for example

Make schema class from server table {SchemaName,^LIST}

You can use the *Make schema class from server table* command to generate schema classes even when not using the DAM. The schema specification used by *Make schema class from server table* and created by a *Describe server table (Columns)* is:

Col	Column description
1	Column name
2	SQL data type for the column
3	Column width
4	Number of decimal places (for numeric columns); empty for floating Numbers
5	NULL or NOTNULL; for some servers only
6	Empty; for future expansion
7	Description for the column where available

Maximize window instance

Reversible: NO   **Flag affected:**   NO

Parameters: Window instance name

Syntax: Maximize window instance *window-instance-name*

This command maximizes the specified window instance, that is, it sizes the window to the maximum size of the OMNIS application window (the Finder window under MacOS). You can maximize a window only if it has a maximize button.

```
Maximize window instance MyWin2
; full screen for data entry, etc.
Minimize window instance MyWin2
; reduces window to an icon at the bottom of screen
```

You can maximize a window using the $maximize() method. To maximize the current window use

```
Do $cwind.$maximize()
```

Merge list

Reversible: NO   **Flag affected:**   YES

Parameters: List or row name
        ☐ Clear list
        ☐ Use search

Syntax: Merge list **list-name** [*([Clear list][,Use search])*]

This command adds the specified list to the end of the list previously specified as the current list. Once the list reaches its maximum size, the command finishes and clears the flag. OMNIS does not check that the same fields are stored in the two lists (which they should be). If the same fields are not present, data is not transferred.

If you use the **Clear list** option, the current list is initially cleared and defined to hold the same fields as the specified list. This is the same as copying a list.

If you use the **Use search** option, only lines matching the search class are merged or added to the current list. All lines match if there is no current search class.

In the following example, list *LIST1* is merged or added to the current list, namely, *LIST2*.

```
Set current list LIST2
Set search name SRCH001
Merge list LIST1 (Clear list,Use search)
If flag true
   Sort list
Else
   OK message {Merge failed at line [LIST1.$linecount]}
End If
```

This example appends selected lines only.

```
Set current list LIST2
Set search as calculation {#LSEL}
Merge list LIST1 (Use search)
```

or do it like this

```
Do LIST.$merge(AnotherList)
```

Message timeout

Reversible: NO   **Flag affected:** NO

Parameters: Interval (in seconds)

Syntax: Message timeout *{interval}*

This command specifies the time OMNIS has to wait for DDE responses to messages sent to other applications. There is a default value of 30 seconds when OMNIS is started.

The following general purpose method sets up a DDE channel by increasing the message timeout by 5 seconds until successful. You pass three parameters to the method, that is, the initial timeout, the channel number and the program 'name|document'.

```
; Open DDE
; declare Parameter LNUM (Short integer (0-255))
; declare Parameter LCHAN (Short integer (0-255))
; declare Parameter LPROGDOC (Character)
Set DDE channel number {LCHAN}
Repeat
   Message timeout {LNUM}
   Open DDE channel {LPROGDOC}
   If flag false
      Yes/No message {Give up 'Open DDE channel'?}
   End If
   Calculate LNUM as LNUM + 5
Until flag true
```

Minimize window instance

Reversible: NO   **Flag affected:** NO

Parameters: Window instance name

Syntax: Minimize window instance *window-instance-name*

This command minimizes the specified window instances, that is, the window is shown as an icon at the bottom of the OMNIS application window (or the Finder window under MacOS).

```
Open window instance WCUSTOMERS/wcust1/20/30/270/230
; let the user enter data, etc.
Minimize window instance wcust1  ;; reduces window to an icon
```

You can minimize a window using the $minimize() method. To minimize the current window use

```
Do $cwind.$minimize()
```

Modify class

Reversible: NO   **Flag affected:** YES

Parameters: Class name (Search or Report only)

Syntax: Modify class *{class-name}*

This command opens a library class in design mode. Method execution continues and does not wait for the design window to be closed. *Modify class* lets users modify new search and report classes created with the *New class* command. Opening a class in design mode when one of its methods is running causes a *Quit all methods* to be carried out before the design window opens. If the class does not exist, the command clears the flag.

```
New class {Search/S_CUSTOMERS}
Modify class {S_CUSTOMERS}
; Now you can:
Set search name S_CUSTOMERS
Print report (Use search)
```

Modify methods

Reversible: NO   **Flag affected:** YES

Parameters: Class name

Syntax: Modify methods *{class-name}*

This command opens the method editor for the specified class. Method execution continues and does not wait for the design window to be closed. Opening a method in design mode first causes a *Quit all methods* if one of the methods for that class is running. The flag is cleared if the specified class does not exist, or if it is a file, search, or report class.

```
New class {Window/W_CUSTOMERS}
Modify methods {W_CUSTOMERS}
; Opens at the $construct() method for window W_CUSTOMERS
```

New class

Reversible:	NO	**Flag affected:**	YES

Parameters: Class type
                    New class name

Syntax: New class *{class-type/new-class-name | super-class-name}*

class type can be one of the following:
    File, Task, Window, Report, Menu, Search, Code, Toolbar, Schema, Table

This command creates a new class with the specified type and class name. For example, you can use *New class* in association with *Modify class* to allow users to create new search and report classes. Attempting to create a class with the same name as one which already exists clears the flag and displays an error message.

```
New class {Window/W_CUSTOMERS}
Modify class {W_CUSTOMERS}
```

Next

Reversible:	YES	**Flag affected:**	YES

Parameters: Field name (must be indexed)
                    ☐ Exact match
                    ☐ Use search

Syntax: Next [on *field-name*] [([*Exact match*][,*Use search*])]

This command locates the next record using the current find table. The *Next* command works in the same way as the corresponding option on the **Commands** menu but with no redraw, allowing you to work through a file. It is usually used after a *Find* command which creates a find table of records.

If the Index field, Exact match and/or Search option used in the *Next* is incompatible with the preceding *Find,* a new find table is built. Normally, the parameters in this command are left blank so that the current find table is used.

If the *Next* command does not follow a *Find*, a find table is built for the current main file before doing the *Next*.

If an indexed field is specified, *Next on SU_NAME* for example, the find table is just the index order for the field. The **Use search** option creates a find table for the current main file in which the search specification is implicitly stored. Thus, changes to the search do not affect the find table once it is created.

Once the next record is located, the main and connected files are read into the current record buffer.

An error occurs whenever a *Next on FIELD* command is performed on a non-indexed field or if the field is not in the main file or its connected files.

If the next record is found, the flag is set; if not, it is cleared.

If the **Exact match** option is chosen, the next record is loaded where the index value of the specified field matches the current value.

If you use *Next* with a search, it builds a find table if necessary and finds the next record listed on the find table which meets the search criteria.

In the following example, *Next* is used without an exact match in order to work systematically through the file. As each next record is found, the flag is set and the commands in the loop are executed. When a next record cannot be found, the flag is cleared and the *Repeat–Until* loop terminated.

```
Find first on SEQ     ;; creates a table equal to the SEQ index
While flag true
   Working message {Processing records}
   Prepare for edit
   Calculate PRICE as PRICE*1.2
   Update files
   Next
End While     ;; Loop terminates at end of table


; This finds all the records where CODE = CVAR1
Single file find on CODE {CVAR1}
While flag true
   OK message {Found [NAME]}
   Next (Exact match)
End While
```

No/Yes message

Reversible: NO   **Flag affected:**   YES

Parameters: Title (for message box)
        ☐ Icon
        ☐ Sound bell
        ☐ Cancel button
        Message (text)

Syntax: No/Yes message [*title*] [([*Icon*] [,*Sound bell*]
        [,*Cancel button*])] *{message}*

This command displays a message box containing the specified message and provides a **No** and a **Yes** pushbutton. Also, you can include a **Cancel** button by checking the **Cancel button** option. When the message box is displayed method execution is halted temporarily;

it remains open until the user clicks on one of the buttons before continuing. The **No** button is the default button and can therefore be selected by pressing the Return key.

The number of lines displayed in the message box depends on your operating system, fonts and screen size. In the message text you can force a break between lines (a line return) by using the notation "//". Also you can add a short *title* for the message box.

For greater emphasis, you can select an **Icon** for the message box (the default "info" icon for the current operating system), and you can force the system bell to sound by checking the **Sound bell** check box.

You can insert a *No/Yes message* at any appropriate point in a method. If the user clicks the No button, the flag is cleared; otherwise, a Yes sets the flag. You can use the *msgcancelled()* function to detect if the user pressed the Cancel button.

```
No/Yes message (Icon,Sound bell) {The balance in this account is now
   [LBAL1].//Are you sure you want to increase the credit limit?}
If flag true
   Do method IncreaseCredit
End If
```

OK message

Reversible: NO   **Flag affected:** NO

Parameters: Title (for message box)
☐ Icon
☐ Sound bell
☐ Cancel button
Message (text)

Syntax: OK message [*title*] [*([Icon][,Sound bell] [,Cancel button])*] *{message}*

This command displays the specified message and waits for the user to click the **OK** or **Cancel** button before continuing. Method execution is halted temporarily while the message box is displayed. The number of message lines displayed depends on your operating system, fonts and screen size. In the message text you can force a break between lines (a line return) by using the notation "//". Also you can add a short *title* for the message box.

For greater emphasis, you can select an **Icon** for the message box (the default "info" icon for the current operating system), and you can force the system bell to sound by checking the **Sound bell** check box.

The message box displayed by this command has an **OK** button by default, but you can add a **Cancel** button by checking the **Cancel button** option. After executing an OK message, the flag is unchanged, but you can use the *msgcancelled()* function to detect if the user pressed the Cancel button.

You can use square bracket notation in the message text to display the current value of fields and variables. For example, the following method executes the SQL text passed to it

and displays the SQL error number and text (*sys(131)* and *sys(132)*, respectively) if there is an error.

```
; ExecSQL
; declare Parameter pSQL (Character 10000000)
; declare Parameter pAction (Character 100)
; pSQL holds the SQL text, and pAction holds the SQL command
Perform SQL {[pSQL]}
If flag false
   OK message [pAction] (Icon,Sound bell) {A SQL error occurred
   while executing [pAction].//[sys(131)]: [sys(132)]}
End If
```

On

Reversible:	NO	**Flag affected:**	NO

Parameters: Event constant or list of event constants

Syntax: On *event-constant*[,*event-constant*,...]

This command is used in an event handling method and marks the beginning of a code segment that executes when the specified event (or one of a number events) is received by the current event handling method. An *On* command also marks the end of any preceding *On* statement. You specify the event or list of events using the event constants.

When OMNIS generates an event it sends the event information as a series of event parameters to the appropriate event handling method. The first parameter is always an event constant. Further parameters, if any, depend on the event and further describe the event. This event information is interpreted by the *On* statements in the event handling methods. Window field events are sent to the $event() method behind the field, then to the $control() method for the window instance, and then to the $control() method for the current task. Events that occur in the window itself, such as a click on the window background, are sent to the class method called $event(), then to the $control() method for the current task. A particular event is sent to the first *On* command which applies, and when the next *On* command is encountered quits the method.

You should place any code which is to be executed for all events before the first *On* command. You cannot nest *On* commands or put them in an *If* or *Else* statement. You can use *On default* to handle any events not handled by an earlier On event command. The *On* commands must be in event handling methods only: if used elsewhere they are not executed. The function *sys(86)* at the start of a method reports any events received by the object.

The following example shows a typical event handling method for a window field.

```
On evBefore
   ; whatever code is needed for evBefore event
On evAfter
   ; whatever code is needed for evAfter event
On evClick, evDoubleClick
   ; whatever code is needed for evClick and evDoubleClick events
```

Seealso *Quit event handler*.

On default

Reversible: NO   **Flag affected:**   NO

Parameters: None

Syntax: On default

This command is used in an event handling method and handles any events not handled by the preceding *On* commands. You use the *On* command to mark the beginning and end of an *On* statement. You should place any code which is to be executed for all events before the first *On* command.

```
On evClick, evDoubleClick
   ; whatever code is needed for evClick and evDoubleClick events
On default
   ; this bit handles all other events
```

Open check data log

Reversible: YES   **Flag affected:**   NO

Parameters: ☐ Do not wait for user

Syntax: Open check data log [*(Do not wait for user)*]

This command opens the check data log. If the **Do not wait for user** option is specified, execution continues with the next command, otherwise execution stops until the user has closed the log. You use the check data log to manage the problems encountered in a data file after the *Check data* command is run. The data log window lets you repair any problems listed in the window, print the contents of the log, or clear the log.

```
Check data (Check indexes)
Open check data log
```

Open client import file

Reversible: NO   **Flag affected:** YES

Parameters: None

Syntax: Open client import file

This command will open the import file specified with the *Set client import file name* command. If the file already exists, OMNIS will open it and move to the end of the file where the incoming data will be appended. However, if the file does not already exist, OMNIS will create and open it. This lets you have an import file that is repeatedly added to until you are ready to import its entire contents. By preceding the *Open client import file* command with a *Delete client import file {file-name}* command, you can guarantee an empty file for the next SQL transaction.

```
Set client import file name {VAX_FILE.TXT}
Begin SQL script
SQL: Select Name, Title from Authors
End SQL script
Execute SQL script
If flag true
   Open client import file
   If flag true
      Retrieve rows to file
   Else
      OK message {couldn't open file for import}
   End If
Else
   OK message {error selecting rows from server}
End If
```

Open cursor

Reversible: NO   **Flag affected:** YES

Parameters: Cursor name

Syntax: Open cursor [*{cursor-name}*]

This command opens the specified cursor and executes any SQL statement set for that cursor. The *cursor-name* must be a currently valid cursor, but when omitted the current cursor is used. You can pass the SQL statement to be executed by this command to the named cursor using *Declare cursor*. The command is the same as:

```
Set current cursor {CURSOR_NAME}
Execute SQL script
```

Open data file

Reversible:	NO	**Flag affected:**	YES

Parameters: ☐ Do not close other data
                 ☐ Read-only Studio/OMNIS 7
                 Data file name
                 Internal name

Syntax: Open data file [*(Do not close other data)*] [*(Read-only Studio/OMNIS 7)*] *{data-file-name*[*/internal-name*]*}*

This command opens the specified data file and makes that file the "current" data file. It clears the flag if the data file cannot be found or opened. If the **Do not close other data** check box option is not specified, all existing data files are closed even if the command fails. Opening a data file which is already open will close and reopen that data file. The **Read-only Studio/OMNIS 7** check box causes the data file to be opened in read-only mode. This lets you open an OMNIS 7 data file in read-only mode in OMNIS Studio without conversion taking place.

You can override the default internal name by specifying your own in the parameter for the command, for example

Open data file {Clients.df1/Names}

If an opened data file uses more than one segment, all segments are opened. The rules for finding the additional segments which form part of the data file are as follows. Under Windows, the PATH and the paths given in the OMNIS environment variable are searched. Under MacOS, root directories of all mounted volumes are searched as well as the folders containing the first segment and the most recently opened library.

```
; example 1
Open data file {SALES.DF1/SALES}
If flag true
   Find first
   If flag true
      Open data file {PURCH.DF1/PURCHASES}(Do not close other data)
      If flag true
         Calculate PURCHASES.FIELD2 as SALES.FIELD1
         Prepare for insert with current values
         Enter data
         Update files if flag set
      End If
   End If
End If
```

```
; example 2
; Transfer from data file 1 to data file 2
Open data file {PORDERS1.DF1/PORDERS1}
If flag true
   Set main file {ORDERS}
   Find first on ORDERNUM
   While flag true
      Prepare for insert with current values
      Open data file {PORDERS2.DF1/PORDERS2}
      Update files if flag set
      Open data file {PORDERS1.DF1/PORDERS1}
      Next on ORDERNUM
   End While
End If
```

Open DDE channel

Reversible: YES   **Flag affected:**   YES

Parameters: Program name|Topic name (include the pipe)

Syntax: Open DDE channel *{program-name|topic-name}*

This command opens the current channel for exchanging data. If there is a valid response, the flag is set and the channel is successfully opened. If the channel is already open, the existing conversation is closed.

When entering the command in a method, you use the parameters to specify the program and the topic to which the message is to be addressed. Note that the "pipe" (or vertical bar) between the program name and topic name is required.

This command is reversible, that is, a previous conversation will reopen if this command is contained within a reversible block.

When the command is used in a method containing a reversible block, and if a new conversation is initiated using the same channel number as an existing conversation, the original continues to process incoming messages only, and at the end of the method, the new conversation is stopped and the original becomes fully active.

```
Set DDE channel number {2}
Open DDE channel {OMNIS|COUNTRY}
If flag false
   OK message {Country library not running}
Else
   Do method TransferData
   Close DDE channel
   OK message {Update finished}
End If
```

Open desk accessory

Reversible: NO   **Flag affected:**   YES

Parameters: Desk accessory name

Syntax: Open desk accessory *{desk-accessory-name}*

This command opens a specified desk accessory while OMNIS continues to run in the
background. Without MultiFinder, the DA opens immediately but cannot be used until
OMNIS stops running methods and waits for an input from the user.

The flag is set if the command opens the DA, and cleared if there is too little memory or the
DA is not found.

```
Yes/No message {Put this entry in SmartPad?}
If flag true
   Open desk accessory {SmartPad}
   If flag false
     OK message {Either there is too little memory
        or SmartPad is not installed}
   End If
End If
```

Open library

Reversible:	NO	**Flag affected:**	YES

Parameters: ☐ Do not close others
☐ Do not open startup task
Library name
Internal name
Password
Parameters list

Syntax: Open library [*([Do not close others*] [,*Do not open startup task*]*)]*
     {library-name[*/internal-name*[*/password*]]
     [*(parameter1*[,*parameter2*]...*)]}*

This command opens the specified library file and closes other libraries, if specified. You specify the library name (including path name if required), internal name, password, and startup method parameters of the library to be opened. If the disk file with the specified path name cannot be opened or is not a valid library, the flag is cleared and no libraries are closed.

If the internal name of an opened library is specified, a check is made to ensure the internal name is unique among the open libraries, and a runtime error occurs if this is not the case. If no internal name is specified, the default internal name is the disk name of the file with the path name and suffix removed. For example, the internal name for 'hd:myfiles:testlib.lbr' is 'testlib'.

Do not close others

The **Do not close others** option lets you keep open all other libraries. Otherwise, all other open libraries are closed (see the *Close library* command for the consequences of closing a library). If an attempt is made to open a library which is already open, that library is closed and reopened.

Startup task

If the **Do not open startup task** option is specified, the startup task construct for the opened library is not called. Otherwise, the startup task $construct() method is called and the parameters for it are passed. The startup task instance name will be either the library name or the library internal name if it has one: it is *not* called Startup_Task.

Passwords

If a password is specified, an attempt is made to open the library with that password. If it is not a valid password or no password is specified, the library is opened in the usual way, that is, if the library does not need a master password, it is opened at the master level; otherwise the usual prompt for password dialog is opened (the library is closed and a flag false returned if this dialog is closed without a password being entered).

```
Open library {MYLIB.LBR} (Param 1)
Open library (Do not close others) {SQLTOOLS.LBR}
```

The following method tries to open the named library, and uses *GetFile* if it fails. The parameter variables accept the library name and internal name passed to the method.

```
; OpenLibrary method
; Libname and IntName are passed to this method
; declare Parameter Libname (Character 10000000)
; declare Parameter IntName (Character 10000000)
If pos('.',IntName)  ;; if IntName has an extension, strip it
   Calculate IntName as mid(IntName,1,pos('.',IntName)-1)
End If
Do $root.$libs.$findname(IntName)
If flag false
   Open library (Do not close others) {[Libname]/[IntName]}
   Do $root.$libs.$findname(IntName)
   If flag false
      OK message {Can't find library}
      Get File (Libname,"Please locate the file")
      Open library (Do not close others) {[Libname]/[IntName]}
      Do $root.$libs.$findname(IntName)
      If flag false
         OK message {Still can't open library!}
      End If
   End If
End If
```

Open lookup file

Reversible: YES   **Flag affected:**   YES

Parameters: Lookup reference or label
                  Data file name
                  File class name
                  Index field number

Syntax: Open lookup file *{[lookup-reference/]data-file-name/file-name/field-number}*

This command opens an OMNIS data file for use as a lookup file. You give each lookup file a reference name which you use in subsequent *lookup()* functions to select a particular data file and file class.

You can open any OMNIS data file as a lookup file. In a lookup file, you can use the file classes to look up field values based on an indexed search. Each file class should consist of

at least two fields: the first is the index (usually a character field), the second is any field type. For example, the data file LAREAS.DF1 has the following file structure:

File name	Field1	Field2
FPIC	Char Indexed	Picture
FCITIES	Char Indexed	Char

The parameters for *Open lookup* are separated by "/". The first parameter is a label that you create to become the reference to that lookup "channel". If you omit this label, OMNIS assumes that you will use only one lookup file whereupon you can use *lookup()* without its first parameter. The label you give to each lookup is case-insensitive and if you use the same one twice, the previous lookup file is closed. A flag true is returned if the data file is found and opened. Here is a typical example:

```
Open lookup file {City/HD:LOOKUP.DF1/FCITIES/1}
If flag true
   OK message {The city you require is [lookup('City','I',2)]}
End If
```

This example opens a data file called LOOKUP.DF1 and assigns the label "City" to the lookup channel. The City lookup uses the file class FCITIES within that data file and uses the first index to search for the required data. The OK message uses *lookup()* to search the first indexed field for an exact match with the value "I". If the match is found, the value of field 2 in the matched record is returned and displayed as part of the OK message. If no match is found, *lookup()* returns an empty value.

Note that the index and field are specified as *numbers* because your particular library may not include the file class used in the lookup data file. If you omit either number, the default is to use the first field as the index, and the second as the field value to be returned in the lookup() function.

OMNIS looks for the data file using the following rules. Under Windows, the current PATH and additional paths included in the OMNIS environment variable are searched. The AUTOEXEC.BAT file sets up the environment variables, for example

```
PATH C:\;C:\WINDOWS;C:\DOS;C:\DOS\TOOLS
SET OMNIS=C:\WINDOWS\LOOKUPS
```

Under MacOS, the System folder, OMNIS folder and then the root of each mounted volume is searched, in that order.

You can open more than one file class within a particular data file by assigning a different label to each lookup, for example

```
Open lookup file {City2/LOOKUP.DF1/FCITIES2}
Open lookup file {City/LOOKUP.DF1/FCITIES}
Open lookup file {Country/LOOKUP.DF1/FCOUNTRIES}
```

The flag is set if the lookup is successful, that is, the data file is opened, the file slot exists and the indexed field is indeed indexed. The lookup file is closed if the command is reversed (see *Begin reversible block*).

You can close lookup files using *Close lookup file*, but this is not necessary: all lookup files associated with a library are closed automatically when the library quits.

You can maintain the data within the lookup file from within the library by:

1. Adding the appropriate file classes to your library,

2. Changing the data file to the lookup file using *Open data file*,

3. Opening a window and editing/ inserting data in the usual way, and

4. Returning to the original data file.

You can also load multiple data files with *Open data file*.

Open runtime data file browser

Reversible: NO   **Flag affected:**   NO

Parameters: None

Syntax: Open runtime data file browser

This command opens a restricted version of the Data File Browser suitable for use in the runtime version of OMNIS. The Runtime Data File Browser lets you check data files and individual data slots. The IDE Data File Browser, and the runtime version are mutually exclusive, that is, opening one closes the other. If the Runtime Data File Browser is already open, executing this command brings it to the front.

```
Open data file {Salaries.df1}
Set current data file {Saleries}
Open runtime data file browser
```

Open task instance

Reversible: NO   **Flag affected:**   NO

Parameters: Task class name
    Instance name (the default is the class name)
    Parameters list

Syntax: Open task instance ***task-class-name***[/task-*instance-name*]
    [(*parameter1*[,*parameter2*]...)]

This command opens the specified task and assigns an instance name. You can include a list of parameters which are sent to the $construct() method in the task instance. Note that startup task instance is normally opened when the library opens: its name will be either the library name or the library internal name if it has one.

```
Open task instance Task1 (1)


; $construct for Task 1
; declare Parameter pMenu of type Boolean
If pMenu
   Install menu mAccounts
End If
```

Open trace log

Reversible: NO   **Flag affected:**   NO

Parameters: None

Syntax: Open trace log

This command opens the trace log.

Open window instance

Reversible:	YES	**Flag affected:**	NO

Parameters: Window class name

Window instance name   default is the class name

left/top/right/bottom/   to position and size window; no. of pixels

/CEN   to center the window

/MAX   to maximize the window

/MIN   to minimize the window

/STK   to stack the window

Parameters list

Syntax: Open window instance ***window-class-name***[/window-*instance-name*]
[/*left*[/*top*[/*right*[/*bottom*]]]] [/*CEN*] [/*MAX*] [/*MIN*] [/*STK*]
[(*parameter1*[,*parameter2*]...)]

This command opens an instance of the specified window class. You can specify the position and size of the window instance (using the left, top, right, bottom coordinates in pixels), and you can center, maximize, minimize, and stack the window. Furthermore, you can send a list of parameters to the window's $construct() method.

Open window instance lets you open multiple instances of the same window class. The default instance name for a window is the class name, but if you want to open multiple instances of the same window class you must assign a unique name to each instance. Window instance names are case-sensitive.

```
Open window instance WCLIENT/winst1/stk
Open window instance WCLIENT/winst2/stk
; will open and stack two instances of the WCLIENT window
```

Alternatively you can let OMNIS assign enumerated names to multiple instances by specifying '*' as the instance name.

```
Open window instance WCLIENT/*
Open window instance WCLIENT/*
; will open two instances WCLIENT1 and WCLIENT2
```

Window Position and Size

You can specify the position of the top-left corner of the window instance by adding the coordinates to the end of the window name/instance name parameter, that is, *window-name*/*instance-name*/left/top. You specify the position in pixels, the origin being /0/0, that is, *under* the menu bar. By providing all four coordinates, you can specify the position and size of the window instance.

```
Open window instance WCLIENT/winst1/20/30/200/300/stk
Open window instance WCLIENT/winst2/20/30/300/400/stk
```

You can use variables to locate a window instance, for example

```
Open window instance
   WPALETTE/wpal1/[LVLeft]/[LVTop]/[LVRight]/[LVBott]
```

Centering and Stacking Windows

The **/CEN** option automatically centers the window instance. You can include the four window size coordinates with the **/CEN** option so the window is sized and centered.

```
Open window instance WCLIENT/winst1/20/30/200/300/CEN
```

The **/STK** option opens the window instance about 12 pixels (the stack offset) to the right and down from the current top window. When a stacked window reaches the edge of the screen, it is placed back at the top of the stack, offset slightly from the first window.

```
Open window instance WPALETTE/wpal1/STK
```

Maximizing and Minimizing Windows

The **/MAX** option opens and maximizes the window instance. If you include the position and size coordinates with this option, the window is opened with the specified position and size and then maximized.

```
Open window instance WCLIENT/winst2/20/30/200/300/MAX
; opens the window at 20/30/200/300 and then maximizes it
```

The **/MIN** option opens and minimizes the window instance. If you include the position and size coordinates with this option, the window is opened with the specified position and size and then minimized.

```
Open window instance WCLIENT/winst3/30/40/250/350/MIN
; opens the window at 30/40/250/350 and then minimizes it
```

$construct() Method and Passing Parameters

When you open a window instance, the $construct() method for that instance is run. In this method, you place commands which set up the conditions required by the window. For example, you may want to set the main file, build particular lists, and so on. Just as with *Do method* and *Do code method* you can send parameters to the window using *Open window instance*, for example

```
Open window instance WCLIENT/winst2 (CVAR1,LVAR1,CO_NAME)
```

In this case, the values held in CVAR1, LVAR1, and CO_NAME are passed to the $construct() method for the WCLIENT window instance.

Reversible blocks in the $construct() method do not reverse until the window instance is closed, unlike a normal method whose reversible blocks reverse on termination of the method.

Alternatively you can use the $open() method to open a window.

```
$windows.WINDOW.$open('instancename'[,location,constructparams])
```

Optimize method

Reversible: YES   **Flag affected:**   NO

Parameters: None

Syntax: Optimize method

This command stores an optimized form of the method so when the method is executed for a second time it runs much faster. You should put this command at the beginning of a method, except when you put it in a reversible block. Methods which are executed frequently, such as control methods and loops, are best optimized. The command is reversible and does not change the flag.

When *Optimize method* is executed for the first time it converts the method being executed into its optimized form and continues execution. When the method terminates, the optimized form of that method is kept in RAM; the optimized form is executed if the method is called again. If *Optimize method* is in a reversible block the optimized form of the method is disposed of when the method terminates; so it will be rebuilt each time the method executes. The optimized method is also discarded whenever the design window is open for the method or the method is modified using the notation.

```
Optimize method
Set main file fRequests
Set current list BookRequests
Define list {ReqId,BookName,CollegeName}
Set search name sPhysics
Find first on ReqId (Use search)
While flag true
   Single file find on BookId (Exact match) {ReqBkId}
   Single file find on CollegeId (Exact match) {ReqCollegeId}
   Add line to list
   Next
End While
```

WARNING Optimizing too many methods will increase the memory used which may eventually result in a slowdown or worse.

OR selected and saved

Reversible: NO   **Flag affected:**   YES

Parameters: Line number (can be calculation, default is current line)
            ☐ All lines

Syntax: OR selected and saved [*(All lines)*] [*{line-number}*]

This command performs a logical OR of the Saved selection with the Current selection. To allow sophisticated manipulation of data via lists, a list can store two selection states for each line; the "Current" and the "Saved" selection. The Current and Saved selections have nothing to do with saving data on the disk; they are no more than labels for two sets of selections. The lists may be held in memory and never saved to disk: they will still have a Current and Saved selection state for each line but they will be lost if not saved. When a list is stored in the data file, both sets of selections are stored.

You can specify a particular line in the list by entering either a number or a calculation.

The *OR selected and saved* command performs a logical OR on the Saved and Current states and puts the result into the Current selection. Hence, if either or both the Current and Saved states are selected, the Current state becomes selected, but if both states are deselected, the resulting Current state will remain deselected.

Logic Table (S=selected, D=deselected)

Saved	Current	Resulting Current State
S	S	S
D	S	S
S	D	S
D	D	D

The **All lines** option performs the OR on all lines of the current list. The following example selects all lines of the list.

```
Set current list LIST1
Define list {LVAR1}
Calculate LVAR1 as 1
Repeat
   Add line to list
   Calculate LVAR1 as LVAR1+1
Until LVAR1=6
Select list line(s) (All lines)
Save selection for line(s) (All lines)
Invert selection for line(s) {3}
OR selected and saved (All lines)
Redraw lists
```

Paste from clipboard

Reversible: NO   **Flag affected:** YES

Parameters: Field name
   ☐ Redraw field
   ☐ All windows

Syntax: Paste from clipboard [*field-name*] [([*Redraw field*][,*All windows*])]

This command pastes the contents of the clipboard into the specified field, current selection or at the insertion point. When the field name parameter is specified, *Paste from clipboard* pastes the contents of the clipboard into the field replacing the contents of the whole field. However, when the field name parameter is not specified the command will paste the contents of the clipboard at the current selection (a range of selected characters) or the insertion point within the current field.

```
; copies one field to another then clears the first field
Copy to clipboard C_NAME
Paste from clipboard C_COMPANY (Redraw field)
Clear data C_NAME (Redraw field)
```

Perform SQL

Reversible: NO   **Flag affected:**   YES

Parameters: SQL script

Syntax: Perform SQL *{sql-script}*

This command sends a SQL statement direct to the current session without loading the SQL buffer. It replaces the sequence

```
Begin SQL script
SQL: Select Name from Clients where Name like J
End SQL script
Execute SQL script
```

The flag is set if the server accepts the SQL statement. You can use the functions *sys(131)* and *sys(132)* to report any errors returned from the server, for example

```
Set current session {Session_Ora}
Reset cursor(s) (Current)
Perform SQL {Select Name from Clients where Name like J}
If flag false
   OK message {Error returned: [sys(132)]}
End If
```

Popup menu

Reversible: NO   **Flag affected:**   YES

Parameters: Menu name
                    x coordinate, y coordinate

Syntax: Popup menu *menu-name ([x-coordinate,y-coordinate])*

This command installs the specified menu as a popup menu at the specified location. The location is the *x,y* screen coordinate relative to the (0,0) position. Under Windows, the coordinate (0,0) is the point directly under the menu bar within the OMNIS application window. Under MacOS, (0,0) is literally the top left corner of the screen. If you omit the x,y coordinates the menu pops up at the current mouse position.

The *mouseover()* function returns the mouse position relative to the open window and not the OMNIS application window. Using this function to specify the x and y position of the popup menu may not produce the effect you want.

Popup menu behaves much like *Popup menu from list* except the source of the popup is a user-defined menu. It clears the flag if the user does not select a menu line, otherwise, the line selected from the popup is executed.

```
; $event for window class
On evRMouseDown
   Popup menu mContext3
```

Popup menu from list

Reversible: NO   **Flag affected:**   YES

Parameters: List name
                  x coordinate, y coordinate

Syntax: Popup menu from list *list-name ([x-coordinate,y-coordinate])*

This command installs the specified list as a popup menu at the specified *x,y* screen location. Under Windows, the coordinate 0,0 is the point directly under the menu bar within the application area. Under MacOS, 0,0 is literally the top left corner of the screen. If you omit the x,y coordinate from this command the menu pops up at the current mouse position.

Popup menu from list behaves much like *Popup menu* except the source of the menu is a list. The specified list can contain any number of rows but only the first column and a limited number of rows are displayed in the popup menu.

This command clears the flag if the user does not select a list line and *LIST.$line* is unaffected. After the command has executed you can use *lst()* to return the line selected.

The *mouseover()* function returns the mouse position relative to the open window and not the OMNIS application window. Using this function to specify the x and y position of the popup menu may not produce the effect you want.

```
; $event for window class
On evRMouseDown
   Popup menu from list cList
```

Prepare current cursor

Reversible: NO   **Flag affected:**   YES

Parameters: None

Syntax: Prepare current cursor

This command sends the SQL statement given to the current cursor to the DAM in order that it can interpret and understand the statement. This is normally done implicitly as part of an execute, but you can make this explicit using this command. Consider that a statement has 3 stages:

1. Prepare—always needed

2. Execute—always needed

3. Describe—only needed if the statement has results, a Select, for example

When sending a SQL statement using *Perform SQL* and *Execute SQL script* the SQL script is parsed and interpreted by the server. Should you wish to send the same statement again, this preparation stage can be bypassed to save time by using *Prepare current cursor*. For example:

```
Begin SQL script
SQL: INSERT INTO Sales(col1,col2,..) VALUES (@[col1],@[col2], .. )
End SQL script
Prepare current cursor
Execute SQL script
```

Subsequent use of *Execute SQL script* on the same cursor will execute the same statement without having to set up the SQL buffer each time since the indirection and bind variables are already prepared. This can greatly speed up the process of, say, inserting many rows into a server table within a loop.

```
Set current cursor { Cursor1 }
Repeat
   ; get next row
   Execute SQL script   ;; insert the row
Until ..                ;; no more rows to insert
```

Prepare for edit

Reversible: YES   **Flag affected:**   YES

Parameters: None

Syntax: Prepare for edit

This command prepares OMNIS for editing data. It brings records into memory ready for updating and rereads the current records when in multi-user mode in case another user has made a change to a record since it was read in. Your method can then alter the values of the records. The contents of the current record buffer are not written back to disk until *Update files* is encountered.

If there is a window open and you require data to be entered via that window, *Enter data* is required after the *Prepare for edit*.

The **Edit** option on the **Commands** menu is, in fact, equivalent to the commands:

```
Prepare for edit
Enter data
If flag true
   Update files
Else
   Clear main & connected
   Redraw MyWindow
End If
```

Prepare for edit/insert mode is cleared only by a *Cancel prepare for update, Update files* or *Quit all methods* command. You can build lists, print reports and change the main file in the middle of an update without canceling the Prepare for... mode.

Multi-user considerations

Records in the current record buffer from Read/write files will be locked when *Prepare for edit* is executed, so as to prevent simultaneous editing of a record. The lock is removed by *Update files* or any command which cancels the Prepare for mode.

If *Wait for semaphores* is active, a *Prepare for edit* will wait for a record to become available if another workstation has locked it. If the user presses Ctrl-Break (or Cmnd-period under MacOS) while waiting for access, the command fails and processing halts. With *Do not wait for semaphores* active, a record lock returns control to the method with the flag false.

In the following method, the Edit mode is used to process the whole of a file. *Enter data* is not used as no user intervention is required. *Update files* writes data to the disk and clears the Prepare for.. mode and record locks.

In 'Wait for semaphores' mode:

```
Set main file MYFILE
Find first on MYINDEX
While flag true
   Prepare for edit
   Calculate CLBALN as CLBALN - TCCOST
   Update files
   Next
End while
```

In 'Do not wait for semaphores' mode:

```
Set main file MYFILE
Find first on MYINDEX
While flag true
   Repeat
      Prepare for edit
   Until flag true
   Calculate CLBALN as CLBALN - TCCOST
   Repeat
      Update files
   Until flag true
   Next
End while
```

In the next Edit example, the *Enter data* command is included in the method so that the user can edit the record from the keyboard. Again, the command *Update files* cancels the Prepare for update mode and writes data to the disk.

```
Prepare for edit
Enter data
Update files if flag set
```

The next example has been written to control record locking by preventing OMNIS from waiting for a record lock. It takes the form of general purpose 'prepare for edit' which you can call with a number which tells it how many times to try for a lock if the record is locked by another user:

```
; general Prepare for edit
; declare Parameter TRIES (Number 0 dp)
Do not wait for semaphores
Calculate COUNT as 1
Repeat
   Prepare for edit
   Calculate COUNT as COUNT+1
Until #F | (COUNT>TRIES)
; Keeps trying until flag true OR counter>TRIES
Wait for semaphores
```

Prepare for export to file

Reversible: NO   **Flag affected:** YES

Parameters: Export format

Syntax: Prepare for export to file *{export-format}*

> *export-format* is one of the following: *Delimited (commas),*
> *Delimited (tabs), One field per line, OMNIS data transfer*

This command prepares to export records to a file in one of the specified data formats. The file must previously have been set using *Set print or export file name*.

```
Set print or export file name {Export.txt}
Prepare for export to file {Delimited (commas)}
Export data LIST1
End export
```

Prepare for export to port

Reversible: NO   **Flag affected:** YES

Parameters: Export format

Syntax: Prepare for export to port *{export-format}*

> *export-format* is one of the following: *Delimited (commas),*
> *Delimited (tabs), One field per line, OMNIS data transfer*

This command prepares to export records to a port in one of the specified data formats. The file must previously have been set using *Set port name* or *Prompt for port name*.

```
Set port name {COM1:}
Prepare for export to port {Delimited (commas)}
Export data LIST1
End export
```

Prepare for import from client

Reversible: NO   **Flag affected:** YES

Parameters: Data format

Syntax: Prepare for import from client *{data-format}*

> *data-format* is one of the following: *Delimited (commas),*
>   *Delimited (tabs), One field per line, OMNIS data transfer*

This command prepares to import records from the DDE client in the specified data format; it is a DDE command, OMNIS as server. The data referred to in the subsequent *Import data* commands are imported from the DDE client. A single DDE Poke message received from the client contains a complete record. As each field is received, it is read into the fields in the top window in field order.

If the imported record contains more fields than there are in the window, the extra ones are ignored. Conversely, if there are too few, the extra fields in the window are left blank.

```
Open window instance W_import_data
Prepare for import from client {Delimited (tabs)}
If flag true
   Import data {ImportList}
End If
End import
```

Prepare for import from file

Reversible: NO   **Flag affected:** YES

Parameters: Data format

Syntax: Prepare for import from file *{data-format}*

> *data-format* is one of the following: *Delimited (commas),*
>   *Delimited (tabs), One field per line, OMNIS data transfer*

This command prepares OMNIS for a series of *Import data* commands. You must specify the format for the import data as the parameter, otherwise an error will occur. The parameter can contain square bracket notation but must evaluate to a valid import format name. You should use the *Set import file name* command to specify the name of the file to be read in.

If the data matches the specified import format, the flag is set. However, if the data does not match the import format, the flag is cleared.

When data is imported via a method rather than the **Utilities** menu, you must open a window which defines the fields in which the incoming data must be placed. The example below shows a typical import data method.

You can use a $control() method in conjunction with the *Import data* command.

```
Open window instance W_import_window
; Set control method (optional)
Set import file name {data1.DBF}
Prepare for import from file {Delimited (tabs)}
If flag true
   Import data {ImportList}
End If
End import
Close import file
```

If there are too few fields on the window, imported fields will be lost. If there are too many, the extra fields are cleared. You can use the *Do not flush* command to speed up the import when there is only one user logged into the data file.

Prepare for import from port

Reversible: NO   **Flag affected:**   YES

Parameters: Data format

Syntax: Prepare for import from port *{data-format}*

       data-format is one of the following: *Delimited (commas),
          Delimited (tabs), One field per line, OMNIS data transfer*

This command prepares OMNIS for importing data from a port. It is similar to the *Prepare for import from file* command. The user can cancel the import of data while *Prepare for import* is waiting for data from the port. If this happens, OMNIS clears the flag.

Set port name defines which port is used. Under MacOS, the choice is 1 (Modem port) or 2 (Printer port). Under Windows 3.x, the choices are Com1:, Com2:, and so on.

```
Open window instance W_import_window
; Set control method (optional)
Set port name {1 (Modem port)}
Set port parameters {1200,n,7,2}
Prepare for import from port {Delimited (commas)}
If flag true
   Import data {ImportList}
End If
End import
Close port
```

Prepare for insert

Reversible: YES **Flag affected:** YES

Parameters: None

Syntax: Prepare for insert

This command prepares OMNIS for inserting new data into the main file. It clears the main file and prepares to insert a new record into the main file. All Read/write non-main file records in the current record buffer are reread if a record has been changed. You can edit data in all the read/write files in the buffer, other than the main file.

Prepare for insert is *not* the same as the **Insert** option on the **Commands** menu which is in fact equivalent to:

```
Prepare for insert
Enter data
If flag true
   Update files
Else
   Clear main & connected
   Redraw MyWindow
End If
```

The *Enter data* command is required only if the user is to enter data via a window. Data is not written to the disk until *Update files* is executed.

Prepare for edit/insert mode is cleared only by a *Cancel prepare for update, Update files* or a *Quit all methods* command. You can build lists, print reports and change the main file in the middle of an insert without canceling the Prepare for... mode.

If the main file is changed while in Prepare for insert mode, the main file at the time of the *Prepare for insert* is used when *Update files* is encountered.

In multi-user mode, the *Prepare for...* commands reread the current records from the data file if another user has edited a record.

Prepare for insert with current values

Reversible: YES   **Flag affected:** YES

Parameters: None

Syntax: Prepare for insert with current values

This command prepares OMNIS for inserting new data into the main file using the values in the current record buffer as a starting point. *Prepare for insert with current values* differs from *Prepare for insert* in that the fields in the main file are not cleared.

In multi-user mode, the *Prepare for...* commands reread the current records from the data file if another user has edited a record.

```
Set main file {ORDERS}
Prepare for insert with current values
If flag false
   Quit method kFalse
End If
Enter data
Update files if flag set
```

Prepare for print

Reversible: YES   **Flag affected:** YES

Parameters: ☐ Ask for job setup
                 ☐ Do not finish other reports
                 Report instance name (default is the class name)
                 Parameters list

Syntax: Prepare for print [*(Do not finish other reports)*]
                 [*{report-instance-name}*] [*(parameter1*[*,parameter2*]*...)*]

This command prepares OMNIS for record-by-record report printing. You specify the report instance name and you can add a list of $construct parameters for the report instance. The default instance name is the name of the report class itself.

You must put *Prepare for print* after any *Set report name*, *Select destination...*, *Set port name*, *Set print file name*, *Set sort field* and *Report parameter* commands and before the first *Print record* command.

The **Ask for job setup** option opens the job setup dialog that lets you select the number of copies, paper trays, the printer, and so on, for the current print job.

Prepare for print has the **Do not finish other reports** option which when checked allows multiple reports to be in progress at the same time. If this is unchecked (the default) all reports in progress are terminated before the new report is started, which is compatible with earlier versions of OMNIS.

The flag is set if the command is successful, errors cause a message to be displayed. If placed in a reversible block, the *Prepare for print* mode is canceled and the totals printed when the command is reversed.

All the Print commands give an error if no report is selected, or if the report is printed to a port and no port is selected.

When reports are printed record-by-record using *Print record* in a loop, the sort fields set up in the report class still trigger the subtotals. No sorting takes place and, therefore, you must take care in the choice of index. You can trigger subtotals from the method by including a variable on the first line of the report class, including it in the sort fields and then using the method to change its value when required.

The Prepare for print mode is terminated or canceled by *End print*; You *must* include an *End print* after a *Prepare for print* even if a totals section is not required.

```
Perform SQL {Select * from FELEMENTS}
Prompt for destination
If flag true
   Fetch row
   Set report name report1
   Prepare for print
   Repeat
      Working message (Repeat count) {Printing}
      Print record
      Fetch row
   Until flag false
   End print
End If
```

```
; Example 2
; CVAR1, used to trigger subtotals section 1, is a sort field and
; placed on line 1 of the report class
Set report name RS_CONTACTS
Set main file {CONTACTS}
Prompt for destination
If flag true
   Prepare for print
   Find first
   While flag true
      Calculate CVAR1 as TOWN
      Print record
      Next
   End While
   End print
End If
```

Previous

Reversible:	YES	**Flag affected:**	YES

Parameters: Field name (must be indexed)
       ☐ Exact match
       ☐ Use search

Syntax: Previous [on *field-name*] [*([Exact match][,Use search])*]

This command locates the previous record using the current find table. The *Previous* command works in the same way as the corresponding option on the **Commands** menu but with no redraw, allowing you to work through a file. It is usually used after a *Find* command which creates a find table of records.

If the Index field, Exact match and/or Search option used in the *Next* is incompatible with the preceding *Find,* a new table is built. Normally, the parameters in this command are left blank so that the current find table is used.

If the *Previous* command does not follow a *Find*, a find table is built for the current main file before doing the *Previous*.

If an indexed field is specified, *Previous on SU_NAME* for example, the find table is just the index order for the field. The **Use search** option creates a table for the current main file in which the search specification is implicitly stored. Thus, changes to the search do not affect the find table once it is created.

Once the previous record is located, the main and connected files are read into the current record buffer and the flag is set, otherwise, the flag is cleared. An error occurs whenever *Previous on FIELD* is performed on a non-indexed field.

If the **Exact match** option is chosen, the previous record with the same index value is found, or the flag is cleared if no previous records exist with the same index value.

If you use *Previous* with a search, it finds the previous record listed on the index table which meets the search criteria.

