OC.doc

COLLABORATORS			
	TITLE : OC.doc		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		September 19, 2022	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

OC.doc

1.1 OC.doc

```
                    $RCSfile: OC.doc $
Description: Documentation for the Oberon-A compiler

 Created by: fjc (Frank Copeland)
  $Revision: 4.2 $
    $Author: fjc $
      $Date: 1994/08/08 21:09:14 $

Copyright © 1994, Frank Copeland.
```

```
New links are marked with a "+".
Changed sections are marked with a "*" in the link.


                Description
                            What is OC?

                Distribution
                           Copyright and distribution

                Requirements
                           What do I need to run OC

Running OC...

                From the CLI *

                From the Workbench

                From FPE
                  Oberon-2 *             The programming language Oberon-2

                Extensions *
                         Language extensions supported by the compiler

                Module SYSTEM *
                       The pseudo-module SYSTEM
```

1.2 What is OC?

OC is a (fairly) fast single pass compiler that directly generates MC68000 machine code. The object files it produces are in standard AmigaDOS format and are linkable with BLink.

The compiler translates source code written in the Oberon-2 language described in the Oberon-2 Report by Niklaus Wirth and Hanspeter Mössenböck. It also supports a number of compiler options and language extensions that allow direct access to the Amiga operating system without messy assembler "glue code".

1.3 Distribution and Copyright

OC is part of Oberon-A and is:

1.4 System requirements

OC requires an Amiga personal computer with at least 1 MB of RAM ↩
,
running Kickstart version 1.3 or greater. Depending on the module being compiled, 500K or more of free RAM must be available to run the program. A conscious effort has been made to make sure that OC will run under Kickstart 1.3, but it was developed under Kickstart 2.04 and it was not possible to test this. It is likely that at some future date OC will require Kickstart 2.04 or greater. If this is a problem for you, please contact
the author
.

1.5 Running OC from the CLI

```
Usage:          OC {option} {<filename>}

Options:        NS | NEWSYMFILE
                {SYM | SYMBOLS <directory>}
                DST | DESTINATION <directory>
                BATCH
                DEBUG
                TEXTERR

Purpose:        To translate an Oberon-2 source text into MC68000
                machine code.

Path:           Oberon-A/C/OC
```

OC can be operated in three modes: interactive, command line and batch.

If no filename is given in the arguments passed to OC, it will enter interactive mode and repeatedly prompt the user for the name of a file to be compiled. It will exit when the user presses <enter> in response to the prompt. If a name is entered, it will attempt to compile that file.

If the BATCH argument is passed to OC and one or more filenames have been specified it will enter batch mode. In batch mode OC will attempt to open all the files passed as arguments and interpret their contents as the names of files to be compiled.

If one or more filenames are given and the BATCH keyword is omitted, OC

will attempt to compile the files named in the command line.

OC skips anything in a source file before the first "MODULE" symbol. If there isn't one, it will scan the whole file before reporting an error. This feature allows the programmer to include a header in the file which may be meaningful to another translator. For instance, the file might start with a sequence of commands that the AmigaDOS Execute command can interpret as commands to compile and link the module contained in the file.

If any errors are detected, their location and description are output in the file "<module>.err" in the current directory. By default, this file is in a binary format, intended to be read by an error listing utility (see Error file format). However, if the TEXTERR argument is specified, OC will output an ASCII file, readable by humans.

If there are no errors, an object file (containing machine code, data and relocation information) is output in the current directory, with a name consisting of the name of the module specified in the text plus a ".Obj" extension.

If the compiler cannot find a symbol file for the module, it will output one in the current directory. A symbol file contains information about the constants, types, variables and procedures exported by a module and is used by the compiler if the module is imported by another module. If the NEWSYMFILE option is specified and the module's definition has been changed, the compiler will replace the existing symbol file. If the module's definition has changed and NEWSYMFILE is NOT specified, an error (obsolete symbol file) is reported. The symbol file's name consists of the module name plus a ".Sym" extension.