In the following example, *Previous* is used without an exact match in order to work systematically through the file. As each previous record is found, the flag is set and the commands in the loop are executed. When a previous record cannot be found, the flag is cleared and the *Repeat*–Until loop terminated.

```
Find last on SEQ     ;;  this creates a table equal to the SEQ index
While flag true
   Working message {Processing records}
   Prepare for edit
   Calculate PRICE as PRICE*1.2
   Update files
   Previous
End While
```

Print check data log

Reversible: NO   **Flag affected:**   NO

Parameters: None

Syntax: Print check data log

This command prints the current contents of the check data log to the current report destination. There is no need for the log to be open.

```
Check data (Check indexes)
If flag true
   Print check data log
Else
   OK message {Check data works only if one user is logged on}
End If
```

Print class

Reversible: NO   **Flag affected:**   NO

Parameters: Class name

Syntax: Print class *class-name*

This command prints the field list and methods (if any) for the specified class. The following example prints the field list and/or methods for all the classes in the current library.

```
; declare local variable FLIST of List type
Set current list FLIST
Calculate FLIST as $clib.$classes.$makelist($ref.$name)
Redefine list {CVAR5}
For each line in list
   Print class {[lst(CVAR5)]}
End for
```

Print record

Reversible: NO   **Flag affected:**   YES

Parameters: Report instance name

Syntax: Print record [*{report-instance-name}*]

This command prints a single record of the specified report instance. You use it when printing a report on a record-by-record basis and usually within a loop. It provides greater control over the report generator than *Print report*. If you omit the report instance name *Print record* is applied to the most recently started report instance ($ireports.$first).

Each time *Print record* is encountered, a record section of the report is printed to the selected output using the data in the CRB. Any page heading, subtotal heading and subtotal sections before the record section are printed where necessary.

Subtotal sections are printed whenever the sort fields change value, provided that the fields entered in the Sort Fields dialog have **Subtotals** set to True.

The flag is cleared if:

❑   no *Prepare for print* is used, or

❑   the user cancels the report by pressing Ctrl-Break or Cmnd-period, or

❑   there is an error.

These errors will not cause OMNIS to execute a *Quit all methods*. If the flag is cleared, OMNIS will not execute any further *Print record* commands until it encounters another *Prepare for print*.

```
; example 1
Set main file {f_client}
Set report name r_letters
Send to screen
Prepare for print
Find first
While flag true
   Print record
   Next
End While
End print


; example 2
Perform SQL {Select C_CODE,C_NAME from CUST}
Set report name R_CUST
Prepare for print
Fetch row
While flag true
   Print record
   Fetch row
End While
End print
```

Print report

Reversible:	NO	**Flag affected:**	YES

Parameters: ☐ Ask for job setup
                    ☐ Use search
                    ☐ Do not finish other reports
                    Report instance name (default is the class name)
                    Parameters list

Syntax: Print report [*([Use search][,Do not finish other reports])*]
                     [*{report-instance-name}*] [*(parameter1[,parameter2]...)*]

This command prints the specified report instance to the selected output. You specify the report instance name and you can add a list of $construct parameters for the report instance. The default instance name is the name of the report class itself.

Subtotal sections are printed whenever the sort fields change value, provided that the fields entered in the Sort Fields dialog have **Subtotals** set to True.

You specify sort fields and the main file or list as part of the report parameters. If the main file has not been set in the report class, the current main file is used. You can override all the parameters in the class using the appropriate commands, for example, *Set left margin*.

Print report does not use the current record buffer but a special memory buffer to load in and sort records. Thus *Print report* does not affect Prepare for mode and does not lose current records. If the report is printed from a list, data is read directly from the report main list, as specified in the parameters for the report. *LIST.$line* is unaffected.

The **Ask for job setup** option opens the job setup dialog that lets you select the number of copies, paper trays, the printer, and so on, for the current print job.

All records are printed unless the **Use search** option is specified. In this case, only the records matching the current search class are printed. It is not necessary to use *Prepare for print* before *Print report*.

The **Do not finish other reports** option allows multiple reports to be in progress at the same time. If this is unchecked (the default) all reports in progress are terminated before the new report is started, which is compatible with earlier versions of OMNIS.

The flag is cleared if the report is canceled before completion by the user or in the event of an error. Most errors will display a message but will not cause OMNIS to *Quit all methods*.

```
Set report main file {f_client}
Set report name r_letters
Clear sort fields
Set sort field ORDERCODE
Send to screen
Print report
```

Print report from disk

Reversible: NO   **Flag affected:**   YES
Parameters: File name
Syntax: Print report from disk {*file-name*}

This command prints the contents of the specified disk file to the current report destination. The specified file must contain output generated using the Disk printing device.

Print report from memory

Reversible: NO   **Flag affected:**   YES

Parameters: Field or variable name

Syntax: Print report from memory *var-name*

This command prints the contents of the specified binary field or variable to the current report destination. The specified field or variable must contain output generated using the Memory printing device.

Print top window

Reversible: NO   **Flag affected:**   YES

Parameters: None

Syntax: Print top window

This command prints the top window to the current print destination. It behaves the same as the **Window>>Print Top** menu option.

```
Send to printer
Print top window
```

Process event and continue

Reversible: NO   **Flag affected:**   NO

Parameters: ☐ Discard event

Syntax: Process event and continue [*(Discard event)*]

This command causes the current event to be processed immediately allowing the event handler method containing the command to continue to execute. Normally, the default processing for an event takes place when all the event handler methods dealing with the event have finished executing. It is not possible to have active unprocessed events when waiting for user input so the default processing is carried out for any active events after an Enter data command has been executed or at a debugger break Therefore if required, you can use this command to override the default behavior and force events to be processed allowing the event handler method to continue.

The **Discard event** option lets you discard the active event. For example, in an event handler for evOK the following code would cause the OK event to be thrown away before the subsequent enter data starts.

```
On evOK
   Process event and continue (Discard event)
   Open window instance {window2}
   Enter data
```

Prompt for data file

Reversible: NO   **Flag affected:**   YES

Parameters: Internal name
                   ☐ Do not close other data
                   ☐ Read-only Studio/OMNIS 7

Syntax: Prompt for data file [*(Do not close other date)*] [*(Read-only Studio/OMNIS 7)*] *{internal-name}*

This command prompts the user to enter the name of a data file. A dialog box is displayed that lets the user choose a data file. An error message "Unable to find data file" is generated if the selected file cannot be opened, and the user is forced to select another file name or Cancel. If the user selects Cancel, the flag is cleared and the original data file remains selected.

The selected file is opened in shared mode unless the volume does not support record locking.

The existing open data files remain open if the **Do not close other data** option is selected. In this case, the new data file becomes the "current" data file and this becomes the default data file for file classes which have not been associated with a particular data file using the *Set default data file* command. If the **Do not close other data** option is not specified, all other open data files are closed even if the command fails.

If an attempt is made to open a data file which is already open, that data file is closed and reopened. The **Read-only Studio/OMNIS 7** check box causes the data file to be opened in read-only mode. This lets you open an OMNIS 7 data file in read-only mode in OMNIS Studio without conversion taking place.

```
Test if file exists {ORDERS.DF1}
If flag true
   Open data file {ORDERS.DF1}
Else
   Prompt for data file
   If flag false
      Quit method
   End If
End If
```

Prompt for destination

Reversible: YES   **Flag affected:**   YES
Parameters: None
Syntax: Prompt for destination

This command displays the Set report destination window so the user can select the destination for the report. The user can choose the following destinations: printer, screen, page preview, file, port, clipboard, or DDE channel.

If the command is part of a reversible block, the destination reverts to its former identity when the method terminates. If the user selects the Cancel button on the dialog, the flag is cleared.

```
Set report name Orders
Prompt for destination
If flag true
   Print report
End If
```

Prompt for event recipient

Reversible: NO   **Flag affected:**   YES
Parameters: Recipient tag name
Syntax: Prompt for event recipient [*{recipient-tag}*]

This command prompts the user to select the name of an application which will become the destination of all subsequent events. The "recipient tag" is entered also. Recipient tags may have a maximum of 31 characters to comply with the MacOS Finder. Several recipients may be prompted for, each with a different tag, but you can use only one at a time. A complete list of current recipients is built with the *Build list of event recipients* command.

If no recipient tag is specified for the application, the tag will be supplied by OMNIS. Its name will be capitalized and spaces removed. Once an event recipient has been tagged, you can use the tag as a parameter for the *Use event recipient* command without further prompting, thus allowing recipients to be changed easily.

```
; This example is a pushbutton method which sets up two recipients
On evClick
   OK message {Locate the Excel spreadsheet for me}
   Prompt for event recipient {Sheet}     ;; tagged as "Sheet"
   OK message {Locate the remote database for me}
   Prompt for event recipient {Data}      ;; tagged as "Data"
   Set current list LIST1
   Build list of event recipients
   Redraw lists
   Set event recipient {[LIST1(1,1)]}
   ; Uses the first recipient in list, that is, R1,C1
On default
   Quit event handler
```

Prompt for import file

Reversible: YES   **Flag affected:**   YES

Parameters: None

Syntax: Prompt for import file

This command prompts the user to select the name of the import file. The flag is set if the import file is successfully selected, otherwise a Cancel clears the flag, closes the current file and closes the dialog. You use the selected file in any subsequent *Import data* commands.

If you use *Prompt for import file* in a reversible block, the import file is closed when the method containing the reversible block terminates.

```
Open window instance W_IMPORT
Prompt for import file
Prepare for import from file {Delimited(tabs)}
If flag true
   Import data {ImportList}
End If
End Import
Close import file
```

Prompt for input

Reversible: NO   **Flag affected:**   YES

Parameters: □ Sound bell
                  □ Cancel button
                  □ Upper case only
                  □ Password entry
                  □ Prompt above entry
                  Prompt text
                  Title
                  Icon id
                  Maximum characters
                  Return field

Syntax: Prompt for input [([*Sound bell*] [,*Cancel button*]
              [,*Upper case only*] [,*Password entry*] [,*Prompt above entry*])]
              prompt-text/[*title*]/[*icon-id*]/[*max-chars*] Returns ***return-field***

This command opens a message box requesting a value from the user. You can specify the text for the prompt, title and icon for the message box, and the maximum number of characters for the input. If the user enters a value and presses OK, the command sets the flag and returns the user value. The command is not reversible.

The first parameter for the *Prompt for input* command is the *prompt-text* which is the prompt displayed to the left of the entry field by default; you can place the prompt text above the entry field using the **Prompt above entry** option. You can also enter a *title* for the message box. The prompt and title default to empty. Note that if you want to enter an empty title, you need to enter '/ /' to avoid ambiguity with the newline convention.

You can specify an icon for the message box using the *icon-id* of an icon from the OMNISPIC or USERPIC icon data file. Zero is the default which means no icon. You can use one of the icon size constants enclosed in square brackets with the icon id to specify a non-default size, for example, [1710+k48x48]. You can specify the maximum number of characters that the user can enter in *max-chars*. This defaults to the maximum length defined in your return field. The *return-field* can specify an initial value for the entry field on the message box, and receives the value entered after the user clicks OK.

The **Sound bell** option causes the system beep to sound when the message box opens. The **Cancel button** option adds a Cancel button to the message box. The flag returns false if the user presses the Cancel button. The **Upper case only** option forces all input to be upper case, while the **Password entry** option hides the input, by displaying '*' for each character entered.

```
Prompt for input Please enter your name Returns lv_input (Sound
  bell,Cancel button,Prompt above entry)
; lv_input now contains value entered by user
```

Prompt for library

Reversible: NO   **Flag affected:** YES

Parameters: ☐ Do not close others
☐ Do not open startup task
Internal name
Parameters list

Syntax: Prompt for library [*([Do not close others]* [*,Do not open startup task]*)]
[*{internal-name* [*(parameter1*[*,parameter2]...*)]*}*]

This command prompts the user for a library file. You can specify the internal name and
startup task construct parameters of the library to be opened, together with the **Do not close
others** and **Do not open startup task** options.

If the internal name of an opened library is specified, a check is made to ensure the internal
name is unique among the open libraries; a runtime error occurs if this is not the case. If no
internal name is specified, the default internal name is the disk name of the file with the path
name and suffix removed. For example, the internal name for 'hd:myfiles:testlib.lbr' is
'testlib'.

If an attempt is made to open a library which is already open, that library is closed and
reopened. Refer to *Close library* for the consequences of closing a library. If the user
cancels the Select Library dialog, the flag is cleared and no libraries are closed.

Do not close others

The **Do not close others** option lets you keep open all other libraries. If the **Do not close
others** option is not selected, then all other open libraries are closed when the user opens a
new library, including the one containing the currently executing method.

Passwords

If the library does not need a master password, it is opened at the master level, otherwise the
usual prompt for password dialog is opened. The library is closed and a flag false returned
if this dialog is closed without a password being entered.

Startup task

If the **Do not open startup task** option is specified, the startup task construct for the
opened library is not called and there is no startup task instance. Otherwise, the startup task
$construct() method is called and the parameters for it are passed.

```
Prompt for library {MyLbr (Param1)}
Prompt for library (Do not close others) {SQLTool}
```

Prompt for page setup

Reversible: NO   **Flag affected:**   YES

Parameters: None

Syntax: Prompt for page setup

This command displays the Printer page setup dialog box. This dialog allows the page size, orientation and printer's effects to be chosen before a report is printed. The flag is set if the dialog is closed by clicking on the OK pushbutton. Cancel clears the flag and leaves the page parameters unchanged. Note the *Prepare for print* and *Print report* commands have the **Ask for job setup** option which opens the setup dialog before printing.

```
Prompt for page setup
If flag true
   Print report
End If
; You will need to repeat the Prompt
; to reset the page setup to its former values
```

Prompt for port name

Reversible: YES   **Flag affected:**   YES

Parameters: None

Syntax: Prompt for port name

This command displays the Set port dialog box that lets the user select a port. The flag is set if the port is successfully selected; if the user cancels, the flag is cleared and the port closed.

You can set the baud rate and other parameters for the port using *Set port parameters*.

If the command is in a reversible block, the port is closed when the method terminates.

```
Open window instance WIMPORT
Prompt for port name
Prepare for import from port {Delimited (commas)}
If flag true
   Import data {ImportList}
End If
End Import
Close port
```

Prompt for print or export file

Reversible: YES **Flag affected:** YES

Parameters: None

Syntax: Prompt for print or export file

This command displays the Select Print or Export File dialog. The flag is set if the file is successfully selected. If the file exists already, a further dialog lets you delete it. If the user cancels, the flag is cleared and the file is closed.

If the command is in a reversible block, the file is closed when the method terminates.

```
Prompt for print or export file
If flag true
   Send to file
   Set report name R_Addresses
End If
Print report
```

Prompt for word server

Reversible: NO **Flag affected:** YES

Parameters: Word server tag

Syntax: Prompt for word server *{word-server-tag}*

This command prompts the user to specify an application for text checking using the standard dialogs offered by the Apple interface. Apple Word Servers are available for both spelling and grammar.

Once chosen, OMNIS will remember the checker, using an alias record in the OMNIS Preferences file, and the checker need not be reselected each time the Library is opened. Since you can open only one word server at a time, prompting for a new word server replaces the original, which must be recalled with *Prompt for word server* if required.

By giving the command a tag parameter (*Prompt for word server {NetSpeller}*, for example) you can use the tag name to quit the word server. This is useful if memory is running short.

```
Prompt for word server {NetSpeller}
If flag true
   OK message {Speller found, tagged 'NetSpeller'}
   ; NetSpeller is added to the Application Menu
Else
   OK message {Your speller is not available}
   Quit method
End If
```

```
; following line could be under a radio button
; Send Core event {Quit Application ('NetSpeller')}
```

NetSpeller remains available until the Quit event, or until an alternative word server is prompted for. You can use the next example behind a pushbutton to check a particular field.

```
On evClick
   Prompt for word server  ;; opens a dialog box
   Send Word Services event {Check field text('CVAR1')}
   ; checks the text in CVAR1 using the checker prompted for
   Quit event handler
```

Prompted find

Reversible:	YES	**Flag affected:**	YES
Parameters:	☐ Exact match		
Syntax:	Prompted find [*(Exact match)*]		

This command prompts the user to enter a value in an indexed field on the current window and locates the record which most closely matches that value. The user can use the Tab key to select an indexed field. The Find field is the current field for the window when the user clicks on the OK button.

Once the user enters a value in the Find field and clicks OK, OMNIS locates the record most closely matching this value, the main and connected files are read into the current record buffer and the flag is set. If the indexed field is in a connected file, the find continues until a record connected to a valid main file record is located. The current index, as used by *Next* and *Previous*, is set to the Find field.

If the exact field value cannot be matched, the next highest value in the index is located. You use the **Exact match** option if you want only the exact match.

```
Open window instance {wSuppliers}
Prompted find
If flag true
   Redraw {wSuppliers}
End If
```

Publish field

Reversible:	YES	**Flag affected:**	YES

Parameters: Field name
Edition name

Syntax: Publish field *field-name* [*{edition-name}*]

This command publishes the specified OMNIS field in the specified edition. A full pathname can be given for the edition, that is, a specification for the volume and folder(s) in which you want to create the edition. For example

```
If sys(113)
   Publish field SALESTOTAL {HD80:Public Folder:OMNIS-MyLbr-Sales Total}
End If
```

This creates the edition file "OMNIS-MyLbr-Sales Total" in the folder "Public Folder" on the hard disk volume HD80. If the edition already exists, the data is published using the existing edition. Before other network users can "see" the edition, you must enable sharing for the Public Folder, this is possible only from the System 7 Finder (see System 7 user guide).

If you do not specify an edition name, the existing edition for that field is used; if there is no existing edition for that field, the default edition name "library name-field name" is used.

The flag is set if the field is already published in that edition or if the field is successfully published. The command does nothing and clears the flag if System 7 is not running. If the command is used within a reversible block, the edition is canceled when the command is reversed.

When a field is newly published, none of the publisher options are set, so a *Publish now* command must be used to update the edition. If you want the edition to be updated automatically, the *Set publisher options* command must be used.

Fields published with this command are not shown with borders and are invisible to the user, that is, the **Edit** menu's **Publisher options...** cannot affect them. If you publish a local variable, its edition is canceled when the method terminates. Lists are published as tab-delimited text and pictures as PICT.

```
Publish field CNAME {HD80:Public:Sales-Name}
Publish field CTOTAL {HD80:Public:Sales-Total}
Set publisher options (Publish on save) {CNAME,CTOTAL}
Prepare for edit
Enter data
Update files if flag set
```

Publish now

Reversible:	NO	**Flag affected:**	YES
Parameters:	File or field list		
Syntax:	Publish now [*{file|field1*[,*file|field2*]...*}*]		

This command updates the editions for the specified fields. Field values are written from the current record buffer to the editions. The field list can take a file name (for all fields in a file) or a range of fields, which includes a range of fields in the order listed in the Field names window. If no file and/or field list is given, all publications for the library are updated.

The flag is set if the command publishes one or more fields.

```
Publish field CNAME {HD80:Public:Sales-Name}
Publish field CTOTAL {HD80:Public:Sales-Total}
Find first
Publish now {CNAME,CTOTAL}
```

Queue bring to top

Reversible:	NO	**Flag affected:**	NO
Parameters:	Window instance name		
Syntax:	Queue bring to top *window-instance-name*		

This command queues a "bring to top" event for the specified window instance as if the user had clicked on the window instance with the mouse. The command brings the window instance to the fore and generates evWindowClick and evToTop events. If, at runtime, the specified window instance does not exist, the command will do nothing.

```
Open window instance WCLIENT/W1
Open window instance WCLIENT/W2
Queue bring to top W1   ;; brings W1 to the top
```

Queue cancel

Reversible: NO   **Flag affected:**   NO
Parameters: None
Syntax: Queue cancel

This command queues a "cancel" event as if the user had clicked on the Cancel button or pressed the Cancel key combination, that is, the user pressed Ctrl-break under Windows or Cmnd-period under MacOS. The command takes no parameters.

```
; Timer method cancels enter data after 120 secs
Set timer method (120 sec) Win1/Timer
Prepare for edit
Enter data
Update files if flag set


; Timer method for Win1/Timer
Queue cancel
```

Queue click

Reversible: NO   **Flag affected:**   NO
Parameters: ☐ Shift
                   ☐ Command/Ctrl
                   Field name
                   Selection point or Start value, end value
Syntax: Queue click [([*Shift*][,*Command/Ctrl*])]
                   {field-name [*(selection-point|start-value, end-value)]}*

This command queues a "mouse click" event on a specified field, that is, it simulates a user-generated mouse click/drag operation on a field. You must specify the name of the field as a parameter, including the click positions within the field (that is, Start Row, Finish Row for lists and Start Character, Finish Character for text field selection). The specified field will get the focus.

There are options for including up to three modifier keys (that is, Shift, Ctrl/Cmnd) along with the click.

The field name parameter must be the name of a window field, not the name of the method associated with the field or the data name (*$fieldname*, not *$dataname*).

Queue click on Edit fields

You can specify a range of characters. For example, the parameter *field-name (2,5)*, highlights the characters within cursor positions 2 to 5 (that is, characters 3 to 5). Note that

cursor position 0 is to the left of character 1, and cursor position 1 is to the right of character 1 (or to the left of character 2).

If Shift is selected and 5 is passed as the selection point, all characters between the current cursor position and cursor position 5 will be highlighted.

```
Queue click {field-name (7,2)}
; Characters 3 to 7 will be highlighted

Queue click {field-name (5,9)}
; Characters 6 to 9 will be highlighted

Queue click (Shift) {field-name (8)}
; Current cursor is at 15
; Characters 9 to 15 will be highlighted

Queue click (Shift) {field-name (22)}
; Current cursor is at 15
; Characters 16 to 22 will be highlighted

Queue click (Shift) {field-name (7,9)}
; Current cursor is at 15
; Characters 10 to 15 will be highlighted

Queue click (Shift) {field-name (9,7)}
; Current cursor is at 15
; Characters 8 to 15 will be highlighted
```

As the examples show, the two parameters act as a "click on, drag to" key operation.

Queue click for lists

If the specified field is a window list box or grid, the range is interpreted as a range of list lines. For example, the parameter *list-field-name (2,5)*, selects the lines 2 to 5 (if $multipleselect for the list field is set), and the current line will be set to 2. An evClick event is generated after the specified lines have been selected.

```
Queue click {List-field-name (7,3)}
; Lines 7 to 3 will be selected, the current line will
; be set to 7

Queue click (List-field-name (2,9)}
; Lines 2 to 9 will be selected, the current line will
; be set to 2

Queue click (Shift) {List-field-name (12)}
; The current line to line 12 will be selected,
; The current line does not change
```

```
Queue click (Shift,Command/Ctrl) {List-field-name (13)}
; Line 13 will be selected and lines already selected will stay
; selected. The current line does not change
Queue click (Shift,Command/Ctrl) {List-field-name (4,8)}
; Lines 4 to 8 will be selected and lines already selected will stay
; selected. The current line does not change
```

Queue click for pushbuttons

If the specified field is a pushbutton it is activated and an evClick event is generated as if the user had clicked on the button.

Queue click for Radio buttons and check boxes

If the specified field is a check box or set of radio buttons, the check box field or group of radio buttons is checked/unchecked accordingly, and an evClick event is generated. Methods behind radio buttons and check boxes run as if the user had clicked on the window fields.

Queue close

Reversible: NO   **Flag affected:**   NO

Parameters: Window instance name

Syntax: Queue close *window-instance-name*

This command queues a "close window" event for the specified window instance as if the user had selected the close option (system menu under Windows or close box under MacOS).

The specified window instance is closed, but an evClose event is not produced. If the specified window instance does not exist, the command has no effect. If you omit the window instance name, the top window instance at the time of execution will be closed. In this case, a proper evClose event is generated.

```
Open window instance WCUSTOMERS/winst1
Open window instance WCUSTOMERS/winst2
; do something in winst2
Queue close        ;; closes winst2, the top instance
```

Queue double-click

| **Reversible:** | NO | **Flag affected:** | NO |

Parameters:	□ Shift
	□ Command/Ctrl
	Field name
	(Selection point or Start value, end value)

| **Syntax:** | Queue double-click [*([Shift][,Command/Ctrl])*] |
| | *{field-name* [*(selection-point|start-value, end-value)*]*}* |

This command queues a "double-click event" on the specified field, that is, it simulates a user-generated double-click event on the field. A double-click event always generates an evClick before an evDoubleClick. You must specify the name of the field as a parameter, including the click positions within the field (that is, Start Row, Finish Row for lists and Start Character, Finish Character for text field selection).

There are options for including up to three modifier keys (that is, Shift, Ctrl/Cmnd) along with the click.

The field name parameter must be the name of a window field, not the name of the method associated with the field or the data name (*$fieldname*, not *$dataname*).

Queue double-click for edit fields

Double-clicks on text within an edit field will select the complete word. If a range was specified, all COMPLETE words falling within the start and end positions will be highlighted. For example, if the text in the field is:

```
Good books are the lifeblood of a master spirit
```

and the command is:

Queue double-click {field-name (7,23)}

The selected text will be:

```
books are the lifeblood
```

Queue double-click for list fields

Double-clicks on list fields will generate an evClick followed by an evDoubleClick. The behavior in other ways is the same as described for *Queue click*.

Queue double-click for other field types

Pushbuttons, radio buttons and check boxes behave in the same way as described for *Queue click*. An evDoubleCLick event is *not* generated.

```
; method for pushbutton:  Opens a new window while in Enter data
; mode and selects all the text in field2
```

```
On evClick
   Open window instance WCLIENTS
   Queue double-click {field2}
On default
   Quit event handler
```

Queue keyboard event

Reversible: NO   **Flag affected:**   NO

Parameters: Key sequence (can be a calculation)

Syntax: Queue keyboard event *{key-sequence}*

This command queues a "keyboard" event or series of events. It simulates keyboard entry by the user from within your methods. You can enter the key sequence in several ways:

1. **Recording a key sequence**
   You can use the **Start Recording** and **Stop Recording** buttons to specify the keys to be generated. During the recording, all key events are echoed to the Key sequence parameter field, and are not acted on by OMNIS in any other way (for example, pressing Ctrl/Cmnd-Q will NOT suddenly quit OMNIS). Click events, however, behave normally so you can click on **Stop recording** button.

2. **Entering into the text field**
   You can enter the text representation manually to generate the keys. Syntax checking is done at design time. When recording is off, you can edit the Key sequence parameter manually. This lets you delete key combinations or enter key sequences by hand. Since spaces are used to automatically separate key presses, the special key name SPACE will have to be manually entered to generate a "space key" event.

3. **Specifying a calculation**
   You can enter a calculation like concatenating text fields, which will contain the text representation of the keys to be generated. Syntax checking is done at runtime. Incorrect key sequence syntax will result in a runtime error. When you use a calculation, the general calculation syntax applies, which is checked at design time.

Key names

Special keys or key combinations are represented using the names of the keys. When a given key combination is run on another platform, a conversion is carried out internally so that, for example, alt-c under Windows becomes opt-c under MacOS. The list below summarizes the conversion:

Windows

Modifier Key names: shift-, alt-, ctrl-

Special Key names: Space, Up, Down, Left, Right, PgUp, PgDn, PgLeft, PgRight, Home, End, Tab, Return, Enter, Bkspc, Clear, Cancel, Minus, Move, Del, Ins, Exit

MacOS

Modifier Key names: shift-, opt-, com-

Special Key names: Space, Up, Down, Left, Right, PgUp, PgDn, PgLeft, PgRight, Home, End, Tab, Return, Enter, Bkspc, Clear, Cancel, Minus, Move, Del.

Set current field

If queued key events are intended for an edit field or a list, it is advisable to queue a "set current field" event before generating the key events. On the other hand, general key events, for example, menu accelerators or shortcut keys, do not require a specific current field.

Key event restrictions under Windows

Under Windows, you can use alt-<key> sequences to select menu options from the menu bar. Since the menu bar is handled by Windows, and *Queue keyboard event* generates internal OMNIS events, queuing alt-<key> events will NOT drive the menu bar. Thus, for example, queuing alt-f will not drop the **File** menu.

As a consequence of the above restriction, evKeyPress events are not generated for queued alt-<letter> sequences either.

A second situation where evKeyPress events are not generated is when you queue alt-control-<letter> events. These key combinations are normally used to produce accented characters, and this facility exists only in some but not all keyboards. Since Windows does not generate character messages, these events do not generate evKeyPress.

WARNING When queuing events on pushbuttons a danger of recursion can occur under Windows, but also under MacOS if buttons have been given Windows behavior, that is, they get the focus. Normally, when the focus is on a pushbutton, you can activate it by pressing the space bar. If that pushbutton receives an evClick event and has a queued space key event WITHOUT a set current field, the space key event will be sent back to the pushbutton, thereby generating another evClick, which again activates the space key event. A recursion occurs and exhibits an apparent crash.

Key event restriction under MacOS

Under MacOS, you use opt-<letter> to generate extended characters. When queued key events include such opt-<letter> sequences, evKeyPress is not generated.

```
; Button method
On evClick
   Open window instance ADDRESS
   Queue keyboard event {Y o u r N a m e}
On default
   Quit event handler


; Paste button
On evClick
   Queue keyboard event {com-v}   ;; does a paste operation
On default
   Quit event handler
```

Queue OK

Reversible: NO   **Flag affected:**   NO

Parameters: None

Syntax: Queue OK

This command queues an "OK" event. It simulates the user clicking on the OK button or pressing the Enter key.

```
; This field method traps the Tab event and issues an OK event
On evTab
   Queue OK
On default
   Quit event handler
```

Queue quit

Reversible: NO   **Flag affected:** NO

Parameters: None

Syntax: Queue quit

This command queues a quit event. It simulates the user selecting the **Exit/Quit** option in the **File** menu. In enter data mode, a *Queue OK* or *Queue Cancel* should precede a *Queue quit* to close the enter data correctly.

```
; Button method to terminate data entry and quit
If flag true
   Queue OK
   Queue quit
Else
   Queue cancel
   Close top window
End If
```

Queue scroll

Reversible: NO   **Flag affected:** NO

Parameters: ☐ Page
Left|Right|Up|Down scroll direction
Field name
(Units), that is, the number of lines (up/down) or characters (left/right)

Syntax: Queue scroll *(scroll-direction*[,*Page*]*) {field-name*[*(units)*]*}*

This command queues a "scroll" event in the specified scrollable field, that is, it simulates a mouse click or page key event on a scrollable field. With this command you can scroll a field up or down, left or right provided the appropriate scroll bar is available. You cannot use this command to scroll a window instance.

The field name parameter must be the name of a window field, not the name of the method associated with the field or the data name (*$fieldname*, not *$dataname*).

The **Units** parameter specifies the number of lines to scroll up or down in a vertical scroll bar for a field; one unit represents one line. For a horizontal scroll bar, the unit is approximately one character.

If the **Page** option is selected, the event simulates clicking above or below the "thumb" and is the same as using the Page up or Page down key.

```
; $construct() for window instance to display end of text field
```
Queue scroll (Down, Page) {FIELD1}
Queue scroll {LIST1 (5)}
```
; scrolls the field list by 5 lines
```

Queue set current field

Reversible: NO   **Flag affected:**   NO

Parameters: Field name

Syntax: Queue set current field *{field-name}*

This command queues a "set current field" event in the specified field, that is, it simulates a user-generated click or tab to the specified field. In enter data mode, the contents of the field is selected. The command does not generate an evClick. However it will produce proper evBefore and evAfter events during *Enter data*.

The field name parameter must be the name of a window field, not the name of the method associated with the field or the data name (*$fieldname*, not *$dataname*).

```
; field method to jump to another field on the window instance
On evAfter
   Queue set current field {SALARY)  ;; jumps to SALARY field
On default
   Quit event handler
```

Queue tab

Reversible: NO   **Flag affected:**   NO

Parameters: □ Shift

Syntax: Queue tab [*(Shift)*]

This command queues a "tab" or "shift-tab" event. It simulates a user-generated tab event. With the **Shift** option, it simulates a shift-tab keypress.

```
; Field method for field on a window instance to simulate auto tab
; when the 6th character is entered; $keyevents must be true
On evBefore
   Calculate COUNTER as 0
On evKeyPress
   Calculate COUNTER as COUNTER + 1
   If COUNTER >= 4
      Queue tab
   End If
On default
   Quit event handler
```

Quick check

Reversible: NO   **Flag affected:**   YES

Parameters: ☐ Perform repairs

Syntax: Quick check [*(Perform repairs)*]

This command performs a quick check on the current data file. It examines the status of the current data file by reading only the internal tables in which records of any inconsistencies are stored. These records indicate corruption caused by either hardware or software failure. No attempt is made to systematically check the entire data file for problems (you use the *Check data* command for this purpose).

The command is not reversible: it sets the flag if it completes successfully and clears it otherwise.

If the **Perform repairs** option is specified, any repairs required are automatically carried out, otherwise the results of the check are added to the check data log. The check data log is not opened by this command but is updated if it is already open. If the **Perform repairs** option is specified, the following applies:

If you are not running in single user mode, OMNIS automatically tests that only one user is logged onto the data file (the command fails with flag false if not), and further users are prevented from logging onto the data until the command completes.

If a working message with a count is open while the command is executing, the count will be incremented at regular intervals. The command may take a long time to execute and it is not possible to cancel execution even if a working message with cancel box is open.

```
Quick check
Yes/No message {View the check data log}
If flag true
   Open check data log
End If
```

Quit all if canceled

Reversible: NO   **Flag affected:**   NO

Parameters: None

Syntax: Quit all if canceled

This command quits all methods that are running when the user clicks on a Cancel button inside a working message dialog box. The keyboard equivalent to the Cancel pushbutton is the Escape key under Windows or Cmnd-period under MacOS. Note that the test for cancel is carried out in *Working message* only if *Disable cancel test at loops* has first been executed.

```
Begin reversible block
   Disable cancel test at loops
End reversible block
Repeat
   Working message (Cancel box, Repeat count)
   Quit all if canceled
   Calculate LVAR1 as LVAR1+1
Until LVAR1=200
OK message {Finished method, counter = [LVAR1]}
```

Quit all methods

Reversible: NO   **Flag affected:**   NO

Parameters: None

Syntax: Quit all methods

This command quits all methods that are running.

If the command is executed during a method which has been called, OMNIS quits both the current method and the calling method.

```
; Calling method
Do method QuitMethod
OK message {This never runs}

; Quitmethod
Quit all methods
```

Quit cursor(s)

Reversible: NO   **Flag affected:**   YES

Parameters: Current, Session, or All option (Current is the default)

Syntax: Quit cursor(s) *(Current|Session|All)*

This command disposes of the specified cursor and frees all memory it occupied. It has three possible values: Current, Session, or All.

The **Current** option quits the current cursor. If the current cursor is the only remaining cursor, a logoff is carried out and communication with the remote database is disconnected. All memory used by the cursor is released. A new *Set current session* or *Set current cursor* followed by *Start session* and *Logon to host* would be required to recreate the cursor.

This command will close any open import file and will close the SQL driver. If you have used the automatic logon process, closing the driver will log off from the remote computer and disconnect the modem, otherwise, it will leave you connected and logged on. After *Quit cursor(s)* is executed, you cannot perform any other SQL transactions until another *Start session* is issued.

The **Session** option performs a quit for all cursors in the session containing the current cursor. A logoff and disconnect from the remote database used by that cursor will be carried out.

The **All** option performs a quit for all cursors. A logoff and disconnect from all the remote databases in use will be carried out.

```
; select a cursor and close it
Set current cursor {SQL_1}
Quit cursor(s) (Current)
; or use
Close cursor {SQL_1}
```

Quit event handler

Reversible: NO   **Flag affected:**   NO

Parameters: □ Discard event
                  □ Pass to next handler

Syntax: Quit event handler [*([*Discard event*] [*,Pass to next handler*])*]

This command is used to quit out of the currently executing event handling method and is only used to terminate an *On* clause. It is not reversible and does not affect the flag.

If the **Discard event** option is checked, the event is thrown away and OMNIS quits the event handling method.

If the **Pass to next handler** option is checked, the event is passed to the next level of handler such as the window $control() method or task $control() method.

```
On evAfter
   If CoName = ''
      OK message {You must enter a name}
      Queue set current field {eCompanyName}
      Quit event handler (Discard event)
   End If

; $event() for a window field
On default
   Quit event handler (Pass to next handler)
   ; passes all events to the window $control() method
```

Quit method

Reversible: NO   **Flag affected:**   NO

Parameters: Return field

Syntax: Quit method

This command quits the current method and returns control to the calling method, if any.

```
Do method Print


; Print
Set report name rStock
Prompt for destination
If flag false
   Quit method
Else
   .. print the report
```

Quit OMNIS

Reversible: NO   **Flag affected:**   NO

Parameters: ☐ Force quit

Syntax: Quit OMNIS [*(Force quit)*]

This command quits OMNIS closing all libraries and data files. It is equivalent to the **Exit/Quit** option in the **File** menu. However, if the **Force quit** option is not checked *Quit OMNIS* will set the flag false and do nothing if an instance or library cannot be closed.

If the **Force quit** check box is checked OMNIS will force any class instances to close so that the quit can take place, even if they have custom $canclose logic which would normally prevent them from closing.

```
Yes/No message {Do you want to quit OMNIS?}
If flag true
   Quit OMNIS (Force quit)
   ; closes all instances and tasks, then quits OMNIS
End If
```

Redefine list

Reversible: NO   **Flag affected:**   NO

Parameters: List of file and/or field names

Syntax: Redefine list *{file|field1*[*,file|field2*]*...}*

This command redefines the column headings of the current list. No change is made to the internal data type and structure of the list; columns can neither be added nor removed, merely renamed. If you place more field names in *Redefine list* than there were in the original list, the extra names are ignored. Changing the field name of a column may cause a data conversion to take place as subsequent lines are added. List boxes on windows will no longer display the data in the list unless you change their $calculation property to include the new variable or field name(s).

```
Set current list LIST1
Define list {Field1Date,Field2Num,Field3Char}
Add line to list
Redefine list {,,Field4Boolean}
; the third column is now defined Field4Boolean
Add line to list
; the Boolean field value is converted to a character field
; format 'YES' etc., then added to the list
```

or do it like this

```
Do List.$redefine(Field1Date,Field2Num,Field4Boolean)
```

Redraw

Reversible: NO   **Flag affected:**   NO

Parameters: ☐ Refresh now
                   Field or window instance name (or list of fields or windows)

Syntax: Redraw [*(Refresh now)*] ***field1/window1***[,*field2/window2,...*]

This command redraws the specified field or window instance (or list of fields or window instances). The **Refresh now** option ensures the redraw is completed when the command is executed. Without this option the redraw occurs when the method has finished executing.

```
Prepare for edit
Enter data
If flag true
   Update files
Else
   Clear main & connected
   Redraw WinDataEntry
End If
```

Alternatively you can use the $redraw(setcontents,refresh) method to redraw the contents and/or refresh a field or window; 'setcontents' defaults to true, 'refresh' defaults to false.

```
Do $cfield.$redraw()    ;; redraws the current field
Do $cwind.$redraw()     ;; redraws the current window
Do $root.$redraw()      ;; redraws all window instances
```

Redraw lists

Reversible: NO   **Flag affected:**   NO

Parameters: ☐ All windows
                   ☐ All lists
                   ☐ Selection only

Syntax: Redraw lists [*(*[*All windows*][,*All lists*][,*Selection only*]*)*]

This command redraws the current list window field or all list fields. It lets you update the display of the current list field after you delete, change, or insert a line, so that the screen list reflects the changes. When OMNIS executes *Redraw lists*, the selected line is scrolled into view and the visible lines recalculated.

OMNIS can execute a *Redraw lists* command for all window instances and for all lists using the **All windows**, and **All lists** options. If neither option is selected, only the fields on the top window instance which display the current list are redrawn.

Chapter 5—Commands

The **Selection only** option causes the redraw to affect the highlighting of the selected lines, the contents are not redrawn.

OMNIS also redraws any fields which are local to the list field so that they will display the new values. It also redraws the grid fields associated with the current list.

```
Open window instance LAYOUT
Set current list LIST1
Define list {LVAR1,CVAR1}
Calculate LVAR1 as 42
Add line to list {(LVAR10,CHR(LVAR10))}
Redraw lists
```

Redraw menus

Reversible: NO **Flag affected:** NO

Parameters: None

Syntax: Redraw menus

This command redraws all instances of your own custom menus. When executing *Redraw menus*, OMNIS re-evaluates any square-bracket notation contained in the menu titles and lines before redrawing the menu bar.

This example assumes that the menu instance uses [CVAR5] as its title.

```
Parameter LNUM (Number 0 dp)
Calculate CVAR5 as pick(LNUM,'Purchases','Invoices')
Redraw menus
; If LNUM = 0, menu called Purchases, otherwise called Invoices
```

Redraw toolbar

Reversible: NO **Flag affected:** NO

Parameters: ☐ Droplists only
Instance name

Syntax: Redraw Toolbar [*(Droplists only)*] *{instance-name}*

This command redraws the toolbar instance. You can redraw droplists only using the **Droplists only** option.

```
Show docking Area {kDockingAreaTop}
Install Toolbar {T_Formats}
; do something
Redraw Toolbar (Droplists only) {T_Formats}
```

Redraw working message

Reversible: NO   **Flag affected:**   NO

Parameters: None

Syntax: Redraw working message

This command redraws the text in the working message after evaluating any square bracket notation. OMNIS does not increment the working message count and does nothing if there is no open working message.

When a library is being debugged, you can monitor the values of critical variables and fields with the following line:

```
Working message (Cancel box) {[CVAR1],[TOTAL], [sys(84)]}
```

Once the message has been displayed, you can use the command *Redraw working message* to refresh the values monitored in the message box.

```
; declare local variable COUNT of type Number 0 dp
Working message {COUNT = [COUNT]}
For COUNT from 1 to 100 step 1
   Redraw working message
End For
```

Reinitialize search class

Reversible: NO   **Flag affected:**   NO

Parameters: None

Syntax: Reinitialize search class

This command reloads the current search definition into memory. *Reinitialize search class* is useful if square bracket notation has been used in the search class. The square bracket expressions are re-evaluated using current field values before reloading the search definition. Each find table keeps its own copy of the search conditions so you must reissue the *Find* command if a search needs reinitializing.

For example, a search class uses the comparison line *TOWN Begins with [S5].* Window wStarts is used to allow the user to specify a value for S5.

```
Set search name STOWN
Repeat
   Open window instance wStarts
   Enter data
   Close window wStarts
   If flag true
      Reinitialize search class
      Do method PrintReports
   End If
   ; assumes no rev. blocks in window construct to change flag
Until flag false
```

Remove all menus

Reversible: YES   **Flag affected:**   NO

Parameters: None

Syntax: Remove all menus

This command removes all menu instances from the menu bar, *excluding* the standard OMNIS menus such as **File, Edit,** and **Help** (under Windows only). If you use *Remove all menus* in a reversible block, the menu instances are reinstalled when the method containing the block finishes.

```
Begin reversible block
   Remove all menus
End reversible block
OK message {Menus are now removed}
; now all menu instances are reinstalled
```

Remove final menu

Reversible: YES   **Flag affected:**   NO

Parameters: None

Syntax: Remove final menu

This command removes the final or right-most menu instance from the menu bar, *excluding* the standard OMNIS menus such as **File, Edit,** and **Help** (under Windows only). If you use *Remove final menu* in a reversible block, the final menu instance is reinstalled when the method containing the block terminates.

```
Begin reversible block
   Remove final menu
End reversible block
OK message {Menu is now removed}
; now the final menu is reinstalled
```

Remove menu

Reversible: YES   **Flag affected:**   YES

Parameters: Menu instance name

Syntax: Remove menu *menu-instance-name*

This command removes the specified menu instance from the menu bar and sets the flag. You can choose the menu name from a list containing any custom and standard built-in menus, such as *File, *Edit, and so on.

If you use this command to remove a menu instance which has previously been installed in place of the standard **File** or **Edit** menu (using the *Replace standard File/Edit menu* command) the previously replaced standard **File** or **Edit** menu is restored.

If you use *Remove menu* in a reversible block, the specified menu instance is reinstalled when the method containing the reversible block terminates.

```
Begin reversible block
   Remove menu STARTUP
End reversible block
OK message {STARTUP is now removed}
; now the menu instance is reinstalled
```

or do it like this

```
Do $imenus.INSTANCE.$close()
```

Remove toolbar

Reversible: NO   **Flag affected:**   NO

Parameters: Instance name

Syntax: Remove Toolbar *{instance-name}*

This command removes the specified toolbar instance.

```
Show docking area {kDockingAreaRight}
Install Toolbar {T_Tennis}
; do something
Remove Toolbar {T_Tennis}
Hide docking area {kDockingAreaRight}
```

or do it like this

```
Do $itoolbars.INSTANCE.$close()
```

Rename class

Reversible: NO   **Flag affected:**   YES

Parameters: ☐ Perform find and replace
Class name/New class name

Syntax: Rename class [*(Perform find and replace)*]
{class-name/new-class-name}

This command renames the specified library class and can perform a find and replace. Errors, such as attempting to use a name that is already in use, simply clear the flag and display an error message. You can rename a class which is in use.

When renaming a class, you can use the **Perform find and replace** option to search through all the classes in the library and replace the references to the old class name with the new name.

```
New class {Search/S_My}
Modify class {S_My}
Delete class {S_User}
Rename class {S_My/S_User}
Set search name S_User
Print report (Use search)
```

Rename data

Reversible: NO   **Flag affected:**   YES

Parameters: File class name
                  New file slot name

Syntax: Rename data {file-name/new-slot-name}

This command renames the data for a specified file class in a data file so that the data will then belong to a file with a different name; that is, it renames a slot. The existing file class name and the new slot name are specified as parameters.

The specified file class is disconnected from the data, and an empty slot and indexes for that file will be created as soon as that file is accessed again.

If the specified file name does not include a data file name as part of the notation, the default data file for that file is assumed.

If the file is closed or memory-only, the command does not execute and returns flag false.

If you are not running in single user mode, OMNIS automatically tests that only one user is logged onto the data file (the command fails with flag false if this is not true), and further users are prevented from logging onto the data until the command completes.

This command sets the flag if it completes successfully and clears the flag otherwise. The command is not reversible.

```
Rename data {C_CONTACTS/C_ARCHIVE}
If flag true
   OK message {File archived}
Else
   OK message {Can't archive when more than one user is logged on}
End If
```

Reorganize data

Reversible: NO   **Flag affected:**   YES

Parameters: ☐ Test only
                  ☐ Optimize
                  ☐ Convert pictures
                  File or list of files (the default is all files)

Syntax: Reorganize data [*([Test only][,Optimize] [,Convert pictures])*]
                  [*{file1*[,*file2*]*...}*]

This command reorganizes the data for the specified file or list of files. Reorganization is the process by which the data structures held in the OMNIS data file are brought into line with the file class definitions.

Reorganize data reorganizes the data for the specified list of files, and is equivalent to the option on the **Slot** menu in the Data File Browser.

If you omit a file name or list of files, *all* the files with slots in the current data file are reorganized.

If a specified file name does not include a data file name as part of the notation, the default data file for that file is assumed. If the file is closed or memory-only, the command does not execute and returns with the flag false.

If you are not running in single user mode, OMNIS automatically tests that only one user is logged onto the data file (the command fails with the flag false if this is not true), and further users are prevented from logging onto the data until the command completes.

If a working message with a count is open while the command is executing, the count will be incremented at regular intervals. The command may take a long time to execute, and it is not possible to cancel execution even if a working message with cancel box is open.

The command sets the flag if it completes successfully and clears the flag otherwise. The command is not reversible.

If the **Test only** checkbox option is specified, no reorganization is actually carried out. The flag is set if at least one file needs reorganization.

The **Optimize** checkbox option specifies whether reorganize with optimize is to be carried out. This distributes the free space to make the data storage more efficient.