When searching for symbol files, the compiler will look in the current directory first. If any SYMBOL parameters have been specified, it will then search those directories in the order they have been given. If unsuccessful, it will search "OLIB:". If a DESTINATION parameter is specified, the compiler will output any object and symbol files it generates in the specified directory instead of in the current directory.

If the DEBUG parameter is specified, the compiler will generate symbol hunks containing the names of all procedures and other objects in the module. These hunks are for the benefit of third-party debuggers. To exclude them from the final program, use the NODEBUG option with BLink or the equivalent if you use another linker.

The CLI stack should be set to at least 10000 bytes. See the Stack command in the AmigaDOS manual.

Typing in the full command line can become tedious. It is suggested that you adopt a consistent strategy for storing the source, symbol and object files of a project. The author keeps each project in a seperate directory and creates a sub-directory called "Code" to hold the symbol and object files. It is suggested that all library modules' symbol and object files be kept in the "OLIB:" directory, which the compiler automatically searches. A Shell alias can then be created to simplify calling the compiler:

```
  alias OComp OC SYM Code DST Code DEBUG []

A module can then be compiled by typing:

  OComp <module>.mod

Other aliases can be created for compiling library modules and doing
batch compiles. See the file Oberon-A:S/Oberon-Startup for some
suggested aliases.

Examples:

    OC DST OLIB: DEBUG Intuition.mod
    OC NS SYM Code/ DST Code/ OCE.mod
```

1.6 Running OC from the Workbench

```
The compiler cannot be run from the Workbench.  Sorry.  This will come
in a future version.
```

1.7 Running OC from the FPE utility

```
A tool button in the FPE window can be configured to  run  the  compiler
(see FPE.doc).  In the button editor, set the Command field to the  full
path name of the OC program.  Set the Arguments field to "!F"  plus  any
options that are desired.  Specify  a  console  window  as  the  Console
field.  Put at least 10000 in the stack field.

For example:

Command="DH1:Oberon-A/OC"
Arguments="!F SYM OLIB: SYM Code/ DST Code/"
Console="CON:0/11/540/189/Compiling.../CLOSE/WAIT"
Stack=10000

To compile a source file:

  1. select the module in the Module gadget.
  2. select the file extension from the Files gadgets.
  3. click on the tool button the compiler is bound to.
  4. sit back and relax for a bit.
```

1.8 Language extensions supported by the compiler

```
                There are two justifications for a compiler allowing deviations ↩
                    from  a
computer  language's  formal  definition.   One  is  to  "improve"   the
language; the other is to provide  machine-dependant  facilities.   With
one exception, all the language extensions supported  by  this  compiler
are  machine-dependant  facilities.   In  order  to  use  any  of  these
```

extensions, the global compiler switch $P must be set to FALSE (see

Compiler Switches
).

String and char literals *

Pointer variants

Assignable procedures

Amiga library functions

Foreign code interface +

Register parameters +

Variable length parameter lists *

Examples

1.9 String and char literals

The Oberon-A compiler allows the use of escaped characters in character and string constants. An escaped character consists of a "\" character followed by one or more characters. The "\" character indicates to the compiler that the following character(s) has special meaning. The meanings are:

```
\0, \o : insert a nul (NUL, 0X) character.
\b     : insert a backspace (BS, 08X) character.
\e     : insert an escape (ESC, 1BX) character.
\t     : insert a tab (HT, 09X) character.
\n     : insert a newline (LF, 0AX) character.
\v     : insert a vertical tab (VT, 0BX) character.
\f     : insert a form-feed (FF, 0CX) character.
\r     : insert a carriage return (CR, 0DX) character.
\xnn   : insert the character with ASCII value nn hex.
```

For any other combination, the compiler ignores the "\" character and inserts the following character. So, to insert a "\" character, use the sequence "\". The most common use for this mechanism is to insert formatting characters in strings to be output to the console, making the task of console IO simpler.

If two literals are seperated only by whitespace, the compiler will concatenate them. This can be used either to improve the formatting of source code, or to get around the 256-character limit for strings.

Examples:

```
CONST
```

```
  ugly   = "This is an ugly multi-line string\nfor an EasyRequest\n";
  pretty = "This is a prettier multi-line string\n"
           "for the same EasyRequest\n";
```

1.10 Pointer variants

Oberon defines only one kind of pointer; this compiler provides two variants. Pointer variants are indicated by alternatives to the normal POINTER keyword. A normal, non-variant pointer is referred to here as an Oberon pointer.

The first pointer variant is known as a CPointer. It is declared with the CPOINTER keyword. It corresponds to the notion of a pointer used by the C language. Such pointers are endemic in the Amiga operating system; this is not surprising since it is largely written in that language. They differ from normal Oberon pointers in two ways: they may have ANY type as a base type; and there is no type tag associated with the object they point to. This last feature means that they cannot be used in type guard statements. Otherwise, they may be used in the same way as any normal Oberon pointer. However, it is recommended that they only be used when declaring objects required by a module providing an interface to the Amiga operating system.

The second variant is known as a BPointer. It is declared with the BPOINTER keyword. It represents a BCPL pointer, a curious object that is found in many data structures used by AmigaDOS. It is a longword pointer; that is, a byte address divided by 4. It is treated by the compiler in much the same way as a CPointer. The compiler automagically generates the code needed to convert between longword and byte addresses when such a pointer is dereferenced.

Examples:

```
TYPE
  Object = POINTER TO ObjDesc;                (* An Oberon pointer *)
  STRPTR = CPOINTER TO ARRAY 32767 OF CHAR;   (* A CPointer *)
  FileHandlePtr = BPOINTER TO FileHandleRec; (* A BPointer *)
```

The standard procedure NEW may be used to allocate memory for CPointer and BPointer variables, as well as the SYSTEM.NEW procedure. Both will handle the special requirements of BPointers transparently. SYSTEM.DISPOSE can be used to free the memory once it is no longer required.

You are also free to use the Amiga memory allocation functions such as AllocMem() and FreeMem() with CPointer and BPointer variables. Note that if you do this with BPointers, you will have to handle the necessary shifting yourself.

DO NOT USE AMIGA MEMORY ALLOCATION FUNCTIONS WITH OBERON POINTERS. The compiler makes assumptions about the structure of the memory allocated to an Oberon pointer that are only valid if it is allocated with NEW. The compiler will not allow you to use SYSTEM.NEW with Oberon pointers where this is inappropriate.

1.11 Assignable procedures

Procedures that are to be assigned to procedure variables must be marked with a "*" character, unless they are marked as exported. This tells the compiler to treat them as if they were exported, without making them visible outside the module. This mainly involves generating code to save, set and restore register A4 as the global variable base pointer. The mark character must be placed immediately after the PROCEDURE keyword. For example:

```
PROCEDURE * Assignable;
          ^
          Mark character
```

1.12 Amiga library functions

Amiga system software is accessed through shared code libraries. An Amiga shared library consists of a block of variables and a table of jump instructions. There is one of these jump instructions, known as a function vector, for each function provided by the library. Each vector is accessed by a negative offset (known as the function vector offset) from the base of the library's variables. A library function is called by placing the address of the library variables in register A6 and coding a "JSR offset(A6)" instruction, where "offset" is the vector offset of the desired function. Parameters are placed in specific registers before the function call and results are also returned in registers. See the Amiga ROM Kernel Manual for an in-depth discussion of this process.

The simplest method for a compiler to interface with Amiga library calls is to require that the programmer declare a normal procedure and use assembly language stubs or facilities such as SYSTEM.PUTREG to set up the parameters and make the call. This is an inefficient and error-prone system and most recent compilers, including Oberon-A, provide a means for describing library calls in such a way that the compiler can generate the call directly.

Syntax
The formal syntax of LIBCALL declarations

Semantics
The semantics of LIBCALL declarations

1.13 The formal syntax of LIBCALL declarations

The syntax for declaring Amiga library calls used by OC is similar to the syntax for type-bound procedures in Oberon-2, which Amiga library calls closely resemble:

```
$ LibCallDeclaration = LibCallHeading ";"
$ LibCallHeading = LIBCALL Receiver identdef [RegParameters]
    ["-"] integer.
```

The integer value is the library function vector offset. It may be negative; if not, the compiler will negate it anyway.

```
$ Receiver = "(" ident ":" ident ")".
```

The first ident is a dummy. The second must be a type identifier that has been declared earlier in the same module. The type must be a CPOINTER TO RECORD (a CPointer to a record, see Pointer variants) and should be an extension of the Exec.LibraryPtr type (Amiga Resources are extensions of the Exec.NodePtr type).