The **Convert pictures** checkbox option causes all pictures in the data to be converted to a shared picture format.

```
Reorganize data (Test only)
If flag true
   Yes/No message {Reorganize now?}
   If flag true
      Reorganize data
   End If
Else
   OK message {No reorganization required}
End If
```

Repeat

Reversible: NO   **Flag affected:** NO

Parameters: None

Syntax: Repeat

This command repeats a command or series of commands that are contained in a loop closed by an *Until* command. Each time the command is repeated, OMNIS tests the condition attached to the *Until* command to ensure that the condition is true. If the condition is true, the commands in the loop are not executed and the command after the *Until* is executed. However, if the condition is false, OMNIS jumps back to the first command following the *Repeat* command. An error will result if there is a *Repeat* command without a matching *Until* command. Repeat loops always execute at least once. The *Repeat–Until* logic test is carried out at the *end* of the loop, after the commands in the loop are executed, whereas the *While–End While* logic test is carried out at the beginning of the loop.

You can use the *Repeat* command to step through a Find table in order to print each row, as follows.

```
; Perform SQL to select your data
Fetch next row
If flag true
   Prepare for print
   Repeat
      Print record
      Fetch next row
   Until flag false
   End print
Else
   OK message (Sound bell) {No rows to print}
End If
```

You can use *Repeat* to write general purpose methods to insert data, and can include a working message in the Repeat loop that displays while the loop is executing.

```
; Perform SQL to select your data
Set main file {[FILENAME]}
Fetch next row
Repeat
   Working message (Repeat count) {Inserting...}
   Prepare for insert with current values
   Update files
   Fetch next row
Until flag false
```

You can use an *Until flag true/false* command at the end of a Repeat loop to force the loop to repeat until the true or false state is met. In the following case, the window WCHOOSE remains open until the user enters a valid value for LETCODE.

```
Repeat
   Open window instance WCHOOSE
   Enter data          ;; user enters a value for CVAR1
   Close all windows
   Find on LETCODE (Exact match) {CVAR1}
Until flag true        ;; if false, loops again
```

Replace line in list

Reversible: NO   **Flag affected:** YES

Parameters: Line number (can be a calculation, default is current line)
Field values

Syntax: Replace line in list [*{*[*line-number*] [(*value1*[,*value2*]...)]*}*]

This command transfers field values from the current record buffer to the corresponding fields in the current list. Alternatively, it is possible to specify a comma-separated list of values enclosed in brackets after the line number. In this case, the values stored in the specified line of the list are set up from the values in the brackets and not from the variables specified when the list was defined. For example

Replace line in list {LIST.$linecount('abc',,LVAR12+3)}

will store 'abc' into the first column of the final line of the current list, leave the value of the second column unchanged, and load the result of LVAR12+3 into the third column. If too few values are specified, the other columns will be left unchanged; if too many values are specified, the extra values are ignored. Any conversions required between data types are carried out.

If the line number specified in the command line is empty, or if it evaluates to zero, the current line is used. If the list is empty or if the line is beyond the current end of the list, the flag is cleared.

```
Set current list LIST2
Define list {CODE,NAME,CREDIT}
Build list from file on CLIENTS
Calculate CVAR3 as 'New string'
Calculate LVAR1 as 23
Replace line in list {4(,CVAR3,LVAR1)}
If flag false
   OK message {line 4 is beyond the end of the list}
Else
   OK message {New value in list is [LIST2.4.CVAR3]}
End If
```

Replace standard Edit menu

Reversible: YES   **Flag affected:**   NO

Parameters: Menu class name (must be user-defined or Edit)
                      Instance name

Syntax: Replace standard Edit menu [*{menu-name/instance-name}*]

This command removes the standard built-in **Edit** menu from the menu bar and replaces it with a custom menu. You can assign an instance name for the replacement menu. The default instance name of the replacement menu is the menu class name. If no replacement menu name is specified, the **Edit** menu is reinstated.

The replacement menu will remain enabled even when commands such as *Disable all menus* are issued, or modal user-defined windows are opened. The only time the replacement menu will not remain enabled is when a report is printed to screen with *Send to screen*, and the check box option **Do not wait for user** is not checked (that is, OMNIS is awaiting user input).

You can disable the **Edit** menu or its replacement menu by using *Disable menu line*.

```
Replace standard Edit menu {MY_EDIT1}
Set main file {FMAIN}
Prepare for insert
Enter data
Update files if flag set
; Now put system Edit menu back
Replace standard Edit menu
```

Replace standard File menu

Reversible: YES   **Flag affected:**   NO

Parameters: Menu class name (must be user-defined or File)
                      Instance name

Syntax: Replace standard File menu [*{menu-name/instance-name}*]

This command removes the standard built-in **File** menu from the menu bar and replaces it with a custom menu. You can assign an instance name for the replacement menu. The default instance name of the replacement menu is the menu class name. If no replacement menu name is specified, the **File** menu is reinstated.

The replacement menu will remain enabled even when commands such as *Disable all menus* are issued, or modal user-defined windows are opened. The only time the replacement menu will not remain enabled is when a report is printed to screen with the *Send to screen* command, and the check box option **Do not wait for user** is not checked (that is, OMNIS is awaiting user input).

You can disable the **File** menu or its replacement menu by using *Disable menu line*.

```
Replace standard File menu {MY_FILE1}
Set main file {FMAIN}
Prepare for insert
Enter data
Update files if flag set
;  Now put system File menu back
Replace standard File menu
```

Request advises

Reversible:	YES	**Flag affected:**	YES

Parameters:	Field name
	Server data item name

Syntax: Request advises *field-name {server-data-item-name}*

DDE command, OMNIS as client. This command sends a request to the server asking to be advised of any changes made to a specified data item. An error occurs if the channel is not open. The command takes the OMNIS field name and the server data item name as parameters. The data item name can contain square bracket notation.

Whenever OMNIS is advised of a change in field value, that value is changed providing your library is in enter data mode.

The flag is set if the command is successful.

You can use a control method to detect the arrival of data from the server using evSent.

```
Request advises C_COMPANY {C_COMPANY}
Request advises C_ADDRESS {C_ADDRESS}
Prepare for insert
Enter data
Update files if flag set
```

Request field

Reversible: NO   **Flag affected:** YES

Parameters: Field name
Server data item name

Syntax: Request field ***field-name*** [*{server-data-item-name}*]

DDE command, OMNIS as client. This command requests a data item from the DDE channel. An error occurs if the channel is not open. The command takes the OMNIS field name and the server data item name as parameters. The data item name can contain square bracket notation. If the data item name is not specified, the OMNIS field name is used. The flag is set if the command is successful.

```
Set DDE channel number {1}
Calculate Tries as 1
; Keeps trying until conversation opened or number of tries > 10
Repeat
   Open DDE channel {OMNIS|DDE2}
   Calculate Tries as Tries + 1
Until #F | Tries > 10
Calculate CVAR1 as '[TakeControl]'
Send command {[CVAR1]}
If flag false
   OK message {Error: [CVAR1], Open tries = [Tries]}
End If
```

Request field C_COMPANY {C_COMPANY}
Request field C_ADDRESS {C_ADDRESS}
```
Prepare for insert with current values
Enter data
Update files if flag set
```

Reset cursor(s)

Reversible: NO   **Flag affected:** YES

Parameters: Current, Session, or All option (Current is the default)

Syntax: Reset cursor(s) *(Current|Session|All)*

This command resets the specified cursor(s) for a server. It has three possible values: Current, Session or All.

The **Current** option clears or empties the SQL buffer, the error status and select table for the current cursor.

The **Session** option resets all cursors in the session containing the current cursor.

The **All** option resets all the open cursors.

```
Reset cursor(s) (Current)
Perform SQL {Select * from elements}
Build list from select table
If [LIST.$linecount] > 0
   OK message {[LIST.$linecount] records found}
Else
   OK message {No records found}
End If
```

Restore selection for line(s)

Reversible: NO   **Flag affected:** YES

Parameters: Line number (can be calculation, default is current line)
                 ☐ All lines

Syntax: Restore selection for line(s) [*(All lines)*] [*{line-number}*]

This command copies the Saved selection state to the Current selection state and sets the flag. To allow sophisticated manipulation of data via lists, a list can store two selection states for each line; the "Current" and the "Saved" selection. The Current and Saved selections have nothing to do with saving data on the disk; they are no more than labels for two sets of selections. The lists may be held in memory and never saved to disk: they will still have a Current and Saved selection state for each line but they will be lost if not saved. When a list is stored in the data file, both sets of selections are stored.

The *Restore selection for line(s)* command allows the Saved selection state of the specified line (or All lines) to be copied into the Current set. You can specify a particular line in the list either by entering a number or a calculation. You are required to redraw the list to refresh the state of the displayed list field. The **All lines** option restores the selection states for all lines of the current list. The following example selects the middle line of the list:

```
Set current list LIST1
Define list {LVAR1}
Calculate LVAR1 as 1
Repeat
   Add line to list
   Calculate LVAR1 as LVAR1+1
Until LVAR1 = 6
Select list line(s) {3}
Save selection for line(s) (All lines)
Deselect list lines (All lines)
Restore selection for line(s) (All lines) ;; line 3 selected
Redraw lists
```

Retrieve rows to file

Reversible: NO   **Flag affected:** YES

Parameters: None

Syntax: Retrieve rows to file

This command copies the select table, row by row, into the current client import file. A *Retrieve rows to file* must follow a Select statement, for example, *Perform SQL {Select * from table}*. Any data returned by the remote computer is appended to the import file in tab-delimited format.

It is faster to use *Retrieve rows to file* than to use Fetch/Export loops.

```
Set client import file name {test}
Open client import file
Perform SQL {Select * from table}
Retrieve rows to file
Close client import file
```

Revert class

Reversible: NO   **Flag affected:** YES

Parameters: Class name

Syntax: Revert class *{class-name}*

This command reads the specified class from the library file on disk into RAM, so that any changes made to that class using the notation are lost. The flag is set if the class is successfully re-read. A runtime error occurs if the specified class cannot be found.

```
Calculate $windows.MYWIND.$objs.Field1.$visible as kfalse
; makes change to window
Open window instance MYWIND
Prepare for edit
Enter data
Update files if flag set
Revert class {MYWIND}   ;; puts window back to saved version
```

Rollback current session

Reversible: NO   **Flag affected:**   YES

Parameters: None

Syntax: Rollback current session

This command cancels all transactions for the current session. It causes any SQL transactions sent to the server since the last commit to be rolled back. *Rollback current session* is usually used in conjunction with *Autocommit (Off),* and allows finer control of transaction management than the default *Autocommit* system. With *Autocommit (On)*, the default action for a session is only to rollback all unsuccessful statements after an unsuccessful *Execute SQL script*. A standard management strategy is:

```
Autocommit (Off)
Begin SQL script
; SQL transaction
End SQL script
Execute SQL script
If flag false
   Rollback current session
Else
   Commit current session
End If
; Commit current session and
; Rollback current session   override Autocommit (On)
```

Save class

Reversible: NO **Flag affected:** YES

Parameters: Class name

Syntax: Save class *{class-name}*

This command writes the specified class, which normally contains changes made by notation, into the library file on disk. You use *Save class* to make the changes permanent. The flag is set if the class is successfully saved. A runtime error occurs if the specified class cannot be found.

```
; Example to hide a field on a window and save the new version
Open window instance WCLIENT
Do $iwindows.WCLIENT.$objs.C_FIELD.$visible.$assign(false)
Save class {WCLIENT}
Redraw WCLIENT
Bring window instance to front WCLIENT
```

Save selection for line(s)

Reversible: NO **Flag affected:** YES

Parameters: Line number (can be calculation, default is current line)
☐ All lines

Syntax: Save selection for line(s) [*(All lines)*] [*{line-number}*]

This command saves the selection state of the specified line(s) in memory and sets the flag. To allow sophisticated manipulation of data via lists, a list can store two selection states for each line; the "Current" and the "Saved" selection. The Current and Saved selections have nothing to do with saving data on the disk; they are no more than labels for two sets of selections. The lists may be held in memory and never saved to disk: they will still have a Current and Saved selection state for each line but they will be lost if not saved. When a list is stored in the data file, both sets of selections are stored.

Save selection for line(s) allows the selection state of the specified line (or All lines) to be copied into the Saved set. You can specify a particular line in the list by entering either a number or a calculation. If the line number is not specified, the current line selection is saved. The **All lines** option saves the selection for all lines of the current list. This example selects the middle line of the list:

```
Set current list LIST1
Define list {LVAR1}
Calculate LVAR1 as 1
Repeat
   Add line to list
   Calculate LVAR1 as LVAR1+1
Until LVAR1=6
Select list line(s) (All lines)
Save selection for line(s) (All lines)
Invert selection for line(s) {LIST.$linecount/2}
XOR selected and saved (All lines)  ;; 1 AND 1 = 0, 1 AND 0 = 1
Redraw lists
```

SEA continue execution

Reversible: NO   **Flag affected:**   NO

Parameters: None

Syntax: SEA continue execution

This command continues method execution at the command following the command which called an error handler; SEA stands for Set Error Action. Using it is, in effect, like saying "Error is acknowledged. Now, skip over the error line and proceed with the succeeding good lines."

Using this command is similar to setting the go point in the debugger at L+1 where L is the error line. The command is always used within an error handler.

```
; Error handler to trap break key when waiting for semaphore
If #ERRCODE = KerrCantlock
   OK message {user canceled request for record lock}
   SEA continue execution
End If
; Edit method must test flag to prevent error on update
```

SEA repeat command

Reversible: NO   **Flag affected:** NO

Parameters: None

Syntax: SEA repeat command

This command attempts to repeat the command that caused an error; SEA stands for Set Error Action. This is most useful after an out of memory condition. The command is always used within an error handler. It is your responsibility to ensure that an endless looping situation between the error handler and the command is not created. Also, you must ensure that any side effects of the original execution of the command which caused the error are taken into account.

```
; error handler traps attempt to edit locked
; record and the user presses break key
If #ERRCODE = kerrCantlock
   Yes/No message {Cancel edit}
   If flag true
      Quit all methods
   Else
      SEA repeat command
   End If
End If
```

SEA report fatal error

Reversible: NO   **Flag affected:** NO

Parameters: None

Syntax: SEA report fatal error

This command causes the default action for a fatal error to occur; SEA stands for Set Error Action. If the debugger is available, it is invoked, otherwise, execution halts with an error message. This command, like the other SEA commands, should only be used from within an error handler. The SEA commands determine the behavior following fatal or warning errors.

```
; This causes warning error to generate same action as fatal error
If #ERRCODE = KerrUnqindex  ;; KerrUnqindex is a warning error code
   SEA report fatal error
   ; your method ..
End If
```

Search list

Reversible: NO   **Flag affected:** YES

Parameters: ☐ From start
☐ Only test selected lines
☐ Select matches (OR)
☐ Deselect non-matches (AND)
☐ Do Not Load Line

Syntax: Search list [([*From start*][,*Only test selected lines*] [,*Select matches (OR)*]][,*Deselect non-matches (AND)*]] [,*Do Not Load Line*])]

This command searches the current list for field values that match the current search class or search calculation and loads them into the Current Record Buffer. The search starts at the beginning of the list if **From start** is checked, otherwise at the line after the current line.

If OMNIS finds a line that matches the search class, that line number becomes the current line $line and the flag is set. If OMNIS cannot find a matching line, the $line is cleared and the flag is cleared. If there is no current search class, all lines are said to match and OMNIS sets the flag.

When checked, the **Do Not Load Line** option ensures the line found by the search is not loaded into the current record buffer.

The **Only test selected lines** option restricts the list scan to selected lines only. If the **Select matches (OR)** option is checked, the command scans all the lines from the line after the current line to the end and selects all those that match the search; if you also use the **From start** option, the whole of the list is scanned, that is, the search starts at line 1. Lines that are already selected before the command is executed remain selected. This is equivalent to ORing the existing selected lines with the lines that match the search. The current line is not affected.

If the **Deselect non-matches (AND)** option is used, the command scans all the lines from the line after the current line to the end and deselects all those which do not match the search; if you also use the **From start** option, the whole of the list is scanned, that is, the search starts at line 1. Lines which are already selected before the command is executed are deselected if they do not match the search, that is, the only lines left selected are those which were already selected and which match the search. This is equivalent to ANDing the existing selected lines with the lines which match the search. The current line is not affected.

Using the Select and the Deselect options together alters the selection state so that matching lines are selected, non-matching lines are deselected. The current line is not affected.

This example selects line 3 of the list:

```
Set current list LIST1
Define list {LVAR1}
Calculate LVAR1 as 1
Repeat
   Add line to list
   Calculate LVAR1 as LVAR1+1
Until LVAR1=6
Set search as calculation {LVAR1=3 | LVAR1=1}
Search list (From start)      ;; the current line is now 1
Search list (Select matches (OR))  ;; Selects line 3
Redraw lists
```

or do it like this

```
Do LIST.$search(SearchCalc,FromStart,OnlySelected, ..)
```

Select list line(s)

Reversible: NO   **Flag affected:** YES

Parameters: Line number (can be calculation, default is current line)
☐ All lines

Syntax: Select list line(s) [*(All lines)*] [*{line-number}*]

This command selects the specified list line. The specified line of the current list is selected and is shown highlighted (or checked on popup lists) on any window list fields provided that the field has $multipleselect on. If the line number is not specified, the current list line is selected. The **All lines** option selects all lines of the current list. The current line is not affected. When a list is saved in the data file, the line selection is stored. The following example selects the middle line of the list:

```
Set current list LIST1
Define list {LVAR1}
Calculate LVAR1 as 1
Repeat
   Add line to list
   Calculate LVAR1 as LVAR1+1
Until LVAR1=6
```
Select list line(s) {LIST.$linecount/2}
```
; or we could use Select list line(s) 3
Redraw lists (Selection only)
```

You can select the current line by assigning to its $selected property.

```
Do LIST.$line.$selected.$assign(kTrue)
```

Select printer

Reversible: NO   **Flag affected:**   YES

Parameters: ☐ Discard previous settings
                    Printer name (this parameter Windows only)

Syntax: Select printer *{printer-name}*

This command prompts the user to select a printer. Under Windows, you can choose the required printer from a list of all installed printer drivers. Under MacOS you cannot specify a printer name, the Chooser is opened, but since method execution does not pause while the user makes a choice from the available printers, the following example does not work. When this command is executed, the flag is set if the printer is selected successfully.

The **Discard previous settings** option causes OMNIS to reload the OMNIS page setup with the default system settings for the specified printer.

You can use the function *sys(101)* to return the name of the current printer.

```
Switch sys(6) = 'M'
   Case kTrue
      Select printer
   Default    ;; Windows, NT, or 95
      Select printer {POSTSCRIPT Printer}
      If flag true
         Prompt for report
         Prompt for destination
         If flag true
            Print report
            Quit method
         End If
      End If
End Switch
```

Send advises now

Reversible: NO   **Flag affected:** NO
Parameters: None
Syntax: Send advises now

DDE command, OMNIS as server. This command advises the client applications of all the field values for all the fields for which Advise requests have been received. The values are taken from the CRB.

```
Set main file {FCUST}
Find on CUSTOMER (Exact match) {CVAR1}
Send advises now
```

Send command

Reversible: NO   **Flag affected:** YES
Parameters: Command text
Syntax: Send command *{command-text}*

DDE command, OMNIS as client. This command sends a command or a series of commands as text to the current channel.

The command-text syntax must conform to whatever syntax rules apply to the server program.

The DDE syntax dictates that the commands be enclosed in square brackets and OMNIS attaches special meaning to them in strings. Therefore, it may be necessary to put the command text into one of the OMNIS string variables. For example

```
Calculate #S1 as '[command-text]'
```

puts the command text into #S1 and

```
Send command{[#S1]}
```

sends the command to the server.

Alternatively you can enter the command directly into the command parameter by doubling the first set of brackets, for example

```
Send command {[[releasecontrol]]}
```

The flag is set if the server accepts the command(s).

Syntax and errors

When you send commands to OMNIS, the syntax is defined by the text shown in the method editor. You can enter scripts in OMNIS, copy them to the clipboard and paste them

into the client application. If the sent command returns an error to OMNIS, the hash
variables *#ERRCODE* and *#ERRTEXT* store the error code and message.

```
Set DDE channel number {2}
Open DDE channel {OMNIS|COUNTRY}
If flag false
   OK message {Country library not running}
Else
   Calculate #S4 as 'OK message {Hi, this is DDE magic}'
   Send command {[#S4]}
   Send command {'Next'}
   Close DDE channel
   OK message {Update finished}
End If
```

Send Core event

Reversible:	NO	**Flag affected:**	YES
Parameters:	Core event message (see below)		
	Parameters list		
Syntax:	Send Core event *{event-message* [*(parameter1*[,*parameter2*]...*)*]*}*		

The following core events are available:
Quit Application
Open Documents
Print Documents
Do Script
Create Publisher
Set Data

This command sends one of the "Core" events. The event is sent to the current event
recipient unless an application name is provided as a parameter. Note that *Open
application*, *Quit application, Open Document* and *Print Document* are compulsory events
and will be accepted by an Apple-event-aware recipient at all times. To return a value use
Send Core event with return value.

You can disable the compulsory events using the *Disable receiving of Apple events*
command with the **Disable compulsory events** option checked.

Quit Application

Send Core event {Quit application ('APPNAME')}

Quit application is a compulsory event, and always responded to by an Apple event aware
application. *Send Core event {Quit Application ('APPNAME')}* quits the named application
in the Application Menu. If no parameter is given, the current recipient quits by default.

CAUTION If the event recipient is the Finder, the Finder is quit, and you will need to restart your machine.

```
Send Finder event {Open Files ('MyHD:TeachText')}
; Use TeachText
Ok message {Now quit TeachText}
Send Core event {Quit Application (MyHD:TeachText')}
; Quits TeachText. Now re-set OMNIS by default
```

Open Documents

```
Send Core event {Open Documents ('PATH:DOC1','PATH:DOC2')}
```

Open Document loads the named documents into the target application. If received by OMNIS, the event uses the **Open Library** item in the **File** menu to open each library in turn. If the documents are ad hoc reports, they are opened.

Print Documents

```
Send Core event {Print Documents ('PATH:DOC1','PATH:DOC2')}
```

Print Documents loads the named documents into the target application and prints each one. If received by OMNIS, and the documents are ad hoc reports, they are opened and printed.

Create Publisher

```
Send Core event {Create publisher ('OBJECT','EDITION NAME')}
```

Create Publisher sends a request to the current recipient to publish the OBJECT (a document, spreadsheet, or other database, for example) so that your OMNIS library can use the data in the EDITION NAME. For example

```
Send Core event {Create Publisher ('MySheet','MonthResults')}
```

Set Data

```
Send Core event {Set data ('TARGETFIELD',SOURCEFIELD)}
```

Set data sends an event to the current recipient that takes the data in SOURCEFIELD and puts it in TARGETFIELD. (Notice the use of quotes; if SOURCEFIELD is quoted, the actual string is passed to TARGETFIELD.)

This pushbutton method takes the data in the field CHARFIELD and puts it into a spreadsheet cell.

```
On evClick
   Set current list LIST2
   Send Core event {Set Data ('R1:C1',CHARFIELD)}
   If flag false
      Ok message {Error sending core event}
   End if
   Quit event handler
```

Do Script

Send Core event {Do Script (SCRIPT)}

Do script sends a script to the current recipient which will be executed. When sending methods to OMNIS the *Do script* message is only accepted when OMNIS is not already executing a method or performing an operation. If an event is not accepted, the event errAEEventNotHandled is returned to the sender.

The syntax of the script to be sent to OMNIS is defined simply by the form of the commands as displayed by OMNIS in the method design window (top right-hand list area).

When sent to another Apple application, such as Hypercard, a script must, of course, use the script language and syntax of that application.

The following pushbutton method assumes that a script has been entered in the field SCRIPT. If the script can be run by the current recipient (local OMNIS by default), the results can be seen; otherwise the OK error message appears.

```
On evClick
   Send Core event {Do Script (SCRIPT)}
   If flag false
      OK message {Do Script Failed}
   End If
   Quit event handler
```

When receiving scripts, OMNIS opens the debugger window if it is available and an error occurs when interpreting the script.

Send Core event with return value

Reversible:	NO	**Flag affected:**	YES

Parameters: Return field name or variable
                    Core event message (see below)
                    Parameters list

Syntax: Send Core event *{event-message* [*(parameter1*[,*parameter2*]...)]*}*
                    with return value *field-name*

This command sends either a *Get Data* or *Do Script* event to the current event recipient and returns a value. The flag is set if the event is accepted.

Send Core event {Get data ('TARGETFIELD'} **Returns** FIELD

Get Data

Get Data sends an event to the current recipient and returns data to the specified field or variable.

```
Send Core event {Get Data ('Container1')} Returns LBOOL1
; Container1 is data container in target Lib
If LBOOL1
   OK message {It does!}
Else
   OK message {Sorry, not today}
End If
Send Core event {Do Script (SCRIPT)} Returns FIELD
```

Do Script

Do Script lets you execute a script in a remote application (for example a macro in a spreadsheet) and return a value to OMNIS.

```
; declare local variable LBOOL1 of Boolean type
Use event recipient {HYPERCARD}
; Previously prompted for, and tagged
Send Core event {Do Script (LSCRIPT)} Returns LBOOL1
```

The result of the Hypercard Answer script (LSCRIPT) is a value Yes/No which is returned to the OMNIS source library in the local field LBOOL1.

When a script is sent to OMNIS, the syntax of the commands is defined by what is shown in the method design window. In freetype entry mode, you can create scripts in OMNIS and transfer them via the clipboard to your chosen application.

When sent to another Apple application, such as Hypercard, a script must, of course, use the syntax of that application.

Send Database event

Reversible: NO   **Flag affected:** YES

Parameters: Database event message (see below)
Parameters list

Syntax: Send Database event *{event-message* [*(param1*[,*param2*]...)]*}*

The following database events are available:
Does field exist ('fieldname')
Get field type ('fieldname', 'datatype')
Get field size ('fieldname', 'fieldsize')
Set Field ('myfieldname', yourfield)
Get field ('yourfield', 'myfield')
Does table exist ('format')
Use table ('thatformat')
Define Returns ('source1'[,'source2']...)
Next
Previous
Insert
Delete
Update

This command sends one of the database events to the current event recipient. The flag is set if the event is accepted by the recipient. These events let you send and receive data from other applications that contain fields and row/column database structures (tables; file class names, for OMNIS), provided they implement the Database events. They use the standard terminology of "Table" where OMNIS uses file classes.

You can run OMNIS as a networked data server for any other application on the network and in this configuration would be Client/server.

Database events

The following tables show the OMNIS event messages used with *Send Database event*. The name in the first column is the Apple term for the event. CRB is Current Record Buffer. RSN is Record Sequence Number.

General Database events

The following table shows the OMNIS event messages used with *Send Database event*, and the parameters for each message. These general commands are used to interrogate or set values in a database. The last two columns show the results when OMNIS is the target and when it is the source of the events.

Apple Event name	OMNIS Database command message	Command parameters to send event	Action when event received by OMNIS	Action when event sent by OMNIS
Get Structure	Get field type	FieldName, ResultField	Returns field type of FieldName to client	Sets text in ResultField to be field type of FieldName
Does Object Exist	Does field exist	FieldName	Returns Boolean (0 /1) if FieldName does not /does exist	Sets OMNIS flag to true / false if FieldName does / does not exist at the server
Get Data	Get field	FieldName, ResultField	Returns value from CRB corresponding to FieldName	Gets data from FieldName and return data into ResultField (CRB)
Set Data	Set field	Value, FieldName	Sets data in OMNIS CRB field 'FieldName' to be Value	Sets the data of FieldName in the remote server to be Value
Get Data Size	Get field size	FieldName, ResultField	Returns data size in bytes for FieldName	Returns the data size of FieldName in bytes into ResultField

Record events

The following database events are used with complete OMNIS records (records or rows in other applications). The *Send Database event {Define Returns ('FIELD1','FIELD2','FIELD3' . . .)}* allows OMNIS to define fields as the source and destination of "Next", "Previous", "Insert" and "Update".

Apple Event name	OMNIS Database command message	Database command parameters to send event	Action when event received by OMNIS	Action when event sent by OMNIS
Does Object Exist	(none)	N/A	Returns true/false if RSN.	N/A
Get Data	Next and Previous	...	Returns record with requested RSN (or nearest following/previous RSN)	Returns next/ previous sequenced record into previously defined OMNIS fields
Set Data	Update	...	Sets requested RSN to data specified.	Sets server record with OMNIS defined fields
Delete Element	Delete record	...	Deletes OMNIS record with RSN specified by client.	Delete record as defined by defined fields from server table.

Table events

A table is simply described as a collection of rows and columns in a database or spreadsheet. For OMNIS, this equates to the combination of an OMNIS file class and corresponding data file, and takes the name of the file class as a parameter. OMNIS keeps a "table index" (record pointer) for the table currently in use for database events so record (row) operations can be performed.

The *Send Database event{Use table ('TABLENAME')}* command must be issued with a valid table name (file class name for OMNIS) that will be used for all subsequent record (row) operations. This OMNIS command sends the Does Object Exist event before setting the current active table to ensure that there is such a valid table, and also resets the "table index" to point to the first record in that table. Use Table may also be used to reset the table index to the first record in a table.

Apple Event name	OMNIS Database command message	Database command parameters to send event	Action when event received by OMNIS	Action when event sent by OMNIS
Does Object Exist	Does table exist	TableName. (File class name when sent to OMNIS)	Returns Boolean (1/ 0) if TableName does/does not exist	Sets OMNIS flag to true / false if TableName does / does not exist at the server
New Element	Insert	...	Inserts OMNIS record with values specified by event issued by client	Adds record to server table with values defined by defined fields

Data entry example

This set of methods shows how you can handle data entry remotely. Several pushbuttons are put on the local window to mimic the standard OMNIS buttons, with methods behind them to handle data in the server library with file class "f2".

The following commands are demonstrated:

Send Database event *{Define Returns ('source1'*[*,'source2'*]*...)}*

Send Database event *{Use table ('thatformat')}*

Send Database event *{Insert}*

Send Database event *{Previous}*

Send Database event *{Next}*

Send Database event *{Update}*

```
; Declare class variable EditType of type SHORT INTEGER (0 TO 255)
Set main file {f2}
Send Database event {Define Returns ('CVAR1','LVAR1','CVAR2')}
; defines local fields for values from table f2
Send Database event {Use table ('f2')}  ;; the name of a file class


$control   ;; window control method
On evOK
   If EditType = 1
      Send Database event {Insert}
   Else If EditType = 2
      Send Database event {Update}
   End If
   Calculate EditType as 0
   If len(CVAR1) = 0
      Enable fields {entry1014,entry1016}
   Else
      Disable fields {entry1014,entry1016}
   End If
   Redraw {entry1014,entry1016}
```

The following methods run behind pushbuttons.

```
; Example of 'Next' pushbutton
On evClick
   Send Database event {Next}
   Redraw DataEntryWin
   If flag false
      OK message {No more records}
   End If
   Quit event handler
; Example of 'Insert' pushbutton
On evClick
   Calculate EditType as 1
   Clear range of fields CVAR1 to CVAR5
   Clear range of fields #1 to #60
   Redraw DataEntryWin
   Enter data
   Quit event handler
```

Changing a field value

The following example method prompts for a recipient library, and then changes the value of a field in the current record buffer.

The following commands are demonstrated:

Send Database event *{Does field exist ('fieldname')}*

Send Database event *{Set Field ('myfieldname', yourfield)}*

```
; Declare local variable TEMP of Character type
On evClick
   Calculate TEMP as 'Not known'
   Prompt for event recipient {Betas}
   ; prompts for the library, and tags it
   Send Database event {Does field exist ('CONTACT')}
   ; just to confirm its name
   If flag true
      OK message {Contact name [CONTACT] found;
        OK to change to 'Not known'}
   Else
      OK message {Sorry, can't find CONTACT; quitting method}
      Quit method
   End If
   Send Database event {Set Field ('CONTACT', TEMP)}
   If flag true
      OK message {CONTACT now changed to[TEMP]}
   Else
      OK message {Failed to set remote field}
   End If
   Quit event handler
```

The following additional commands are shown:

Send Database event *{Does table exist ('format')}*

Send Database event *{Get field ('yourfield', 'myfield')}*

```
Send Database event {Does table exist ('Beta sites')}
If flag false
   OK message {Sorry, 'Beta sites' not found}
Else
   Send Database event {Get field ('CO_NAME','%%S4')}
   ; Note use of quotes round local variable %%S4
   OK message {Returned value is [%%S4]}
End If
```

The following method fragments demonstrate two further commands:

Send Database event *{Get field size ('fieldname', 'fieldsize')}*

Send Database event *{Get field type ('fieldname', 'datatype')}*

```
; Declare local variable DATASIZE of Character type
; Declare local variable DATATYPE of Character type
Send Database event {Get field size ('CHARFIELD','DATASIZE')}
OK message {Field 'CHARFIELD' has room for [DATASIZE] characters}
Send Database event {Get field type ('CHARFIELD','DATATYPE')}
OK message {Field CHARFIELD is of type [DATATYPE]}
```

Finally, the current record buffer is deleted by:

Send Database event *{Delete}*

```
; Example of 'Delete' pushbutton
On evClick
   OK message {Are you sure you want to delete the current record?}
   If flag true
      Send Database event {Delete}
      If flag true
         OK message {Current record now deleted}
      Else
         Ok message {Delete event not accepted}
      End if
   End If
   Quit event handler
```

Send field

Reversible: NO   **Flag affected:** YES

Parameters: Field name
Server data item name

Syntax: Send field *field-name* [*{server-data-item-name}*]

DDE command, OMNIS as client. This command sends the value of an OMNIS field to the current DDE channel. An error occurs if the channel is not open. The command takes the OMNIS field name and the server data item name as parameters. The data item name can contain square bracket notation. If the data item name is not specified, the OMNIS field name is used.

The flag is set if the server program accepts the value.

```
Set DDE channel number {1}
Open DDE channel {OMNIS|DDE2}
Calculate CVAR1 as '[TakeControl]'
Send command {[CVAR1]}
If flag false
   OK message {Error sending: [CVAR1]}
End If
Send field C_CLIENT {S_NAME}
Send field C_TOTAL {S_TOTALS}
```

Send Finder event

Reversible:	NO	**Flag affected:**	NO

Parameters: Finder event message (see below)
                  Parameters list

Syntax: Send Finder event *{event-message* [*(parameter1*[*,parameter2*]*...)*]*}*

The following Finder events are avalailable:

Show About	*Reveal Files*
Get File Info	*Share Files*
Duplicate Files	*Empty Trash*
Make Alias For Files	*Restart Macintosh*
Open Files	*Show Clipboard*
Print Files	*Shutdown Macintosh*
	Sleep Macintosh

This command sends one of the Finder events to the standard MacOS Finder. With the exception of the *Open Files* and *Print Files* messages, events in this group can only be sent to the local Finder.

The Finder event suite lets you manipulate files on your hard disk. If the events are accepted, the flag is set to true.

You might be familiar with the following events that act directly on the local Finder, since you can find them on the Finder's pull-down menus.

```
Send Finder event {Get File Info}
Send Finder event {Make Alias For Files}
Send Finder event {Reveal Files}
Send Finder event {Share Files}
Send Finder event {Duplicate Files}
```

If run without parameters, they bring up a standard dialog window, allowing one or more files or folders to be selected for action. Pathname parameters can also be entered from the keyboard, using Apple syntax; see the appropriate Apple reference manuals.

```
Send Finder event {Get File Info
                   ('MyHD:Desktop folder:Microsoft Word')}
```

The other four events above behave in a similar way.

The following messages are self-explanatory and take no parameters.

```
Send Finder event {Empty Trash}
; permanently removes deleted files.
Send Finder event {Show About}
; shows the 'About' information for the computer.
Send Finder event {Restart}
Send Finder event {Show Clipboard}
Send Finder event {Shutdown}
Send Finder event {Sleep} ;; for PowerBooks and other portables
Send Finder event {Open Files}
Send Finder event {Print Files}
```

You can use these last two events to launch and print files under MacOS, for example:

```
Send Finder event {Open Files('YourMac:MyHD:Apps:AnApp:Doc2')}
Send Finder event {Print Files('MacNum:MyHD:Apps:AnApp:MyDoc')}
Send Finder event {Open Files ('MyHD:AppleDoc')}
; this is the same as double-clicking on the AppleDoc icon.
```

Send to a window field

Reversible: YES   **Flag affected:**   NO

Parameters: ☐ Show printer pages
Screen report field name

Syntax: Send to a window field [*(Show printer pages)*] *{field-name}*

This command directs the output of a report to a window Screen Report field; you cannot print to any other type of window field. When you print the report the field is changed into a standard screen report window that has all the features of the standard screen report. The **Show printer pages** option show the outline of the current paper size in the report field.

An error is generated if the field name is invalid for the current window. If you use *Send to a window field* in a reversible block, the report destination reverts to its former setting when the method terminates.

```
; $event() method for a button
On evClick
   Send to a window field {ScreenReportField}
   Set report name RLABELS
   Print report (Show printer pages)  ;; prints to screen report
   field showing current paper margins
```

Send to clipboard

Reversible: YES   **Flag affected:** NO

Parameters: None

Syntax: Send to clipboard

This command sends the output of any subsequent reports to the clipboard. The report is printed as a text-only file and all text formatting is ignored. If two reports are sent to the clipboard, the second report overwrites the first. Once a report has been sent to the clipboard, you can launch another program, such as a word processor, and paste the report into it.

If you use *Send to clipboard* in a reversible block, the report destination reverts to its former setting when the method terminates. The contents of the clipboard are not altered by the command or its reversal.

If you want to copy pictures from a report to the clipboard, you can print the report to screen and use the mouse to select the area required. The standard Edit menu Copy option will copy the graphic to the clipboard.

```
Send to clipboard
Set report name Orders
Print report
; Now launch word processor and paste
```

Send to DDE channel

Reversible: YES   **Flag affected:** NO

Parameters: None

Syntax: Send to DDE channel

This command directs any subsequent reports to a DDE channel. The current channel is defined by *Set DDE channel number.* An error occurs if the channel is not open or if the report is not printed with an export format.

Each record within the report is prepared and sent by OMNIS as the data in a Poke message. The term "Poke" is defined by the DDE protocol and refers to messages carrying data which set field values in the target program. The server's item names, into which the exported data is read, are defined by *Set DDE channel item name*.

The subsequent print commands will send to the channel number which is current at the time of the print command, not at the time of the *Send to DDE channel* command.

If you use *Send to DDE channel* in a reversible block, the report destination reverts to its former setting when the method terminates.

It may be the case that an export format for a particular OMNIS report does not correspond to any of the formats supported by DDE. If a mismatch occurs, there will be an error message at the *Print report* or *Prepare for print* command.

```
Send to DDE channel
Set export format {Delimited (Tabs)}
Set report name DDEReport
Clear DDE channel item names
Set DDE channel item name {Name}
Set DDE channel item name {Tel}
Print report
Close DDE channel
```

Send to file

Reversible: YES **Flag affected:** NO
Parameters: None
Syntax: Send to file

This command directs the report output to the currently selected print file. The report is sent as a text file (no text style or formatting) with the appropriate line terminators. The print file is not closed when a report finishes so you can print multiple reports without changing the destination or the name of the print file.

When you select the destination using the dialog window (see *Prompt for destination*), the Page size pushbutton lets you set up the form feeds and lines per page. These settings are stored in the preferences file.

Set lines per page lets you specify page length from methods. If the Send form feed option is selected, the end of each page is marked by a form feed character; otherwise, the pages are forced by sending multiple line feeds. You use *Set print file name* to designate the file name.

If you use *Send to file* in a reversible block, the report destination reverts to its former setting when the method terminates.

```
Send to file
Set lines per page {46}
Set print file name {Output.txt}
Print report
```

Send to page preview

Reversible:	YES	**Flag affected:**	NO

Parameters: ☐ Do not wait for user
                   ☐ Hide until complete
                   Report title
                   /left/top/right/bottom page preview position and size (coords in pixels)
                   /STK   to stack the preview
                   /CEN   to center the preview

Syntax: Send to page preview [([*Do not wait for user*][,*Hide until complete*])]
                [*report-title*] [/*left*[/*top*[/*right*[/*bottom*]]]] [/*STK*][/*CEN*]

This command sends the report instance to a page preview screen. This lets the user check the final page layout before printing. On small screens, the text is Greeked, that is, each character is represented by a dot.

The **Do not wait for user** option allows subsequent method lines to execute or lets the user do other things without closing the report; the default is to gray out all menus while a screen report is displayed. You may want to have several reports on the screen for reference while doing some other work with the library. Without the option, the user must close the window before doing anything else. The number of screen report instances is limited by the operating environment. Under Windows, you should refer to the OMNIS.INI settings if you are prevented from opening enough report windows.

The **Hide until Complete** option suppresses the output until all the report data is ready. Normally, you can view the first part of the report before all the records have been prepared.

Title and Position

You can give each page preview a title and control its position and size. The Left/Top/Right/Bottom values fix the positions of the four corners to screen pixel resolution. The /STK parameter offsets the top left-hand corner from the last page preview and /CEN positions the page preview in the middle of the screen. The following example stacks two page preview showing US and UK customers.

```
Set report name RS_FCUSTOMERS
Send to page preview (Do not wait for user) UK customers/STK
Set search as calculation {CU_COUNTRY = 'UK'}
Print report (Use search, Do not finish other reports) {rinst1}
Send to page preview (Do not wait for user) USA customers/STK
Set search as calculation {CU_COUNTRY = 'USA'}
Print report (Use search, Do not finish other reports) {rinst2}
```

If you change the shape and size of the page preview window it will no longer reflect the paper size.

If you use *Send to page preview* in a reversible block, the report destination reverts to its former setting when the method terminates.

Send to port

Reversible: YES   **Flag affected:**   NO

Parameters: None

Syntax: Send to port

This command directs the report output to the currently selected port. The report is sent as a stream of text with the appropriate line terminators. The port is selected with the *Set port name* command.

If you use *Send to port* in a reversible block, the report destination reverts to its former setting when the method terminates.

```
; Set port name {Com2:}                  ;; for Windows
; Set port name {2 (Printer port)}       ;; for MacOS
Send to port
Set port parameters {9600,n,7,0}
Print report
```

Send to printer

Reversible: YES   **Flag affected:**   NO

Parameters: None

Syntax: Send to printer

This command sends the report to the current printer. You can choose the printer using the *Select printer* command.

If you use *Send to printer* in a reversible block, the report destination reverts to its former setting when the method terminates.

```
Set report name MyOrder
Send to printer
Print report
```

Send to screen

Reversible:	YES	**Flag affected:**	NO

Parameters: ☐ Page for screen
☐ Do not wait for user
☐ Hide until complete
Report title
/left/top/right/bottom report position and size (coords in pixels)
/STK   to stack the report instance
/CEN   to center the report instance

Syntax: Send to screen
[([*Page for screen*][,*Do not wait for user*] [,*Hide until complete*])]
[*report-title*] [/*left*[/*top*[/*right*[/*bottom*]]]] [/*STK*][/*CEN*]

This command sends the output of the current report to the screen. The screen report uses the appropriate fonts and page size, but ignores the margins.

The **Page for screen** option paginates the report for screen-length pages.

The **Do not wait for user** option allows subsequent method lines to execute or allow the user to do other things without closing the report; the default is to gray out all menus while a screen report is displayed. You may want to have several reports on the screen for reference while doing some other work with the library. Without the option, the user must close the window before doing anything else. The number of open screen reports is limited by the operating environment. Under Windows, you should refer to the OMNIS.INI settings if you are prevented from opening enough report windows.

The **Hide until Complete** option suppresses the output until all the report is ready. Normally, you can view the first part of the report before all the records have been prepared.

Title and Position

You can give each screen report a title and control its position and size. The Left/top/right/bottom parameters set the positions of the four corners to screen pixel resolution. The /STK parameter offsets the top left-hand corner from the last report instance and /CEN positions the report in the middle of the screen. The following example stacks two report instances showing US and UK customers.

```
Set report name RS_FCUSTOMERS
Send to screen (Do not wait for user) UK Customers/STK
Set search as calculation {CU_COUNTRY = 'UK'}
Print report (Use search, Do not finish other reports) {rinst1}
Send to screen (Do not wait for user) USA Customers/STK
Set search as calculation {CU_COUNTRY = 'USA'}
Print report (Use search, Do not finish other reports) {rinst2}
```

If you use *Send to screen* in a reversible block, the report destination reverts to its former setting when the method terminates.

The prompt for destination dialog sets the **Do not wait** option and clears all the others.

As with all scrolling fields in OMNIS, screen reports can have panes. This lets you split the window into panes and scroll each pane separately.

Send to trace log

Reversible: NO   **Flag affected:**   NO
Parameters: Text
Syntax: Send to trace log *{Text}*

This command sends a specified line of text to the trace log. The text can contain square bracket notation.

```
Send to trace log {Value of CLIENT field is [CLIENT]}
Send to trace log {Current task is [$ctask().$name]}
Send to trace log {Win1 $control: events are [sys(86)]}
```

Send Word Services event

Reversible: NO   **Flag affected:**   NO
Parameters: Word Services event message (at present, Check field text only)
                Field name
Syntax: Send Word Services event *{Check field text* [*('field-name')*]*}*

This command performs the specified Word Services event on the parameter, typically an OMNIS field. Together with *Prompt for word server* this event allows spell checking and other text services to be carried out on data entry text and text variables in OMNIS fields. It uses the current word services application, spell checker or grammar checker, for example. Once you have set up the word server, OMNIS stores the path in the preferences file and there is no need to prompt for the server each time you use OMNIS.

```
On evClick
   Prompt for word server    ;; opens a dialog box
   Send Word Services event {Check field text ('CVAR1')}
   ; checks the text in CVAR1
   Quit event handler
```

Server specific keyword

Reversible: NO   **Flag affected:** YES

Parameters: Server keyword

Syntax: Server specific keyword *{server-keyword}*

This command sends a server-specific keyword to the current DAM. This mechanism supports server-specific functionality which you cannot access via SQL scripts. At present, the Sybase error and message handling keywords are supported. For example

Server specific keyword `{<SQLMESSAGE>MSQL/sqlMessage}`

causes the DAM to call OMNIS method MSQL/sqlMessage each time a message is returned by SQLServer.

Similarly

Server specific keyword `{<SQLERROR>MSQL/Error}`

calls method MSQL/Error each time an error is returned.

Set 'About...' method

Reversible: YES   **Flag affected:** NO

Parameters: Number or class name/number (of method)

Syntax: Set 'About...' method [*class-name/*]**number**
 [*{method-name}*]

This command changes the "About..." option by calling the specified method which you should set to open a different About window. OMNIS executes the specified method when this option is selected in exactly the same way as if it had been selected from a menu, for example, standard windows are closed. If you use *Set 'About...' method* in a reversible block, the command is reversed when the method terminates.

There are no restrictions on what you can do in the *Set 'About...' method*, that is, the method that is called. Extra care is needed to ensure that the method does not alter any variables, lists or the status of the flag.

```
Set 'About...' method Code1/About {About Library}
; End of method


; About Library              ;; the 'About' method
Open window instance WABOUT  ;; your own About window
Enter data
Close window WABOUT
```

Set advise options

Reversible: YES   **Flag affected:** NO

Parameters: ☐ Find/next/previous
                 ☐ OK
                 ☐ Redraw

Syntax: Set advise options [*([Find/next/previous][,OK][,Redraw])*]

DDE command, OMNIS as server. This command determines when OMNIS is permitted to send requested Advise messages to the client application. When the *Accept advise requests* option is active, OMNIS will accept Advise requests from the client program. By default, the client program will only be advised of the values requested from OMNIS when *Send advises now* is executed.