1.14 The semantics of LIBCALL declarations

A LIBCALL procedure is bound to a library base type (which is specified in parentheses before the procedure name). This means that to call it you must first declare a variable of the library base's type and initialise the variable with the address of the base of the corresponding shared code library (use the Exec OpenLibrary function for this). The procedure is then called as if it were a type-bound procedure bound to the library base type.

LIBCALL procedures are NOT inherited by extensions of the type they are bound to.

1.15 Variable-length parameter lists

Utility library taglists are now commonly used to pass parameters to Amiga system functions that deal with complex objects. Passing tags as arrays of TagItems is effective but verbose. Oberon-A allows the programmer to avoid this by passing a variable number of parameters to a

LIBCALL or foreign code procedure, in a manner similar to C vararg parameters. Note that this facility is only available for LIBCALL and foreign code procedures, which pass their paramaters in CPU registers.

A parameter is declared to be a VarArg parameter by placing an ellipsis ("..") symbol after the register specification. Only one parameter can be so marked, and it must be the LAST parameter. It cannot be a VAR parameter. It must be a register parameter.

A formal VarArg parameter may be replaced with one or more actual

```
parameters, seperated by commas. Each actual parameter must be
assignment compatible with the VarArg formal parameter.
```

1.16 Examples of declaring and using LIBCALLs

```
Libcall example:

LIBCALL (base : ExecBasePtr) OpenLibrary*
  ( libName {9} : ARRAY OF CHAR;
    version {0} : Types.ULONG )
  : Types.APTR;
  - 552;

This defines the Amiga Exec library function OpenLibrary.  It  indicates
that it is bound to the ExecBasePtr type.  It is marked for export.   It
has two parameters: libName is an ARRAY OF CHAR whose address is  to  be
passed in register A1; version is a ULONG (effectively a LONGINT) to  be
passed by value in register D0.  The function  returns  an  APTR  value.
Its jump vector can be found 552 bytes before the library base  address.

Assuming that it has been declared in module Exec, along with a variable
called Base of type ExecBasePtr, a call of  OpenLibrary  ()  might  look
like this:

  DiskFontBase := Exec.Base.OpenLibrary ("diskfont.library", 33);


VarArgs example:

...

LIBCALL (base : IntuitionBasePtr) OpenWindowTagsA*
  ( newWindow {8}   : NewWindowPtr;
    tagList   {9}.. : U.Tag )
  : WindowPtr;
  -606;
...

VAR w : I.WindowPtr;

BEGIN
  ...

  w := I.Base.OpenWindowTagsA (
      NIL,
      I.waFlags,     { I.wflgDepthGadget, I.wflgDragBar,
                       I.wflgCloseGadget, I.wflgSizeGadget },
      I.waIDCMP,     { I.idcmpCloseWindow },
      I.waMinWidth,  minWindowWidth,
      I.waMinHeight, minWindowHeight,
      U.tagEnd );
  ...
END ...
```

1.17 The pseudo-module SYSTEM

Every Oberon implementation includes a pseudo-module called ↩ SYSTEM,
defined internally in the compiler. Its purpose is to provide machine-dependant and low-level facilities that cannot otherwise be expressed in the Oberon language. The SYSTEM module provided with Oberon-A is based on the module defined for the Ceres compiler but contains several differences.

Data types *
Data types exported by SYSTEM

Memory management *
Allocating and deallocating memory

Memory access
Peeking and poking and addresses

Bit operations
Bit twiddling

Inline code
Why bother with a compiler?

Type tag handling
Manipulating type tags

Miscellaneous *
And all the rest...