However, *Set advise options* specifies other events which will cause the values to be sent. There are three checkbox options available for this command: Find/next/previous, OK, and Redraw.

The **Find/next/previous** option sends the requested Advise value whenever a Find/next/previous command or a Clear command is executed. The **OK** option sends the requested Advise value whenever an *Enter Data* or *Prompted Find* ends with an OK. The **Redraw** option sends the requested Advise value whenever a *Redraw* is executed.

Each of these options in *Set advise options* has its command equivalent within the *Exchanging Data...* group, whose function is identical. These commands are listed as *Advise on Find/next/previous, Advise on OK,* and *Advise on redraw*.

```
Set server mode (Field requests,Advise requests)
Set advise options (Find/next/previous,OK)
OK message {Server mode for DDE enabled}
```

Set batch size

Reversible: NO   **Flag affected:**   YES

Parameters: Number of rows

Syntax: Set batch size *{number}*

This command sets the number of rows read into the local buffer by each *Fetch next row* command. Following a SQL Select statement, rows of data are held on the server ready for the client. With some servers, you can maximize network efficiency by adjusting the number of rows transferred to the client. *Fetch next row* transfers the first row of data to the client and reads it into the CRB. The next *Fetch next row* reads the next row from the local buffer and no network traffic is generated.

You can set the batch size high for small record sizes and low for large record sizes. Optimal values depend on the size of data packets used by the network. Sybase servers generally perform their own batching of rows and do not need tuning.

At the time of writing, the Oracle and ODBC DAMs support *Set batch size*.

```
Set batch size {50}
Perform SQL {Select * from AUTHORS}
Fetch next row
While flag true  ;; process rows
   Fetch next row
End While
OK message {[sys(135)] rows processed}
```

Set bottom margin

Reversible: NO   **Flag affected:**   NO

Parameters: Measurement
          ☐ Measurement in cms (leave unchecked for inches)

Syntax: Set bottom margin [*(Measurement in cms)*] *{number}*

This command specifies the bottom margin for the current report class. It overrides the $bottommargin property until such time as the current report is reset.

```
Set report name ROrders
Yes/No message {Print on metric A4 paper?}
If flag true
   Set bottom margin (Measurement in cms) {2.34}
   Set top margin (Measurement in cms) {1.2}
Else
   Set bottom margin {1.0}
   Set top margin {1.0}
   ; Default measurement is inches
End If
Print report
Set report name RTOTALS
; The settings for ROrders are now deleted
```

Set break calculation

Reversible: NO   **Flag affected:**   NO

Parameters: Field name
          Calculation

Syntax: Set break calculation on *field-name {calculation}*

This command stops method execution when the specified calculation evaluates to true; all values except zero are considered true. You use *Set break calculation* after a *Variable menu command: Set break on calculation {field-name}* command. The field used in the command does not have to feature in the calculation but is used to "label" the break within OMNIS.

At breakpoints, a method design window is opened with the current method loaded and the breakpoint command highlighted. You can examine field values by right button/ Ctrl-clicking on the field or step through the remaining method.

Setting up calculated breakpoints slows down method execution considerably so you should use them sparingly. In runtime the command does nothing.

```
Variable menu command: Set break on calculation {CVAR1}
Set break calculation on CVAR1 {sys(131)<<>>0}
Variable menu command: Set break on calculation {#F}
Set break calculation on #F {#F=0} ;; this monitors for flag false
```

Set character mapping

Reversible: NO   **Flag affected:**   YES

Parameters: Name of map file

Syntax: Set character mapping *{map-name}*

This command loads a character mapping file for the current session. You may need character translation if the data stored on the server did not originate in OMNIS, and the data uses a different character set. The differences usually affect extended character sets which support non-ASCII values (that is, greater than 127).

You must create two translation tables, one for characters coming IN to OMNIS, the other for characters OUT of OMNIS to the server. They are given the same name but with extensions .IN and .OUT, respectively. You must place them in a subdirectory/folder called CHARMAPS under EXTERNAL (under Windows) or in the EXTERNAL folder (under MacOS). For example, you would define the mapping for EBCDIC in files EBCDIC.IN and EBCDIC.OUT, and you would load them with:

```
Set character mapping {EBCDIC}
```

You can load different tables for each session in use. The mapping affects the current session only.

```
Set current session {Session_1}
Set character mapping {ANSIDOS}
Set current session {Session_2}
Set character mapping {EBCDIC}
```

Set class description

Reversible: NO   **Flag affected:** YES

Parameters: Class name/description

Syntax: Set class description *{class-name*[*/description*]*}*

This command sets the description text for the specified library class. When a class is created, you must specify a class name and also an optional description of up to 255 characters. This command lets you set the description string for the specified library class. The original description for the specified class is cleared if the description parameter is left blank (or evaluates to an empty string). The flag is set if the description is changed.

```
New class {Search/S_My}
Modify class {S_My}
Delete class {S_User}
Rename class {S_My/S_User}
Set class description {S_User / [CVAR1]}
; Sets the description to the string value CVAR1
Set search name S_User
Print report (Use search)
```

Set client import file name

Reversible: NO   **Flag affected:** YES

Parameters: File class name

Syntax: Set client import file name *{file-name}*

This command defines the name of the import file into which you wish to store the data returned from a SQL transaction. It moves data from the server to a tab-delimited import file on the local disk. The only parameter is the name of the OMNIS import file. It is important to remember that the import file name you supply here should match the one you have used in the OMNIS methods that import the data.

```
Set client import file name {xprImportFile}
Open client import file
Perform SQL {select cust_name, cust_city, credit_line from customer}
Retrieve rows to file
Close client import file
```

Set closed files

Reversible:	YES	**Flag affected:**	YES
Parameters:	List of files		
Syntax:	Set closed files *{file1*[*,file2*]...*}*		

This command sets the file mode of the specified file(s), other than a main file, to closed. Closing a file prevents any data from being read or changed in that file.

If you attempt to close the main file an error occurs. If you use *Set closed files* in a reversible block, the file mode is reset when the method terminates. *Set closed files* does not cancel the Prepare for update mode. In multi-user libraries, closing a file prevents OMNIS from locking it.

Closing a parent file when editing a child has the effect of protecting the connections from child to parent from change and saves time when locating child records because the parent record is not loaded.

In the method editor, a list of files is displayed. You can Ctrl/Cmnd-click on the file names to select multiple names.

```
Set main file {FPORDERS}
Set closed files {FINVOICES,FINVITEMS}
Set memory-only files {FCONSTANTS,FNAMES}
Set read-only files {FREADFILES}
```

Set current cursor

Reversible:	NO	**Flag affected:**	YES
Parameters:	SQL cursor name		
Syntax:	Set current cursor [*{cursor-name}*]		

Creates a cursor in the current session with the specified cursor name. If that cursor already exists, it becomes the current cursor. The cursors in the same session are always logged onto the same database. Logging on or logging off any of the cursors in a session always logs on or logs off the other cursors in the same session.

Each cursor maintains its own select table, error status, and so on, but the transactions for all the cursors in the same session are all committed or rolled back at the same time. This means that the *Start session*, *Set hostname*, *Set username*, *Set password*, *Logon to host*, *Logoff from host*, *Commit current session*, *Rollback current session* and *Autocommit* commands act on *all* the cursors in the same session as the current cursor. The other SQL commands only act on the current cursor.

If there is no current cursor, *Set current cursor* and *Set current session* are equivalent.

If the cursor name is left blank or a SQL command is sent before a *Set current session* or *Set current cursor* command is encountered, a default session called CHANNEL_1 is automatically created and made the current session.

Once you have created a cursor, you can use either *Set current session* or *Set current cursor* to make it the current cursor but you cannot use *Set current cursor* to move the cursor into another session.

```
Set current session {SESSION_1}
Set current cursor {Cursor_1B}
Logon to host
; Deal with errors here
Perform SQL {Select * from CLIENTS}
Set current session {SESSION_1}
Perform SQL {Select * from ORDERS}
; You now have two select tables available, to access CLIENTS
Set current session {Cursor_1B}
; process CLIENTS
Set current session {SESSION_1}
; Process ORDERS
```

Set current data file

Reversible: YES   **Flag affected:**   NO

Parameters: Internal name (of data file)

Syntax: Set current data file *{internal-name}*

This command sets the specified data file the "current" data file. If your methods refer to file class names without specifying the data file, it is essential to make the appropriate data file current before setting a main file.

```
Open data file {Archive/DataA}
Open data file (Do not close other data) {MYDATA/DataC}
Set current data file {DataA}
Set main file {File1}
; File1.Field1 now refers to DataA.File1.Field1
```

Set current list

Reversible: YES   **Flag affected:** NO

Parameters: List or row name

Syntax: Set current list *list-name*

This command sets the current list, that is, the list to be processed in the subsequent list commands. You can make any type of list the current list, including local, class, and library variables of list data type. If you use this command as part of a reversible block, the current list reverts to its former value when the method containing the reversible block finishes.

```
; declare variable CLIST of List type
Set current list CLIST
Define list {ASSIGNDATE,FIRST_NAME,LAST_NAME}
Set main file {FCUSTOMERS}
Build list from file on CCODE
```

See *Define list.*

Set current session

Reversible: NO   **Flag affected:** YES

Parameters: SQL session name

Syntax: Set current session [*{session-name}*]

This command creates a session with the specified session name. If the named session already exists, it becomes the current session. It allows multiple simultaneous conversations with different remote databases and multiple simultaneous select tables. Session names can be up to 15 characters long and are case-insensitive. If the session name is left blank or a SQL command is sent before a *Set current session* command is encountered, a default session called CHANNEL_1 is automatically created and made the current session.

The first use of *Set current session* with a particular name creates the session which becomes the current session. All successive commands are sent to that session until another *Set current session* is issued. Each session has its own select table, import file, error status, and so on. There is no limit to the number of sessions that you can have open at one time, apart from the limits imposed by available memory and other resources.

```
Set current session {Session_O}
Start session {ORACLE}
Set database version {ORACLE5}
Set current session {Session_S}
Start session {SYBASEDB}
Set current session {Session_O}
; Now log on to ORACLE and so on
```

Set database version

Reversible:	YES	**Flag affected:**	NO
Parameters:	Server type		
Syntax:	Set database version *{server-type}*		

This command sets the server type or version used by the current DAM. You should issue this command after the *Start session* and before the *Logon to host*. For example

```
Start session {ORACLE}
Set database version {ORACLE7}
```

Some of the database versions you can use are:

DAM	Database Version
INFORMIX	INFORMIX
ODBC	
ORACLE	ORACLE7
SybaseDB & CT	SQLSERVER
EDA	EDASERVER

Set DDE channel item name

Reversible:	NO	**Flag affected:**	YES
Parameters:	Server data item name		
Syntax:	Set DDE channel item name *{server-data-item-name}*		

DDE command, OMNIS as client. This command specifies the server data item name to which you can send the exported report. When transmitting a *Send to DDE channel* report, OMNIS takes the channel item name and uses it as the server item name which is to be sent.

The flag is cleared if the item name is too long, thus causing a memory allocation error to take place.

The item names set in the command accumulate over each use of the command until a *Clear DDE channel item names* is issued.

Within a client library, for example, a report class is created which sends the fields ClF1, ClF2...ClF5 to the current channel. At the server end of the conversation, the fields are to be read into five fields SVR1, SVR2...SVR5. Before you can print the report, the method must contain the following commands:

```
Set report name Export_to_channel
Send to DDE channel
Set DDE channel number {1}
Open DDE channel {PROG|LIBRARY}
Send command {[[TakeControl]]}
If flag true
   Set DDE channel item name SVR1
   Set DDE channel item name SVR2
   Set DDE channel item name SVR3
   Set DDE channel item name SVR4
   Set DDE channel item name SVR5
   Print report
End If
```

Set DDE channel number

Reversible: YES **Flag affected:** YES

Parameters: Channel number (can be a calculation)

Syntax: Set DDE channel number *{number}*

DDE command, OMNIS as client. This command sets the channel number to be used in subsequent DDE commands.. Each channel number identifies a particular conversation.

The channels are numbered from 1 to 8, and the flag is cleared if an invalid channel number is used. The channel number in a newly selected library defaults to 1. The channel number selected can be the result of a calculation. All subsequent channel commands function on the current channel number. To select another channel, you must use a new *Set DDE channel number* command.

```
Set DDE channel number {2}
Open DDE channel {OMNIS|COUNTRY}
If flag false
   OK message {Country library not running}
Else
   Send command {Do method Invoice}
   Do method TransferData
End If
```

Set default data file

Reversible: YES   **Flag affected:**   NO

Parameters: File or list of files

Syntax: Set default data file *{file1*[,*file2*]*...}*

This command sets the default data file to be the current data file. Normally, file classes are associated with whatever the current data file is, at the time of execution. You use *Set current data file* to change the identity of the current data file. As the current data file changes, the file classes are associated with the changed current data file.

Set default data file sets the data file, for the specified file class or list of file classes, to be fixed at whatever is the current data file at the time when the command executes. In other words, it creates an association between a list of file classes and the particular data file that was current. For these file classes, the data file becomes fixed (that is, the "default" data file) and does not change whenever the current data file changes. You can break the association with either a new *Set default data file* or a *Floating default data file* command.

When you close the default data file for a file, that file reverts to a floating state. This means that the default data file for that file reverts to the current data file and changes when the current data file changes.

Set default data file does not change the flag but is reversible, that is, when the command is reversed, the previous default data files are restored. A runtime error occurs if there are no data files open when the command is executed.

```
Open library {MYLIB}
Open data file {D1}
Open data file (Do not close other data) {D2}
Set default data file {FCLIENTS, FINVOICES}
Set current data file {D1}
Set main file {FCLIENTS}
; this refers to the D2 data file,
; not D1 (which is the current data file)
Open window instance WCLIENT
```

Set event recipient

Reversible: YES   **Flag affected:** YES

Parameters: Application name

Syntax: Set event recipient
                [{('*application_name*[:*mac_name*[@*zone_name*]]')}]

This command specifies the name of the application to which subsequent Apple events are to be sent. The name of the application must exactly match the name in the System 7 Application menu, for example "Microsoft Excel". This name becomes the "recipient tag" by which you can select it from all the current event recipients.

You can access another machine by specifying its name and zone together with the application name. The *zone_name* is where the Mac or PowerMac and MacOS applications reside (when you specify the zone you must also specify the Mac or PowerMac name). If you omit *zone_name*, the current zone is the default. The *mac_name* is the Mac or PowerMac on which the event recipient resides. If you omit *mac_name* (and *zone_name*) your machine (the host) receives the events by default. When you launch OMNIS, the recipient defaults to OMNIS, that is, events are sent to itself. In the same way, if you use this command without a parameter, the recipient reverts to OMNIS.

The *application_name* must exactly match the name of the application. If a match is found, the flag is set. The application name is stored in this form as an event recipient, as seen in a list created with *Build list of event recipients*.

The following example shows the difference between *Use event recipient*, which is used with a tag previously assigned by the user with *Prompt for event recipient*, and *Set event recipient*, which takes a local application name as a parameter, and turns it into a recipient tag.

```
Prompt for event recipient {MyApp1}
; Prompt user and select application
; do something with 'MyApp1'
Set event recipient {Microsoft Excel}
; This is the name of a current application, as shown on
; the Apple Application menu
; do something in 'Microsoft Excel' for example
Use event recipient {MyApp1}
; go back to the tagged recipient, previously prompted for
; do something else. Finally go back to OMNIS by resetting
; recipient with no prompt
Use event recipient
```

Set export format

Reversible: NO   **Flag affected:**   NO

Parameters: Export format

Syntax: Set export format [*{export-format}*]

export-format is one of the following: *Delimited (commas),*
Delimited (tabs), One field per line, OMNIS data transfer

This command specifies the export format to be used with the current report. The *Set export format* command lets you to override the parameters stored in the report class. You should use it after selecting a report class.

If you leave the name empty, the report is printed without an export format. An error occurs if the name is not a valid export format name. The name specified for the command can contain square bracket notation.

Translation

Export format names are not tokenized and therefore are not understood by foreign language versions of OMNIS. To avoid this portability problem, you can always build a list of export formats and use the list to select a format (see Example 2 below).

```
; Example 1
Send to file
Set report name Export1
Set print file name {Output.TXT}
Set export format {Delimited (tabs)}
Print report

; Example 2
Set current list LEXP
Build export format list (Clear list)
Set export format {[LEXP(1,2)]}
; selects format on second line of column one: Delimited (tabs)
```

Set final line number

Reversible: NO   **Flag affected:**   YES

Parameters: Line number (can be a calculation)

Syntax: Set final line number [*{line-number}*]

This command explicitly sets the value of *LIST.$linecount* by specifying a line number or a calculation. OMNIS expands or contracts any list as necessary and maintains the value of the LIST.$linecount property as the last line number. If the number of lines in the list is less than the number set for LIST.$linecount, OMNIS adds empty lines to the end. If the number of lines is greater than LIST.$linecount, OMNIS shortens the list and reduces the memory needed by the list.

You can use *Set final line number* to speed up list handling by setting the final line number to shorten lists, for example. The list is effectively cleared of data when the line number parameter is left blank (or evaluates to zero).

```
Calculate LVAR1 as 0
Set current list LIST1
Define list {LVAR1}
Repeat
   Calculate LVAR1 as LVAR1+1
   Add line to list
Until LVAR1>100
Set final line number {50}
OK message {List has [LIST.$linecount] lines}
```

Set hostname

Reversible: YES   **Flag affected:**   YES

Parameters: Server host name

Syntax: Set hostname *{host-name}*

This command sets the name of the remote computer you wish to access, that is, the hostname. The content of the logon parameters set up by *Set hostname* is server-specific.

```
Set username {SA}
Set password {Lion}
Set hostname {Serve300}
Logon to host
```

Set import file name

Reversible: YES   **Flag affected:**   YES

Parameters: Import file name

Syntax: Set import file name *{import-file-name}*

This command specifies the name of the import file. The flag is set if the import file is successfully selected. You use the current import file in any subsequent *Import field from file* commands.

If you use *Set import file name* in a reversible block, the import file is closed when the method containing the reversible block terminates.

```
Set import file name {DATA.DB1}
Repeat
   Import field from file into CVAR1
Until CVAR1 = 'start_data'
Do method ImportData
Close import file
```

Set label width

Reversible: NO   **Flag affected:**   NO

Parameters: Measurement
                ☐ Measurement in cms (leave unchecked for inches)

Syntax: Set label width [*(Measurement in cms)*] *{number}*

This command specifies the width of the labels when printing labels. It overrides the value set in the report parameters dialog until the current report is next reset. The width is measured from the edge of one label to the corresponding edge of the next.

You can set up the vertical spacing between labels using *Set record spacing*.

```
Set report name RLABELS
Set labels across page {4}
Set record spacing {3}
Set repeat factor {2}   ;; two of each label
Set label width (Measurement in cms) {4.5}
; Default measurement is inches
Print report
```

or do it like this

```
Do $clib.$reports.MyReport.$labelwidth.$assign(4.5)
```

Set labels across page

Reversible: NO   **Flag affected:**   NO

Parameters: Number (of labels)

Syntax: Set labels across page *{number}*

This command specifies the number of labels across the page for label printing. It overrides the setting in the report parameters dialog for the current report class. The setting remains in force until the next *Set report name* command.

When labels are printed, the vertical spacing from the top of one label to the next is set up using the $recordspacing property or from a method using *Set record spacing*.

```
Set report name RLABELS
Set labels across page {4}
Set record spacing {3}
Set label width (Measurement in cms){4.5}
Print report
```

Set left margin

Reversible: NO   **Flag affected:**   NO

Parameters: Measurement
            ☐ Measurement in cms (leave unchecked for
            inches)

Syntax: Set left margin [*(Measurement in cms)*] *{number}*

This command specifies the left margin for the current report class. It overrides the left margin setting in the report properties until such time as the current report is reset.

```
Set report name Rorders
Yes/No message {Print on A4 paper?}
If flag true
   Set bottom margin (Measurement in cms) {2.34}
   Set top margin (Measurement in cms) {1.2}
   Set left margin (Measurement in cms) {1.2}
   Set right margin (Measurement in cms) {1.2}
Else
   Set bottom margin {0.5}
   Set top margin {0.5}
   Set left margin {0.5}
   Set right margin {0.5}
   ; Default measurement is inches
End If
Print report
```

or do it like this

```
Do $clib.$reports.MyReport.$leftmargin.$assign(0.5)
```

Set lines per page

Reversible: NO   **Flag affected:**   NO

Parameters: Number (of lines per page)
            ☐ Send form feed

Syntax: Set lines per page [*(Send form feed)*] *{number}*

This command changes the number of lines per page for reports printed to file or port. You can send any report to a port or file using the Report destination dialog. When the destination is selected in this window, the number of lines is automatically set to the default number for the destination, so you must use *Set lines per page* after you have selected the report destination. The default lines per page setting is stored in the configuration file.

The **Send form feed** option lets you send a form feed character at the end of each page of the report; otherwise, multiple line feeds are sent.

```
Set report name RTEXTOUT
Send to port
Set lines per page (Send form feed) {66}
Print report
```

Set main file

Reversible: YES   **Flag affected:**   NO

Parameters: File class name

Syntax: Set main file *{file-name}*

This command selects the "main file" class. *Set main file* is an essential command which you must execute before manipulating any data. You can insert or delete data *only* in the file designated as the main file. The designated file cannot be memory-only or closed.

The main file setting also determines which connected files are located when finding records with Find/Next/Previous, and which connections are updated. As each main file record is read, the connected records are automatically read in and made available for editing. When the main file is edited or inserted, all connections to its parent files are updated, unless the parent file is closed.

If OMNIS attempts to execute a command which requires a main file before the main file is set, an error occurs. If the data file is not opened when the main file is set, OMNIS will try to open the default data file and, if this is unsuccessful, will display the Change data file dialog box so that the user can select or create a data file.

Changing the main file after a *Prepare for...* command does not cancel Prepare for mode. When an update is encountered, the main file set at the time of the last Prepare for is used. (See *Prepare for edit, Prepare for insert*.)

If you use *Set main file* in a reversible block, the main file is reset to its previous value when the method containing the reversible block finishes.

Multiple open data files

If more than one data file is open, there is only one main file setting shared by all open data files. If you do not qualify a file class name with a data file, the current data file is assumed unless you have created an association between the file class and another data file using the *Set default data file* command.

```
; Stock_control
If COST > PRICE
   Set main file {FSTOCK1}
   ; ... process FSTOCK1
Else If COST < PRICE
   Set main file {FSTOCK2}
   ; ... process FSTOCK2
End If
Set main file {FMAIN}
```

This example uses a reversible block to return the main file to its former setting after the method terminates

```
Begin reversible block
   Set main file {FMINV}
End reversible block

Do method InsertInvoices
;  Now quit method and put main file back
```

This changes main file after the Prepare for... command:

```
Set main file {FSTOCK1}
; FSTOCK1 is cleared
Prepare for insert
Set main file {FSTOCK2}
Enter data
Update files
; Record is inserted into FSTOCK1
; All read/write files in CRB are updated and
; parent connections to FSTOCK1 are updated.
```

Set memory-only files

Reversible:	YES	**Flag affected:**	YES
Parameters:	File or list of files		
Syntax:	Set memory-only files *{file1*[,*file2*]...*}*		

This command sets the file mode of the specified file(s), other than the main file, to memory-only. You can use the fields from a memory-only file as global variables. To do this:

1. Create a file class with some fields of the required type (Character, Numeric, and so on).

2. Designate the file class as a memory-only file using this command.

3. Use the fields in your methods as temporary storage for data.

When a memory-only file is changed to read/write, its fields are not cleared from the current record buffer. Similarly, when a file is changed from read/write to memory-only, its records are not cleared. Memory-only fields are initialized as empty when the library is launched.

If used in a reversible block, *Set memory-only files* is reversed when the method containing the block finishes. This command does not clear the Prepare for update mode.

In the method editor, a list of files is displayed. You can Ctrl/Cmnd-click on the file names to select multiple names.

```
Set memory-only files {fGlobals}
```

Set OMNIS window title

Reversible: YES   **Flag affected:**   NO

Parameters: Text (window title)

Syntax: Set OMNIS window title *{text}*

This command changes the title on the OMNIS application window (available under Windows only). The *text* parameter provides the new title which may contain square bracket notation. Unless reversed as part of a reversible block, the new title will remain until OMNIS is restarted.

```
Begin reversible block
   Set OMNIS window title {Call Tracker}  ;; for Windows only
End reversible block
Install menu MCalls
```

Set page width

Reversible: NO   **Flag affected:**   NO

Parameters: Number (of characters across page)

Syntax: Set page width *{number}*

This command changes the width of reports printed to file or port. The default setting is stored in the preferences file and is selected automatically when the destination is chosen. *Set page width* overrides this setting and must be used after selecting the report destination.

```
Set report name ROUTEXT
Send to file
Set print file name {OUT.TXT}
Set lines per page {66}
Set page width {45}
Print report
```

Set palette when drawing

Reversible: NO   **Flag affected:**   NO

Parameters: ☐ Color shared pictures
                   ☐ Never

Syntax: Set palette when drawing [([*Color shared pictures*][,*Never*])]

This command controls the color drawing system used to display certain types of color bitmap. When drawing pictures which contain more colors than can be accurately rendered by the display adapter, you can adjust the system's palette of available colors to match the palette stored inside the picture data. This will affect all colors used on the screen, not just OMNIS. This command lets you selectively turn on this option for shared pictures.

```
Set palette when drawing (Color shared pictures)
Open window instance WPICTS
Set palette when drawing (Never)
Open window instance WNOPICTS
```

Set password

Reversible: NO   **Flag affected:** YES

Parameters: Server password

Syntax: Set password *{password}*

This command sets the password of the remote database server. This password should not be confused with any passwords required by the file server or operating system software. A simple logon sequence for a local ORACLE database is:

```
Start session {ORACLE}
If flag true
   Set username {Scott}
   Set password {Tiger}
   Logon to host
   If flag false
      OK message {Error logging on: [sys(132)]}
   End If
Else
   OK message {Can't start ORACLE}
End If
```

Set port name

Reversible: NO   **Flag affected:** YES

Parameters: Port name (COMn: or LPTn:)

Syntax: Set port name *{port-name}*

This command specifies the name of the port to be used with subsequent input or output via the port. The flag is set if the port is successfully selected. The command should follow *Send to port.* You can set the baud rate and other parameters for the port using *Set port parameters.*

Set port name is not reversible, but if you use it in a reversible block the specified port is closed when the method terminates.

```
Set report name RPORT
Send to port
Switch sys(6)='M'
  Case kTrue
    Set port name {1 (Modem port)}
  Default      ;; if Windows
    Set port name {COM1:}
End Switch
Set port parameters {1200,n,7,2}
Print report
```

Set port parameters

Reversible: NO   **Flag affected:** YES

Parameters: ☐ Convert for ImageWriter (this parameter MacOS only)
Baud rate, Parity, Data bits, Stop bits,
X or H (XON/XOFF protocol or Hardware handshake), CPI,
LPI; include a comma for X|H and/or CPI parameters when not specifying a value; see examples

Syntax: Set port parameters [*(Convert for ImageWriter)*] *{baud-rate, parity, data-bits, stop-bits*[,*X|H*][,*cpi*][,*lpi*]*}*

This command sets the serial port parameters. When you use *Select port* in a method, the baud rate and other parameters are set to the Control panel settings. If you need to change the settings you can do so with this command, which should follow a *Send to port.* The flag is set if the command is successful.

For a baud rate of 9600, no parity, eight data bits and 1 stop bit, the command is:

Set port parameters {9600,n,8,1}

The fifth character in the parameter string can be 'X' (for XON/XOFF protocol) or 'H' for hardware handshake. The 'H'/'X' can be in upper or lower case.

The CPI and LPI parameters are numbers which specify characters and lines per inch. These are used by OMNIS to justify fields in the report - not sent as control characters to the printer.

Under MacOS, you use the **Convert for imagewriter** option to insert control codes suitable for an Apple ImageWriter. On the PC, you use **Output translation** in conjunction with the .INI file settings to convert characters with ASCII codes greater than 128 into combinations of backspace and other characters suitable for simple output devices with limited or differing character sets.

```
; example 1
Set port parameters {9600,n,8,1,,10,6}
; extra comma indicates no change to the Handshake parameters (X/H)
Set port parameters {9600,n,7,1,X}
; Sets up XON/XOFF handshake protocol

; example 2
Set report name RPORT
Send to port
Switch sys(6)='M'
  Case kTrue
    Set port name {1 (Modem port)}
  Default       ;; if Windows
    Set port name {COM1:}
End Switch
Set port parameters {1200,n,7,2}
Print report
```

Set print or export file name

Reversible: YES   **Flag affected:**   YES

Parameters: File name (full path can be specified)

Syntax: Set print or export file name *{print-file-name}*

This command specifies the print file name to which printed output is to be directed. The flag is set if the print file is successfully selected. If you use *Set print or export file name* in a reversible block, the print file is closed when the method containing the reversible block terminates.

Once the file name has been specified, *Send to file* directs the report output to the file. As each report is printed, its output is added to the end of the last report in the file.

```
If sys(6)='M'
   Set print or export file name {HD80:Work:Output file2}
Else
   Set print or export file name {C:\work\output2.prn}
End If
Send to file
Set report name r_addresses
Print report
```

Set publisher options

Reversible:	YES	**Flag affected:**	YES

Parameters: ☐ Publish on save
                    File or field list

Syntax: Set publisher options [*(Publish on save)*] *{file|field1*[,file|field2]...*}*

This command sets up the conditions under which the editions for the published fields in the list are updated. The **Publish on save** option causes the field values to be published when the current record buffer values for the fields are changed. For list fields, the value is published when the evAfter message is sent to the field.

Set publisher options alters the publisher options for all the published fields in the list. The field list can take a file name (for all fields in a file) or a range of fields, which includes a range of fields in the order listed in the Field names window. If no field list is given, the command operates on all published fields (in the library).

The flag is set if the command alters the options for one or more fields successfully. The flag is cleared if not running under System 7. If placed within a reversible block, the options are returned to their former status when the method terminates.

```
Publish field CNAME {HD80:Public:Sales-Name}
Publish field CTOTAL {HD80:Public:Sales-Total}
Set publish options (Publish on save) {CNAME,CTOTAL}
Prepare for edit
Enter data
Update files if flag set
```

Set read-only files

Reversible: YES   **Flag affected:** YES

Parameters: File or list of files

Syntax: Set read-only files *{file1*[,*file2*]...*}*

This command sets the file mode of the specified file(s) to read-only. You can read but not write to a read-only file. *Set read-only files* does not cancel the Prepare for update mode.

If you use this command in a reversible block, the file reverts to its original mode when the method containing the command block terminates.

In multi-user systems, you use *Set read-only files* to prevent OMNIS from locking certain files. When you make files read/write, they are locked and re-read. In multi-user systems, records such as invoice numbers and totals, accessed by a number of users, should be made read-only to prevent delays caused by record locking. You must return the file to read/write status momentarily while it is updated.

In the method editor, a list of files is displayed. You can Ctrl/Cmnd-click on the file names to select multiple names.

```
Set read-only files {FCONSTANTS,FSUPPLIERS}
Set main file {FORDERS}
Prepare for insert
Enter data
Update files if flag set
```

Set read/write files

Reversible: YES   **Flag affected:** YES

Parameters: File or list of files

Syntax: Set read/write files *{file1*[,*file2*]...*}*

This command sets the file mode of the specified file(s) to read/write. The read/write file mode is the default type of OMNIS file; you can read and write data to a read/write file. The other three file modes are read-only, closed and memory-only. If a file is changed to read/write mode when in Prepare for update, the data for the file class is reread from disk. In multi-user systems, read/write files are locked when a *Prepare for...* command is executed.

The file mode will revert to its former state if you use the command in a reversible block.

In the method editor, a list of files is displayed. You can Ctrl/Cmnd-click on the file names to select multiple names.

```
; NewInvoice
Set read-only files {fInvNumber}
Prepare for insert
Enter data
If flag true
   Set read/write files {fInvNumber}
   ; waits for and locks fInvNumber file
   Calculate InvNo as cInvNum+1
   Calculate cInvNum as InvNo
   Update files    ;; update both Invoice and Constants file
End If
Redraw wInvoice
```

Set record spacing

Reversible: NO   **Flag affected:**   NO

Parameters: Measurement (number)
                ☐ Measurement in cms (leave unchecked for inches)

Syntax: Set record spacing [*(Measurement in cms)*] *{number}*

This command specifies the line spacing for the record section of the current report class. It overrides the setting in the record section properties for the current report. The setting remains in force until the next *Set report name*.

```
Set report name RLABELS
Set labels across page {3}
Set record spacing (Measurement in cms) {5.2} ;; Default is inches
Print report
```

or do it like this

```
Do $clib.$reports.MyReport.$recordspacing.$assign(5.2)
```

Set reference

Reversible: NO   **Flag affected:**   NO

Parameters: Variable name (of type Item reference)

                     Notation for an item (can be a calculation)

Syntax: Set reference ***variable-name*** to ***notation***

This command sets up and stores a reference to an item in a reference variable. It assigns an *alias* for an item of notation that you do not want to type each time the item is referenced in the code. The variable can be a local, class, or task variable of type Item reference.

```
; Declare class variable FREF of type Item reference
Set reference FREF to LIBRARY1.$windows.SALESWINDOW.$objs.TOTAL
; now set the color of the TOTAL field
Do FREF.$forecolor.$assign(6)   ;; set the color of the TOTAL field
```

Set repeat factor

Reversible: NO   **Flag affected:**   NO

Parameters: Number (that is, the repeat factor)

Syntax: Set repeat factor [*{number}*]

This command specifies the number of copies of the record section to be printed. It overrides the repeat factor specified in the report properties for the current report. *Set repeat factor* is particularly useful when printing multiple labels. The setting remains in force until the next *Set report name*. If the repeat factor is left blank (or evaluates to zero), the printing of the record sections of a report is suppressed completely; all heading sections, totals and subtotals are still calculated correctly.

```
Set report main file {FLABELS}
Set report name RLABELS
Set labels across page {3}
Set repeat factor {2}
Set label width {3.4}
Print report
```

or do it like this

```
Do $clib.$reports.MyReport.$repeatfactor.$assign(2)
```

Set report main file

Reversible: NO   **Flag affected:** NO

Parameters: File name

Syntax: Set report main file *{file-name}*

This command specifies the main file for the current report. When a report is printed, OMNIS uses the main file set by the last *Set main file*. *Set report main file* overrides the main file setting by specifying a new main file specifically for the report. The setting remains in force until the next *Set report name*.

Printing connected files

When printing connected files, it is essential that the child file is made the main file. Only the main file and its connected parent files are automatically read into the current record buffer.

If no sort fields are specified in the report class, the report generator steps through the records in the order defined by the record sequencing number for the main file. Sort fields let you reorder the report records.

```
Set report name RORDERS
Set report main file {FORDERS}
Clear sort fields
Set sort field ORD_CODE
Prompt for destination
Print report
```

or do it like this

```
Do $clib.$reports.MyReport.$mainfile.$assign(FORDERS)
```

Set report main list

Reversible: NO   **Flag affected:** NO

Parameters: List or row name

Syntax: Set report main list *list-name*

This command specifies a list as the source for the data for the current report. When a report is printed, OMNIS uses the main file specified either in $mainfile or the file set by the last *Set main file* command. *Set report main list* lets you override the main file setting by specifying a list, from which data is read for the next printed report.

A list-based report prints one record for each line in the list. The data file is not used unless the report contains auto find fields. Sorting, searching, subtotals, and so on, continue to work the same way as for file-based reports. All field values are taken from the list and records are read in list order.

When a *Prepare for print* command is encountered, the current list or file setting overrides the Main file setting used in the report parameters dialog.

```
Set report name RLNAMES
Set report main list LIST1
Prompt for destination
Print report
```

or do it like this

```
Do $clib.$reports.MyReport.$mainlist.$assign(LIST1)
```

Set report name

Reversible: YES   **Flag affected:**   NO

Parameters: Report name

Syntax: Set report name *report-name*

This command selects a report class for use with subsequent *Print...* commands. It terminates any report in progress.

If you use *Set report name* in a reversible block, the previous report name will be restored when the method terminates.

```
Set report name RLABEL
Set sort field CXTITLE (Upper case)
Print report
```

Set right margin

Reversible: NO   **Flag affected:**   NO

Parameters: Measurement
                   ☐ Measurement in cms (leave unchecked for
                   inches)

Syntax: Set right margin [*(Measurement in cms)*] *{number}*

This command specifies the right margin for the current report class. It overrides the right
margin setting in the report properties until such time as the current report is reset.

```
Set report name Rorders
Yes/No message {Print on A4 paper?}
If flag true
   Set bottom margin (Measurement in cms) {2.34}
   Set top margin (Measurement in cms) {1.2}
   Set left margin (Measurement in cms) {1.2}
   Set right margin (Measurement in cms) {1.2}
Else
   Set bottom margin {0.5}
   Set top margin {0.5}
   Set left margin {0.5}
   Set right margin {0.5}
   ; Default measurement is inches
End If
Print report
```

or do it like this

```
Do $clib.$reports.MyReport.$rightmargin.$assign(0.5)
```

Set search as calculation

Reversible: NO   **Flag affected:**   NO

Parameters: Calculation

Syntax: Set search as calculation [*{calculation}*]

This command sets the current search as the single line calculation specified. The
calculation replaces the current search class if one has been set. A subsequent report, *Search
list* or a *Find* command with **Use search** will use the search calculation.

Search calculations allow the index optimization routine in OMNIS to select a suitable
index, provided that such an index is available. Leaving the calculation blank has the effect
of clearing the previous search calculation.

```
Set main file {f_client}
Open window instance w_Address
Set search as calculation {SURNAME = 'Smith'}
Find on TOWN {'London'}(Exact match,Use search)
; Uses TOWN index, locates Londoners, and uses search to locate
; Smiths.  Exact match applies to the 'London' match
Redraw w_Address
```

This example moves selected lines only between lists.

```
Set current list LIST2
Set search as calculation {#LSEL}
Merge list LIST1 (Use search)
Redraw lists
```

Set search name

Reversible:	YES	**Flag affected:**	NO

Parameters: Search class name

Syntax: Set search name [*{search-name}*]

This command sets the search class to be used with reports, *Search list* and *Find* (using search) commands. If no search class name is included, the current search is cleared. Search classes allow subsets of the records to be printed or worked on.

A *Find first (Use search)* command reads in the first record which matches the current search criterion and creates a find table. Subsequent *Next* commands print out the records in the table.

If used within a reversible block, the search name reverts to its former setting when the method terminates.

```
; example 1
Set search name S_Area1
Set report name R_list
Print report (Use search)

; example 2
Set search name S_Area2
Set main file {FORDERS}
Clear main & connected
Prepare for print
Find first (Use search)   ;;  Creates table of records which match
While flag true
   Print record
   Next
End While
End print
```

Set server mode

Reversible:	YES	**Flag affected:**	NO

Parameters:	☐ Field requests
	☐ Field values
	☐ Advise requests
	☐ Commands

Syntax: Set server mode [*([Field requests][,Field values]*
                     [,Advise requests][,Commands])]

This command sets OMNIS to act as a DDE server and specifies which DDE commands it will accept. With one or more of the check box options selected OMNIS will respond to the corresponding commands and demands from a client. If none is selected, server mode is deselected.

All four server mode check box options have equivalent DDE commands which are described separately: *Accept field requests, Accept field values, Accept advise requests* and *Accept commands*.

Irrespective of the mode selected, OMNIS will only accept field values and commands when in enter data mode, and accept commands when no methods are running.

OMNIS will only respond to a request to act as a server if the Initiate message from the client contains at least the name of the program, that is, OMNIS. If the client specifies a topic, it has to be equal to the OMNIS library name without the .LBR extension. OMNIS responds with the current library name if the client does not specify the topic.

If no options are set, OMNIS is disabled as a server except for the System Topic. If OMNIS is already a server when the options under *Set server mode* are disabled, one of two things will happen:

1. If the options have been disabled during a reversible block, the client sending the Initiate message will get busy acknowledgments until the reversible command method finishes. You cannot initiate any new conversations during this time.

2. OMNIS will end the communication by sending the client a Terminate message.

All four server mode options have equivalent commands which are described separately: *Accept field requests, Accept field values, Accept advise requests* and *Accept commands*.

Set sort field

Reversible: YES   **Flag affected:**   NO

Parameters: Field name
                  ☐ Descending
                  ☐ Upper case
                  ☐ Subtotals
                  ☐ New page

Syntax: Set sort field *field-name* [*([Descending]*
                      [,*Upper case*][,*Subtotals*][,*New page*]*)*]

This command specifies a field on which a list or report is to be sorted. The report generator systematically works through the records in the main and connected files and prints them using the report class definition. You can use sort fields to sort the records into a specific index order.

A report can be sorted on up to nine fields: you can specify sort fields in the report class or by using *Set sort field*. Since sort fields are cumulative, use *Clear sort fields* first to clear any that already exist.

When a report name is selected, the report class sort fields are used but you can override these sort fields by clearing them and specifying new sort ones with *Set sort field*. For nine sort fields, you use the *Set sort field* command nine times in succession. Using this method, however, can be slower than sorting on fields that are already indexed.

You can set the sort fields for lists using *Set sort field*. The *Sort list* command sorts the current list in the order specified by the current sort fields. Note that lists have to be explicitly redrawn before you can view the results of a sort.

If used within a reversible block, the sort field setting reverts when the method terminates.

```
; to sort a report on fields AREA, DEPT and NAME
Set report name RCOMMISSION
Clear sort fields
Set sort field {AREA}
Set sort field {DEPT}
Set sort field {NAME}
Print report
```

The **Descending** option sorts the records in descending order. The **Upper Case** option converts lower case characters to upper case for the purpose of sorting. The **Subtotals** option causes the Subtotal section in the report to be printed when the value of the sort field changes. Thus, in the above example, when AREA changes, subtotals 1 is printed, when DEPT changes, subtotals 2 is printed, and so on. The **New Page** option starts a new page when the field value changes.

Set SQL blob preferences

Reversible:	YES	**Flag affected:**	NO

Parameters: Default, Load all, Segment, or Threshold
Chunk size/Threshold

Syntax: Set SQL blob preferences *(Default|Load all|Segment|Threshold)*
        {chunk-size/threshold}

This command controls the way pictures, large strings, and BLOBs (Binary Large Objects) are read across the DAM interface. The capabilities of each DAM dictate how the blob is handled. *Set SQL blob preferences* sets the preferences for the current cursor.

Blob buffering is the ability of the API to bring a blob to or from the server in "chunks". Although OMNIS and many servers support blob sizes of up to 2 gigabytes, the size that may be totally buffered in memory is limited. This command lets you pass a blob as a series of smaller chunks. You can split strings, pictures, and binaries over 256 bytes into chunks.

Set SQL blob preferences has four options: Default, Load all, Segment, or Threshold.

With the **Default** option the DAM deals with splitting the blob.

With the **Load all** option the blob is passed as one chunk. The DAM currently attached may not be able to pass chunks of that size.

With the **Segment** option you specify the chunk size.

With the **Threshold** option you specify the threshold/chunk size.

Set SQL script

Reversible: NO   **Flag affected:**   YES

Parameters: Field name or variable

Syntax: Set SQL script *{field-name/variable}*

This command takes a string held in a specified field and loads it directly into the SQL statement buffer. The field can be any OMNIS character field or variable. The field value can include square bracket notation and indirect square bracket notation. Carriage returns are converted to spaces before being sent to the remote database. *Set SQL script* will clear the flag and leave the SQL buffer unaltered if the text contains square bracket notation with invalid field names or calculations.

You should use the command with care since, apart from evaluating square bracket notation, this process bypasses the normal syntax checking carried out when using *SQL:* or *Perform SQL*.

```
Calculate CVAR1 as 'Insert publishers (pub_name,pub_id)
                    values (@[NAME],@[ID])'
Set SQL script {CVAR1}
If flag false
   OK message {Error loading SQL buffer with [CVAR1]}
   Quit method
End If
Execute SQL script
; Handle errors from server
```

Set SQL separators

Reversible: YES   **Flag affected:**   NO

Parameters: Thousand and/or Decimal separator type

Syntax: Set SQL separators *{[thousand-separator][/decimal-separator]}*

This command sets the thousand and decimal separators for numbers that are sent to a remote server. You can set the separators in this way for each session. You must use different characters for each separator type. The separators revert back to their default when the session is closed.

To enter a decimal separator only you should use the syntax */decimal-separator*, that is, include the forward slash.

Most SQL servers use English/American numeric separators, that is, commas representing the thousand separator and a period representing the decimal separator. When using European numeric separators, that is, periods representing the thousand separator and a comma representing the decimal separator, there would be a mismatch between numbers

you send to the server and what the server expects. This command lets you set the SQL separators and remedy the mismatch.

```
Set SQL separators {,/.}
```

```
; set to comma/period which English/American server expects
```

Set subscriber options

Reversible: YES   **Flag affected:**   YES

Parameters: ☐ Subscribe automatically
                  File or field list

Syntax: Set subscriber options [*(Subscribe automatically)*]
                  [*{file|field1*[,*file|field2*]...*}*]

This command controls whether the subscribed fields in the list are to be automatically updated. When **Subscribe automatically** has been selected, the values are set up when the library starts up and updated whenever OMNIS is notified that the values have been changed. The new values are not made available to the library while there is a design window open as the top window. When a subscriber is updated, the evSent message is sent to any window or library $control() methods.

The command alters the subscriber options for all the subscribed fields in the list. The field list can take a file name (for all fields in a file) or a range of fields, which includes a range of fields in the order listed in the Field names window. If no field list is given, the command operates on all subscribed fields within the library.

The flag is set if the command alters the options for one or more fields successfully. If placed within a reversible block, the options are returned to their former status when the command is reversed.

```
Subscribe field CNAME {HD80:Public:Sales-Name}
Subscribe field CTOTAL {HD80:Public:Sales-Total}
Set subscriber options (Subscribe automatically) {CNAME,CTOTAL}
Enter data
Set subscriber options {CNAME,CTOTAL}
```

Set timer method

Reversible: YES   **Flag affected:**   NO

Parameters: Interval in seconds (must be an integer, e.g. 300 sec)
                  Code class name
                  Method name

Syntax: Set timer method *interval* sec [*code-class-name*/] [*{method-name}*]

This command calls the specified method at regular intervals while waiting for a keyboard input; the called method should preferably be one contained in a code class. You could use this command for automatic telephone dialing, regular checks for electronic mail, and so on.

The command specifies the timer method and the interval in seconds between calls to the timer method. This interval can be between 1 and 30,000 in the form "n sec" where n is the number of seconds. OMNIS will start the next timer method when the method which is currently executing, finishes. Timer methods cannot operate in real time as OMNIS will not execute a timer method while another method is running or when an OK or Yes/No message is displayed on the screen.

The timer method in your code class should not contain a *Quit all methods* as this will terminate any *Enter data* commands which are running. You can also use an *Enter data* inside a timer method: if so and you do not clear the timer method, the timer method continues to be active while OMNIS carries out the Enter data part of the timer method.