Reference *
Module SYSTEM Reference

1.18 Data types

All data types imported from the pseudo-module SYSTEM must be qualified with the name of the module or an alias. For example, WORDSET must be referred to as SYSTEM.WORDSET.

The SET type in Oberon-A is a 32 bit entity. However, many Amiga data structures contain the equivalent of sets that are 8 and 16 bit entities. These smaller sets are represented by the BYTESET (8 bit) and WORDSET (16 bit) types exported by module SYSTEM. All the normal set operations may be performed on these types. The different set types are NOT compatible; sets of different types may not be mixed in expressions or assigned. Set constants have their types automatically adjusted by the compiler to conform to the type of set they being used with.

The operation of the LONG and SHORT standard procedures has been extended to deal with set type conversions. The $P compiler switch must

be set to FALSE to get access to these extensions. The LONG procedure will convert a BYTESET to a WORDSET and a WORDSET to a SET. The SHORT procedure will convert a SET to a WORDSET and a WORDSET to a BYTESET. This is the only supported method of mixing set types in assignments and expressions.

Module SYSTEM exports three anonymous types, BYTE, WORD and LONGWORD. These types are compatible with any other type with the same or fewer number of bits. Any 8-bit type (SHORTINT, CHAR, and BOOLEAN) may be assigned to a variable or parameter of type BYTE. In addition, a variable of any type may be passed to a formal variable parameter of the type ARRAY OF BYTE. Any 8-bit (see above) or 16-bit type (INTEGER and WORDSET) may be assigned to a variable or parameter of type WORD. Any 8-bit, 16-bit (see above) or 32-bit type (LONGINT, SET, real types, pointers and procedures) may be assigned to a variable or parameter of type LONGWORD. Where the value being assigned is smaller than the variable or parameter type, it is extended to fit. Integers are sign-extended and all other types are zero-extended.

Three pointer types are exported: PTR, CPTR and BPTR. These are used as anonymous pointer types and are analogous to the LONGWORD type. Any Oberon pointer may be assigned to a variable or parameter of type PTR. Any CPointer may be assigned to a variable or parameter of type CPTR. In addition, a CPTR may be assigned to any variable or parameter of a CPointer type. Any BPointer may be assigned to a variable or parameter of type BPTR. No other operations except comparisons with and assignment of NIL are allowed for these types.

The VAL function procedure is used to cause the compiler to treat an object of one type as if it had another type. This version of the compiler does not insist that the two types have the same size. This can cause unexpected problems with a big-endian processor like the MC68000. For example, if you convert a 32 bit type to a 16 bit type, you may end up accessing the _upper_ 16 bits of the original object when you really wanted the _lower_ 16 bits.

1.19 Memory management

The SYSTEM.NEW procedure is used to allocate a block of memory with an arbitrary size. Such a block does NOT have a type tag associated with it, so do not use this procedure to allocate a record structure through an Oberon pointer. An optional third parameter can be used to indicate the memory requirements for the allocated block.

The DISPOSE procedure is used to explicitly free the memory associated with any pointer variable. Great care must be taken with this procedure, since it introduces the possibility of errors such as hanging pointers that Oberon is attempting to eliminate. The only valid use for DISPOSE is to free memory allocated to a CPointer or BPointer, using SYSTEM.NEW. DISPOSE makes sure that it has been passed a valid pointer and causes a processor trap to occur if it has not. It can be quite slow to execute in some circumstances (especially when freeing a pointer allocated in the middle of a large number of other allocations).

1.20 Memory access

The ADR procedure is used to find the run-time address of any variable or string constant. The result may be assigned to any CPointer. The BIND procedure is used to assign the address of a record variable to a CPointer variable while ensuring that the types of the two objects are compatible.

The BIT procedure is used to test an individual bit at a given memory location. Procedure GET is used to read a value at a given memory location while PUT is used to write one.

1.21 Logical operations

LSH, ROT, LOR, AND and XOR perform bit operations on most basic types. The legal types are: BYTE, WORD, LONGWORD, CHAR, BYTESET, WORDSET, SET, SHORTINT, INTEGER and LONGINT.