You can use *Set timer method* in a reversible block, in which case the timer method is cleared when the executing method terminates.

```
; a menu method
Set timer method 60 sec CODECLASS/Timer
OK message {Now play the minute waltz!}

; Timer method in CODECLASS
OK message {Timer method triggered once only}
Clear timer method
```

Set top margin

Reversible:	NO	**Flag affected:**	NO

Parameters: Measurement
☐ Measurement in cms (leave unchecked for inches)

Syntax: Set top margin [*(Measurement in cms)*] *{number}*

This command specifies the top margin for the current report class. It overrides $topmargin until such time as the current report is reset.

```
Set report name Rorders
Yes/No message {Print on metric A4 paper?}
If flag true
   Set bottom margin (Measurement in cms) {2.34}
   Set top margin (Measurement in cms) {1.2}
Else
   Set bottom margin {1.0}
   Set top margin {1.0}
   ; Default measurement is inches
End If
Print report
```

or do it like this

```
Do $clib.$reports.MyReport.$topmargin.$assign(1.0)
```

Set top window title

Reversible:	YES	**Flag affected:**	NO

Parameters: Window title

Syntax: Set top window title [*{window-title}*]

This command specifies the title for the top window instance. You can use square bracket notation within the window title. The title of the top window instance is cleared if you omit the window title parameter (or it evaluates to an empty string). The title reverts to the normal title if the window instance is closed and reopened. An error occurs if there is no window instance.

If you use *Set top window title* in a reversible block, the title reverts to its normal value when the method containing the reversible block terminates.

```
Open window instance WACCOUNTS/wacc1
Yes/No message {Do journals?}
If flag true
   Set top window title {'Journals for [#D]'}
Else
   Set top window title {'Invoices'}
End If
```

Set transaction mode

Reversible: YES   **Flag affected:** NO

Parameters: Automatic, Generic, or Server

Syntax: Set transaction mode *(Automatic|Generic|Server)*

This command manages SQL transactions and controls when a transaction is committed to the DBMS. *Set transaction mode* has three options: Automatic, Generic, and Server.

The **Automatic** option commits or rolls back each SQL statement automatically as soon as the next SQL command starts (*Begin SQL script*, *Perform SQL*, *Reset cursor*) or if the statement either did not generate a result set immediately after execution, or failed.

The **Generic** option implements basic transaction processing using *Commit current session* and *Rollback current session.*

With the **Server** option the DAM in use takes complete control and uses the server SQL dialect.

For more information on this command and its relationship to *Autocommit*, refer to *OMNIS Studio Data Access Manager.*

```
If fDestType = 'SYBASE'
   Set transaction mode (Server)      ;; (autocommit off)
   Perform SQL {BEGIN TRAN}
Else                    ;; Oracle and OMNISSQL
   Reset cursor(s) (Session)
   Set transaction mode (Generic)      ;; (autocommit off)
End If
```

Set username

Reversible: NO   **Flag affected:**   YES

Parameters: Server user name

Syntax: Set username *{user-name}*

This command sets the username for logging onto a remote database server. A simple logon sequence for a local ORACLE database is:

```
Start session {ORACLE}
If flag true
   Set username {Scott}
   Set password {Tiger}
   Logon to host
   If flag false
      OK message {Error logging on: [sys(132)]}
   End If
Else
   OK message {Can't start ORACLE}
End If
```

The format for the username string when using a remote ORACLE server is:

```
username/password@driver_prefix:database_string
```

Show 'About...' window

Reversible: NO   **Flag affected:**   NO

Parameters: None

Syntax: Show 'About...' window

This command displays the standard "About..." window which is available as an option in the **Help** menu under Windows, or the **Apple** menu under MacOS. You can change the standard "About..." screen with the *Set 'About...' method* command. For example

```
Show 'About...' window
; Now redefine the 'About...' method
Set 'About...' method HELP/About this demo
; Now the only way to see the original 'About'
; is to execute another Show 'About...' window
```

Show fields

Reversible: YES **Flag affected:** NO

Parameters: Field name or list of field names

Syntax: Show fields *{field1*[*,field2,...*]*}*

This command shows the specified window field or list of fields. You can hide fields with *Hide fields* or using the notation. Inactive pushbuttons with the **Do not gray** attribute cannot be made visible with this or any other command.

If you use *Show fields* in a reversible block, the specified fields are hidden when the method containing the reversible block terminates.

```
Hide fields { EntryId,EntryCompany,EntryTel }
Redraw CustWindow      ;; Fields are hidden
If ACCESS < 3
   Show fields { EntryId,EntryCompany,EntryTel }
   Redraw CustWindow
End If
```

To show a single field on the current window

```
Do $cwind.$objs.FieldName.$visible.$assign(kTrue)
```

or to show all fields on the current window

```
Do $cwind.$objs.$sendall($ref.$visible.$assign(kTrue))
```

Show OMNIS maximized

Reversible: NO **Flag affected:** NO

Parameters: None

Syntax: Show OMNIS maximized

This command shows OMNIS at its maximum size within the application window. This command performs the same action as the **Maximize** option in the **System** menu and the Maximize button on the application window.

```
OK message {Printing now}
Show OMNIS minimized
Print report
Show OMNIS maximized
```

Show OMNIS minimized

Reversible: NO   **Flag affected:**   NO

Parameters: None

Syntax: Show OMNIS minimized

This command minimizes OMNIS which subsequently appears as an icon at the bottom of the screen.

```
OK message {Printing now}
Show OMNIS minimized
Print report
Show OMNIS maximized
```

Show OMNIS normal

Reversible: NO   **Flag affected:**   NO

Parameters: None

Syntax: Show OMNIS normal

This command shows OMNIS at its normal size within the application window. Icons for other applications are visible along the bottom of the screen.

```
OK message {Printing now}
Show OMNIS minimized
Print report
Show OMNIS normal
```

Show docking area

Reversible: NO   **Flag affected:**   NO

Parameters: ☐ Show text
                  Toolbar name (a constant)

Syntax: Show docking area [*(Show text)*] **toolbar-name**

This command opens the top, bottom, left, or right docking area into which toolbars may be installed. The docking area is specified using one of the constants: kDockingAreaTop, kDockingAreaBottom, kDockingAreaLeft, kDockingAreaRight or kDockingAreaFloating.

When a toolbar is created each control may have a text label, for example, a Print button may have the word "Print" associated with it. The **Show text** option allows these text labels to be shown beneath the buttons.

```
Show docking area {kDockingAreaTop}
Install Toolbar {T_New}   ;; toolbar installed on Top docking area
```

Alternatively you can use

```
Do $root.$prefs.$dockingarea.$assign(kDockingAreaTop)
```

Signal error

Reversible: NO   **Flag affected:**   NO

Parameters: Error number
                  Error text

Syntax: Signal error *{error-number[,error-text]}*

This command reports a fatal error which can be either a user-defined error or a built-in OMNIS error. A fatal error is any error that normally halts method execution and reports an error (for example, syntax error, or an out of memory error).

The fatal error is reported with the specified error code and text. Any error handler for that code will be invoked. If there is no error handler or the error handler does not make a set error action (SEA), the debugger is invoked, if available. Otherwise, execution halts with the error message.

This command is useful for trapping user-defined errors, and is a convenient tool for triggering an error situation inside OMNIS for whatever condition you may want to specify.

```
Test for only one user
If flag false
   Signal error (99,'Test for one user failed')
End If
```

Single file find

Reversible: YES   **Flag affected:**   YES

Parameters: Field name
Calculation
☐ Exact match

Syntax: Single file find on ***field-name*** [*(Exact match)*] [*{calculation}*]

This command locates a record in a single file only. It is similar to the standard *Find* command but is not dependent on the main file; that is, the field used in *Single file find* does not have to belong to the main file and it does *not* read in the connected records. You can specify a calculation for *Single file find* which determines the value used in the Find. The **Exact match** option with a blank calculation indicates that the command is to be executed using the current value of the field, that is, the file is searched for a record whose index value matches the current value of the specified field.

In multi-user systems, a *Single file find* while in Prepare for... mode causes additional semaphores to be set. If the record is already locked, the user must wait for access to the record.

```
Wait for semaphores
Single file find on PRICE {PRICE <= COST*3}
If flag false
   OK message {Can't find record}
End If
```

Sort list

Reversible: NO   **Flag affected:** YES

Parameters: None

Syntax: Sort list

This command sorts the current list in the order specified by the current sort fields. You can use *Set sort field* to set the sort fields. Note that lists have to be explicitly redrawn before you can view the results of a sort.

```
Set current list CUSTLIST
Define list {NAME,TOWN,CITY}
Set main file {FCUST}
Build list from file (Use search)
Clear sort fields
Set sort field NAME
Set sort field TOWN
Sort list
Redraw lists
; Note, Build list can also use sort fields
```

Or do it like this

```
Do LIST.$sort(SortField1,SortOrder,SortField2,SortOrder, ..)
```

Sound bell

Reversible: NO   **Flag affected:** NO

Parameters: None

Syntax: Sound bell

This command sounds the system beep. You can sound the bell at any point in a method to draw attention to a particular method, field, message, error, and so on.

```
Sound bell
Open window instance FWARNING
Enter data
Close window FWARNING
```

SQL:

Reversible: NO   **Flag affected:**   YES

Parameters: SQL script

Syntax: SQL: sql-script

This command adds a line of script to the SQL buffer. It loads the SQL script buffer with lines of text ready to be sent to the remote database by a subsequent *Execute SQL script*.

The *Begin SQL script* and *Reset cursor(s)* commands clear the buffer ready for a new script. The *sql-script* parameter is a text field which can contain square bracket and indirect square bracket notation (which you use to send data to the server as a "bind" variable).

Each *SQL: sql-script* line is added to the current buffer with a carriage return delimiter. When the script is sent to the server, each carriage return is replaced by a space character. You can split a SQL statement over more than one line but literal values must not be split between lines. A line can contain more than one SQL statement provided you use the appropriate delimiter.

Text loaded into the buffer must be valid SQL script and must be understood by the server. You use square brackets to load the buffer with text obtained from OMNIS functions, variables and calculations. Indirect notation of the form @[Field] is not evaluated in OMNIS but is handled by the DAMs, and lets you pass field values to the server without the need for them to be included in the text of a SQL statement.

```
Set current session {SY_ONE}
Begin SQL script
SQL: Insert into [TABLE]
SQL: (col1,col2)
SQL: VALUES (@[FIELD1],@[FIELD2])
End SQL script
Execute SQL script
; Deal with errors now
```

Start program maximized

Reversible: NO   **Flag affected:** YES

Parameters: Program name
Document or file name (full pathname for document or file)

Syntax: Start program maximized *{program-name*[*document-name*]*}*

This command starts up a Windows application at its maximum screen size. The program name must be the program's module name which is usually, but not necessarily, the same as its executable file name. You can also specify the document or file name which must include the full path name and a space after program-name.

The flag is set if the program is found.

You can pass other parameters such as command line switches by including them after the document name.

Having run the program, OMNIS has no way of determining whether it is running except by initiating a DDE conversation with it.

```
Test if file exists {C:\winword\winword}
If flag true
   Start program maximized  {winword C:\winword\work\readme.txt}
End If
```

Start program minimized

Reversible: NO   **Flag affected:** YES

Parameters: Program name
Document or file name (full pathname for document or file)

Syntax: Start program minimized *{program-name*[*document-name*]*}*

This command starts up a Windows application as a minimized icon. The program name must be the program's module name which is usually, but not necessarily, the same as its executable file name. You can also specify the document or file name which must include the full path name and a space after program-name.

The flag is set if the program is found.

You can pass other parameters such as command line switches, by including them after the document name.

Having run the program, OMNIS has no way of determining whether it is running except by initiating a DDE conversation with it.

```
Test if file exists {C:\winword\winword}
If flag true
   Start program minimized  {winword C:\winword\work\readme.txt}
End If
```

Start program normal

Reversible: NO   **Flag affected:**   YES

Parameters: Program name
                      Document or file name (full pathname for document or file)

Syntax: Start program normal *{program-name*[*document-name*]*}*

This command starts up a Windows application at its normal screen size. The program name must be the program's module name which is usually, but not necessarily, the same as its executable file name. You can also specify the document or file name which must include the full path name and a space after program-name.

The flag is set if the program is found.

You can pass other parameters such as command line switches by including them after the document name.

Having run the program, OMNIS has no way of determining whether it is running except by initiating a DDE conversation with it.

```
Test if file exists {C:\winword\winword}
If flag true
   Start program normal  {winword C:\winword\work\readme.txt}
End If
```

Start session

Reversible: NO   **Flag affected:**   YES

Parameters: DAM name

Syntax: Start session *{DAM-name}*

This command loads the specified DAM and initializes communication between the current session and the remote database. It takes the *DAM-name* as the parameter. It is only necessary to supply this command once per server, or after *Quit session*.

All DAMs are placed in the EXTERNAL folder under the main OMNIS folder. All DAM names begin with the letter d. Under Windows DAMs have the .dll file extension, but you don't need to include it in the *Start session* command. For example under Windows, *Start session {dORACLE}* will cause OMNIS to look for dORACLE.DLL.

After a successful *Start session*, you can use *Set hostname, Set username, Set password,* and *Logon to host* to log on to your database.

```
; You need plenty of memory for these two...
Set current session {Server1}
Start session {dORACLE}
Set current session {Server2}
Start session {dSYBASE}
; now log on to each server
```

Subscribe field

Reversible: YES   **Flag affected:**   YES

Parameters: Field name
                  Edition name

Syntax: Subscribe field *field-name* [*{edition-name}*]

This command subscribes a field to the specified edition. When a field is subscribed in this way, its value is read from a file called an "edition". A full path can be given for the edition, that is, a specification for the volume and folder in which the edition is located. The volume can be another user's public folder or a network server. For example

```
If sys(113)   ;; that is, if Pubs and Subs available
   Subscribe field SALESTOTAL {Fred's Mac:Public Folder:OMNIS-FredsApp-
   Sales Total}
End If
```

If no edition name is given, the existing edition name is used, or if one does not already exist, the default "library name-field name" is used.

The flag is set if the field is already subscribed to that edition or if the field is successfully subscribed. If the field was formerly subscribed to a different edition, that subscription is canceled, the new subscription set up and the flag set. If the command is used within a reversible block, the edition is canceled when the command is reversed (but any former subscription is not recreated if the command canceled one).

When a field is newly subscribed, none of the Subscriber options are set, so a *Subscribe now* command must be used to update the field. If you want the edition to be updated automatically, the *Set subscriber options* command must be used.

```
Subscribe field CNAME {HD80:Public:Sales-Name}
Subscribe field CTOTAL {HD80:Public:Sales-Total}
Set subscriber options (Subscribe automatically) {CNAME,CTOTAL}
Enter data
Redraw windows
Cancel subscriber  {CNAME,CTOTAL}
```

Subscribe now

Reversible: NO   **Flag affected:**   YES

Parameters: File or field list

Syntax: Subscribe now [*{file|field1*[,*file|field2*]...*}*]

This command updates the fields in the parameter list if they have been subscribed. The field list can take a file name (for all fields in a file) or a range of fields, which includes a range of fields in the order listed in the Field names window. If no list is given, all subscriptions for the library are updated.

The flag is set if the command updates one or more fields successfully.

```
Subscribe field CNAME {HD80:Public:Sales-Name}
Subscribe field CTOTAL {HD80:Public:Sales-Total}
Enter data
Subscribe now {CNAME,CTOTAL}
```

Swap lists

Reversible: NO   **Flag affected:**   YES

Parameters: List or row name

Syntax: Swap lists *list-name*

This command swaps the definition and contents of the specified list with that of the current list and sets the flag. After this command, the current list contains the fields and data which were held in the specified list, and the specified list contains the fields and data which were in the current list.

This command cannot be used to copy lists. To do this use *Calculate LIST2 as LIST1*.

```
; declare local vars LIST1 & LIST2 of List type
Set current list LIST1
Set main file {FNUMBERS}
Define list {NUM1,NUM2,BOOL1}
Build list from file
Swap lists LIST2
; LIST2 now contains definition and data from LIST1 (current list)
; LIST1 is now empty
```

Swap selected and saved

Reversible: NO   **Flag affected:**   YES

Parameters: Line number (can be calculation, default is current line)
☐ All lines

Syntax: Swap selected and saved [*(All lines)*] [*{line-number}*]

This command swaps the Saved selection state and the Current selection state and sets the flag. To allow sophisticated manipulation of data via lists, a list can store two selection states for each line; the "Current" and the "Saved" selection. The Current and Saved selections have nothing to do with saving data on the disk; they are no more than labels for two sets of selections. The lists may be held in memory and never saved to disk: they will still have a Current and Saved selection state for each line but they will be lost if not saved. When a list is stored in the data file, both sets of selections are stored.

Swap selected and saved allows the Saved selection state of the specified line (or All lines) to be swapped with the Current set. You can specify a particular line in the list by entering either a number or a calculation. The **All lines** option swaps the selection status for all lines of the current list. The following example selects the middle line of the list:

```
Set current list LIST1
Define list {LVAR1}
Calculate LVAR1 as 1
Repeat
   Add line to list
   Calculate LVAR1 as LVAR1+1
Until LVAR1=6
Select list line(s) {3}
Save selection for line(s) (All lines)
Deselect list lines (All lines)
Swap selected and saved (All lines)
Redraw lists
```

Switch

Reversible: NO   **Flag affected:**   NO

Parameters: Expression or calculation

Syntax: Switch *expression*

This command initiates a Switch method construct. You use a Switch statement to select a course of action from a set of options based on the value of a variable, expression or calculation. It is similar to an *If–Else If* construct although the performance of a Switch construct tends to be faster.

The first line of the construction contains the *Switch* command. This defines the variable, expression or calculation on which the choice of action will depend. Following the *Switch* command, the *Case* commands provide values which, if matched with the expression supplied in the *Switch* line, cause the methods between case lines to be executed.

You can nest multiple Switch statements, and embed other conditional statements such as *If-Else* constructs.

The following example builds a dataformat list for a graphs application. It uses the graph major and minor types to build the correct list of data formats; the data formats are added to the list using *Add line to list*, but for brevity, some have been commented out.

```
; Build Datalist
; Declare parameter vars MajType, MinType and Dataformat
Set current list GraphDataformatList
Switch MajType
   Case kGraphPie      ;; MajType is Pie chart
      Add line to list {('Value',0)}
   Case kGraphSpecial      ;; MajType is Special
      Switch MinType
         Case kHistogram
            Add line to list {('Value',0)}
         Case kSpectralMap
            Add line to list {('Value',0)}
            Add line to list {('Value+Label',1)}
         Case kPolar
            Add line to list {('X+Y',1)}
         Case kHighLowOpenClose,kDualYHighLowOpenClose
            ; Add data format(s) to list...
         Case kContour
            ; Add data format(s) to list...
         Default      ;; MinType must be kScatter or kScatterDualY
            ; Add data format(s) to list...
      End Switch
   Case kGraph3D      ;; MajType is 3D
      Switch MinType
         Case k3DScatter
            ; Add data format(s) to list...
         Default      ;; any other 3D minor type
            ; Add data format(s) to list...
      End Switch
   Default      ;; MajType is kGraphArea, kGraphBars, or kGraphLines
      ; Add data format(s) to list...
End Switch
```

You can write Switch statements that contain other constructs such as *If-Else If* statements. Also note that if the Switch accepts one of a fixed number of possibilities and your method has a *Case* command for each possibility, your method does not need a Default statement. For example

```
Switch LV_Data
   Case kFalse      ;; LV_Data is a label
      If ...
         ; do this
      Else
         ; do this
      End If
   Case kTrue       ;; LV_Data is data
      If ...
         If ...
         Else ..
         End If
      Else
         If ...
         Else ..
         End If
      End If
End Switch
```

Test check data log

Reversible: NO   **Flag affected:**   YES

Parameters: ☐ Perform repairs

Syntax: Test check data log [*(Perform repairs)*]

This command tests if there are any reports of nonrepaired damage in the check data log. If the **Perform repairs** option is not specified, the flag is set if there are any reports of nonrepaired damage.

If the **Perform repairs** option is specified, an attempt is made to repair the damage. There is no need for the check data log to be open. Furthermore, OMNIS automatically tests that only one user is logged onto the data file (if not, the command fails with flag false), and further users are prevented from logging onto the data until the command completes.

If a working message with a count is open while the command is executing, the count will be incremented at regular intervals. The command may take a long time to execute, and it is not possible to cancel execution even if a working message with cancel box is open.

The command sets the flag if it completes the data repair successfully and clears the flag otherwise. The command is not reversible.

```
; First do a check data to obtain list of problems
Quick check
Test check data log
If flag true
   OK message {Problems found in data file}
   Open check data log
End If
```

Test clipboard

Reversible: NO   **Flag affected:**   YES

Parameters: Field name

Syntax: Test clipboard [*field-name*]

This command tests whether the data on the clipboard is suitable for pasting into the specified field or current selection. The command sets the flag to true if and only if there is data on the clipboard "suitable" for pasting into the specified or current field. "Suitability" here is defined by the standard type conversion built into OMNIS, that is, a text field has to be presented with some text, and a picture field with something that can be handled as a picture, for example, a bitmap, metafile, PICT, OLE object, and so on.

```
Test clipboard CVAR1
If flag true
   Paste from clipboard CVAR1 (Redraw field)
End If
```

Test data with search class

Reversible: NO   **Flag affected:**   YES

Parameters: None

Syntax: Test data with search class

This command tests the record in the CRB against the current search class. It sets the flag if the record passes the test or if there is no current search class. If the data does not fit the current search class, the flag is cleared.

Test data with search class uses the current search as the condition of the test which has been set using *Set search name* or *Set search as calculation*.

```
; Declare local variable SCODE of Character type
Calculate SCODE as 'RT'
Set search as calculation {len(SCODE)>2}
Test data with search class
If flag false
   OK message {Test failed, [SCODE] invalid}
End If
```

Test for a current record

Reversible: NO   **Flag affected:**   YES

Parameters: File class name

Syntax: Test for a current record [*{file-name}*]

This command tests for the presence of a current record from a specified file class. The flag is set if a current record for the file is found and cleared if not. The flag is also cleared if the selected file is a memory-only or a closed file. The test is carried out on the main file if no other file class is specified.

```
Test for a current record {FCLIENTS}
If flag false
   OK message {No client record, locating first}
   Find first
End If
Prepare for edit
Enter data
Update files if flag set
```

Test for a unique index value

Reversible: NO   **Flag affected:**   YES

Parameters: Field name (must be indexed)

Syntax: Test for a unique index value on *field-name*

This command tests the specified indexed field for a unique value. The flag is set if the current field value is a unique index value, and cleared if the value duplicates an existing index value. In a multi-user situation, no account is made of field values in records held by other work stations which are not yet updated to disk.

You use *Test for a unique index value* before storing a new value in a file. In the following example, the proposed new part number is tested against the existing file.

```
Calculate PART_NUM as 'RT100'
Test for a unique index value on PART_NUM
If flag true
   Update files
Else
   OK message {Part number already exists}
   Cancel prepare for update
   Quit method
End If
```

Test for field enabled

Reversible: NO   **Flag affected:**   YES

Parameters: Field name (of window field)

Syntax: Test for field enabled *field-name*

This command tests if the specified field on the top window instance is enabled, that is, if it is not currently disabled with *Disable fields* or by setting $enabled to kFalse. The flag is always cleared if there are no window instances open or if the field does not exist.

```
Test for field enabled {EntryId}
If flag true
   Disable fields EntryId
Else
   OK message {Field 4 is disabled}
   Enable fields EntryId
End If
```

or do it like this

```
If $cwind.$objs.FieldName.$enabled = kTrue
```

Test for field visible

Reversible:　　NO　　　　**Flag affected:**　　YES

Parameters:　　Field name (of window field)

Syntax:　　　　Test for field visible *field-name*

This command tests whether a particular field is visible. If the specified field in the top window instance is visible, that is, $visible is kTrue and the field has not been hidden with *Hide fields*, the flag is set. A field under another field or beyond the edge of the screen, may be reported as visible and the flag set. The flag is always cleared if there are no window instances open or if the field does not exist.

```
Test for field visible EntryId
If flag true
   Hide fields FieldName
End If
```

or do it like this

```
If $cwind.$objs.FieldName.$visible = kTrue
```

Test for menu installed

Reversible:　　NO　　　　**Flag affected:**　　YES

Parameters:　　Menu instance name

Syntax:　　　　Test for menu installed *{menu-instance-name}*

This command tests whether the specified menu instance is installed on the menu bar. The flag is set if the menu instance is on the menu bar and cleared if it is not, regardless of whether the menu instance is enabled or grayed out. The command does not apply to hierarchical and popup menus.

```
Test for menu installed {REP1}
If flag false
   Install menu MREPORTS/REP1
End If
```

Test for menu line checked

Reversible: NO   **Flag affected:**   YES

Parameters: Menu instance name
Line number

Syntax: Test for menu line checked *menu-instance-name/line-number*

This command tests whether the specified line of a menu instance is checked. You specify the *menu-instance-name* and the *line-number* of the menu line you want to test. The flag is set if the specified line of the menu instance is checked, and cleared if the line is not checked. The flag is always cleared if the menu instance is not installed on the menu bar.

You can check menu lines using *Check menu line*. *Uncheck menu line* removes the check.

```
Install menu MREPORTS/rep1
Test for menu line checked rep1/5
If flag true
   Uncheck menu line rep1/5
Else
   Check menu line rep1/5
End If
```

or do it like this

```
If $clib.$imenus.MENU.$objs.LineName.$checked = kTrue
```

Test for menu line enabled

Reversible: NO   **Flag affected:**   YES

Parameters: Menu instance name
Line number

Syntax: Test for menu line enabled *menu-instance-name/line -number*

This command tests whether the specified line of a menu instance is enabled. You specify the *menu-instance-name* and the *method-number* of the menu line you want to test. It sets the flag if the specified line of the menu instance is enabled. The flag is cleared if the menu instance is not installed on the menu bar.

This command may still return false if the current user has no access to the menu line or if the line is disabled because there is no current record, even after *Enable menu line* has been executed.

You can disable or enable menus using *Disable menu line* and *Enable menu line*.

```
Install menu MREPORTS/rep1
Test for menu line enabled rep1/3
If flag true
   Disable menu line rep1/3
Else
   Enable menu line rep1/3
End If
```

or do it like this

```
If $clib.$imenus.MENU.$objs.LineName.$enabled = kTrue
```

Test for only one user

Reversible: NO   **Flag affected:**   YES

Parameters: □ All data files

Syntax: Test for only one user [*(All data files)*]

This command tests whether the current data file is being used by a single user, and if so sets the flag.

If the **All data files** check box option is selected, all open data files are tested for a single user. The flag is cleared if any one data file has more than one user.

If the flag is set, further workstations are prevented from logging on to the tested data file(s) until the method containing the test command is terminated. The workstations will see a padlock cursor until the method terminates.

OMNIS always sets the flag if the program is running in single user mode. Under Windows, this means that the data is on a DOS volume without the SHARE command having been run.

```
Test for only one user
If flag false
   OK message {Sorry, option not allowed}
   Quit method kFalse
End If
Do method INVOICES/Insert New
```

Test for program open

Reversible: NO **Flag affected:** YES

Parameters: Program name

Syntax: Test for program open *{program-name}*

This command tests whether the specified program is running under Windows. The flag is set if the specified program is running.

The program name must be the module name which is usually, but not necessarily, the same as its executable file name. Under Windows 95, you need to specify the full pathname for the program. Under Windows NT, the file PSAPI.DLL must be present in the OMNIS directory or on the Windows path for this command to work. PSAPI.DLL is supplied in the OMNIS directory of the Windows NT version of OMNIS Studio.

```
Test for program open {c:\excel\excel}
If flag false
   Start program minimized {c:\excel\excel}
End If
```

Test for valid calculation

Reversible: NO **Flag affected:** YES

Parameters: Calculation

Syntax: Test for valid calculation *{calculation}*

This command lets you test a calculation before it is evaluated. It is essential to test strings to be evaluated by the *eval(), evalf()* and *fld()* functions before doing the evaluation.

The flag is set True if the calculation is valid.

```
Calculate CVAR1 as 'SALARY >= CVAR5'
Test for valid calculation {evalf(CVAR1)}
If flag true
   Set search as calculation {evalf(CVAR1)}
End If
Find first on SALARY (Use search)
```

See the *eval()* function.

Test for window open

Reversible: NO   **Flag affected:**   YES

Parameters: Window instance name

Syntax: Test for window open *{window-instance-name}*

This command tests if the specified window instance is open. If the window instance is open, OMNIS sets the flag, otherwise the flag is cleared. Window instances are opened with *Open window instance* or the $open() method.

```
Test for window open {winst1}
If flag false
   Open window instance Mywin/winst1
End If
```

Test if file exists

Reversible: NO   **Flag affected:**   YES

Parameters: File name (full file name and path)

Syntax: Test if file exists *{file-name}*

This command tests if the specified file exists. The flag is set if the file exists. Otherwise, it is cleared. You can use this command to prevent the user from overwriting existing files with print files, and so on.

You cannot use this command to check for the existence of a data file if the data file is in use by another workstation. Use *Open data file* for this type of checking.

```
Switch sys(6)='M'
   Case kTrue              ;; for MacOS
      Test if file exists {HD80:Work:Output file1}
      If flag false
         Set print file name {HD80:Work:Output file1}
      Else
         OK message {Overwriting file Output file1}
         Set print file name {HD80:Work:Output file1}
      End If
   Default                 ;; Under Windows, NT, or 95
      Test if file exists {C:\WORK\OUTPUT1.TXT}
      If flag false
         Set print file name {C:\WORK\OUTPUT1.TXT}
      Else
         OK message {Overwriting file OUTPUT1.TXT}
         Set print file name {C:\WORK\OUTPUT1.TXT}
      End If
End Switch
```

Test if list line selected

Reversible: NO   **Flag affected:**   YES

Parameters: Line number (can be calculation, default is current line)

Syntax: Test if list line selected [*{line-number}*]

This command tests the specified line of the current list and sets the flag if it is selected. You can specify a particular line in the list by entering either a number or a calculation. If the number is not specified, the test is performed on the current line of the list, that is, the line number held in *LIST.$line*.

The following example loads the current line of the list if it has been selected:

```
Set current list LIST1
Test if list line selected
If flag true
   Load from list
Else
   Quit method
End If
```

or do it like this

```
If LIST1.$line.$selected = kTrue
```

Test if running in background

Reversible: NO   **Flag affected:** YES

Parameters: None

Syntax: Test if running in background

This command tests if OMNIS is running in the background, that is, it sets the flag if OMNIS is *not* the top application window.

The Windows environment and MacOS Finder both provide you with multi-tasking facilities. When another program is running, with OMNIS in the background, you can continue with tasks such as importing data although the processor's time becomes shared between the current tasks. You can use this test to alter the behavior of the library when it becomes the background task.

```
Test if running in background
If flag false
   Open window instance WMONITOR
End If
```

Text:

Reversible: NO   **Flag affected:**   NO

Parameters: Text and/or variable
                  ☐ Carriage return
                  ☐ Line feed
                  ☐ Platform newline

Syntax: Text: *text*

This command adds text to the global text buffer. The *Text:* command supports leading and trailing spaces and can contain square bracket notation, that is, you can include or add the contents of a variable to the text buffer. You build up the text block using the *Begin text block* and one or more *Text:* commands. The **Carriage return, Line feed,** and **Platform newline** options add the appropriate character to the current *Text:* line. When you have placed one *Text:* line and you press Ctrl/Cmnd-N to create a new method line, the *Text:* command is selected and the current carriage return and line feed options are copied to the new method line automatically. You should end a block of text with the *End text block* command, and you can return the contents of the text buffer using the *Get text block* command.

Note that in some cases the *Text:* command will not uncomment; for example, *Text:* [Carriage return] will uncomment to *Text:* <empty text>.

```
; Declare var lv_TEXT of Character type
; Declare var lv_SUBTEXT of Character type
Calculate lv_SUBTEXT as '"EXCELLENT command or filename!"`
Begin text block
Text: Why doesn't DOS ever        ;; includes trailing space
Text: say [lv_SUBTEXT]            ;; includes contents of lv_SUBTEXT
End text block
Get text block lv_TEXT
; lv_TEXT contains
   Why doesn't DOS ever say "EXCELLENT command or filename!"
```

Trace off

Reversible: NO   **Flag affected:**   NO

Parameters: None

Syntax: Trace off

This command turns off trace mode at a point in a method. See *Trace on* for more information about trace mode and using the debugger.

```
Trace on
; this line is sent to the trace log ..
Trace off
; .. this line is not
```

Trace on

Reversible: NO   **Flag affected:**   NO

Parameters: ☐ Clear trace log

Syntax: Trace on [*(Clear trace log)*]

This command sends all subsequent commands to the trace log and displays the current command in the method editor. It lets you turn on trace mode at a point in a method where you suspect that there may be a problem, or some code which is difficult to follow. In trace mode, the topmost method design window is continually changed to show the command being executed. Also when in trace mode, a **trace log** is maintained; this contains the class name and method name in the Item column and the command line text in the Data column, for all methods which are executed in trace mode or single-stepped. Error messages, breakpoints, and so on, which occur in trace mode are also entered in the trace log. The **Clear trace log** option deletes all existing entries before new lines are added to the log.

The trace log window is opened and brought to top either by **Open trace log** on the method editor **Options** menu or by the *Open trace log* command. This window allows the trace log to be viewed, cleared or printed, and lets you alter the maximum number of lines in the log. Double-clicking on a line in the trace log causes a method design window to be opened or brought to the top with the appropriate command displayed. If Shift is pressed when double-clicking, a new method design window is opened in preference to changing the identity of the class displayed in the existing method design window.

If the double-clicked line in the log is a field value line, the value window for that field is opened. The trace log is not adjusted when methods are modified. This means that trace log lines may point to the wrong command or no command if the class containing that method has been modified.

```
Trace on
; this line is sent to the trace log ..
Trace off
; .. this line is not
```

Translate input/output

Reversible: YES **Flag affected:** NO

Parameters: ☐ Enabled

Syntax: Translate input/output [*(Enabled)*]

This command converts text between ANSI and ASCII when you import or export/print text, when you check the **Enabled** option. Windows applications such as OMNIS use the ANSI character set which differs from the extended ASCII used by non-Windows DOS programs and DOS hardware. This can cause text containing accented and other special characters with ASCII values greater than 127 to be exported or printed incorrectly when the normal Windows drivers are bypassed.

When the **Enabled** option is turned on, text exported or printed to port, clipboard, file or TTY printer is converted from ANSI to extended ASCII. Conversely, imported text is converted from ASCII to ANSI. The command has no effect when printing using the standard Windows printer drivers except for the Generic TTY driver.

If you execute the command with **Enabled** unchecked, text conversion is turned off.

```
Send to file
Set print file name {OUT.TXT}
Translate input/output (Enabled)
Print report
Translate input/output
; Turns off the translation since Enabled is not checked
```

Transmit text to port

Reversible: NO **Flag affected:** YES

Parameters: Text
       ☐ Add newline

Syntax: Transmit text to port [*(Add newline)*] *{text}*

This command sends text to a port; for example, you can send printer control characters. To transmit control characters, you can use the *chr()* function inside square brackets. For example, *[chr(27,14)]* sends escape 14.

The **Add newline** option enables you to send end of line characters after each line of text.

An error occurs and the flag is cleared if the port has not been selected or if the user presses Ctrl-Break/Cmnd-period while waiting for the output buffer to be emptied.

When you use a printer connected to the port, this command lets you send escape codes to control print characteristics.

```
Set port name {1 (Modem port)}
Set port parameters {1200,n,7,2}
Transmit text to port {[chr(14)]}
Print report
Close port
```

Transmit text to print file

Reversible: NO   **Flag affected:**   YES

Parameters: Text
            ☐ Add newline

Syntax: Transmit text to print file [*(Add newline)*] *{text}*

This command sends text to a print file, for example, you can send printer control characters. To transmit control characters, you can use the *chr( )* function inside square brackets. For example, *[chr(27,14)]* sends escape 14.

The **Add newline** option causes OMNIS to add end of line characters after each line of text.

An error occurs if no print file has been selected.

```
Set print file name {output.prn}
Transmit text to print file {[chr(27,14)]}
Print report
Close print file
```

Uncheck menu line

Reversible: YES   **Flag affected:**   NO

Parameters: Menu instance name
                 Line number

Syntax: Uncheck menu line *menu-instance-name*/*line-number*

This command removes the check mark on the specified line of a menu instance. No action is taken if there is no check mark or the menu instance is not installed. You specify the *menu-instance-name* and the *line-number* of the menu line you want to uncheck.

If you use *Uncheck menu line* in a reversible block, the specified menu line is checked again when the method terminates.

```
; Help menu is installed as mHelp1 menu instance
Test for menu line checked mHelp1/6
If flag true
   Uncheck menu line mHelp1/6
   Calculate HELP as 0
Else
   Check menu line mHelp1/6
   Calculate HELP as 1
End If
```

or do it like this

```
Do $clib.$imenus.MENU.$objs.LINE.$checked.$assign(kFalse)
```

Unload error handler

Reversible: NO   **Flag affected:**   YES

Parameters: Number or name/number (of custom menu method)

Syntax: Unload error handler [*menu-name/*]***number*** [*{method-name}*]

This command unloads the specified error handler (a method is taken as its parameter). If there are multiple error handlers at that method, they are all unloaded. The flag is set if an error handler is unloaded. See *Load error handler* for more information about error handlers.

```
Unload error handler Code1/Hndlr1
Load error handler Code1/Hndlr2
```

Unload event handler

Reversible: NO **Flag affected:** NO

Parameters: Library name
      Routine name
      Parameters list

Syntax: Unload event handler [*library-name*/]***routine-name***
      [*(parameter1*[,*parameter2*]...*)*]

This command unloads the specified event handler or, if no handler is specified, all event handlers. If none exists, no action is taken. An event handler is always unloaded when the library is closed or when the program quits. See *Load event handler* for more information on event handlers.

Unload external routine

Reversible: NO **Flag affected:** YES

Parameters: Library name
      Routine name
      Parameters list

Syntax: Unload external routine [*library-name*/]***routine-name***
      [*(parameter1*[,*parameter2*]...*)*]

This command unloads the specified external code from memory. If it is not already loaded or is not found, the flag is cleared and no action takes place. If no external is specified, all externals are unloaded. All loaded external routines are unloaded when the library is closed or when the program quits. See *Load external routine* for more information on external routines.

```
Unload external routine {mathslib/sqroot}
```

Until break

Reversible: NO   **Flag affected:**   NO

Parameters: None

Syntax: Until break

This command terminates a repeat loop if the break command (Ctrl-Break/Cmnd-period) is detected unless the break key itself is turned off with *Disable cancel test at loops*. *Until break* does not perform a test. You can terminate a repeat loop using *Break to end of loop* within the loop.

```
Set main file {f_prices}
Disable cancel test at loops
Repeat  ;; only way out of this loop is to enter a price of zero!
   Open window instance W_enter_price
   Enter data
   If PRICE = 0
     Break to end of loop
   End If
Until break
```

Until calculation

Reversible: NO   **Flag affected:**   NO

Parameters: Calculation

Syntax: Until *calculation*

This command terminates a *Repeat–Until* conditional loop specifying a calculation as the condition. The calculation is evaluated at the end of the loop that continues if the derived value is zero.

```
; This method prints 10 messages
Calculate LVAR1 as 1
Repeat
   OK message {Loop number [LVAR1]}
   Calculate LVAR1 as LVAR1+1
Until LVAR1 >= 11
; Loop ends when LVAR1 >= 11
```

Until flag false

Reversible: NO   **Flag affected:**   NO

Parameters: None

Syntax: Until flag false

This command terminates the *Repeat–Until* conditional loop if the flag is false; execution continues with the command following the *Until*. If the flag is true, execution continues with the command following the *Repeat*.

The following method uploads data to a server, and uses a *Repeat-Until flag false* construct to select the records in turn, until there are no more records.

```
; FILENAME and TABLE are passed to this method
Set main file {[FILENAME]}
Find first
If flag true
  Repeat
    Working message (High position) {Inserting...}
    Perform SQL: Insert into [TABLE] insertnames([FILENAME])
    If flag false
       Do method ErrorHandler
    End If
    Next
  Until flag false
End If
```

Until flag true

Reversible: NO   **Flag affected:** NO

Parameters: None

Syntax: Until flag true

This command terminates the *Repeat–Until* conditional loop if the flag is true; execution continues with the command following the *Until* command. If the flag is false, execution continues with the command following the *Repeat* command.

```
Repeat
   Working message (Repeat count)
   Yes/No message {End the loop?}
Until flag true
```

Update data dictionary

Reversible: NO   **Flag affected:** YES

Parameters: ☐ Test only
File or list of files (the default is all files)

Syntax: Update data dictionary [*(Test only)*] [*{file1*[,*file2]...}*]

This command updates the data dictionary for the specified file or list of files. The data dictionary is a copy of the file class field definitions and is stored in the data file. The command lets you write minor file class changes to the data dictionary. These minor changes do not require data reorganization, and include changes such as adding new fields, altering field names and altering field lengths.

Update data dictionary updates the data dictionary for the specified list of file classes. If you omit a file name or list of files, *all* the files with slots in the current data file are updated.

If a specified file name does not include a data file name as part of the notation, the default data file for that file is assumed. If the file is closed or memory-only, the command does not execute and returns with flag false.

If the **Test only** option is specified, no updating is actually carried out, and the flag is set if at least one file in the data dictionary needs updating.

Certain changes made to a file class (that is, changes in indexes, field type changes and changes in file connections) require data reorganization. In this case, using *Update data dictionary* to keep the file class and the data file "in step" will be inappropriate. *Reorganize data* lets you test whether a data file needs reorganization as well as to reorganize it if necessary.

```
Reorganize data (Test only)
If flag false
   Update data dictionary
   ; used when only minor changes to file class(es) have been made
Else
   Yes/No message {Data needs reorganizing; do it?}
   If flag true
      Reorganize data
   End If
End If
```

Update files

Reversible: NO   **Flag affected:**   YES

Parameters: ☐ Do not cancel pfu (prepare for update)

Syntax: Update files [*(Do not cancel pfu)*]

This command writes the records in the current record buffer to disk and cancels the Prepare for... mode. You must execute the command when OMNIS is in a Prepare for update mode otherwise an error occurs.

If a warning error code *kerrUnqindex* or *kerrNonull* is returned during the execution of this command, the Prepare for update mode is *not* canceled. This means that you can check for these errors and recover without losing the data the user has already typed in. In fact, if you issue a new *Prepare for...* command, OMNIS will reread records, and any data that is already in the CRB will be lost.

```
Open window instance W_addresses
Prepare for edit
Enter data
If flag true
   Update files
Else
   OK message {Files not updated, data invalid}
End If
```

The **Do not cancel pfu** option prevents the command from canceling Prepare for update mode. Thus, you can make more changes to the data, the multi-user locks remain in place, and another *Update files* can be executed, for example

```
Open window instance W_addresses
Prepare for edit
Enter data
If flag true
   Update files (Do not cancel pfu)
   Calculate FCODE as 'New code'
   OK message {Code changed}
   Update files
End If
```

The following example inserts an invoice (in the parent file) and a list of related invoice items (in the child file). The **Do not cancel pfu** option ensures that the parent record remains locked until complete.

```
Set main file {FINVOICE}
Prepare for insert
Enter data
Update files (Do not cancel pfu)
Set main file {FITEMS}
For each line in list from 1 to $linecount step 1
   Prepare for insert
   Load from list
   Update files (Do not cancel pfu)
End For
Update files
```

The *Update* command causes the indexes in the files to be re-sorted. Thus, in multi-user mode, the files are locked while *Update files* is executing. You can control this file locking by running *Do not wait for semaphores*, for example

```
Wait for semaphores
Prepare for edit
Enter data
Do not wait for semaphores
If flag true
   Repeat
      Working message {Waiting for file locks}
      Update files
   Until flag true
End If
```

When *Do not wait for semaphores* is active, *Update files* returns flag false and does nothing if the file is locked.

Update files if flag set

Reversible: NO   **Flag affected:**   YES

Parameters: ☐ Do not cancel pfu (prepare for update)

Syntax: Update files if flag set [*(Do not cancel pfu)*]

This command writes the current values in the current record buffer to disk if the flag is set, that is, true. This is a variation on the *Update files* command and is equivalent to:

```
If flag true
   Update files
End If
```

When the command follows *Enter data*, the Prepare for update mode is canceled, and the record is stored on disk if the user clicks OK or presses the Return/Enter key.

Use event recipient

Reversible: NO   **Flag affected:**   YES

Parameters: Recipient tag

Syntax: Use event recipient *{recipient-tag}*

This command sets the event recipient by specifying the recipient tag. The named event recipient must be currently available on the network; that is, its name must be on the Apple **Application** menu.

When OMNIS is launched, the recipient defaults to OMNIS, that is, events are sent to itself. Similarly if you use this command without a parameter, the recipient reverts to OMNIS.

The following example shows the difference between *Use event recipient*, which is used with a tag previously assigned by the user with *Prompt for event recipient*, and *Set event recipient*, which takes a local application name as a parameter, and turns it into a recipient tag.

```
Prompt for event recipient {MyApp1}
; Prompt user and select application do something with 'MyApp1'
Set event recipient {Microsoft Excel}
; This is the name of a current application, as shown on the
; Apple Application menu
; do something in 'Microsoft Excel' for example
Use event recipient {MyApp1}
; go back to the tagged recipient,
; previously prompted to do something else.
; Finally go back to OMNIS by resetting recipient with no prompt
Use event recipient
```

Variable menu command

Reversible:	NO	**Flag affected:**	NO

Parameters: Command option (see below)
                  List of file and/or field names

Syntax: Variable menu command**: *option {file|field1*[*,file|field2*]*...}*

This command performs one of the Variable context menu options on the specified field or list of fields. You can specify one of the following Variable menu options.

Set break on variable change	*Remove from watch variables list*
Clear break on variable change	*Send value to trace log*
Set break on calculation	*Send minimum to trace log*
Clear break on calculation	*Send maximum to trace log*
Store min & max	*Send all to trace log*
Do not store min & max	*Open value window*
Add to watch variables list	*Open values list*

You use the Variable menu to examine the value of fields and variables. Normally, you open the Variable menu by right-button/Ctrl-clicking on a variable name in the method editor, the Catalog, or anywhere else in OMNIS. The list of field names is entered in any of the following ways (including a mixture of file class and field names):

```
Fieldname1,Fieldname2,Fieldname3
; or
FileName
; includes all the fields in the file class
File1.Fieldname1,File5
; includes Fieldname1 (from a file other than filename5)
; and all the fields in filename5.
```

You can select one of the following options:

Set break on variable change sets a variable change breakpoint for each variable in the list.

Clear break on variable change clears any variable change breakpoint for each variable in the list. If no variable names list is specified, all current variable change breakpoints are cleared.

Set break on calculation sets a calculation breakpoint for each variable in the list. You can set the calculation for each variable using *Set break calculation*. Setting calculation breaks for more than a very few variables will cause methods to run very slowly.

Clear break on calculation clears any variable change breakpoints for each variable in the list. If no variable names list is specified, all current calculation breakpoints are cleared.

Store min & max causes minimum and maximum values to be stored for each variable in the list.

Do not store min & max clears 'Store min and max' mode for each variable on the list. If no variables are specified, all current 'Store min and max' are cleared.

Add to watch variables list marks each variable on the list as a watch variable.

Remove from watch variables list marks each variable on the list as not watched. If no variables are specified, all variables are marked as not watched. Note that variables with breakpoints or with 'Store min and max' mode set always appear in the watch variables list.

Send value to trace log adds a line to the trace log for each variable on the list. If no variables are specified, all values for all variables on the watch variables list are sent to the trace log.

Send minimum to trace log adds a line to the trace log for each variable on the list for which 'Store min and max' is set. If no variables are specified, the minimum values for all variables for which 'Store min and max' is set are sent to the trace log.

Send maximum to trace log adds a line to the trace log for each variable on the list for which 'Store min and max' is set. If no variables are specified, the maximum values for all variables for which 'Store min and max' is set are sent to the trace log.

Send all to trace log adds a value line to the trace log for each variable on the list, and adds minimum and maximum line(s) to the trace log for each variable on the list for which 'Store min and max' is set. If no variables are specified, this is carried out for all appropriate variables on the watch variables list.

Open value window opens a value window for each variable on the list, or for every variable on the watch variables list if no variables are specified. There is a limit on the number of windows that you can open at once.

Open values list opens the values list for each of the variable types given in the command parameters. For example, *Variable menu command: open values list {LVAR1, Local1}* opens two values lists, one for Hash variables, the other for Local variables. There is one values list for each file class, so if more than one variable name in a particular file class is specified the values list for that file will only be opened once. There is also a limit on the number of windows that you can open at once.

Wait for semaphores

Reversible: YES   **Flag affected:** NO
Parameters: None
Syntax: Wait for semaphores

This command causes all the commands which set semaphores to wait with a lock cursor until the semaphores for the required records are available.

When a library is first selected, *Wait for semaphores* is automatically selected to ensure compatibility with existing libraries. It causes all the commands which set semaphores to wait with a lock cursor until the semaphore is available then return with the flag set, or to wait until the user cancels with a Ctrl-Break/Cmnd-period then return with a flag clear.

Semaphores

Semaphores are internal flags or indicators set in the data file to show other users that the record has been required elsewhere for editing. Semaphores are set only when running in multi-user mode, that is, the data file is located on a networked server, a Mac volume or on a DOS machine on which SHARE has been run.

The commands which set semaphores are *Prepare for edit*, *Prepare for insert*, *Update files* and *Delete*, and also, if pfu mode is on, *Single file find*, *Load connected records*, *Next, Previous* and *Set read/write files*. Auto finds on windows always wait for semaphores.

The **Edit/Insert** commands from the **Commands** menu always wait for a semaphore as do automatic find entry fields.

```
Wait for semaphores
Prepare for edit  ;;  Waits for record if locked by another user
Enter data
Do not wait for semaphores
If flag true
   Update files
   If flag false
      OK message {File was locked, update failed}
   End If
End If
```

While calculation

Reversible: NO   **Flag affected:**   NO

Parameters: Calculation

Syntax: While *calculation*

This command starts a *While–End While* loop that continues while a calculated condition remains true. When the condition is not satisfied the method jumps out of the loop and the first command after the closing *End While* is executed. A loop that begins with a *While* command must terminate with an *End While* otherwise an error occurs.

```
While PAID = 'YES'
   Do method DeleteOldRecords
   Next
End While
```

While flag false

Reversible: NO   **Flag affected:**   NO

Parameters: None

Syntax: While flag false

This command starts a *While–End While* loop that continues while the flag is false. While the condition is false, a command or a series of commands is executed until the condition becomes true, at which time the first command after the closing *End While* is executed. A loop that begins with a *While* command must terminate with an *End While*, otherwise an error occurs.

```
Do not wait for semaphores
If flag true
   Update files
   While flag false
      Working message {Waiting for file locks}
      Update files
   End While
End If
```

While flag true

Reversible: NO   **Flag affected:** NO

Parameters: None

Syntax: While flag true

This command starts a *While–End While* loop which continues while the flag is true. While the condition is true, a command or a series of commands is executed until the condition becomes false, at which time the first command after the closing End While command is executed. A loop that begins with a *While* command must terminate with an *End While*, otherwise an error occurs.

```
While ...
   While ...
      [User commands]
   End While
End While
```

In the following example, the loop continues until the *Next (Exact match)* command fails to find a match.

```
Calculate LVAR1 as 0
Find on CODE (Exact match) {BR01}
While flag true
   Calculate LVAR1 as LVAR1 +CBAL
   Next (Exact match)
End While
```

Working message

Reversible: NO   **Flag affected:** NO

Parameters: ☐ High position
                ☐ Large size
                ☐ Cancel box
                ☐ Repeat count
                Message (text)

Syntax: Working message [([*High position*][,*Large size*][,*Cancel box*]
                 [,*Repeat count*])] [*{message}*]

This command displays a message, usually to indicate that the computer is working or waiting for input. An alternating icon indicates that the computer is busy. A working message automatically closes when the method quits and control returns to the user.

For greater emphasis, you can display the working message with **High position**, and also increase the size of the message box by checking the **Large size** option.

If a working message is placed in a loop with a **Cancel** button, pressing the Escape/Cmnd-period or clicking on Cancel quits all methods. However, if you first execute *Disable cancel test at loops*, you can implement an orderly exit.

```
Begin reversible block
   Disable cancel test at loops
End reversible block
Repeat
   Working message (Cancel box)
   If canceled
      Break to end of loop
   End If
   Do LVAR1+1
Until flag true
OK Message {All done}
```

If *Disable cancel test at loops* is executed before the loop, the cancel is detected only on executing the *Working message*.

A **Repeat count** option is available with *Working message*, and displays the value of an internal counter which indicates the number of times a particular *Working message* has been encountered. If the command is in a *Repeat* loop, the counter increments at each pass of the loop.

```
Repeat
   Working message (Repeat count) {FIELD = [FIELD]}
   Redraw working message
   Do method DeleteOldRecords
Until DONE = 1
```

XOR selected and saved

Reversible: NO   **Flag affected:**   YES

Parameters: Line number (can be calculation, default is current line)
                    ☐ All lines

Syntax: XOR selected and saved [*(All lines)*] [*{line-number}*]

This command performs a logical XOR of the Saved selection with the Current selection. To allow sophisticated manipulation of data via lists, a list can store two selection states for each line; the "Current" and the "Saved" selection. The Current and Saved selections have nothing to do with saving data on the disk; they are no more than labels for two sets of selections. The lists may be held in memory and never saved to disk: they will still have a Current and Saved selection state for each line but they will be lost if not saved. When a list is stored in the data file, both sets of selections are stored.

You can specify a particular line in the list by entering either a number or a calculation.

The *XOR selected and saved* command performs a logical XOR (exclusive OR) on the Saved and Current state and puts the result into the Current selection. Hence, if either of the Current and Saved states is selected, the Current state becomes selected, but if both states are equal, the resulting Current state will become deselected.

Logic Table (S=selected, D=deselected)

Saved	Current	Resulting Current State
S	S	D
D	S	S
S	D	S
D	D	D

The **All lines** option performs the XOR for all lines of the current list. The flag is set by this command. The following example selects the middle line of the list:

```
Set current list LIST1
Define list {LVAR1}
Calculate LVAR1 as 1
Repeat
   Add line to list
   Calculate LVAR1 as LVAR1+1
Until LVAR1=6
Select list line(s) (All lines)
Save selection for line(s) (All lines)
Invert selection for line(s) {3}
XOR selected and saved(s) (All lines)
Redraw lists
```

Yes/No message

Reversible:	NO	**Flag affected:**	YES

Parameters: Title (for message box)
                 ☐ Icon
                 ☐ Sound bell
                 ☐ Cancel button
                 Message (text)

Syntax: Yes/No message [*title*] [*([Icon]* [*,Sound bell*]
                 [*,Cancel button*]*)] {message}*

This command displays a message box containing the specified message and provides a **Yes** and a **No** pushbutton. Also, you can include a **Cancel** button, and add a short *title* for the message box. For greater emphasis, you can select an **Icon** for the message box (the default "info" icon for the current operating system), and you can force the system bell to sound by checking the **Sound bell** check box.

When the message box is displayed method execution is halted temporarily; it remains open until the user clicks on one of the buttons before continuing. The **Yes** button is the default button and can therefore be selected by pressing the Return key.

The number of lines displayed in the message box depends on your operating system, fonts and screen size. In the message text you can force a break between lines (a line return) by using the notation '//'.

You can insert a *Yes/No message* at any appropriate point in a method. If the user clicks the Yes button, the flag is set; otherwise, it is cleared. You can use the *msgcancelled()* function to detect if the user pressed the Cancel button.

```
Yes/No message (Cancel button) {Do you want to proceed?}
If flag false
   If msgcancelled()
      ; user chose Cancel
   else
      ; user chose No
   End If
Else
   ; user chose Yes
End If
```

Chapter 6—External Commands

This chapter describes the external commands supplied with OMNIS. They are available for all platforms except where indicated. All the external commands appear in the *External Commands...* group at the bottom of the command list in the method editor. Many of them start with an appropriate prefix which makes them easier to find; for example, the Lotus Notes commands begin with NSF, all the FTP ones begin with FTP, and so on.

You can extend the functionality of OMNIS by adding your own externals, or external code modules. You can implement these as external commands or functions which get called from within OMNIS. Under Windows, they are written as DLLs or code resources under MacOS. To use an external, you must place it in the EXTERNAL folder under the main OMNIS folder; the commands supplied with OMNIS are placed there by default ready for you to use.

External Commands

Call DLL

Reversible: NO   **Flag affected:** NO

Parameters: Library name (the DLL)
Procedure name
Parameters list
Return field

Syntax: Call DLL *(library-name, procedure-name*[,*parameter1*]...*)*
[returns *return-field*]

This command calls the registered DLL. The *library-name* is the name of the DLL containing the procedure specified by *procedure-name*. You can add field parameters that are pushed onto the stack before the DLL is called.

The following example opens the Windows **File Manager.**

```
Do method OpenExe ('winfile.exe',3)


; OpenExe  ;; called method
; Declare Parameter APPNAME (Character   255)
; Declare Parameter INSTRUCTS (Short integer   (0 to 255))
Register DLL ('KRNL386.EXE','WinExec','ICI') returns RESULT
Call DLL ('KRNL386.EXE','WinExec',APPNAME,INSTRUCTS) returns RESULT
If RESULT < 18
   Do method Errors
End If
```

CGIDecode

Reversible: NO   **Flag affected:** NO

Parameters: Stream
Returns: *DecodedField*
Syntax: CGIDecode(*Stream*)

CGIDecode can be used to turn CGI-encoded information back into plain text. It is the converse of CGIEncode. CGI-encoded information is sent over the HTTP protocol in a format that preserves special characters in URLs that delimit CGIs and arguments (that is, fields on Web forms). Errors are reported via the WebDevError callback mechanism.

stream is an OMNIS Character or Binary field containing the information to decode.

DecodedField is an OMNIS Character or Binary field that holds the resulting CGI-decoded representation of the *stream* argument.

Note: The HTTPParse external command automatically performs CGI decoding. Results from HTTPParse are already CGI-decoded.

CGIEncode

Reversible:	NO	**Flag affected:**	NO

Parameters: stream
Returns: *EncodedField*
Syntax: CGIEncode(*stream*)

CGIEncode changes text into a form acceptable as an argument to a Web server CGI. The HTTP protocol specifies that the text of an argument must be alphanumeric plus some other special characters, and does not allow spaces. Certain characters that separate arguments from each other and their values must be specially quoted.

The same rules apply to some HTTP header fields that are normally hidden.

Use this call when you are creating or decoding the text of a URL involving a CGI call or a header attribute.

Note: The HTTPHeader and HTTPPost external commands automatically encode or decode information presented to a Web server. You need not pre-encode arguments if you are using those external commands.

Stream is an OMNIS Character or Binary field containing the information to encode.

EncodedField is an OMNIS Character or Binary field that holds the resulting CGI-encoded representation of the *stream* argument.

Errors are reported via the WebDevError callback mechanism.

Change working directory

Reversible:	NO	**Flag affected:**	NO

Parameters: Path name
          Return field
Syntax: Change Working Directory *(path-name)*
          returns *return-field*

This command changes the current directory in use under Windows. Wild cards are not allowed with this command. Change working directory only switches directories on the same drive, not between drives. It returns any error code (shown at the end of this chapter), or zero if none.

```
Change working directory ("C:\OMNIS\External")
```

Close file

Reversible:	NO	**Flag affected:**	NO

Parameters: Reference number or DOS file handle
Return field

Syntax: Close file *(refnum)* returns ***return-field***

This command closes the file referred to by the file reference number or DOS file handle specified in *refnum*. All open files are automatically closed when OMNIS quits, but not when the current OMNIS library is closed. You should close files correctly.

It returns any error code (shown at the end of this chapter), or zero if none.

CMAttach

Reversible:	NO	**Flag affected:**	NO

Input Parameters: content, ctntID, ctntType, encodingType, mailText
Output Parameters: MIMEcontent
Returns: *Status* (0 if no error, or a non-zero if error)
Syntax: CMAttach(*content,MIMEContent*[,*ctntID,ctntType,
encodingType,mailText*])

CMAttach creates one MIME object for the specified content. This is the quick and simple way to compose a single-part MIME. To compose multipart MIME, use CMMCBegin, CMMInsert, and CMMCEnd.

Input Parameters

content   OMNIS Binary variable

content contains readable text or any binary content. For example, it can be the content of a word-processing document or the content of an image (such as Logo.GIF).

ctntID   OMNIS Character variable containing up to 255 characters

ctntID is an optional parameter describing the content that is being attached. Describe the content in any way you want.

ctntType   OMNIS Character variable containing up to 255 characters

ctntType is an optional parameter specifying the type of content in the file. The default *ctntType* is application/octet-stream. See the Content Header Types section for other content types.

encodingType   OMNIS Character variable

encodingType is an optional parameter specifying the type of encoding to use to encode the *content*. The default encoding type is base64. The supported encoding types are base64 and quoted-printable. For details about using the supported encoding types, see the section

Processing Email Content.

The base64 encoding type is generally used for encoding of all binary data. It is considered much safer than the uuencode/uudecode format. The quoted-printable encoding type is used for encoding non-standard ASCII text.

mailText   OMNIS Binary variable

mailtext is an optional parameter containing the ASCII text of an email body only. The default content type is text/plain. The default content transfer encoding is quoted-printable.

Output Parameters

MIMEContent   OMNIS Binary variable

MIMEContent is the composed, MIME-formatted object derived from the original content. Use the SMTPSend command to send this MIME content.

CMMCBegin

Reversible:	NO	**Flag affected:**	NO

Input Parameters: ctntID
Output Parameters: CSP
Returns: *Status* (0 if no error, or a non-zero if error)
Syntax: CMMCBegin(*CSP*[,*ctntID*])

CMMCBegin begins MIME composition to compose a MIME object. This command is typically used to compose multipart MIME objects. However, you can also use it to compose single-part MIME content.

Input Parameters

ctntID   OMNIS Character variable containing up to 255 characters

ctntID is an optional character string describing the content that is being composed.

Output Parameters

CSP   OMNIS Character variable

CSP (Content State Property) is a handle that related commands use. You must use the same CSP for each composition process. For example, if you use CMMCBegin, CMMInsert, and CMMCEnd to compose a given multipart message, the same CSP is used throughout. A different message can use a different CSP.

CMMCEnd

Reversible:	NO	**Flag affected:**	NO

Input Parameters: CSP
Output Parameters: MIMEContent
Returns: Status (0 if no error, or a non-zero if error)
Syntax: CMMCEnd(*CSP*,*MIMEcontent*)

CMMCEnd ends the MIME composition process that was started with CMMCBegin. CMMCEnd generates a completed and final MIME object.

Input Parameters

CSP   OMNIS Character variable

CSP (Content State Property) is the handle that CMMCBegin generated. You must use the same CSP for each composition process. For example, if you use CMMCBegin, CMMInsert, and CMMCEnd to compose a given multipart message, the same CSP is used throughout. A different message can use a different CSP.

Output Parameters

MIMEContent   OMNIS Binary variable

MIMEContent is the output of the completed MIME object. You can use the Internet email OMNIS SMTPSend command to send this MIME content.

CMMGBegin

Reversible:	NO	**Flag affected:**	NO

Input Parameters: content
Output Parameters: CSP
Returns: *Status* (0 if no error, or a non-zero if error)
Syntax: CMMGBegin(*CSP*,*content*)

CMMGBegin begins MIME decomposition by preparing content into a CSP (Content State Property) structure that defines parts and levels. The generated CSP value is used by CMMGet to retrieve and decode, if necessary, MIME body parts. You must use CMMGEnd to free up and release resources when the process is complete.

Input Parameters

content   OMNIS Binary variable

content is the MIME content object. The content object can contain either an entire email consisting of a MIME attachment, just the MIME portion of an email, or a MIME object that was previously generated by Content Manager.

Output Parameters

CSP   OMNIS Character variable

CSP (Content State Property) is a handle that other related extensions use. You must use the same CSP for each decomposition process. For example, if you use CMMGBegin and CMMGEnd to decompose a given multipart message, the same CSP is used throughout. A different message can use a different CSP.

CMMGEnd

Reversible:	NO	**Flag affected:**	NO

Input Parameters: CSP
Output Parameters: [None]
Returns: *Status* (0 if no error, or a non-zero if error)

Syntax: CMMGEnd(*CSP*)

CMMGEnd completes the decomposition process that was started by CMMGBegin. CMMGEnd cleans up all resources (such as memory) during the decomposition process.

Input Parameters

CSP   OMNIS Character variable

CSP (Content State Property) is the handle that CMMGBegin generated when the current process began. You must use the same CSP for each decomposition process. For example, if you use CMMGBegin and CMMGEnd to decompose a given multipart message, the same CSP is used throughout. A different message can use a different CSP.

CMMGet

Reversible:	NO	**Flag affected:**	NO

Input Parameters: CSP, partNum, levelNum
Output Parameters: content, ctntDisposition
Returns: Status (0 if no error, or a non-zero if error)

Syntax: CMMGet(*CSP*,*content*[,*partNum*,*levelNum*,*ctntDisposition*])

CMMGet gets and decomposes the next body part in multipart or single-part content. Prior to this, CMMGBegin starts the decomposition process.

The body parts are retrieved sequentially (from the first part) until the last body part has been retrieved, or you can retrieve specific body parts by specifying the part or level numbers.

Note: When CMMGet is called after retrieving the last body part, it retrieves the last body part again. Be sure to use CMQuery to determine the number of body part. Then keep count during the CMMGet operation.

Input Parameters

CSP   OMNIS Character variable

CSP (Content State Property) is the handle that CMMGBegin generated when the current process began. You must use the same CSP for each decomposition process. For example, if you use CMMGBegin and CMMGEnd to decompose a given multipart message, the same CSP is used throughout. A different message can use a different CSP.

partNum   OMNIS integer variable

partNum is an optional parameter used to retrieve a specific body part within a multipart MIME.

levelNum   OMNIS integer variable

levelNum is an optional parameter specifying the specific level within a multilevel MIME. MIME content containing embedded MIME content is also known as *multilevel* MIME content.

Output Parameters

content   OMNIS Binary variable

content is the output of the body part retrieved with all the MIME headers removed. The content is automatically decoded if necessary.

ctntDisposition   OMNIS Character variable

ctntDisposition is an optional parameter. If *content* contains the Content-disposition header and a specified filename, as in:

Header	Value
Content-disposition	picture.jpg

this parameter returns the filename associated with the Content-disposition header. If *ctntDisposition* is specified, and the content does not include the Content-disposition header or the filename value is blank, *ctntDisposition* returns the value #NULL#.

CMMInsert

Reversible:	NO	**Flag affected:**	NO

Input Parameters: CSP, content, ctntID, ctntType, encodingType, ctntDisposition
Returns: Status (0 if no error, or a non-zero if error)
Syntax: CMMInsert(*CSP*,*content*.[*ctntID*,*ctntType*,*encodingType*, *ctntDisposition*])

CMMInsert inserts content to be composed as a MIME body part in a process started by CMMCBegin. During the CMMInsert operation, the necessary MIME headers are added and the content is encoded if specified or required.

Input Parameters

CSP   OMNIS Character variable

CSP (Content State Property) is the handle that CMMCBegin generated.

content   OMNIS Binary variable

content is the content to be composed into MIME format and inserted into the current MIME object. Content can be any readable text or binary data, such as a Microsoft Word document or a GIF image.

ctntID   OMNIS Character variable containing up to 255 characters

ctntID is an optional character string describing the content.

ctntType   OMNIS Character variable containing up to 255 characters

ctntType is an optional parameter specifying the type of content. The *ctntType* default is text/plain; charset=us-ascii, for a plain text file.

encodingType   OMNIS Character variable

encodingType is an optional parameter specifying the type of encoding. The supported encoding types are base64 and quoted-printable. The *encodingType* default is quoted-printable, which is used for encoding non-standard ASCII text. The base64 encoding type is generally used for encoding of all binary data and is considered much safer than the uuencode/uudecode format.

ctntDisposition   OMNIS Character variable containing up to 255 characters

ctntDisposition is an optional parameter describing how content should be handled by your local email system. For example, if you specify a string containing a full pathname your email system may attempt to save the content into the same location as the pathname specifies.

CMQuery

Reversible:	NO	**Flag affected:**	NO

Input Parameters: content
Output Parameters: MIMEtype, numParts, numLevel
Returns: Status (0 if no error, or a non-zero if error)
Syntax: CMQuery(*content*,*MIMEtype*,*numParts*,*numLevel*)

CMQuery queries content to determine whether it is MIME, S/MIME, or not MIME; it also returns the number of parts and levels, as applicable. If the content is single-part MIME, the number of parts and number of levels is always 1 (one).

Input Parameters

content   OMNIS Binary variable

content is the MIME content object. This is usually an entire email message with its attachments, or an object containing MIME-formatted content that was previously generated by Content Manager.

Output Parameters

MIMEtype   OMNIS Character variable

MIMEtype specifies one of the following: MIME, S/MIME, or not MIME.

numParts   OMNIS Integer variable

numParts is the number of parts within *content* if *content* is multipart MIME.

numLevel   OMNIS Integer variable

numLevel is the number of levels within the *content* if the content is multilevel MIME content, that is, MIME content containing embedded MIME content.

Copy file

| **Reversible:** | NO | **Flag affected:** | NO |

Parameters: From path (file to be copied)
                    To path (of new file)
                    Return field

Syntax: Copy file (*from-path*[,*to-path*]) returns *return-field*

This command makes a copy of the file specified in *from-path*. You specify the path of the new file in to-path. If *to-path* includes a file name the file is copied and the new file is renamed. The file named in *to-path* must not already exist. If you omit *to-path*, a copy of the file named in *from-path* is created in the current directory using the same name with the extension ".BAK" under Windows or followed by " copy" under MacOS.

It returns any error code (shown at the end of this chapter), or zero if none.

Create directory

| **Reversible:** | NO | **Flag affected:** | NO |

Parameters: Path of new directory or folder
                    Return-field

Syntax: Create directory *(path)* returns *return-field*

This command creates the directory or folder (under MacOS) named in *path*. The directory must not already exist. *Create directory* does not create intervening directories. It only creates the last directory name in *path*.

It returns any error code (shown at the end of this chapter), or zero if none.

Create file

| **Reversible:** | NO | **Flag affected:** | NO |

Parameters: Path of new file
File type (MacOS only)
Creator (MacOS only)
R parameter (specifies if a resource fork is created also,
MacOS only)
Return field

Syntax: Create file (***path***[,*file-type*][,*creator*][,'R']) returns
return-field

This command creates the file specified in *path*. Every directory or folder in *path* must already exist. *Create file* does not create directories or folders.

The *file-type*, *creator,* and 'R' parameters apply to MacOS only, and are ignored by all other versions; 'R' is case-insensitive. If *file-type* and *creator* are not specified, a TeachText text file is created with type "TEXT" and creator "ttxt".

It returns any error code (shown at the end of this chapter), or zero if none.

DB2 Audio disable

Reversible: NO   **Flag affected:**   NO

Parameters: Table name
Column name
Logon switch to remain logged on to database
Return value

Syntax: DB2 Audio disable ([*tablename*][,*columnname*][,*/L'*]) returns
[*return-value*]

This command disables the Audio extender data type for the current DB2 database, or the specified table or column in the current database. To disable a database you do not need to pass any parameters, the currently connected database is used. To disable a table in the currently logged on database, you need to pass the *tablename* only. To disable a column in the current database, you need to pass the *tablename* and *columnname* parameters. You can specify the L switch to remain logged on to the current database, otherwise you are logged off automatically. This command returns a value of 1 if it is successful, otherwise 0 is returned.

```
DB2 Audio disable ( ) returns #2
; disables the current database
DB2 Audio disable ('table1') returns #2
; disables table1 in the current database
DB2 Audio disable ('table1', 'column1') returns #2
; disables column1 in table1 in the current database
```


DB2 Audio enable

Reversible: NO   **Flag affected:**   NO

Parameters: Table name
Column name
Logon switch to remain logged on to database
Return value

Syntax: DB2 Audio enable ([*tablename*][,*columnname*][,*′/L′*]) returns
[*return-value*]

This command enables the Audio extender data type for the current DB2 database, or the specified table or column in the current database. To enable a database you do not need to pass any parameters, the currently connected database is used. To enable a table in the currently logged on database, you need to pass the *tablename* only. To enable a column in the current database, you need to pass the *tablename* and *columnname* parameters. You can specify the L switch to remain logged on to the current database, otherwise you are logged off automatically. This command returns a value of 1 if it is successful, otherwise 0 is returned.

```
DB2 Audio enable ( ) returns #2
; enables the current database
DB2 Audio enable ('table1') returns #2
; enables table1 in the current database
DB2 Audio enable ('table1', 'column1') returns #2
; enables column1 in table1 in the current database
```

DB2 Audio is enabled

Reversible: NO   **Flag affected:**   NO

Parameters: Table name
Column name
Status variable
Logon switch to remain logged on to database
Return value

Syntax: DB2 Audio is enabled ([*tablename*][,*columnname*], **status** [,*/L'*])
returns [*return-value*]

This command checks whether or not the Audio extender data type is enabled for the
current DB2 database, or the specified table or column in the current database. To check
whether or not a database is enabled for Audio, you need to pass the status field only. To
check whether or not a table in the current database is enabled for Audio, you need to pass
the *tablename*, as well as the status field. To check whether or not a column is enabled for
Audio, you need to pass the *tablename* and *columnname* parameters, as well as the status
field. The *status* parameter returns kTrue or 1 if the database, table, or column is enabled
for Audio. You can specify the L switch to remain logged on to the current database,
otherwise you are logged off automatically. This command returns a value of 1 if it is
successful, otherwise 0 is returned, regardless of the value returned in the status field.

```
DB2 Audio Is Enabled ( , , #1 ) returns #2
; checks the current database
DB2 Audio Is Enabled ('table1', , #1 ) returns #2
; checks table1 in the current database
DB2 Audio Is Enabled ('table1', 'column1', #1 ) returns #2
; checks column1 in table1 in the current database
```

DB2 Get logon info

Reversible: NO   **Flag affected:**   NO

Parameters: Database or tablespace name
Username for the specified database
Password for the specified database
Error code
Error text

Syntax: DB2 Get logon info (*tablespace*, *username*, *password* [,*errorcode*] [,*errortext*])

This command returns the logon info for the current DB2 database, as specified by the DB2 Register logon info command or the DB2 DAM. You must supply variables for the *database* name, *username*, and *password* of the current DB2 database. You can include variables for the *errorcode* and *errortext* to return the names of the OMNIS variables where error codes and error text are stored.

DB2 Image disable

Reversible: NO   **Flag affected:**   NO

Parameters: Table name
Column name
Logon switch to remain logged on to database
Return value

Syntax: DB2 Image disable ([*tablename*][,*columnname*][,*/L'*]) returns [*return-value*]

This command disables the Image extender data type for the current DB2 database, or the specified table or column in the current database. To disable a database you do not need to pass any parameters, the currently connected database is used. To disable a table in the currently logged on database, you need to pass the *tablename* only. To disable a column in the current database, you need to pass the *tablename* and *columnname* parameters. You can specify the L switch to remain logged on to the current database, otherwise you are logged off automatically. This command returns a value of 1 if it is successful, otherwise 0 is returned.

```
DB2 Image disable ( ) returns #2
; disables the current database
DB2 Image disable ('table1') returns #2
; disables table1 in the current database
DB2 Image disable ('table1', 'column1') returns #2
; disables column1 in table1 in the current database
```

DB2 Image enable

Reversible: NO   **Flag affected:**   NO

Parameters: Table name
                  Column name
                  Logon switch to remain logged on to database
                  Return value

Syntax: DB2 Image enable ([*tablename*][,*columnname*][,*/L'*]) returns
              [*return-value*]

This command enables the Image extender data type for the current DB2 database, or the specified table or column in the current database. To enable a database you do not need to pass any parameters, the currently connected database is used. To enable a table in the currently logged on database, you need to pass the *tablename* only. To enable a column in the current database, you need to pass the *tablename* and *columnname* parameters. You can specify the L switch to remain logged on to the current database, otherwise you are logged off automatically. This command returns a value of 1 if it is successful, otherwise 0 is returned.

```
DB2 Image enable ( ) returns #2
; enables the current database
DB2 Image enable ('table1') returns #2
; enables table1 in the current database
DB2 Image enable ('table1', 'column1') returns #2
; enables column1 in table1 in the current database
```

DB2 Image is enabled

Reversible:	NO	**Flag affected:**	NO

Parameters: Table name
Column name
Status variable
Logon switch to remain logged on to database
Return value

Syntax: DB2 Image is enabled ([*tablename*][,*columnname*], **status** [,*⁄L*])
        returns [*return-value*]

This command checks whether or not the Image extender data type is enabled for the current DB2 database, or the specified table or column in the current database. To check whether or not a database is enabled for Image, you need to pass the status field only. To check whether or not a table in the current database is enabled for Image, you need to pass the *tablename*, as well as the status field. To check whether or not a column is enabled for Image, you need to pass the *tablename* and *columnname* parameters, as well as the status field. The *status* parameter returns kTrue or 1 if the database, table, or column is enabled for Image. You can specify the L switch to remain logged on to the current database, otherwise you are logged off automatically. This command returns a value of 1 if it is successful, otherwise 0 is returned, regardless of the value returned in the status field.

```
DB2 Image Is Enabled ( , , #1 ) returns #2
; checks the current database
DB2 Image Is Enabled ('table1', , #1 ) returns #2
; checks table1 in the current database
DB2 Image Is Enabled ('table1', 'column1', #1 ) returns #2
; checks column1 in table1 in the current database
```

DB2 Init upload

Reversible:	NO	**Flag affected:**	NO

Parameters: Path to OMNIS executable
Logon switch to remain logged on to database

Syntax: Db2 Init upload (*path* [, *⁄L*])

This command prepares the current DB2 database to receive the *DB2 Upload Data* command. You must specify the *path* to the OMNIS executable as the first parameter. You can specify the L switch to remain logged on to the database, otherwise you are logged off automatically. This command uses the upload.bnd file which must be located in the EXTERNAL folder.

```
Db2 Init upload (sys(115),'/L')
```

DB2 Register error vars

Reversible: NO **Flag affected:** NO

Parameters: Error code fieldname
Error text fieldname

Syntax: DB2 Register error vars (***errorcode, errortext***)

This command specifies the variables to contain any errors reported while the DB2 external commands are in operation. You must specify suitable variables for *errorcode* and *errortext* to contain the code and text for any errors.

```
; declare errorcode of Long int type, and errortext as Character
DB2 Register error vars (errorcode, errortext)
```

DB2 Register logon info

Reversible: NO **Flag affected:** NO

Parameters: Database or tablespace name
Username for the specified database
Password for the specified database

Syntax: DB2 Register logon info (***tablespace***, ***username***, ***password***)

This command registers the logon info to be used when logging on to the DB2 database. The command requires the *database* name, *username*, and *password* of the required DB2 database. The logon info specified in this command overrides the information contained in the DB2 DAM as specified in the current session, if bound.

DB2 Unregister logon info

Reversible: NO **Flag affected:** NO

Parameters: None

Syntax: DB2 Unregister logon info ()

This command unregisters the logon info for the current DB2 database. In this case the logon info contained in the DB2 DAM is used, if bound.

DB2 Upload data

Reversible: NO **Flag affected:** NO

Parameters: SQL statement containing data
Logon switch to remain logged on to database

Syntax: Db2 Upload data (*sql-statement* [, */L'*])

This command uploads data to the current DB2 database. It takes a SQL statement containing a user defined function (UDF) and uploads the specified data in a bind variable. You can specify the L switch to remain logged on to the database, otherwise you are logged off automatically.

The bind variable must be an updateable bind variable (:var) and of type binary. In addition, the bind variable must be cast explicitly as a blob in the SQL statement. The command takes one :var variable only and does not support @[var] bind variables.

```
Db2 Upload data ("INSERT INTO my_table (picture) values
   (DB2Image(CURRENT SERVER, CAST(:my_bin as BLOB(2M)), 'BMP',
   CAST(NULL as LONG VARCHAR), 'comment'))")
```

DB2 Video disable

Reversible: NO **Flag affected:** NO

Parameters: Table name
Column name
Logon switch to remain logged on to database
Return value

Syntax: DB2 Video disable ([*tablename*][,*columnname*][,*/L'*]) returns
[*return-value*]

This command disables the Video extender data type for the current DB2 database, or the specified table or column in the current database. To disable a database you do not need to pass any parameters, the currently connected database is used. To disable a table in the currently logged on database, you need to pass the *tablename* only. To disable a column in the current database, you need to pass the *tablename* and *columnname* parameters. You can specify the L switch to remain logged on to the current database, otherwise you are logged off automatically. This command returns a value of 1 if it is successful, otherwise 0 is returned.

```
DB2 Video disable ( ) returns #2
; disables the current database
DB2 Video disable ('table1') returns #2
; disables table1 in the current database
DB2 Video disable ('table1', 'column1') returns #2
; disables column1 in table1 in the current database
```

DB2 Video enable

Reversible:	NO	**Flag affected:**	NO

Parameters: Table name
Column name
Logon switch to remain logged on to database
Return value

Syntax: DB2 Video enable ([*tablename*][,*columnname*][,*/L'*) returns
[*return-value*]

This command enables the Video extender data type for the current DB2 database, or the specified table or column in the current database. To enable a database you do not need to pass any parameters, the currently connected database is used. To enable a table in the currently logged on database, you need to pass the *tablename* only. To enable a column in the current database, you need to pass the *tablename* and *columnname* parameters. You can specify the L switch to remain logged on to the current database, otherwise you are logged off automatically. This command returns a value of 1 if it is successful, otherwise 0 is returned.

```
DB2 Video enable ( ) returns #2
; enables the current database
DB2 Video enable ('table1') returns #2
; enables table1 in the current database
DB2 Video enable ('table1', 'column1') returns #2
; enables column1 in table1 in the current database
```

DB2 Video is enabled

Reversible: NO   **Flag affected:**   NO

Parameters: Table name
Column name
Status variable
Logon switch to remain logged on to database
Return value

Syntax: DB2 Video is enabled ([*tablename*][,*columnname*], **status** [,*⁄L′*])
returns [*return-value*]

This command checks whether or not the Video extender data type is enabled for the current DB2 database, or the specified table or column in the current database. To check whether or not a database is enabled for Video, you need to pass the status field only. To check whether or not a table in the current database is enabled for Video, you need to pass the *tablename*, as well as the status field. To check whether or not a column is enabled for Video, you need to pass the *tablename* and *columnname* parameters, as well as the status field. The *status* parameter returns kTrue or 1 if the database, table, or column is enabled for Video. You can specify the L switch to remain logged on to the current database, otherwise you are logged off automatically. This command returns a value of 1 if it is successful, otherwise 0 is returned, regardless of the value returned in the status field.

```
DB2 Video Is Enabled ( , , #1 ) returns #2
; checks the current database
DB2 Video Is Enabled ('table1', , #1 ) returns #2
; checks table1 in the current database
DB2 Video Is Enabled ('table1', 'column1', #1 ) returns #2
; checks column1 in table1 in the current database
```

Delete file

Reversible: NO   **Flag affected:**   NO

Parameters: Path of file to be deleted
Return field

Syntax: Delete file *(path)* returns *return-field*

This command deletes the file specified in *path*. Under MacOS, files deleted with *Delete file* are not moved into the Trash. You cannot recover deleted files except with advanced disk utilities such as Norton Utilities.

It returns any error code (shown at the end of this chapter), or zero if none.

Does file exist

Reversible: NO   **Flag affected:**   NO

Parameters: File or folder name (including full path)
                      Return field

Syntax: Does file exist (*file*|*folder-name)* returns ***return-field***

This command returns kTrue if the specified file or folder exists, otherwise it returns kFalse. The file or folder name must include the full path.

```
; Windows
Does file exist ("C:\C700\FileOps\FileOps.C")   ;; test for file
If flag true
   ; do this
Does file exist ("C:\C700")   ;; test for folder
; Macintosh
Does file exist ("HD:Desktop Folder:MyPictureFile") ;; test for file
Does file exist ("HD:Microsoft")   ;; test for folder
```

FTPChmod

Reversible: NO   **Flag affected:**   NO

Parameters: Socket, Filename, Mode
Returns: *Status* (0 if no error, -1 or other negative number if error)

Syntax: FTPChmod(*Socket*, *Filename*, *Mode*

FTPChmod changes the protection mode of a remote file on the connected FTP server.

Socket is an OMNIS Integer field containing the socket previously opened to an FTP server with FTPConnect.

Filename is an OMNIS Character field containing the name of the remote file, by default in the current directory. If the server permits, he filename can be a fully qualified pathname in another directory.

Mode is an OMNIS Character field containing the system-dependent file-protection specifier to apply to the named file. Many FTP daemons accept the Unix-style Owner/Group/World 3-digit Read/Write/Execute scheme (for example, 754 = Owner Read/Write/Execute, Group Read/Execute World Read-Only). Consult the documentation for the remote system to determine the acceptable syntax for this argument.

Status is an OMNIS Long Integer field that contains 0 (zero) if no error occurs. To handle an error, use the FTPGetLastStatus command to get the code.

Using WebDevError, one or more callback methods return error messages and codes.

FTPConnect

Reversible:	NO	**Flag affected:**	NO

Parameters: ServerAddr, Username, Password

Returns: *Socket*

Syntax: FTPConnect(*ServerAddr*,*Username*,*Password*) Returns *Socket*

FTPConnect creates a new socket open to the FTP service or port on a named server or IP address.

ServerAddr is an OMNIS Character field containing the hostname or IP address of the FTP server to which the socket connects.

Username is an OMNIS Character field containing the user ID of the account that will be used for access on the server.

Password is an OMNIS Character field containing the password of the account that will be used for access on the server.

Socket is an OMNIS Long Integer field containing the number of the allocated socket. Error codes are socket numbers less than 0 (zero), shown at the end of this chapter. To get the actual error code, call FTPGetLastStatus.

A WebDevError callback method returns error messages and codes.

FTPCwd

Reversible:	NO	**Flag affected:**	NO

Parameters: Socket, Directory

Returns: *Status* (0 if no error, -1 or other negative number if error)

Syntax: FTPCwd(*Socket,NewDir*)

FTPCwd changes the working directory on the connected FTP server. The working directory is the one for which the FTPList command shows a directory listing. Files are transferred to and from this remote directory.

Socket is an OMNIS Integer field containing the number of a socket open to an FTP server.

NewDir is an OMNIS Character field containing the directory specification to change the remote server's current directory. The contents of this string are system-dependent. FTPCwd accepts anything for this argument, but the remote FTP daemon may not. Most FTP daemons accept Unix-style path and file specifications with path and file separated by slashes, such as

/drive/user/subdirectory/filename.extension

Most FTP daemons accept the Unix conventions for abbreviations for special directory specifications, that is, ".." for the next higher sub-directory, and "~userid" for the home directory of a particular user ID.

Some FTP daemons also accept system-specific directory path formats, that is, Macintosh colon-separated as in Macintosh HD:My Folder:My File or VMS-style path and file specifications, as in SOME$DISK:[USER.SUBDIRECTORY]FILENAME.EXTENSION;1.

Consult the documentation for the server to determine the authoritative acceptable directory path specifications. When in doubt, try the Unix style.

Status is an OMNIS Long Integer field that returns a negative number if an error is encountered, or 0 (zero) otherwise. To get the actual error code, call FTPGetLastStatus.

Using WebDevError, one or more callback methods return error messages and codes.

FTPDelete

Reversible:	NO	**Flag affected:**	NO

Parameters: Socket, Filename

Returns: *Status* (0 if no error, -1 or other negative number if error)

Syntax: FTPDelete(*Socket*, *Filename*) Returns *Status*

FTPDelete deletes a remote file on the connected FTP server.

Socket is an OMNIS Integer field containing the number of a socket that is open to an FTP server.

Filename is an OMNIS Character field containing the name of the remote file to delete, by default in the current directory. If the server permits, the filename can be a fully qualified pathname in another directory.

Status is an OMNIS Long Integer field that returns a negative number if an error is encountered, or 0 (zero) otherwise. To get the actual error code, call FTPGetLastStatus.

Using WebDevError, one or more callback methods return error messages and codes.

FTPDisconnect

Reversible:	NO	**Flag affected:**	NO

Parameters: Socket
Returns: *Status* (0 if no error, -1 or other negative number if error)
Syntax: FTPDisconnect(*Socket)*

FTPDisconnect disconnects a socket from the remote FTP daemon.

Socket is an OMNIS Integer field containing the number of a socket that is open on an FTP server.

Status is an OMNIS Long Integer field that returns a negative number if an error is encountered, or 0 (zero) otherwise. To get the actual error code, call FTPGetLastStatus.

Using WebDevError, one or more callback methods return error messages and codes.

FTPGet

Reversible:	NO	**Flag affected:**	NO

Parameters: Socket, RemoteFile, LocalFile
Returns: *Status* (0 if no error, -1 or other negative number if error)
Syntax: FTPGet(*Socket,RemoteFile,LocalFile*)

FTPGet initiates transfer of a file from an FTP server to a file on the local client. The file is transferred according to the currently set transfer type of ASCII or binary as specified by the FTPType command.

Socket is an OMNIS Integer field containing the number of a socket that is open on the server.

RemoteFile is an OMNIS Character field containing the name of the file on the remote system to transfer to the local client.

Note: The remote filename may not be acceptable to the local system. The file is transferred according to the current transfer type of ASCII or binary, as specified by the FTPType external command. Binary files such as executables, pictures, and archives are not transferred properly in ASCII mode.

LocalFile is an OMNIS Character field containing the specification of the file on the local machine to receive the contents of the remote file.

Status is an OMNIS Long Integer field that returns a negative number if an error is encountered, or 0 (zero) otherwise. To get the actual error code, call FTPGetLastStatus.

Using WebDevError, one or more callback methods return error messages and codes.

FTPGetBinary

Reversible: NO   **Flag affected:**   NO

Parameters: Socket, RemoteFile, BinField

Returns: *Status* (0 if no error, -1 or other negative number if error)

Syntax: FTPGetBinary(*Socket,RemoteFile,BinField*)

FTPGetBinary initiates transfer of a file from an FTP server directly to an OMNIS binary variable. The file is transferred according to the currently set transfer type of ASCII or binary as specified by the FTPType command.

Socket is an OMNIS Integer field containing the number of a socket that is open on a remote FTP server.

RemoteFile is an OMNIS Character field containing the name of the file on the remote system to transfer to the local client.

BinField is an OMNIS Binary field that will receive the contents of the remote file.

Status is an OMNIS Long Integer field that returns a negative number if an error is encountered, or 0 (zero) otherwise. To get the actual error code, call FTPGetLastStatus.

Using WebDevError, one or more callback methods return error messages and codes.

Note: The file is transferred according to the current transfer type of ASCII or binary as specified by the FTPType external command. Binary files such as executables, pictures, and archives are not transferred properly in ASCII mode.

FTPGetLastStatus

Reversible:	NO	**Flag affected:**	NO

Parameters: Socket
Returns: *Status* (error code)

Syntax: FTPGetLastStatus(*Socket*)

Because FTP commands return a negative number (usually -1), rather than an error code and message, you can call FTPGetLastStatus to return one of the error codes listed below. FTPGetLastStatus indicates the most recent status from an FTP operation. It is generally used during development, while WebDevError is used in applications.

Note: FTPGetLastStatus errors are redundant with those returned by WebDevError. However, Web Enabler release 2.0 retains the FTPGetLastStatus function so that release 1.0 applications will not require modification.

Code	Meaning
1	Attempt to connect to server failed (FTPConnect, FTPGet, FTPPut, FTPGetBinary, FTPPutBinary, FTPList)
2	Connection lost
3	Invalid username or password
4	No such file
5	Invalid argument
6	No free sockets (too many connections)
7	No such server (DNS failed)
8	Client configuration error (can't get local IP address)
9	Server protocol error - server response unexpected
10	Client file I/O error (disk full, network volume dismounted, and so on)
11	Out of memory error (common in FTPGetBinary/FTPPutBinary)
12	User cancel (progress method returned flag false)

Socket is an OMNIS Integer field containing the number of a socket that is open for the operation.

Using WebDevError, one or more callback methods return these and other error messages and codes. FTPGetLastStatus returns errors after the callback method.

FTPList

Reversible:	NO	**Flag affected:**	NO

Parameters: Socket, List, Pathname, Mode

Returns: *Status* (0 if no error, -1 or other negative number if error)

Syntax: FTPList(*Socket*,*List*[,*Pathname*[,*Mode*]])

FTPList gets an OMNIS list of file information from the current directory on the remote server.

Socket is an OMNIS Integer field containing the number of a socket that is open on a remote FTP server.

List is an OMNIS List field containing a single column of type Character. This list receives the file listing information, one line per file, returned by the remote FTP daemon. The list is dependent on the type of the remote server and may be a long or short format, depending on the setting.

Note: Very often, FTP daemons return long-format listings in a Unix file listing format. At a minimum, this file information contains the filename, but usually includes other information. The OMNIS method must parse this information to find the filename and other information. For example

ListItem					
total 123					
drwxr-xr-x	4	userid	mygroup	Jan 1 1999	.
drwxr-xr-x	6	root	root	Jan 1 1999	..
-rw-------	1	userid	mygroup	Jan 16 1998	myfile
-rw-r—r—	2	userid	mygroup	Jan 16 1998	myotherfile

Where the columns in the character string correspond to protection, file size, username and group of the file owner, the date last modified and the name of the file. The files "." and ".." represent the current and parent directories, respectively, which may neither be retrieved nor changed.

The file information may not be neatly spaced into columns as in this example. Columns are separated with one or more spacing characters (space, tab, and so on).

Pathname is an optional parameter specifying an OMNIS Character field that contains a pathname or wildcard specification for the files to include in the listing.

Mode is an optional parameter specifying an OMNIS Integer field containing a code that indicates whether the server should return a short or long format listing:

Chapter 6—External Commands

Code	Meaning
0	Filename-only listing
1	Long-format listing

Status is an OMNIS Long Integer field that returns a negative number if an error is encountered, or 0 (zero) otherwise. To get the actual error code, call FTPGetLastStatus.

Using WebDevError, one or more callback methods return error messages and codes.

FTPMkdir

Reversible: NO   **Flag affected:**   NO

Parameters: Socket, DirName

Returns: *Status* (0 if no error, -1 or other negative number if error)

Syntax: FTPMkdir(*Socket,DirName*)

FTPMkdir creates a new subdirectory on the remote system.