LSH is similar to ASH but performs a logical shift instead of an arithmetical shift (the difference is in the treatment of the sign bit). ROT performs a bitwise rotation of the argument. LOR performs a bitwise OR, AND a bitwise AND and XOR a bitwise exclusive-OR. Note that these operations do not change the type of the operand, unlike ASH which promotes its parameter to a LONGINT.

1.22 Inline machine code

PUTREG is used to place a value in a specific CPU register. GETREG is used to read the value in a register. INLINE is used to insert machine code directly in the code buffer. It will output either a word or a longword, depending on the size of the type of the argument. INLINE will accept any number of parameters.

SETREG and REG are provided for compatibility with AmigaOberon. SETREG is exactly the same as PUTREG. REG is similar to GETREG, except that it is a function procedure, whose return type is a LONGWORD.

1.23 Manipulating type tags

A type tag is a pointer to a type descriptor, which contains information used by the memory allocator, the garbage collector, and when calling type-bound procedures. In some circumstances it is useful to have access to type tags, especially when working with persistent objects.

Module SYSTEM exports a type, TYPETAG, which is used by the procedures that deal with type tags. It is similar to the PTR type. The only operations permitted are assignment of other TYPETAG variables, assignment of NIL, and comparison with NIL.

The TAG procedure returns the type tag associated with a RECORD type, or the base type of a POINTER TO RECORD type. It can also be used to get the type tag of a POINTER TO RECORD variable, or a VAR parameter of a RECORD type.

The SIZETAG procedure returns the size in bytes of the type whose tag is passed as a parameter.

The GETNAME procedure copies the name of the type associated with a type tag into an ARRAY OF CHAR variable. The name is of the form "<module>.<type>".

The NEWTAG procedure creates a new object whose type is determined by the type tag passed as a parameter. IT DOES NOT GUARANTEE THAT THE DYNAMIC TYPE OF THE NEW OBJECT IS COMPATIBLE WITH THE STATIC TYPE OF THE VARIABLE(TO WHICH IT IS ASSIGNED.

1.24 Miscellaneous

Procedure ARGS fetches the raw argument data passed to the program by AmigaDOS or Workbench. This will either be a pointer to a string containing the command line received from AmigaDOS, or a pointer to the Workbench startup message. Procedure ARGLEN is used to determine which it is. It returns -1 if the program was started by Workbench and the ARGS result should be interpreted as a message pointer. Otherwise it returns the length of the command line string and the ARGS result is a pointer to that string. See the implementation of module Args for an example of how to use these procedures.

Procedure SETCLEANUP is used to specify a procedure that is to be executed after the program has formally ended, either normally or as the result of a HALT or a processor trap. The procedure must be paramaterless and typeless. It is intended that this procedure be used to clean up after the program, returning resources to the system. There is, however, no obligation on the programmer to use it in this way.

The function procedure RC is used to obtain the current return code for the program. This is normally 0, but if a HALT or ASSERT statement has been executed, it will be set to the value given as a parameter. This function is intended to be used inside a cleanup procedure, so that it can determine the reason for the program's termination.

Procedure MOVE is used to copy an arbitrary sequence of bytes from one memory location to another. It is able to deal correctly with overlapping blocks.

1.25 Module SYSTEM Reference

Function Procedures

v stands for a variable, x, y, a and n for expressions and T for a

type. r stands for a register (0 <= r < 16).

Name	Argument type	Result type	Function
ADR(v)	any	CPTR	address of variable v, or string constant v
AND(x, y)	x, y: basic type	larger type	bitwise AND
BIND(T,v)	T: CPointer type v: record type	T	address of variable v, asserting its type is a base type of T
BIT(a, n)	a: LONGINT n: integer type	BOOLEAN	Mem [a][n]
LSH(x, n)	x, n: basic type	type of x	logical shift
OR(x, y)	x, y: basic type	larger type	bitwise OR
RC()	none	LONGINT	return code passed in HALT or ASSERT statement, or 0.
REG(r)	r: register number	LONGWORD	contents of register r
ROT(x, n)	x, n: basic type	type of x	rotation
SIZE(T)	any type	integer type	size of T in bytes
SIZETAG(tt)	TYPETAG	LONGINT	Returns the size of the type owning the type tag.
TAG(v) TAG(T)	Any pointer or record type	TYPETAG	Returns the type tag for a variable or type.
VAL(T, x)	T, x: any type	T	x interpreted as type T
XOR(x, y)	x, y: basic type	larger type	bitwise exclusive OR

Proper Procedures

v stands for a variable, x, y, a and n for expressions and T for a type.