Socket is an OMNIS Integer field containing a socket open to a remote FTP server.

DirName is an OMNIS Character field containing the name of the new directory to create on the server in the current directory. By default, the current directory is as specified by the external command FTPConnect or FTPCwd and may be determined by the FTPPwd external command.

Note: The name of the new directory must follow the convention and file-naming rules of the remote system. Not all users will have permissions to create new subdirectories on arbitrary directories on the remote system. Default file-access permissions apply to the new directory. You may need to use the FTPCwd external command so that files are subsequently transferred to the new directory.

Status is an OMNIS Integer field that returns a negative number if an error is encountered, or 0 (zero) otherwise. To get the actual error code, call FTPGetLastStatus.

Using WebDevError, one or more callback methods return error messages and codes.

FTPPut

Reversible: NO   **Flag affected:**   NO

Parameters: Socket, RemoteFile, LocalFile

Returns: *Status* (0 if no error, -1 or other negative number if error)

Syntax: FTPPut(*Socket,RemoteFile,LocalFile*)

FTPPut initiates transfer of a file to an FTP server from a file on the local client. The file is transferred according to the currently set transfer type of ASCII or binary as specified by the FTPType external command.

Socket is an OMNIS Integer field containing the number of a socket that is open on a remote FTP server.

RemoteFile is an OMNIS Character field containing the name of the file on the remote machine to receive the contents of the local file.

LocalFile is an OMNIS Character field containing the name of the file on the local machine to send the contents of the remote file.

Status is an OMNIS Long Integer field that returns a negative number if an error is encountered, or 0 (zero) otherwise. To get the actual error code, call FTPGetLastStatus.

Using WebDevError, one or more callback methods return error messages and codes.

Note: The local filename may not be acceptable to the remote system. The file is transferred according to the currently set transfer type of ASCII or binary as specified by the FTPType external command. Binary files such as executables, pictures, archives are not transferred properly in ASCII mode. The permission mode of the current remote directory may not allow the creation of files by the username used in FTPConnect. You may not overwrite a read-only or read/execute file, or a directory.

FTPPutBinary

Reversible: NO   **Flag affected:**   NO

Parameters: Socket, BinField, RemoteFile

Returns: *Status* (0 if no error, -1 or other negative number if error)

Syntax: FTPPutBinary(*Socket,BinField,RemoteFile*)

FTPPutBinary initiates transfer of a file to an FTP server from an OMNIS binary variable. The file is transferred according to the currently set transfer type of ASCII or binary as specified by the FTPType external command.

Socket is an OMNIS Character field containing the number of a socket that is open on a remote FTP server.

BinField is an OMNIS Binary field to be sent to the remote file.

RemoteFile is an OMNIS Character field containing the name of the file on the remote system to receive the contents of the OMNIS Binary field.

Status is an OMNIS Long Integer field that returns a negative number if an error is encountered, or 0 (zero) otherwise. To get the actual error code, call FTPGetLastStatus.

Using WebDevError, one or more callback methods return error messages and codes.

Note: The local filename may not be acceptable to the remote system. The file is transferred according to the currently set transfer type of ASCII or binary as specified by the FTPType external command. Binary files such as executables, pictures, archives are not transferred properly in ASCII mode. The permission mode of the current remote directory may not allow the creation of files by the username used in FTPConnect. You may not overwrite a read-only or read/execute file, or a directory.

FTPPwd

Reversible:	NO	**Flag affected:**	NO
Parameters:	Socket		
Returns:	ServerDir (pathname if no error, -1 or other negative number if error)		
Syntax:	FTPPwd(*Socket*)		

FTPPwd gets the name of the remote server's current directory.

Socket is an OMNIS Integer field containing the number of a socket that is open on a remote FTP server.

ServerDir is an OMNIS Character field that returns the path specification of the current remote directory on the server. A NULL string indicates that an error occurred. Call FTPGetLastStatus for the error code.

Using WebDevError, one or more callback methods return error messages and codes.

Note: The value returned depends upon the operating system of the remote server. Many FTP daemons return a Unix-style path specification, but do not assume that this is the case.

FTPReceiveCommandReplyLine

Reversible: NO   **Flag affected:** NO

Parameters: Socket number
Returns: Reply
Syntax: FTPReceiveCommandReplyLine(*Socket*) Returns *Reply*

FTPReceiveCommandReplyLine returns the next line of the reply following an
FTPSendCommand. You have to determine if the reply is multi-line, and if so issue further
receive commands to get the remainder of the reply. FTPReceiveCommandReplyLine will
timeout after 60 seconds if it does not receive a reply.

Socket is an OMNIS Integer variable containing a socket open to a remote FTP server.

Reply is an OMNIS Character variable containing the reply from the server.

```
FTPSendCommand(lvSocket,'pwd') Returns #1
FTPReceiveCommandReplyLine(lvSocket) Returns lvReply
; might return the string
257 "/vol1/ftp/omnis/" is current directory
```

FTPRename

Reversible: NO   **Flag affected:**   NO

Parameters: Socket, OldName, NewName
Returns: *Status* (0 if no error, -1 or other negative number if error)
Syntax: FTPRename(*Socket,OldName,NewName*

FTPRename renames a remote file.

Socket is an OMNIS Integer field containing the number of a socket that is open on a
remote FTP server.

OldName is an OMNIS Character field containing the name of the file to change on the
remote server. By default, the file is assumed to be in the current remote directory as set at
connection or by the external command FTPCwd. You may specify a path in a different
directory, as long as it is correct and you have permissions in that directory.

NewName is an OMNIS Character field containing the new name for the file on the remote
server. By default, the file is renamed in place in the current remote directory as set at
connection or by the external command FTPCwd. You may specify a path and filename in a
different directory. In such a case on many systems, the file is moved to the new directory
path, as long as the path name is correct and you have permissions in the other directory.

Status is an OMNIS Long Integer field that returns a negative number if an error is
encountered, or 0 (zero) otherwise. To get the actual error code, call FTPGetLastStatus.

Using WebDevError, one or more callback methods return error messages and codes.

Note: Local filename conventions may not be acceptable to the remote system. The permission mode of the current remote directory may not allow files to be renamed. You may not change a read-only or read/execute file, or rename a file to the same name as a directory.

FTPSendCommand

Reversible:	NO	**Flag affected:**	NO

Parameters:	Socket number
	Command
Returns:	Status (0 if no error, -1 or other negative number if error)
Syntax:	FTPSendCommand(*Socket,Command*) Returns *Status*

FTPSendCommand sends a command to the remote server.

Socket is an OMNIS Integer variable containing a socket open to a remote FTP server.

Command is an OMNIS Character variable or quoted literal containing the command and its parameters.

Status is an OMNIS Long Integer variable that returns a negative number if an error is encountered, or 0 (zero) otherwise. Using WebDevError, one or more callback methods return error messages and codes.

```
FTPSendCommand(lvSocket,'pwd') Returns #1
FTPReceiveCommandReply(lvSocket) Returns lvReply
; might return the string
257 "/vol1/ftp/omnis/" is current directory
```

FTPSetProgressProc

Reversible: NO   **Flag affected:**   NO

Parameters: Proc

Returns: Invokes method if no error or returns -1 or other negative number if
error

Syntax: FTPSetProgressProc(*Proc)*

FTPSetProgressProc provides a mechanism to provide progress messages during an FTP operation (FTPGet, for example).

Proc is an OMNIS Character field containing an address for an OMNIS method to be called with progress status messages. The method can be used to display the message, log it, or otherwise change normal execution. For example: MYCODE, MYCODE/MYPROC, MYLIBRARY.MYCODE. You should use the method name qualified by the library if your applications are in a multi-library environment.

An example method might look like this:

```
; Parameter messageText (Character   10000000)
; Display a working message while FTP operation is in progress.
Working message (High position,Large size) {[messageText]}
```

Using WebDevError, one or more callback methods return error messages and codes.

FTPSite

Reversible: NO   **Flag affected:**   NO

Parameters: Socket number
                     Command parameters

Returns: Status (0 if no error, -1 or other negative number if error)

Syntax: FTPSite(*Socket,Parameters*) Returns *Status*

FTPSite issues a host specific command to the remote server.

Socket is an OMNIS Integer variable containing a socket open to a remote FTP server.

Parameters is an OMNIS Character variable or quoted literal containing the host specific command and its parameters.

Status is an OMNIS Long Integer variable that returns a negative number if an error is encountered, or 0 (zero) otherwise.

Using WebDevError, one or more callback methods return error messages and codes.

```
FTPSite(lvSocketNum,"FILETYPE=JES") Returns lvStatus
; issues the FTP command SITE FILETYPE=JES
```

FTPType

Reversible:	NO	**Flag affected:**	NO

Parameters: Socket, FileType

Returns: *Status* (0 if no error, -1 or other negative number if error)

Syntax: FTPType(*Socket*,*FileType*)

FTPType specifies the type of transfer as ASCII or binary. In ASCII mode, line separators and other text formatting characters can be changed to the characters required by the local or remote system. In binary mode, line separators and other text formatting characters are not changed. If the information to be transferred is not text, use FTPType to change the transfer mode to binary. Otherwise, binary files such as archives, images, OMNIS Libraries, and executable files may be corrupted by the processing of bytes that coincide with text-formatting characters.

Socket is an OMNIS Integer field containing a socket open to a remote FTP server.

FileType is an OMNIS Boolean field indicating the type of subsequent transfers on this socket.

Value	Transfer Mode
kFalse/Zero	ASCII
kTrue/One	Binary

Status is an OMNIS Long Integer field that returns a negative number if an error is encountered, or 0 (zero) otherwise. To get the actual error code, call FTPGetLastStatus.

Using WebDevError, one or more callback methods return error messages and codes.

Get file info

| **Reversible:** | NO | **Flag affected:** | NO |

Parameters: Path
File type (the file extension under Windows)
Creator (the pathname of the executable under Windows,
provided the extension is registered with Windows)
Logical size (number of bytes in file)
Physical size (number of bytes file occupies on disk; same as
logical size under Windows)
Creation date (not stored under Windows)
Creation time (not stored under Windows)
Modified date
Modified time
Return field

Syntax: Get file info (***path, type, creator, logical-size, physical-size,***
   creation-date, creation-time, modified-date, modified-time)
   returns ***return-field***

This command returns information about the file specified in *path*.

A file may occupy more physical disk space than is necessary, because disk space is usually allocated in blocks of some fixed size. This is why the logical and physical sizes can be different.

Windows (DOS) does not store the creation date and time. These are therefore the same as the modified date and time. Almost all Windows FileOps commands will take wild-cards as arguments, where the MacOS will not.

It returns any error code (shown at the end of this chapter), or zero if none.

Get file name

Reversible:	NO	**Flag affected:**	NO

Parameters: Path of file selected
                  Dialog title
                  File type or list of file types
                  Return field

Syntax: Get file name (***path***[,*dialog-title*][,*file-type*]...*)
             returns ***return-field***

This command prompts the user to open a file with the specified *file-type* and *path*; it opens the standard Open dialog for the current Operating System. Also you can specify a *dialog-title* for the Open dialog. The optional *file-type* parameter limits the choice of file types available. It returns the full pathname of the file the user selects in *path*, or remains empty if no file is selected (that is, the Cancel button was clicked). The selected file is not opened.

It returns any error code (shown at the end of this chapter), or zero if none.

Windows file types

Under Windows (DOS) files do not have type codes, but they do have extensions which serve the same purpose. You can specify one or more extensions (using wildcard patterns like those used in many DOS commands) separated by semicolons. For example, "*.TXT" would specify text files only.

MacOS file types

Under MacOS file types are four-character codes defined by convention (OMNIS library files are type "O7$A", for example). You can use ResEdit, DiskTop, or other such tools to discover file types. For example, "TEXT" would specify text files only.

```
Switch sys(6) = 'M'
   Case kTrue   ;; if MacOS
      Get file name (PATH, 'Select a file', 'TEXT ttro')
   Default
      Get file name (PATH, 'Select a file', '*.TXT;*.DOC')
End Switch
```

Get file read-only attribute

Reversible: NO   **Flag affected:** NO

Parameters: Path of the file
                    Read-flag setting returned
                    Return field

Syntax: Get file read-only attribute (*path, read-flag*) returns [*return-field*]

This command returns the current read-only attribute of the file specified in *path*. If the *read-flag* parameter returns kTrue the file is read-only, otherwise if kFalse is returned the file is read/write. Note that read-only status is the same as locked under MacOS.

It returns any error code (shown at the end of this chapter), or zero if none.

Get files

Reversible: NO   **Flag affected:** NO

Parameters: List name
                    First column of list
                    Path name
                    File type
                    Creator type (MacOS only)
                    Return field

Syntax: Get files (*list-name, first-column, path-name, file-type*[,*creator-type*])
                    returns *return-field*

This command lists all the files of a specified type in a directory or folder. The list is specified by *list-name* which must have at least one column defined in *first-column*. This column will hold the file name of the files with the specified *file-type* found under the specified *path-name*, including the extension for DOS machines. For *file-type* you can use wildcards, such as '*.LBR'. Under MacOS the *creator-type* can be specified.

It returns any error code (shown at the end of this chapter), or zero if none.

The following example uses *Get files* to build a list of all the libraries in the folder returned by sys(10). Under MacOS, you can select libraries using the file type OO$A, and '*.LBR' for Windows.

```
; Declare local vars LVFILELIST, LVPATHNAME, LVDRIVE, LVDIR,
; LVFILENAME, LVEXT, LVFILETYPE, LVCREATORTYPE
Set current list LVFILELIST
Define list {LVFILENAME}
Calculate LVPATHNAME as sys(10)    ;; path of current library
Split path name(LVPATHNAME,LVDRIVE,LVDIR,LVFILENAME,LVEXT)
Calculate LVPATHNAME as con(LVDRIVE,LVDIR)
If sys(6)='M'        ;; under MacOS
   Calculate LVFILETYPE as 'OO$A'
Else                 ;; else, if on any other platform
   Calculate LVFILETYPE as '*.LBR'
End If
Get files (LVFILELIST,LVFILENAME,LVPATHNAME,LVFILETYPE)
```

Get folders

Reversible: NO   **Flag affected:**   NO

Parameters: List name
                  Column name
                  Path
                  Return field

Syntax: Get folders (*list,column,'path'*) returns *return-field*

This command creates a list of folders for the specified *path,* and places the list in the specified *column* of the specified *list* (you can use any column in the list).

It returns any error code (shown at the end of this chapter), or zero if none.

For example, to get a list of the folders in the root of your Mac or PC use the following method

```
Do LIST1.$define(COL1,COL2,COL3)
If sys(6) = 'M'
   Get folders (LIST1,COL2,'Macintosh HD')
Else
   Get folders (LIST1,COL2,'C:\')
End If
Do LIST1.$sort(COL2)
Redraw lists       ;; if LIST1 is a window list
```

HTTPClose

Reversible:	NO	**Flag affected:**	NO

Parameters: Socket
Returns: *Status* (0 if no error, -1 or other negative number if error)
Syntax: HTTPClose(*Socket*) Returns *Status*

HTTPClose is a client or server command that closes a socket that OMNIS is using for communication with a Web server or client, functionally equivalent to TCPClose. You must use HTTPClose when you have finished using a socket.

Socket is an OMNIS Long Integer field containing a socket previously opened. It can close any socket, not just HTTP-related sockets.

WinSOCK error codes are returned as negative values, shown at the end of this chapter. Using WebDevError, one or more callback methods return error messages and codes.

HTTPGet

Reversible:	NO	**Flag affected:**	NO

Parameters: Hostname, URI, CGIList, HeaderList, Port
Returns: *ConnectedSocket*
Syntax: HTTPGet(*Hostname,URI* [,*CGIList*[,*HeaderList*[,*Port*]]])

HTTPGet is a client command that submits a GET-method CGI request to a Web server.

Note: HTTPPage allows you to get HTML text source through a server, transparently and without additional coding. If you need to customize the process for a proxy server, you can use a combination of HTTPGet and TCPReceive. For this technique, see the sample code in "Accessing a Proxy Server".

Hostname is a Character field containing the hostname of a Web server to which to connect.

URI is a Character field containing the path and the name of the CGI to be run on the Web server. Often this can be determined by looking at the source to an HTML page that requests the CGI.

CGIList is an optional parameter specifying an OMNIS list defined to have two character columns. The list contains information to be sent as the arguments of the CGI. There is one row for each field passed to the GET CGI method. In this way, an OMNIS method can send OMNIS field values to a Web server. For example

Attribute	Value
Name	John Smith
City	Podunk
Alive	On
Submit	Please

Note: Before the values are sent to the Web server, HTTPGet automatically performs any CGI encoding required to pass special characters in the arguments. There is no need to call the CGIEncode external command to encode the value entries in the list.

HeaderList is an optional parameter specifying an OMNIS list field defined to have two character columns. The list contains information added to the HTTP message header as attribute/value pairs on each row of the list. There is one row for each item found on the header.

For example, after the call, the list might contain entries such as:

Attribute	Value
Accept	/
Content-type	text/html

Port is an optional field that includes the port number of the server.

The return value is a positive integer socket number opened to the Web server as a result of the GET CGI. This allows OMNIS to read the results of the CGI request on this socket. In the case of an error, a value of -1 (minus one) is returned for the socket number.

Colons are added to the attributes when HTTPGet constructs the header. Do not end attribute names with a colon. HTTPGet adds the following header fields by default:

Attribute	Value
Date	The current GMT date and time in HTTP header format
Server	OMNIS7/3.5
MIME-Version	1.0

Errors in parsing the message header are reported through the standard WebDevError mechanism.

HTTPHeader

Reversible:	NO	**Flag affected:**	NO

Parameters: Socket, Status, HeaderList

Returns: *Length*

Syntax: HTTPHeader(*Socket,Status,HeaderList*)

HTTPHeader is a server command that sends an HTTP standard header back to an HTTP client, for example, an OMNIS application or a Web browser. HTTP headers are normally hidden from Web clients, but convey very useful information regarding the status and contents of the Web page. An OMNIS method must send a header back to a connected Web browser in order to have results properly displayed.

Socket is an OMNIS Long Integer field containing the number of a socket that has already been opened for a TCP/IP client, usually a Web browser or OMNIS application that requires and can understand an HTTP header message.

Status is an OMNIS Long Integer field containing an HTTP status code. The status code may change the way in which any following HTML or other information displays on the Web browser. Some common codes:

Code	Meaning
200	The request was completed successfully
201	The request was a POST method and was completed successfully. Data was sent to the server, and a new resource was created as a result of the request.
202	A GET method returned only partial results.
204	The request was completed successfully, but there is no new information. The browser will continue to display the document from which the request originated.
304	The GET request included a header with an If-Modified-Since field. However, the server found that the data requested had not been modified since the date in this field. The document was not resent (the Web browser will probably display it from cache).
400	The request syntax was wrong
401	The request requires an Authorization field but the client did not specify one. Usually results in a username and password to be displayed
404	The request URL could not be found.
500	The server has encountered an internal error and cannot continue with the request.
501	The server does not support this method

HeaderList is an OMNIS list defined to have two character columns. The list contains information to be included in the HTTP message header as attribute/value pairs on each row of the list. There is one row for each item in the header.

At a minimum, for OMNIS to return normal Web-page HTML text to the client, you should send a header containing the line:

Attribute	Value
Content-type	text/html

HTTPHeader automatically includes the following lines in all HTTP response headers:

Attribute	Value
Date	The current GMT date and time in HTTP header format
Server	OMNIS7/3.5
MIME-version	1.0

Length is an OMNIS Long Integer field containing the number of characters sent.

Standard WinSOCK and Web Enabler errors are reported using WebDevError.

HTTPOpen

Reversible:	NO	**Flag affected:**	NO	

Parameters: Hostname, Port
Returns: *Socket*
Syntax: HTTPOpen(*Hostname*[*Port*])

HTTPOpen is a client or server command that opens a socket to a Web server.

Hostname is a Character field containing the IP address or domain name of an HTTP server that accepts HTTP requests from an OMNIS client. For example:

host.myhost.com or 255.255.255.254

Port is an optional field that specifies the local port to use for the socket.

Socket returns a positive number indicating the socket number to which the command attached. If an error is raised, a negative error number is returned in *Socket*. WinSOCK error codes are returned instead of a valid socket number. Error codes are numbers less than 0 (zero), shown at the end of this chapter.

Using WebDevError, one or more callback methods return error messages and codes.

HTTPPage

Reversible:	NO	**Flag affected:**	NO

Parameters: URL, Port
Returns: -1 if there is an error
Syntax: HTTPPage(*URL*[,*Port*])

A client command that retrieves the HTML text of the Web page specified by URL into an OMNIS character variable.

Note: HTTPPage allows you to get HTML text source through a server, transparently and without additional coding. If you need to customize the process for a proxy server, you can use a combination of HTTPGet and TCPReceive. For this technique, see the sample code in "Accessing a Proxy Server".

URL is an OMNIS Character field containing a standard Web page URL of the form http://domaininfo.xxx/path/webpagepage

Port is an optional parameter that includes the port number to use on the server.

The primary role of HTTPPage is to grab, simply and quickly, the HTML text source of the page specified by the URL. The URL may also specify a CGI name and arguments, but it is simpler to access CGIs by using the HTTPPost or HTTPGet functions.

The command returns -1 (one) if there is an error.

Using WebDevError, one or more callback methods return WinSOCK errors.

HTTPParse

Reversible:	NO	**Flag affected:**	NO

Parameters: Message, HeaderList, Method, HTTPVersion, URL, CGIList
Syntax: HTTPParse(*Message,HeaderList,Method,HTTPVersion,URL,CGIList*)

HTTPParse is a server utility command to parse HTTP header information from an incoming request message.

Errors in parsing the message header are reported through the standard WebDevError mechanism. One or more callback methods return error messages and codes.

Message is an OMNIS Character field containing the full text of an HTTP request message.

HeaderList is an OMNIS list defined to have two character columns. The list contains information culled from the HTTP message header as attribute/value pairs on each row. There is one row for each item found on the header.

For example, after the call, the list might contain entries such as:

Attribute	Value
Date	The current GMT date and time in HTTP header
format	
User-Agent	NCSA Mosaic for the X Window System/2.4 libwww/2.12 modified
Accept	/
Content-type	application/x-www-form-urlencoded
Content-length	1234

Note: HTTPParse automatically strips the colons after the attribute names.

Method is an OMNIS character field that receives the type of HTTP method being requested: GET, POST, or HEAD.

HTTPVersion is an OMNIS Character field containing the version of HTTP. Currently this is the constant 1.0.

URL is an OMNIS Character field that receives the name of the URL to be processed for the GET, POST, or HEAD. This contains the name of the URL, possibly preceded by a path. At a minimum, the path is a single slash, so every URL returned from HTTPParse is of the form /URLName.

Note: Due to the presence of the leading slash, a simple OMNIS equality string comparison to the name of the URL fails. Use the pos() function or similar parsing mechanism to find the URL name. The trailing question mark of a GET-method CGI, which separates the URL path from the CGI arguments, is stripped by HTTPParse.

CGIList is an OMNIS list field defined to have two character columns. The list contains information culled from the arguments (if any) that are passed to a CGI. There is one row for each field by the CGI method. In this way, an OMNIS method can acquire the values from a Web form. For example, if the following HTML form is the submitted from a browser to the OMNIS Web listener server:

Name:
City:
Are you alive?

and the user types in *John Smith, Podunk* and checks the City field, then after HTTPParse, *CGIList* contains:

Attribute	Value
Name	John Smith
City	Podunk
Alive	Yes
Submit	Please

Note: Before the values are placed in the list, HTTPParse automatically decodes any CGI encoding required to pass special characters in the entry. There is no need to call the CGIDecode command to decode the value entries in the list.

HTTPPost

Reversible: NO   **Flag affected:**   NO

Parameters: Hostname, URI, CGIList, HeaderList, Port
Returns: Returns *Socket*

Syntax: HTTPPost(*Hostname,URI,*[*CGIList*[,*HeaderList*[,*Port*]]])

HTTPPost is a client command that submits a POST-method CGI request to a Web server. HTTPPost returns a positive integer socket number opened to the Web server as a result of the POST CGI in *Socket*. This allows OMNIS to read the results of the CGI request on this socket. In the case of an error, a value of -1 (minus one) is returned for the socket number. Errors in parsing the message header are reported through the standard WebDevError mechanism.

Hostname is an OMNIS character field containing the hostname of a Web server to which to connect.

URI is an OMNIS Character field containing the path and the name of the CGI to be run on the Web server. Often this value can be determined by looking at the source to an HTML page that requests the CGI.

CGIList is an optional parameter that specifies a 2-column OMNIS list. The list contains information to be sent as the arguments of the CGI. There is one row for each field passed to the POST CGI method. In this way, an OMNIS method can send OMNIS field values to a Web server. For example:

Attribute	Value
Name	John Smith
City	Podunk
Alive	Yes
Submit	Please

Note: Before the values are sent to the Web server, HTTPPost performs any CGI encoding required to pass special characters in the arguments. There is no need to call the CGIEncode external command to encode the value entries in the list.

HeaderList is an optional parameter specifying an OMNIS list defined to have two character columns. The list contains information added to the HTTP message header as attribute/value pairs on each row. There is one row for each item found on the header. For example, after the call, the list might contain HTTP Header entries such as:

Attribute	Value
Accept	*
/*	
Content-type	text/html

Port is an optional parameter that designates a local client port for the return of data.

HTTPPost adds colons to the attributes when it constructs the header. Do not end attribute names with a colon. HTTPPost adds the following header fields by default:

Attribute	Value
Date	The current GMT date and time in HTTP header format
Server	OMNIS7/3.5
MIME-version	1.0

HTTPRead

Reversible:	NO	**Flag affected:**	NO

Parameters: Socket, Message
Returns: *Length*
Syntax: HTTPRead(*Socket,Message*)

HTTPRead is a server command that reads a character stream from a socket, functionally equivalent to TCPReceive.

Socket is an OMNIS Integer field containing the number of a socket previously opened.

Stream is an OMNIS Character field used to receive the characters waiting on the socket.

Length is an OMNIS Long Integer field containing the number of characters read, if greater than or equal to 0 (zero).

If an error occurs, *Length* contains a WinSOCK error code in the form of a number less than 0 (zero), shown at the end of this chapter. If the socket is set to non-blocking, an error of -10035 is returned to indicate that there is nothing to read. Otherwise, the socket blocks indefinitely. Using WebDevError, one or more callback methods return error messages and codes.

HTTPSend

Reversible:	NO	**Flag affected:**	NO

Parameters: Socket, HTML
Returns: *Length*
Syntax: HTTPSend(*Socket,HTML*)

HTTPSend is a server command that sends a character stream to a socket, functionally equivalent to TCPSend.

Socket is an OMNIS Integer field containing the number of a socket previously opened.

HTML is an OMNIS Character field containing the characters to send through the socket.

Length is an OMNIS Long Integer field containing the number of characters sent if greater than or equal to 0 (zero). If an error occurs, *Length* contains a WinSOCK error code in the form of a number less than 0 (zero), shown at the end of this chapter. If the socket is set to non-blocking, an error of -10035 is returned, indicating that the socket is blocked and the send has failed or is incomplete.

HTTPServer

Reversible: NO   **Flag affected:**   NO

Parameters: WebProc, Port

Syntax: HTTPServer(*WebProc*[,*Port*])

HTTPServer invokes a listening socket on port 80, or a user-specified port, to receive incoming HTTP Web requests. This function shows an OMNIS working message with the count of accepted connections. HTTPServer calls back into a user-specified OMNIS method when a connection is accepted on port 80 or on the specified port. The user function receives the socket number connected to the client. Even though HTTPServer is meant to allow OMNIS to accept incoming HTTP connections, it can serve any other purpose requiring a fast accept loop on a user-specified port.

Standard WinSOCK and command argument errors are reported using WebDevError.

WebProc is an OMNIS Character field containing an address for an OMNIS method to be called when a connection is accepted. The method receives one parameter, the number of the socket on which the connection has been accepted. For example: MYCODE or MYLIBRARY.MYCODE. You should use the method name qualified by the library name if your applications are in a multi-library environment.

```
;Parameter ConnectedSocket (Long integer)
```

You may read and write to the parameter socket with HTTPRead, HTTPSend, or HTTPHeader external commands or a TCP equivalent (TCPSend; for example). All sockets are created equal.

Port is an OMNIS Integer field that is optionally used to indicate a default port number other than 80.

Caution: You must close the connected socket with HTTPClose before quitting the OMNIS method.

Stopping the Server Listener

Once initiated, the server runs indefinitely until it is stopped. There are two ways to stop HTTPServer listeners:

1. Press the Cancel button on the working dialog displayed by the external command. HTTPServer is in a very tight listening loop. Sometimes you may have to click on the Cancel button more than once to get the external command's attention.

2. You may set the OMNIS flag variable to false before returning from the AcceptCallback method. The HTTPServer checks the flag and stops, continuing execution from the next method after the original call to HTTPServer.

HTTPSplitHTML

Reversible: NO   **Flag affected:**   NO

Parameters: Message, TagTextList

Syntax: HTTPSplitHTML(*Message*,*TagTextList*)

HTTPSplitHTML is a client utility function to parse the HTML from a Web page into an OMNIS list. The HTML tags are parsed out of the text, so that it easy to write a program that grabs the Web page content or interprets the tags from a form.

Message is an OMNIS Character field containing the text of the content portion of a Web page, including HTML tags.

TagTextList is an OMNIS list defined to have three columns, all character. Column 1 contains the opening HTML tag, column 2 the actual page text, and column 3 the closing HTML tag.

Using WebDevError, one or more callback methods return error messages and codes.

HTTPSplitURL

Reversible: NO   **Flag affected:**   NO

Parameters: URL, Hostname, URI

Syntax: HTTPSplitURL(*URL*,*Hostname*,*URI*)

HTTPSplitURL is a server or client utility function to split a full URL into a hostname name and a path (that is, a URI). Useful for following HREF links on pages. Errors in parsing the URL are reported through the standard WebDevError mechanism.

URL is an OMNIS Character field containing a standard Web page URL of the form http://host.mydomain.com/path/webpage.html

Hostname is an OMNIS character field that receives the domain name parsed out of the URL argument. For example, given the URL, above, the domain portion would be host.mydomain.com

URI is an OMNIS Character field that receives the path and page name spec parsed out of the URL argument. For example, given the URL, above, the URI would be path/webpage.html.

MAILSplit

Reversible:	NO	**Flag affected:**	NO

Parameters: Message, HeaderList, Body
Returns: *Status* 1 (one) if successful and a 0 (zero) if error
Syntax: MAILSplit(*Message,HeaderList,Body*)

MAILSplit is a utility command to parse RFC 822 mail headers. It strips the mail header from the body of a mail message.

Message is an OMNIS Character field containing the complete text of an Internet e-mail message, including the header. These are returned in the MailList argument of the POP3Recv external command. For example

```
Received: by omnis-software.com with SMTP; 12 Aug 1996 11:49:59 -0700
Received: (from someone@localhost) by netcom8.netcom.com (8.6.13/Netcom)
id LAA09789; Mon, 12 Aug 1996 11:46:45 -0700
Date: Mon, 12 Aug 1996 11:46:45 -0700
From: someone@somedomain.com (PersonalName here)
Message-Id: <199608121846.LAA09789@netcom8.netcom.com>
To: someoneelse@somedomain.com
Subject: This is an e-mail subject
Hello from OMNIS Software, Inc.
```

HeaderList is an OMNIS list defined to have two character columns. The list receives the information from the e-mail message header as attribute/value pairs on a row of the list. There is one row for each item in the header. This function can format the message for simpler display or to find when a message has been sent for filing and other purposes. For example, assuming the e-mail message above:

Attribute	Value
Received	by omnis-software.com with SMTP; 12 AUG 1996 11:49:59 -
	netcom8.netcom.com0700 (from someone@localhost) by netcom8.netcom.com
Received	(8 .6 .13/Netcom) id LAA09789 ; MON, 12 AUG
	1996 11:46:45 -0700
Date	Mon, 12 Aug 1996 11:46:45 -0700
From	someone@somedomain.com (Personal Name here)
	<199608121846.LAA09789@
Message-Id	netcom8.netcom.com>
To	someoneelse@somedomain.com
Subject	This is an e-mail subject

Note: Two header lines may have the same attribute name. This is within the RFC822 message header specification. In this case, the HeaderList has two lines with the same Attribute name, as with Received in the above example. Long header lines that are split and continued in the message header are concatenated into one line in the list, as with the second Received attribute in the above example. The colon at the end of the attribute is stripped.

Body is an OMNIS character field. The body of the e-mail message is returned into this variable, minus the header. For example: Hello from OMNIS Software, Inc.

Errors are reported using the WebDevError callback mechanism.

Move file

Reversible:	NO	**Flag affected:**	NO

Parameters: From path (file to be moved)
To path (the new location)
Return field

Syntax: Move file *(**from-path, to-path**)* returns ***return-field***

This command moves the file specified in *from-path* to the directory named in *to-path*. It returns any error code, shown at the end of this chapter, or zero if none. If *to-path* is a directory only, the file is moved to that directory. If *to-path* includes a filename and directory name the file is moved and renamed. This may fail if the *to-path* directory contains a file with the same name as *from-path* filename.

Move file cannot move a file across volumes (disks). Use *Copy file* and *Delete file* instead. The Windows version of *Move file* cannot move directories; the MacOS version can.

NSF Add fields

Reversible: NO   **Flag affected:**   NO

Parameters: Note ID
                  Commit/NoCommit flag
                  Field list
                  Status return field

Syntax: NSF Add fields *(note-id, commit-flag, field1*[,*field2*]*...)*
              returns *status-field*

This command writes new field values to the Note specified by the Note ID. For example

NSF Add fields (Note_ID,'Commit') Returns R

In this form *NSF Add fields* does the following:

1. Opens the Note

2. For each field in the current 'map' table it adds a field to the Note

3. Writes the Note to disk

4. Returns the number of fields updated

The calls used to add the fields will delete and replace any fields that are already there.

You can specify a list of fields in which case the map table is ignored and the value of the field or fields is added, for example

NSF Add fields (Note_ID,'NoCommit','Field1','Field2' Returns R

The *Commit/NoCommit* string controls the flushing of the note from the disk cache on the server.

NSF Attach file

Reversible: NO   **Flag affected:**   NO

Parameters: Note ID
                  File path
                  File name
                  Status return field

Syntax: NSF Attach file *(note-id, file-path, file-name)* returns
              status-field

This command attaches a file attachment to a Note. The file path and name are separate parameters, the file name being the name of the file as stored in the attachment and the path being the location of the file on the local hard disk.

NSF Build view

Reversible: NO   **Flag affected:**   NO

Parameters: View name
                 List name
                 Text key
                 Partial
                 Number return field

Syntax: NSF Build view *(view-name, list-name*[,*text-key*][,*'Partial'*]*)*
                 returns *number*

This command moves data from a view into an OMNIS list. You can also search the primary index with a text key, either using a partial or full match on the index. The search is insensitive to diacritical marks. The *view-name* and *list-name* parameters are compulsory. As each note is opened, its fields are read into the OMNIS record buffer and added to the list. Thus, the last Note found in the view is always loaded into the "mapped" variables. There is no way to prevent the values from being added to the list unless you were to redefine the columns of the list to be different to the "map".

```
Set current list LIST2
Define list {Note_ID,LastName,FirstName,PhoneNumber}
NSF Map Fields ('LIST2') Returns R
NSF Build View ('People','LIST2') Returns R
Redraw windows NotesWindow
```

NSF Build view does the following:

1. Opens the view note

2. Creates a collection of notes from the view

3. For each note, it uses its ID to open the view

4. For each field in the OMNIS map it tries to read the named field from the note

5. If a matching field is found, it reads the value into OMNIS

6. When all the "mapped" fields have been processed, it adds a line to the specified list.

If you use a list with no mapped fields a blank line will be added for each note. The *NSF Build View* command returns the number of notes found in the view.

If you add the *text-key* parameter a search is carried out for a matching value in the primary index for that view. For example

```
NSF Build View ('People','LIST2','Pon') Returns R
```

The '*Partial'* parameter will search for a partial match beginning with the text value supplied in parameter *text-key*. For example

```
;  Beginning with 'P'
```
NSF Build View ('People','LIST2','P','Partial') Returns R

NSF Close all files

Reversible:	NO	**Flag affected:**	NO
Parameters:	Status return field		

Syntax: NSF Close all files returns *status-field*

This command closes all open Notes database files.

NSF Close file

Reversible:	NO	**Flag affected:**	NO
Parameters:	Pathname or Mail_File		
	Status return field		

Syntax: NSF Close file *(path-name|'mail_file')*
                       returns *status-field*

This command closes the specified file and writes any data to disk. A database remains open until it is closed with this command.

NSF Close file ('Mail_file')

NSF Copy Note

Reversible:	NO	**Flag affected:**	NO
Parameters:	Note ID		
	Return field		

Syntax: NSF Copy Note *(note-id)* returns *return-field*

This command copies a Note from the current database to a specified database. If the target database is not open, it will be opened, but not made current.

NSF Delete Note

Reversible: NO   **Flag affected:**   NO

Parameters: Note ID
                    Status return field

Syntax: NSF Delete Note *(note-id)* returns *status-field*

This command deletes the specified Note from the currently open file. For example

```
NSF Delete Note (Note_ID) Returns #F
If flag false
   OK message {Error}
End If
```

NSF Describe fields on form

Reversible: NO   **Flag affected:**   NO

Parameters: Form name
                    List name
                    Field name
                    Field type
                    Status return field

Syntax: NSF Describe fields on form *(form-name, list-name, field-name,
                    field-type)* returns *status-field*

This command builds a list of objects on the specified form. Forms in Notes contain a
certain amount of data relating to the fields and their data types. This command describes
the field names and types of the form and places the description in the named list.

This method builds a list of fields on "Myform":

```
NSF Describe fields on form (MyForm,'FieldsList','Field','Type')
   Returns #F
```

NSF Find forms

Reversible: NO   **Flag affected:**   NO

Parameters: List name
                    Field name
                    Status return field

Syntax: NSF Find forms *(list-name, field)* returns *status-field*

This command builds a list of forms in the current Notes database. Forms in Notes contain a certain amount of data relating to the fields and their data types. The following example builds a list of forms, stripping out any aliases in the names.

```
Set current list FormList
Clear list
NSF Find forms ('FormList','Form') Returns R
For each line in list from 1 to $linecount step 1
   If pos(';',lst(Form))
     Calculate FormList('Form',LIST.$line) as
               mid(lst(Form),1,pos(';',lst(Form))-1)
   End If
End For
```

NSF Get info

Reversible: NO   **Flag affected:**   NO

Parameters: Info string return field

Syntax: NSF Get info returns *info-string*

This command gets the file info for the current open Notes database file.

NSF List open NSF files

Reversible: NO   **Flag affected:**   NO

Parameters: List name
                    Status return field

Syntax: NSF List open NSF files *(list-name)*
                    returns *status-field*

This command builds a list of open NSF files. The list is built in the specified list for which a single column must have been defined.

NSF Mail Note

Reversible: NO   **Flag affected:**   NO

Parameters: Note ID
                  Note ID return field

Syntax: NSF Mail Note *(note-id)* returns *note-id*

This command sends a Note to the mail file. The Notes DLL uses the API call

```
OSPathNetConstruct(NULL,szMailServerName,"MAIL.BOX",
szMailBoxPath);
```

to create the path to the file. After writing to the Mail file, the data is not flushed to disk until the file is closed with *NSF Close file*.

NSF Make Note

Reversible: NO   **Flag affected:**   NO

Parameters: Form name
                  Note ID return field

Syntax: NSF Make Note *form-name* returns *note-id*

This command inserts a new note in the currently open file, sets its default form and returns the Note_ID. For example

NSF Make Note ('SimpleDataForm') returns Note_ID
OK message {Made [Note_ID]}

NSF Make response

Reversible: NO   **Flag affected:**   NO

Parameters: NoteID
                  Response flag
                  Return field

Syntax: NSF Make response *(note-id, response-flag)* returns *number-field*

This command creates a "response" document to a note specified by the *note-id*. You can then add fields to the new note using *NSF Add fields*.

```
Set current list LIST2
Define list {PLAIN_TEXT,NUMBER,TIME_DATE,TEXT_LIST,Note_ID}
NSF Map fields ('LIST2') Returns R
```
NSF Make response (Note_ID,'Response') Returns R
Calculate Note_ID as R   ;; now points to the new note
OK message (High position,Large size) {Made reponse [R]}

NSF Make server path

Reversible: NO   **Flag affected:**   NO

Parameters: Server
                NSF file
                Path return field

Syntax: NSF Make server path *(server, nsf-file)* returns *path*

This command returns the path to the specified server and NSF file. To access a database on a server and open a mail file, the user must have access to the server itself. Otherwise an error is returned when the API program attempts to open the database.

```
NSF Make server path ('LANSERVE','Specs') Returns NPATH
NSF Open Notes file (NPATH) Returns #F
If flag false
   Ok message {Error opening note file Specs}
End If
```

NSF Map fields

Reversible: NO   **Flag affected:**   NO

Parameters: List name
                Status return field

Syntax: NSF Map fields *(list-name)* returns *status-field*

This command maps Notes fields onto OMNIS field names and field types; you can map up to 32 fields. Once the map has been set up, an array of field references and associated OMNIS field types is held in RAM by the DLL interface that you can use with subsequent *NSF Select* and *NSF Build view* commands. The OMNIS field names have to exactly match the Notes field names for Notes to return any information.

```
; set up variables for list
Set current list LIST2
Define list {Note_ID,LastName,FirstName,PhoneNumber}
NSF Map fields ('LIST2') Returns R
NSF Build view ('People','LIST2') Returns R
Redraw windows NotesWindow
```

NSF Open file

Reversible: NO   **Flag affected:**   NO

Parameters: Pathname
                  Status return field

Syntax: NSF Open file *(path-name)* returns *status-field*

This command opens the database file with the specified pathname. To determine the correct path for the file, open the file in Notes and use the **Synopsis...** option to read the path to that database. When the path is not specified *NSF Open file* will look in the Notes directory for the named file. A return value 1 indicates that the file was already open and was made the 'current' file, return value 2 indicates that the file was opened for the first time.

The number of simultaneous open databases is set to 8. The last opened database is the "current" one and reopening an open database simply makes it the "current" one. You must give the *same* path to the database file each time it is opened.

```
NSF Open file ('Names') returns Statusfield
NSF Make Note ('SimpleDataForm') Returns #F
If flag false
   ; OK message {Error}
End if
; continue
```

NSF Select

Reversible: NO   **Flag affected:**   NO

Parameters: Listname
                  Select macro
                  Date
                  View title
                  Status return field

Syntax: NSF Select *(list-name, select-macro, date, view-title)*
                       returns *status-field*

This command uses the API call "NSFSearch". This function carries out a sequential search of the current Notes database. It scans all the notes in a database or files in a directory. Based on several search criteria, the function calls a user-supplied routine that fills the OMNIS list for every note or file that matches the criteria. NSFSearch is a powerful function that provides the general search mechanism for tasks that process all or some of the documents in a database or all or some of the databases in a directory.

The *date* argument limits the search to notes created or modified since a certain time or date.

The *view-title* string contains the view name. If the selection formula specified by the second argument contains the @ViewTitle function, Notes uses the view name specified by this argument to resolve this @ViewTitle function. If the selection formula does not contain the @ViewTitle function do not include the *view-title* parameter.

```
Set current list LIST1
Define list
   {PLAIN_TEXT,NUMBER,TIME_DATE,TEXT_LIST,RichStuff,Note_ID}
Clear list (All lists)
Set current list TEXT_LIST
Define list {CVAR3}
NSF Select ('LIST1','@All') Returns R
For each line in list
   ; Process list
End for
Redraw NotesWindow
```

NSF Servers

Reversible: NO   **Flag affected:**   NO

Parameters: List name, Status return field

   Syntax: NSF Servers *(list-name)* returns *status-field*

This command returns a list of servers visible on the network. The list is built in the specified list for which you must have defined a single column. For example,

```
Set current list slist
Define list {FV_Server}
NSF Servers ('slist') Returns R
```

NSF Set error field

Reversible: NO   **Flag affected:**   NO

Parameters: Error field name
                    Status return field

Syntax: NSF Set error field *(error-field -name)* returns *status-field*

This command defines an error field which reports errors during method execution. Once
the error is reported to the error field execution continues. Most Notes commands return an
integer value where 0 indicates an error.

```
NSF Set error field ('Error')
NSF Build view ('VIEW','LIST') Returns #F
If flag false
   Ok message {Error [Error]} ;; or call error routine to log it
End if
```

NSF Unpack file

Reversible: NO   **Flag affected:**   NO

Parameters: Note ID
                    File name
                    File path
                    Status return field

Syntax: NSF Unpack file *(note-id, file-name, file-path)* returns
                    status-field

This command unpacks any files attached to the Note. The file name and path are separate
parameters, the *file-name* being the name of the file attachment and the *file-path* being the
location of the file on the local hard disk.

NSF Where's my mail?

Reversible: NO   **Flag affected:**   NO

Parameters: Info string return field

Syntax: NSF Where's my mail? returns *info-string*

This command returns the server name where the current mail file resides.

```
; Declare variable SERVERNAME (Character 1000)
NSF Where's my mail? Returns SERVERNAME
```

NSF Who am I

Reversible: NO   **Flag affected:** NO

Parameters: Info string return field

Syntax: NSF Who am I returns *info-string*

This command returns your user name for the current mail file.

`NSF Who am I` Returns Username

NSF Write composite

Reversible: NO   **Flag affected:** NO

Parameters: Note ID
Commit
Field list
Status return field

Syntax: NSF Write composite *(note-id, commit-flag, field1*[*,field2*]*...)*
returns *status-field*

This command writes a composite field (or RTF) to a Notes field. The current implementation limits the size of text fields written to Notes as the size limit on the Notes summary buffer of 15K. However, the *NSF Write composite* command can append text to an existing RTF field no matter how big it is and you can read fields of any size into OMNIS.

When reading composite or RTF fields into OMNIS, they are converted to plain text via the NSFItemConvert routines in the API. To append a text value to an existing RTF field you issue the following command:

`NSF Write Composite` (Note_ID,'Commit','RichStuff') Returns R

This command never uses the 'map' table and always appends the text in the OMNIS field to the composite field in Notes with the same name. The text is converted using the API convert to composite call, using the default fonts and styles. You have no control over the style of the composite field. There are no size constraints imposed by the OMNIS interface, thus the Notes API calls to convert an RTF to text and the NSFSetText would be the only constraints when dealing with large fields.

Open file

Reversible: NO   **Flag affected:**   NO

Parameters: Path of file to be opened
                  Reference number or DOS file handle
                  R parameter (specifies read-only, otherwise read/write)
                  Return field

Syntax: Open file *(path, refnum*[,*'R'*]*)* returns ***return-field***

This command opens the file named in *path*. The file reference number is returned in
refnum (under Windows, this is a DOS file-handle). You use this reference number to refer
to the open file when calling *Close file, Read file as character, Read file as binary, Write
file as character,* and *Write file as binary.* The *R* parameter is optional and is case-
insensitive. When included this ensures the file opens as read-only, otherwise the file is
opened as read/write.

It returns any error code (shown at the end of this chapter), or zero if none.

Open resource fork

Reversible: NO   **Flag affected:**   NO

Parameters: Path (of file)
                  Reference number
                  R parameter (specifies read-only, otherwise read/write)
                  Return field

Syntax: Open resource fork (***path***, ***refnum***[,'*R*']) returns ***return-field***

This command, available under MacOS only, opens the resource fork of the file specified in
path. The file reference number is returned in *refnum.* If you include the *R* parameter the
file opens as read-only, otherwise the file is opened as read/write.

It returns any error code (shown at the end of this chapter), or zero if none.

POP3Recv

Reversible:	NO	**Flag affected:**	NO

Parameters: Server, Username, Password, List, Delete, Status

Syntax: POP3Recv(*Server,Username,Password,MailList*[,*Delete*,*Status*])

POP3Recv retrieves Internet e-mail messages from a POP3 server into an OMNIS list. If an error is raised, the *Status* field returns a string containing the word ERROR. When an error occurs, all mail may not have been received, and all sockets are closed.

Server is an OMNIS Character field containing the IP address or host name of a POP3 (Post Office Protocol Level 3) server that will serve e-mail to the client running OMNIS. Examples: pop3.mydomain.com or 255.255.255.254.

Username is an OMNIS Character field containing the account that receives the mail on the designated server. Usually an account username, for example, Webmaster.

Password is an OMNIS Character field containing the password for the account specified in the Username parameter, for example, Secret.

List is an OMNIS list field defined to contain a single column of typed characters. The column receives the Internet e-mail messages, one per line. The column variable should be large enough to receive the e-mail message, including the header. The list should be defined with store long data option selected.

Delete is an OMNIS Boolean field which, if set, indicates that the message will be deleted from the server once it has been downloaded into the row in MailList. The default is false, so messages remain on the server if the argument is omitted.

Status is an optional parameter specifying an OMNIS Character field that contains an OMNIS method to be called with mail receive status messages. This parameter overrides WebDevError settings. The method can display a status message in a window or status line of a window while the SMTP process proceeds, for example, MYCODE or MYLIBRARY.MYCODE

Note: Use the method name qualified by the library name if your applications are in a multi-library environment.

POP3Stat

Reversible:	NO	**Flag affected:**	NO

Parameters: Server, Username, Password

Returns: *WaitingMsgs*

Syntax: POP3Stat(*Server,Username,Password*)

The POP3Stat command retrieves the number of Internet e-mail messages waiting for a particular username on a specified POP3 server. If an error is raised, the command returns a string containing the word ERROR. When an error occurs, not all mail may have been received and all sockets are closed.

Server is an OMNIS Character field containing the IP address or hostname of a POP3 server that will serve e-mail to the client running OMNIS. For example: pop3.mydomain.com or 255.255.255.254.

Username is an OMNIS Character field containing the account that receives the mail on the designated server (usually an account username, for example, Webmaster).

Password is an OMNIS character field containing the password for the account specified in the Username parameter, for example, Secret.

WaitingMsgs is an OMNIS Long Integer field containing number of e-mail messages waiting to be collected on the specified server for the specified account.

Put file name

Reversible: NO   **Flag affected:**   NO

Parameters: Path of output file
Dialog title
Prompt (ignored under Windows)
Default
Return field

Syntax: Put file name *(path*[,*dialog-title*][,*prompt*][,*default*]*)*
      returns ***return-field***

This command prompts the user to enter a file name and path; it opens the standard Save as... dialog. You can enter the title of the dialog. The optional *prompt* is put above the name of the file the user enters. The default filename is displayed in the dialog. It returns the full pathname of the file the user entered in *path*, or empty if no file was entered (that is, the Cancel button was clicked). The named file is not opened or created.

It returns any error code (shown at the end of this chapter), or zero if none.

The *prompt* parameter is ignored under Windows. If no default name is specified, MacOS uses "Untitled" and under Windows the field is left empty.

```
Switch sys(6) = 'M'
  Case kTrue        ;; if MacOS
    Put file name (PATH, 'Save your file',
          'Save as','My file') returns ERROR
  Default           ;; if anything else
    Put file name (PATH, 'Save your file',
          'MYFILE.TXT') returns ERROR
End Switch
```

ReadBinFile

Reversible: NO   **Flag affected:** NO

Parameters: Pathname, Binfld, Start, Length

Returns: *Numbytes*

Syntax: ReadBinFile(*Pathname,Binfld*[,*Start*[,*Length*]])

ReadBinFile reads binary data from the file system or data fork (not the resource fork).

Note for Macintosh Users: ReadBinFile and WriteBinFile are useful for reading and writing documents but not system and application files.

Pathname is an OMNIS Character field containing the full path of the file to read.

Binfld is an OMNIS Binary field in which the data is stored.

Start is an optional parameter specifying an OMNIS Integer field that contains the byte position in the file where the command should start reading. Defaults to 0 (zero), that is, the beginning of the file.

Length is an optional parameter specifying an OMNIS Integer field containing the number of bytes to read. If the parameter is not used, the value defaults to the length of the file.

NumBytes is an OMNIS Long Integer field that is the number of bytes read, if no error occurs. Otherwise, an error code is returned, shown at the end of this chapter.

A WebDevError callback method returns error messages and codes.

Read entire file

Reversible:　　NO　　　**Flag affected:**　　NO

Parameters:　　Path of file to be read
　　　　　　　　　Binary variable (for the returned data)
　　　　　　　　　R parameter (specifies read-only, otherwise read/write)
　　　　　　　　　Return field

Syntax:　　　Write entire file *(path, binary-variable* [,'R']*)* returns ***return-field***

This command reads an entire file into a binary field. It returns any error code (shown at the end of this chapter), or zero if none. The Binary value is in the following format:

1. 12 byte header containing the Type (4 bytes), Creator (4 bytes), and Data fork size (4 bytes).

2. Data fork information.

3. Resource fork information.

The size of the data fork determines where the resource fork data is stored, as shown below. Under Windows, the Type defaults to 'TEXT', the Creator to 'mdos', and the resource fork is not stored.
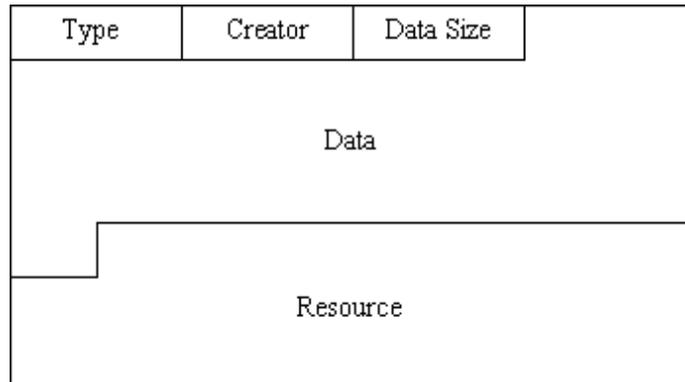
Read file as binary

Reversible: NO   **Flag affected:**   NO

Parameters: Reference number or DOS file handle
                  Binary variable (for the returned data)
                  Start position
                  Number of bytes
                  Return field

Syntax: Read file as binary *(refnum, binary-variable*
              [,*start-position*][,*num-bytes*]*)* returns *return-field*

This command reads a file, or part of a file, into a binary variable. You specify the file reference number or DOS file handle of the file in *refnum*. The binary data read from the file is returned in *binary-variable*.

If you specify the *start-position*, the file is read at that absolute byte position (0 is the first byte in the file, 1 is the second byte in the file, and so on), otherwise it begins at the current position (0 when the file is first opened).If you specify the number of *num-bytes*, only that many bytes are read, otherwise the file is read until the end of the file is reached.

If you specify a *start-position* of 0 and *num-bytes* equal to 0, the file pointer is reset to byte position 0 in the file. If a *start-position* of -1 is given, the file pointer is reset to the end of the file. For both cases an empty *binary-variable* buffer is returned.

It returns any error code (shown at the end of this chapter), or zero if none.

Read file as character

Reversible: NO   **Flag affected:** NO

Parameters: Reference number or DOS file handle
          Character variable for the returned text
          Start position
          Number of characters
          Return field

Syntax: Read file as character (*refnum, character-variable*
        [,*start-position*][,*num-characters*]) returns ***return-field***

This command returns a file, or part of a file, into a character variable. You specify the file reference number or DOS file handle of the file in *refnum*. The text read from the file is returned in *character-variable*.

If you specify the *start-position*, the file is read at that absolute character position (0 is the first character in the file, 1 is the second, and so on), otherwise it begins at the current position (the first character when the file is first opened). If you specify *num-characters*, only that many characters are read, otherwise the file is read until the end of the file is reached.

If you specify a *start-position* of 0 and *num-characters* equal to 0, the file pointer is reset to character position 0 in the file. If a *start-position* of -1 is given, the file pointer is reset to the end of the file. For both cases an empty *character-variable* buffer is returned.

It returns any error code (shown at the end of this chapter), or zero if none.

Register DLL

Reversible: NO   **Flag affected:** NO

Parameters: Library name (of the DLL)
          Procedure name
          Type definition string
          Return field

Syntax: Register DLL (***library-name, procedure-name, type-definition***)
        [returns *return-field*]

This command registers a DLL and its parameters. The *library-name* is a text string specifying the name of the DLL that contains the procedure specified by *procedure-name*. The *type-definition* is a text string specifying the data type of the return value and the data type of all arguments to the DLL. The first letter of *type-definition* specifies the return value. The following table contains the codes to be used in *type-definition* including a description of how the argument or return value is passed and a typical declaration for the data type in the C programming language.

Code	Description	Pass By	C declaration
A	Logical	Value	short int
B	IEEE 8-byte floating point	Value	double
C	Null-terminated string	Reference	char *
D	Pascal string	Reference	unsigned char *
E	IEEE 8-byte floating point	Reference	double *
H	Unsigned 2-byte integer	Value	unsigned short int
I	Signed 2-byte integer	Value	short int
J	Signed 4 byte integer	Value	long int
L	Logical	Reference	short int *
M	Signed 2-byte integer	Reference	short int *
N	Signed 4-byte integer	Reference	long int *
V	void		void

All procedures in the DLL are called using the Pascal calling convention.

The following example opens the Windows **Character Map** Editor.

```
Do method OpenExe ('charmap.exe',1)

; OpenExe
; Declare Parameter APPNAME (Character 255)
; Declare Parameter INSTRUCTS (Short integer (0 to 255))
Register DLL ('KRNL386.EXE','WinExec','ICI') Returns RESULT
Call DLL ('KRNL386.EXE','WinExec',APPNAME,INSTRUCTS) Returns RESULT
If RESULT < 18
   Do method Errors
End If
```

Set creator type

Reversible:	NO	**Flag affected:**	NO

Parameters: File name (including full path)
                    New file type
                    New creator
                    Return field

Syntax: Set creator type (***file-name***[,*file-type*][,*creator*]*)*
                         returns ***return-field***

This command changes the creator and/or file type of a MacOS file; the *file name* must include the full path. If either the *file-type* or *creator* is left empty, the old file type or creator is used.

```
Set creator type ('HD:PicFile','PICT','SPNT')
Set creator type ('HD:OMNIS:SimpSql.LBR','TEXT','ttxt')
```

It returns any error code (shown at the end of this chapter), or zero if none.

Set file read-only attribute

Reversible:	NO	**Flag affected:**	NO

Parameters: Path of the file
                    Read-flag setting
                    Return field

Syntax: Set file read only attribute (***path, read-flag***) returns [*return-field*]

This command lets you set the read-only attribute of the file specified in *path*. If you set the *read-flag* parameter to kTrue the file is set to read-only, or if kFalse the file is set to read/write. Note that read-only status is the same as locked under MacOS.

It returns any error code (shown at the end of this chapter), or zero if none.

SMTPSend

Reversible: NO   **Flag affected:**   NO

Parameters: Server, From, To, Subj, Body, CC, BCC, FromName, StatCall, Priority

Syntax: SMTPSend(*Server,From,To,Subj,Body*
                      [*,CC,BCC,FromName,StatCall,Priority*])

SMTPSend sends Internet e-mail messages via an SMTP server.

Server is an OMNIS Character field containing the IP address or hostname of an SMTP server that will accept e-mail requests from the client running OMNIS, for example, smtp.mydomain.com or 255.255.255.254.

From is an OMNIS Character field containing the RFC 822 Internet e-mail address that will be placed in the header to identify the sender. Recipients can reply to this address, for example, webmaster@www.omnis-software.com.

To is either an OMNIS Character field or an OMNIS list field. If the field is character, it contains the RFC 822 Internet e-mail address to which the e-mail will be sent. The To: line of the message header, for example, webmaster@www.omnis-software.com. If the field is a list, it is defined to contain a single character column, which contains one RFC 822 Internet e-mail addressee per row. The addresses appear in the To: line of the message header. For example:

ToAddresslist
webmaster@www.omnis-software.com
info@omnis-software.com

Subj is an OMNIS character field containing the subject of the e-mail message. The text appears on the Subject: line of the message header, for example, Regarding use of MAILSend ...

Body is an OMNIS Character field containing the body of the e-mail message. The text appears as the actual e-mail message.

CC is either an OMNIS Character field or an OMNIS list field. If the field is character, it contains the RFC 822 Internet e-mail address to which the e-mail will be sent. The To: line of the message header might be, for example, webmaster@www.omnis-software.com. If the field is a list, it is defined to contain a single character column, which contains one RFC 822 Internet e-mail addressee per row. The addresses appear in the CC: line of the message header. For example:

CCAddresslist
webmaster@www.omnis-software.com
info@omnis-software.com

BCC is either an OMNIS Character field or an OMNIS list field. If the field is a Character field, it contains the RFC 822 Internet e-mail address to which the e-mail will be sent (the To: line of the message header, for example, webmaster@www.omnis-software.com). If the

field is a list, it is defined to contain a single character column, which contains one RFC 822 Internet e-mail addressee per row. The addresses appear in the BCC: line of the message header. For example:

BCCAddresslist
webmaster@www.omnis-software.com
info@omnis-software.com

FromName is an OMNIS Character field containing a personal name that will appear in the header to identify the user by a more descriptive name than just the e-mail address, for example, OMNIS Webmaster

StatCall is an OMNIS character field containing an OMNIS method that will be called with status messages about the mail-sending operation. The method can display a status message in a window or status line of a window while the SMTP process proceeds, for example, MYCODEor MYLIBRARY.MYCODE.

Note: Use the method name qualified by the library name if your applications are in a multi-library environment.

Priority is on OMNIS Short Integer field that sets the priority of the e-mail. It accepts a single value in the range of 1 through 5, a 1 (one) indicating the highest priority.

Split path name

Reversible:	NO	**Flag affected:**	NO

Parameters: Path (to be split)
Drive name
Directory
File name
File extension
Return field

Syntax: Split path name (*path, drive-name, directory, file-name,*
file-extension) returns *return-field*

This command splits a full path name into its component parts: the drive name, directory and file name, and file extension. It returns any error code (shown at the end of this chapter), or zero if none. The following examples show how *Split path name* operates.

Mac

Path	Dr	Directory	Filename	Extn
HD:TESTDIR:TESTSDIR:TESTFILE	HD	:TESTDIR:TESTSDIR:	TESTFILE	
HD:TESTDIR:TESTFILE.EXT	HD	:TESTDIR:	TESTFILE	.EXT
HD:TESTFILE	HD	:	TESTFILE	

Non-Mac

Path	Dr	Directory	Filename	Extn
C:\TESTDIR\TESTSDIR\TESTFILE	C:	\TESTDIR\TESTSDIR\	TESTFILE	
C:\TESTDIR\TESTFILE.EXT	C:	\TESTDIR\	TESTFILE	.EXT
C:\TESTFILE	C:	\	TESTFILE	

TCPAccept

Reversible:	NO	**Flag affected:**	NO

Parameters: Socket
Returns: *Socket*

Syntax: TCPAccept(*Socket*)

TCPAccept accepts the first connection on the queue of pending connections on a socket, creates a new accept socket with the same properties, and returns the number of the new socket. If no pending connections are present on the queue, and the listenable socket is marked as blocking, TCPAccept blocks the caller until a connection is present. If the socket is marked non-blocking and no pending connections are present on the queue, TCPAccept returns an error as described below. The accept socket is used for all further communication with that client. The original socket remains open and can accept additional connections.

Socket is an OMNIS Long Integer field containing a socket number for a new socket connection to the client if there is no error.

A negative value indicates an error. Check the error status in the WinSOCK error table.

TCPAddr2Name

Reversible:	NO	**Flag affected:**	NO

Parameters: Address
Returns: *Hostname*

Syntax: TCPAddr2Name(*Address*)

TCPAddr2Name is a domain name service external command to resolve the hostname for a given IP address.

Address is an OMNIS Character field containing the IP address to convert to a hostname. The IP address is of the form 255.255.255.254

Hostname is an OMNIS Character field containing a hostname converted from the given IP address. The hostname is of the form machine[.*domainame.dom*]

WinSOCK error codes are returned. Error codes are numbers less than 0 (zero), shown at the end of this chapter.

Note: This command fails if the address of a Domain Name Server has not been defined in your computer. Not all host IP Addresses may be known to the Domain Name Server. If the Domain Name Server is busy or unavailable, the command times out and returns an error. Defining often -used servers to a local host's file or using a caching Domain Name Server increases performance of this command.

TCPBind

Reversible:	NO	**Flag affected:**	NO

Parameters: Socket, Service/Port
Returns: *Status*
Syntax: TCPBind(*Socket,*[*Service/Port*])

TCPBind binds a socket created with TCPSocket() to a particular local port.

Socket is an OMNIS Long Integer field, containing the number of the socket.

Service/Port is an optional parameter, either an OMNIS integer field containing either the number of the port to which the socket should be bound or an OMNIS character field containing a name from the Windows Services file.

Status is an OMNIS Long Integer field containing a 0 (zero) if the bind was successful, or a standard WinSOCK error code, shown at the end of this chapter, if the bind was unsuccessful.

TCPBlock

Reversible:	NO	**Flag affected:**	NO

Parameters: Socket, option
Returns: *Status*
Syntax: TCPBlock(*Socket,option*)

The TCPBlock command makes a socket blocking or non-blocking. If a socket is blocking, an Accept, Receive, Send, or Connect stops processing, that is, "blocks" until satisfied. A receive waits until the remote machine has performed a send. For example, a non-blocking socket returns -10035 if no information is available to read or if the socket is not ready to send information. WinSOCK error codes are returned, shown at the end of this chapter.

Please note that this is a *Windows command only*. Therefore, before issuing this command, test sys(6) for 'W' or 'N'. This prevents the following dialog from appearing to a Macintosh end user: TCPBlock not implemented yet.

Non-blocking sockets are usually preferable, although more coding is necessary. WinSOCK defaults to non-blocking. The Mac is non-blocking only. If you attempt to use a blocking-type socket on the Windows 16-bit platform, the machine hangs while the socket is waiting for a message.

Socket is an OMNIS Long Integer field containing a number identifying a valid socket.

option is an OMNIS field with the value of 1 (one) or 0 (zero). One (1) is for non-blocking and 0 (zero) is for blocking.

Status is an OMNIS Long Integer field that, if no error occurs, returns a 0 (zero). If an error occurs, a negative value indicates an error.

TCPClose

Reversible: NO   **Flag affected:**   NO

Parameters: Socket
Returns: *Status*

Syntax: TCPClose(*Socket*)

TCPClose closes and releases a socket.

Socket is an OMNIS Integer field containing a number representing a previously opened socket.

Status is an OMNIS Long Integer field that, if no error occurs, returns a socket number for the new socket connected to the client. If an error occurs, a standard WinSOCK error code is return, shown at the end of this chapter.

Note: Non-blocking sockets may return an error code of -10035 if the socket is busy and cannot be closed.

TCPConnect

Reversible: NO   **Flag affected:**   NO

Parameters: Server, Service
Returns: *Socket*

Syntax: TCPConnect(*Server,Service*

TCPConnect creates a new socket open to a particular service or port on a named server or IP address.

Server is an OMNIS Character field containing the domain name or IP address of the server to which the socket is to connect.

Service is an OMNIS Character field containing either a port number or a name (from the Services file on a Windows system) of the port to connect with on the named server.

Socket is an OMNIS Long Integer field that returns the number of the allocated socket. If an error occurs, a standard WinSOCK error code, shown at the end of this chapter, is returned in *Socket*.

Note: This differs from the more standard implementation of the connect-sockets call. Instead of creating a socket with one command (such as TCPSocket), then sending the socket number to a connect command, TCPConnect creates the socket and returns the socket number in one step.

TCPGetMyAddr

Reversible: NO   **Flag affected:**   NO

Parameters: None
Returns: *Address*
Syntax: TCPGetMyAddr()

TCPGetMyAddr is a domain name service external command to resolve the IP address of the local computer running OMNIS.

Address is an OMNIS Character field containing an IP Address of the local host. The IP address is of the form 255.255.255.254

WinSOCK error codes are returned. Error codes are numbers less than 0 (zero), shown at the end of this chapter.

TCPGetMyPort

Reversible: NO   **Flag affected:**   NO

Parameters: Socket
Returns: *Port*
Syntax: TCPGetMyPort(*Socket*)

TCPGetMyPort is a command to return the number of the TCP port to which a given socket is connected.

Socket is an OMNIS Long Integer field containing a socket connected to a peer or bound to a port.

Port is an OMNIS Long Integer field containing the number of the port to which the socket is bound.

WinSOCK error codes are returned. Error codes are numbers less than 0 (zero), shown at the end of this chapter.

Note: This command fails if the socket is not connected or bound to a port. Some implementations of WinSOCK return a number that is offset from the actual port number.

TCPGetRemoteAddr

Reversible:	NO	**Flag affected:**	NO
Parameters:	Socket		
Returns:	*Address*		
Syntax:	TCPGetRemoteAddr(*Socket*)		

TCPGetRemoteAddr is a command to return the IP address of the remote computer to which a given socket is connected.

Socket is an OMNIS Long Integer field containing a socket connected to a peer.

Address is an OMNIS Character field containing the IP Address host to which the socket is connected. The IP address is of the form 255.255.255.254

WinSOCK error codes are returned. Error codes are numbers less than 0 (zero), shown at the end of this chapter.

Note: This command fails if the socket is not connected.

TCPListen

Reversible:	NO	**Flag affected:**	NO
Parameters:	Socket		
Returns:	*Status*		
Syntax:	TCPListen(*Socket*)		

TCPListen puts a socket created with TCPSocket into passive mode. Incoming connections are acknowledged and placed in a queue pending acceptance via TCPAccept.

Socket is an OMNIS Long Integer field containing the number of a socket that has been bound to a port.

Status is an OMNIS Long Integer field that, if no error occurs, TCPListen returns a 0 (zero). If there is an error, a value of -1 (one) is returned.

TCPName2Addr

Reversible:	NO	**Flag affected:**	NO

Parameters: Hostname
Returns: *Address*
Syntax: TCPName2Addr(*Hostname*)

TCPName2Addr is a domain name service external command that resolves the IP address for a given hostname.

Hostname is an OMNIS Character field containing a hostname to convert to an IP address. The hostname is of the form machine[.*domainname.dom*]

Address is an OMNIS Character field containing the IP Address corresponding to the given hostname. The IP address is of the form 255.255.255.254

WinSOCK error codes are returned. Error codes are numbers less than 0 (zero), shown at the end of this chapter.

Note: This command fails if the address of a Domain Name Server has not been defined in your computer. Not all host IP Addresses may be known to the Domain Name Server. If the Domain Name Server is busy or unavailable, the command times out and returns an error. Defining often-used servers to a local host's file or using a caching Domain Name Server increases performance of this command.

TCPPing

Reversible:	NO	**Flag affected:**	NO

Parameters: server, size, timeout
Returns: *Milliseconds*
Syntax: TCPPing(*Server*[,*Size*[,*Timeout*]])

TCPPing sends an ICMP request packet to a specified IP address or named host. It returns the round-trip packet time in milliseconds. If the host is unreachable or not available, the command will returns a negative number and an error message.

Server is an OMNIS Character field containing the IP address or domain name of the host to ping.

Size is an optional parameter specifying an OMNIS Long Integer field containing the size, in bytes, of the packet to ping the specified host. Typical values are from 512 to 2,048 bytes.

Timeout is an optional parameter specifying an OMNIS Long Integer field containing the number of milliseconds to use as a timeout value for the ping request. If the host is unavailable or does not respond in the specified number of milliseconds, the TCPPing function cancels the ping request and returns an error.

Milliseconds is an OMNIS Long Integer field. When no error occur, TCPPing returns the number of milliseconds that it took to receive the ping response from the host. On very fast LANs, it is possible that the ping can complete so quickly that the value may be 0 (zero). A negative number indicates a WinSOCK error code, shown at the end of this chapter.

A value of -1 (one) is returned if the ping times out. Error messages are returned using the standard WebDevError mechanism.

TCPReceive

Reversible:	NO	**Flag affected:**	NO

Parameters: Socket, Buffer
Returns: *receivedCharCount*
Syntax: TCPReceive(*Socket,Buffer*)

TCPReceive receives a message on a socket.

HTTPPage allows you to get HTML text source through a server, transparently and without additional coding. If you need to customize the process for a proxy server, you can use a combination of HTTPGet and TCPReceive. For this technique, see the sample code in "Accessing a Proxy Server" .

Socket is an OMNIS Long Integer field containing the number of a socket previously opened.

Buffer is an OMNIS Character field containing a character variable to receive the characters waiting on the socket.

receivedCharCount is an OMNIS Long Integer field containing the number of characters received into the message.

WinSOCK error codes are returned when error codes are values less than 0 (zero), shown at the end of this chapter.

Note: Non-blocking sockets may return an error code of -10035 if the socket is not ready or able to read the characters. Some implementations of socket libraries may have limits on the number of characters you can receive at one time. Consult the documentation for your installed sockets libraries. You may have to read the message of characters in multiple chunks and assemble the entire message. Always check the number of characters returned to make sure there was no error.

TCPSend

Reversible: NO   **Flag affected:** NO

Parameters: Socket, Message
Returns: *sentCharCount*
Syntax: TCPSend(*Socket,Message*)

TCPSend sends a message on a socket.

Socket is an OMNIS Long Integer field containing a socket previously opened.

Message is an OMNIS Character field containing characters to send on the socket.

receivedCharCount is an OMNIS Long Integer field containing the number of characters sent.

WinSOCK error codes are returned when error codes are values less than 0 (zero), shown at the end of this chapter.

Note: Non-blocking sockets may return an error code of -10035 if the socket is not ready or able to send the characters. Some implementations of socket libraries may have limits on the number of characters you can send at one time. Consult the documentation for your installed sockets libraries. You may have to read the message of characters in multiple chunks in order to send a very long message. Always check the number of characters returned to make sure it matches the length of the message argument.

TCPSocket

Reversible: NO   **Flag affected:** NO

Parameters: None
Returns: *Socket*
Syntax: TCPSocket()

TCPSocket creates a new socket.

Socket is an OMNIS Long Integer field containing the number of the allocated socket.

WinSOCK error codes are returned when error codes are socket numbers less than 0 (zero), shown at the end of this chapter.

Truncate file

Reversible: NO   **Flag affected:**   NO

Parameters: Reference number or DOS file handle
                    End position
                    Return field

Syntax: Truncate file (refnum[,end-position]) returns return-field

This command truncates a file. You specify the file reference number or DOS file handle of the file in the *refnum*. The file is truncated at the current position of the file pointer or the specified *end-position* if given.

It returns any error code (shown at the end of this chapter), or zero if none.

UUDecode

Reversible: NO   **Flag affected:**   NO

Parameters: stream
Returns: *DecodedField*
Syntax: UUDecode(*stream*)

UUDecode turns Uuencoded information back into text or binary information. It is the converse of UUEncode. Uuencoded information is commonly sent over the Internet in a manner that preserves binary information. Errors are reported via the WebDevError callback mechanism.

Stream is an OMNIS Character or Binary field containing the information to UUDecode.

DecodedField is an OMNIS Character or Binary field that holds the resulting Uudecoded representation of the *stream* argument. Because Uuencoding is generally used for binary information, a Binary field is the norm.

UUEncode

Reversible: NO   **Flag affected:**   NO

Parameters: stream
Returns: *EncodedField*
Syntax: UUEncode(*stream*)

UUEncode can send binary information via e-mail or other Internet facilities that might otherwise not transfer binary data correctly. UUEncoding turns a file into a stream of 64-character lines of print-only ASCII characters. The encoded version is approximately 1.25 times larger than the original. Errors are reported via the WebDevError callback mechanism.

Stream is an OMNIS Character or Binary field containing the information to UUEncode.

EncodedField is an OMNIS Character or Binary field that hold the resulting Uuencoded representation of the *stream* parameter.

WebDevError

Reversible: NO   **Flag affected:**   NO

Parameters: Proc
Returns: *Error*
Syntax: WebDevError(*Proc)*

The WebDevError external command provides for the specification for an OMNIS method to be called when an error occurs in any of the Web Enabler external commands. The methodis called with the mehtod name. Generally, when one of the external commands encounters an error while processing, the WebDevError callback mehtod is called *prior* to returning to the caller of the original function.

Note: The Web Enabler commands cannot determine whether a particular command is being executed on a machine acting as a server or a client. If you do not use WebDevError to set up a mehtod for receiving errors, Web Enabler external commands report errors by displaying a modal OK message containing the text of the error. This stops processing until the dialog is dismissed. If Web Enabler is being used as a server, having processing stop is highly undesirable.

Proc is an OMNIS Character field containing an address for an OMNIS mehtod to be called with error status messages. The mehtod can be used to display or log the error message or otherwise change normal execution. For example: MYCODE, MYCODE/MYPROC, MYLIBRARY.MYCODE. You should use the method name qualified by the library if your applications are in a multi-library environment.

In the example, MYCODE would invoke a method defined as:

```
;Parameter ErrorMsg (Character 1000000)
```

```
;Parameter ErrorID (Short Integer)
;Parameter CommandName (Character)
OK {Web Enabler Error raised: ErrorMsg]}
Quit method
```

In the method, the parameters are as follows:

ErrorMsg is an OMNIS Character field containing the text of the error message.

ErrorID is an OMNIS Short Integer field containing the error code for the error message, shown at the end of this chapter.

CommandName is an OMNIS Character field containing the name of the command that generated the error.

ProtocolErrorID is an optional OMNIS Character field containing the WinSOCK protocol error that was generated by a socket operation (for example, one of the TCP sockets external commands).

Note: See the WinSOCK and Web command error codes at the end of this chapter. Code 1011 means an Error setting callback method: %s, where '%s' is at least the beginning of the error callback method specification.

WriteBinFile

Reversible: NO   **Flag affected:**   NO

Parameters: Pathname, Binfld, Start, Length
Returns: *Numbytes*

Syntax: WriteBinFile(*Pathname,Binfld*[,*Start* [,*Length*]])

WriteBinFile writes binary data to the file system or data fork (not the resource fork).

Note for Macintosh Users: ReadBinFile and WriteBinFile are useful for reading and writing documents but not system and application files.

Pathname is an OMNIS Character field containing the full path of the file to which to write. If the output file does not already exist, WriteBinFile() creates it.

Binfld is an OMNIS Binary field from which to write the data.

Start is an OMNIS Integer field specifying the byte position in the file where writing should begin. If the parameter is not used, the command defaults to 0 (zero), that is, the beginning of the file. To append data to an existing file, set *Start* to -1 (minus one).

Length is an OMNIS Integer field containing the number of bytes to write. If the parameter is not used, the value defaults to the length of the Binary field.

NumBytes is an OMNIS Long Integer field that is the number of bytes written if no error code is returned.

Using WebDevError, one or more callback methods return error messages and codes.

Write entire file

Reversible: NO   **Flag affected:**   NO

Parameters: Path of file to be written to
                    Binary variable containing data
                    Return field

Syntax: Write entire file *(path, binary-variable)* returns
                            return-field

This command writes an entire file from a binary field, previously populated by using Read entire file. It returns any error code (shown at the end of this chapter), or zero if none. The Binary value is in the following format:

1. 12 byte header containing the Type (4 bytes), Creator (4 bytes), and Data fork size (4 bytes).

2. Data fork information.

3. Resource fork information.

The size of the data fork determines where the resource fork data is stored, as shown below. Under Windows, the Type defaults to 'TEXT', the Creator to 'mdos', and the resource fork is not written.
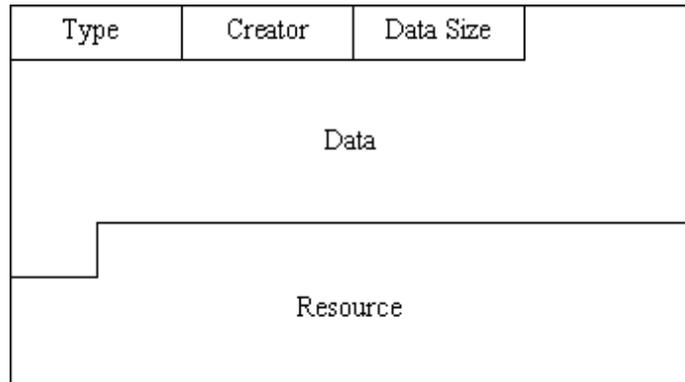
Write file as binary

Reversible: NO   **Flag affected:**   NO

Parameters: Reference number or DOS file handle
Binary variable containing data
Start position
Return field

Syntax: Write file as binary *(refnum, binary-variable* [,*start-position*]*)* returns *return-field*

This command writes the contents of the specified *binary-variable* to a file. You specify the file reference number or DOS file handle of the file in *refnum*.

If you specify the *start-position*, writing begins at that byte (0 is the first byte in the file, 1 is the second byte, and so on), otherwise it begins at the current position (the first byte when the file is first opened).

It returns any error code (shown at the end of this chapter), or zero if none.

Write file as character

Reversible: NO   **Flag affected:**   NO

Parameters: Reference number or DOS file handle
Character variable containing text
Start position
Return field

Syntax: Write file as character *(refnum, character-variable* [,*start-position*]*)* returns *return-field*

This command writes the contents of the specified *character-variable* to a file. You specify the file reference number or DOS file handle of the file in *refnum*.

If you specify the *start-position*, writing begins at that absolute character position (0 is the first character in the file, 1 is the second character, and so on), otherwise it begins at the current position (the first character when the file is first opened).

It returns any error code (shown at the end of this chapter), or zero if none.

FileOps External Command Error Codes

The following errors are returned from the FileOps external commands.

Error Codes returned under Windows

Error Code	Description
1	Too few parameters passed on the command line
12	Out of memory error
998	Undefined error
999	No operation on this platform
-30	Unable to delete directory or file
-36	Disk IO error (or error during operation)
-43	File not found
-48	File or directory already exists
-51	Bad file reference number
-59	Problem during rename

Error Codes returned under MacOS

Error Code	Description
1	Too few parameters passed on the command line
998	Undefined error
999	No operation on this platform
-33	File or directory full
-34	All Allocation blocks on the volume are full
-35	Specified volume doesn't exist
-36	Disk IO error
-37	Bad file name or volume name (perhaps zero-length)
-38	File not open
-39	Logical end-of-file reached during read operation

Error Code	Description
-40	Attempt to position before start of file
-42	Too many files open
-43	File not found
-44	Volume is locked by a hardware setting
-45	File is locked
-46	Volume is locked by a software flag
-47	One or more files are open
-48	A file with the specified name already exists
-49	Only one access path to a file can allow writing
-50	No default volume
-51	Bad file reference number
-53	Volume not on-line
-54	Read/write permission doesn't allow writing
-55	Specified volume is already mounted and on-line
-56	No such drive number
-57	Volume lacks MacOS-format directory
-58	External file system error
-59	Problem during rename
-60	Master directory block is bad; must re-initialize volume
-61	Read/write permission doesn't allow writing
-120	Directory not found
-121	Too many working directories open
-122	Attempted to move into offspring
-123	Attempt to do HFS operation on a non-HFS volume
-127	Internal file system error

Web Command Error Codes

The Web commands return two types of error as negative values: WinSOCK protocol error codes, and Web command error codes. With the exception of the FTPGetLastStatus errors, the error message, error code, and the name of the Web command causing the error are reported by the WebDevError callback mechanism. Macintosh TCP/IP error codes are not used. You can specify an OMNIS method to handle errors using the ***Error! Bookmark not defined.*** command.

WinSOCK Error Codes

The following WinSOCK Error Codes are returned by Web commands as negative values, together with a brief explanation.

Error	Error Code	Explanation
WSAEINTR	-10004	Interrupted system call
WSAEBADF	-10009	Bad socket number
WSAEACCES	-10013	Permission denied
WSAEFAULT	-10014	Bad address
WSAEINVAL	-10022	Invalid argument
WSAEMFILE	-10024	Too many sockets open
WSAEFBIG	-10027	Data too large
WSAWOULDBLOCK	-10035	Operation would block
WSAEINPROGRESS	-10036	Operation now in progress
WSAEALREADY	-10037	Operation already in progress
WSAENOTSOCK	-10038	Socket operation on invalid socket
WSAEDESTADDRREQ	-10039	Destination address required
WSAEMSGSIZE	-10040	Message too long
WSAEPROTOTYPE	-10041	Protocol wrong type for socket
WSAENOPROTOOPT	-10042	Protocol not available
WSAEPROTONOSUPPORT	-10043	Protocol not supported
WSAESOCKTNOSUPPORT	-10044	Socket type not supported
WSAEOPNOTSUPP	-10045	Unknown service or bad port number
WSAEADDRINUSE	-10048	Address already in use
WSAEADDRNOTAVAIL	-10049	Cannot assign requested address

Error	Error Code	Explanation
WSAENETDOWN	-10050	Network is down
WSAENETUNREACH	-10051	Network is unreachable
WSAENETRESET	-10052	Network dropped the connection on reset
WSAECONNABORTED	-10053	Software caused connection abort
WSAECONNRESET	-10054	Connection reset by peer
WSAENOBUFS	-10055	No buffer space available
WSAEISCONN	-10056	Socket is already connected (in use)
WSAENOTCONN	-10057	Socket is not connected
WSAESHUTDOWN	-10058	Cannot send after socket shutdown
WSAETOOMANYREFS	-10059	Too many references: cannot splice
WSAETIMEDOUT	-10060	Connection timed out
WSAECONNREFUSED	-10061	Connection refused
WSAELOOP	-10062	Too many levels of symbolic links
WSAENAMETOOLONG	-10063	File name too long
WSAEHOSTDOWN	-10064	Host is down
WSAEHOSTUNREACH	-10065	No route to host
WSAENOTEMPTY	-10066	Directory not empty
WSAEPROCLIM	-10067	Too many processes
WSAEUSERS	-10068	Too many users
WSAEDQUOT	-10069	Disk quota exceeded
WSAESTALE	-10070	Stale NFS file handle
WSAEREMOTE	-10071	Too many levels of remote in path
WSASYSTEMNOTREADY	-10091	Network subsystem is unusable
WSAVERNOTSUPPORTED	-10092	WinSOCK DLL cannot support this app
WSANOTINITIALISED	-10093	WinSOCK not initialized
WSAEDISCON	-10101	Disconnected
WSAHOST_NOT_FOUND	-11001	Host not found
WSATRY_AGAIN	-11002	Nonauthoritative host not found
WSANO_RECOVERY	-11003	Nonrecoverable DNS error
WSANO_DATA	-11004	Valid name no data record of request type
WSAEHOSTUNREACH	-11065	Connect failed (DNS)

Web Command Error Codes

The following errors are returned by the Web commands as negative values, together with a brief explanation. To avoid collisions and provide an easy way to test for a message type, certain error codes are reserved for Web errors.

With the exception of the FTPGetLastStatus error codes, the following scheme applies to error-code numbering:

Code	Description
0 to -999	Programming error
-1000 and higher	Runtime error caused by a programming error or by an occurrence outside the application's control, such as a network error, non-responding server, and so on

The error message, error code, and command causing the error are reported by the WebDevError callback mechanism, and handled by the OMNIS method specified in the *Error! Bookmark not defined.* command.

Note: The TCP commands return socket errors only. See the WinSOCK Errors.

Reserved Codes

Class/Subclass	Range	Type
Common caused by more than one Web command	-500 to -599	Programming error
FTP	-600 to -649	FTP programming error. Reported by WebDevError callback mechanism.
HTTP	-650 to -699	HTTP programming error. Reported by WebDevError callback mechanism.
E-mail	-700 to -749	E-mail programming error. Reported by WebDevError callback mechanism.
Common caused by more than one Web command	-1000 to -1099	Runtime error caused by programming or by events outside application control
FTP	-1100 to -1149	FTP runtime error caused by programming or by events outside the control of the application
HTTP	-1150 to -1199	HTTP runtime error caused by programming or by events outside the control of the application
E-mail	-1200 to -1249	E-mail runtime error caused by programming or by events outside the control of the application

Web Errors

The following Web Error Codes are returned by Web commands as negative values, together with a brief explanation.

❑ Common errors are generated by more than one command set.

❑ FTP runtime client errors are generated by FTP commands.

❑ HTTP runtime errors are generated by HTTP commands.

❑ E-mail runtime errors are generated by e-mail commands.

Where a number or name is returned within the message, the following abbreviations are used in citing the error below:

%u	Numerics (for example, a column number)
%s	Text (for example, a fieldname)

Common Programming errors

Error Code	Error Text
-501	Invalid argument type for: %s (arg %u)
-502	Problem obtaining argument (arg %u)
-503	Incorrect number of arguments
-504	Must supply a %s (arg %u)
-505	Invalid value for %s (arg %u)
-520	%s list must contain %u columns (arg %u)

Common Runtime errors

Error Code	Error Text
-1010	Insufficient memory to satisfy the request
-1011	Error setting callback method: %s
-1012	Invalid %s (arg %u)
-1051	Unable to locate the required service on the server: %s
-1052	Error establishing communications with server
-1053	Error establishing a connection with the server
-1054	Error while receiving response from server
-1055	Error while sending data to server
-1056	Error while responding to the client
-1057	The current command failed because it timed out on the server

FTP Errors

Error codes returned by FTPGetLastStatus command or reported by WebDevError callback mechanism. FTPGetLastStatus returns only codes -1 to -12. The command is redundant but is retained for backward compatibility. WebDevError callback returns codes -500 to -599, -600 to -649, and -1101 to -1140. FTP errors are client runtime errors.

Error Code	Error Text
-1	Attempt to connect to server failed
-2	Connection lost
-3	Invalid username or password
-4	No such file
-5	Invalid argument
-6	No free sockets (too many connections)
-7	No such server (DNS failed)
-8	Client configuration error (i.e., can't get local IP address)
-9	Server protocol error - server response unexpected
-10	Client file I/O error (disk full, network volume dismounted, etc.)
-11	Out of memory error (common in FTPGetBinary/FTPPutBinary)
-12	User cancel (progress method returned flag false)
-501	Invalid argument type for: %s (arg %u)
-502	Problem obtaining argument (arg %u)
-503	Incorrect number of arguments
-504	Must supply a %s (arg %u)
-505	Invalid value for %s (arg %u)
-1010	Insufficient memory to satisfy the request
-1011	Error setting callback method: %s
-1012	Invalid %s (arg %u)
-1051	Unable to locate the required service on the server: %s
-1052	Error establishing communications with server
-1053	Error establishing a connection with the server
-1054	Error while receiving response from server
-1055	Error while sending data to server
-1056	Error while responding to the client
-1057	The current command failed because it timed out on the server

Error Code	Error Text
-1101	Error while setting FTP mode on server
-1102	Error establishing a connection with server
-1103	The connection with the FTP server was lost
-1104	A severe error occurred while retrieving data from the server
-1110	The specified file was not found: '%s'
-1111	Unable to open the specified file: '%s'
-1112	Error while writing to file: '%s'
-1140	Operation cancelled by user

HTTP Errors

All HTTP errors are runtime errors reported by the WebDevError command

Error Code	Error Text
-1150	Unknown header type. Must be GET, POST, or HEAD.
-1151	Invalid HTML content header received from server
-1152	Badly formed header fields (arg 1)
-1153	Badly formed header, no POST fields found
-1154	Unable to determine end of header
-1160	Missing value for %s in %s list
-1161	Mismatched tag brackets
-1170	Error sending the response header from the server, the failing tag was '%s'.
-1171	Error sending the request header to the server, the failing tag was '%s'.
-1172	Error occurred while sending content data.
-1173	Error occurred while reading the HTML content header from the client

E-mail Errors

E-mail errors are runtime errors reported by the WebDevError command.

Error Code	Error Text
-1200	Badly formed message header (arg %u)
-1201	Recipient list or name is empty
-1210	Error sending the mail header to the server, the failed tag was '%s'. Mail was not sent.
-1211	Error sending the mail text to the server. Mail was not sent.
-1212	Error completing the sending of the mail text to the server. Mail may not have been properly delivered.

Index

OMNIS

OMNIS
Reference

Version 2

How to use this manual

The on-line documentation is designed to make the task of identifying and accessing information about OMNIS Studio as easy and intuitive as possible.

You can navigate this document, or find topics, in a number of different ways.

Bookmarks

Bookmarks mark each topic in a document. To view the bookmarks in this document, click on the Bookmark icon on the Acrobat toolbar or select the **View>>Bookmarks and Page** menu item.

Click on an arrow icon ▷ to open or close a topic, and click on a topic name or double-click a page icon ▢ to move directly to a topic.

Thumbnails

Thumbnails are small images of each page in the document. To view the Thumbnails in this document click on the Thumbnails button on the Acrobat toolbar, or select the **View>>Thumbnails and Page** menu item.

You can click on a thumbnail to jump to that page. Also you can adjust the view of the current page by moving and/or sizing the gray page-view box shown on the current thumbnail.

Links

Links in this document connect related information or take you to a specific location in the document. Links are indicated with *blue italic* text. To jump to a related topic, move the pointer over a linked area (the pointer changes to a pointing finger) and simply click your mouse. Try it!

To return to your last view or location, click on the **Go back** button on the Acrobat toolbar.

Browsing

You can use the Browse buttons on the Acrobat toolbar to move back and forth through the document on a page by page basis. You can also click on the **Go Back** to return to your last view or location.

Find

You can find a text string using the **Tools>>Find** menu item. To find the next occurrence of the text you can use the **Tools>>Find Again** option. If you reach the end of the document, you can use the Ctrl-Home key to go to the beginning and continue your find.

Search

If you have the Acrobat Search plug-in (available under the **Tools>>Search** menu in some versions of Acrobat Exchange and Reader), you can use the Studio Index to perform full-text searches of the entire OMNIS Studio on-line documentation set. Searching the Studio Index is much faster than using the **Find** command, which reads every word on every page in the current document only.

To Search the Studio Index, select **Tools>>Search>>Indexes** to locate the Studio Index (Studio.pdx) on the OMNIS CD. Next, select **Tools>>Search>>Query** to define your search text: you can use Word Stemming, Match Case, Sounds Like, wildcards, and so on (refer to the Acrobat Search.pdf file for details about specifying a query). In the Search Results window, double-click on a document name (the first one probably contains the most references). Acrobat opens the document and highlights the text. To go to the next or previous occurrence of the text, use the Search Next or Search Previous button on the Acrobat toolbar.

Grabbing Text from the Screen

You can cut and paste text from this document into the clipboard using the Text tool. For example, you could copy a code segment and paste it into the OMNIS method editor.

Getting Help

For more information about using Acrobat Reader see the PDF documents installed with the Reader files, or select the **Help** menu on the main Reader menu bar.

Start manual