Name	Argument types	Function
ARGLEN(v)	v : LONGINT	v := length of command line, or -1 if the program started from Workbench
ARGS(v)	v : LONGINT	v := command line or workbench startup message

DISPOSE(v)	any pointer type	free memory allocated to v
GC	none	garbage collect memory
GET(a, v)	a: LONGINT v: any basic type	v := Mem [a]
GETNAME(tt,v)	tt: TYPETAG v: ARRAY OF CHAR	Copies the type name into v
GETREG(r, v)	r: register number v: any basic type	v := R[r]
INLINE(x1,..,xn)	integer constant	insert x1 .. xn into code
MOVE(v0, v1, n)	v0, v1: any type n: integer type	assign first n bytes of v0 to v1
NEW(v, n)	v: any pointer type n: integer type	allocate block of n bytes and assign its address to v
NEW(v, n, a)	v: any pointer type n: integer type a: SET	allocate block of n bytes and assign its address to v. Memory requirements are passed in a.
NEWTAG(v,tt)	v: Pointer type tt: TYPETAG	Allocates a new object using the type tag.
PUT(a, x)	a: LONGINT x: any basic type	Mem [a] := x
PUTREG(r, x) SETREG(r, x)	r: register number x: any basic type	R[r] := x. SETREG and PUTREG are synonyms.
SETCLEANUP(p)	p: procedure	make p the global cleanup procedure

1.26 Controlling the compiler with switches

The behaviour of the compiler is to some extent under programmer control. This control is exercised through compiler switches.

Compiler switches are embedded in comments and consist of a "$" character, a single letter and a "+", "-" or "=" character. There must be no spaces between any of these characters. Any number of switches may appear in the same comment. The switches are boolean variables and their value is set by the trailing character, TRUE for "+", FALSE for "-" and the default value for "=".

Module switches must be specified before any imports or declarations and apply to the entire module. Global switches may be specified at any point in the source file and apply until changed or until the end of the module. Procedure switches apply to the first procedure body following the switch; at the end of the procedure body the switch returns to its default value. Note that it is procedure *body*, not

```
procedure *declaration*; watch out for this with nested procedures.
```

```
The currently supported switches are:

Module switches


               $P
                Portable code

Global switches


               $C
                Case checking

               $I
                Index checking

               $L
                Addressing global variables

               $N
                NIL pointer checking

               $R
                Range checking

               $S
                Stack checking

               $T
                Type checking

               $V
                Overflow checking

               $Z
                Zeroing variables

Procedure switches


               $A
                Saving all registers

               $D
                Copying open arrays

               $r
                RETURN checking

               $s
                Saving non-scratch registers
```

1.27 Copying open arrays

```
(* $D+ enable copying of open arrays *)
(* $D- disable copying of open arrays *)
```

$D+ is the default. When TRUE, the compiler will generate code at the entry to each procedure to make copies of any open array value parameters. When FALSE, this code is suppressed. Use this switch for efficiency if you know an open array value parameter will be strictly read-only. Great care must be taken with this switch, since it effectively converts value parameters into variable parameters, bypassing the compiler's normal safety checks. The programmer MUST make sure not to make any assignments to such parameters, otherwise undesirable side effects such as writing over string constants may occur. This is a procedure switch.

1.28 Index checking

```
(* $I+ enable array index checking *)
(* $I- disable array index checking *)
```

$I+ is the default. When FALSE, the compiler will suppress the generation of code to check that variables used to index arrays contain legal values. This is a global switch.

1.29 Addressing global variables

```
(* $L- access global variables through A4 *)
(* $L+ use absolute long addressing for global variables *)
```

$L- is the default. When FALSE, the compiler generates code at the entry to each exportable procedure to set up the A4 register to point to the module's global variables. The old value in A4 is saved and restored on exit from the procedure. This allows the most efficient access to global variables at the expense of several bytes (currently 10) of overhead per procedure. When TRUE, this code is suppressed and global variables are accessed with less efficient absolute 32-bit addresses. Set this switch to TRUE to minimise code size when you know that no global variables will be accessed in a procedure, OR IN ANY LOCAL PROCEDURE CALLED DIRECTLY OR INDIRECTLY BY IT. This is a global switch.

1.30 Checking memory allocations

```
(* $N+ check for NIL pointers. *)
(* $N- ignore NIL pointers. *)
```

$N+ is the default. When TRUE, the compiler generates code to check for NIL pointers and procedure variables whenever they are dereferenced or used in type guards or procedure calls. This is a global switch.

1.31 Portable code

```
                (* $P+ allow only portable code *)
(* $P- allow non-portable code *)
```

$P+ is the default. When TRUE, the compiler will only translate the Oberon-2 language as defined in the Oberon-2 Report. When FALSE, the compiler will recognise a number of language extensions (see

Language Extensions
). This is a module switch.

1.32 Range checking

```
(* $R+ enable range checking *)
(* $R- disable range checking *)
```

$R+ is the default. When TRUE, the compiler will generate code to check that variables contain legal values for particular operations. This includes the IN, INCL and EXCL set operations and type conversions using CHR and SHORT. When FALSE, this code is suppressed. This is a global switch.

[Not all of these checks have been implemented as yet. FJC]

1.33 Type checking

```
(* $T+ enable type checking *)
(* $T- disable type checking *)
```

$T+ is the default. When FALSE, the compiler will suppress the generation of code to perform run-time type checks on variable record parameters and pointers. This is a global switch.

1.34 Overflow checking

```
(* $V+ enable overflow checking *)
(* $V- disable overflow checking *)
```

$V+ is the default. When FALSE, the compiler will suppress the generation of code to check for arithmetic overflow at run-time. This is a global switch.

1.35 Zeroing variables

```
(* $Z- suppress zeroing of variables *)
(* $Z+ zero all global and local variables *)
```

$Z- is the default. When TRUE, the compiler generates code in the main body of the module and on entry to each procedure to fill all variables with zeroes. This guarantees that variables have a predictable initial value. Numeric variables are set to zero; booleans are set to FALSE; pointers are set to NIL; sets are empty, ie - {}. When FALSE, this code is suppressed and variables have their values undefined. Why is the default set to FALSE? Because in the language report the initial values of variables are undefined and if you write programs that make assumptions about initial values they will not be portable. The default forces you to explicitly state that your program is relying on non-standard behaviour from the compiler. This is a global switch.

1.36 RETURN checking

```
$r+ enable checking for RETURN statements in function procedures
$r- disable checking for RETURN statements in function procedures
```

$r+ is the default. When TRUE, the compiler will check that there is at least one RETURN statement in a function procedure (a procedure with a return type) and will also generate code to cause a processor trap if the procedure exits without executing one. This is a procedure switch.

1.37 Using the garbage collector

Oberon-2 was designed under the assumption that programs written in it would be running in an environment that provided automatic garbage collection of memory. This is the reason why it has a NEW standard procedure but no DISPOSE. The Amiga's operating system does not provide this facility, so Oberon-A implements a garbage collector in the run-time support code linked with every program. This garbage collector must be used carefully, as it has the potential to free memory that is still in use.

The garbage collector is invoked by calling the GC procedure in the pseudo-module SYSTEM. When called, it works in two phases: a mark phase and a sweep phase. During the mark phase it traces all the global pointer variables and marks the memory they point to. If the marked memory contains other pointers, either as record fields or array elements, these are also traced and marked. When the mark phase is completed, the sweep phase processes a list of memory blocks allocated by the program, unmarking any marked blocks and freeing all unmarked blocks.

The point in the program at which the garbage collector is called is very important. The mark phase can only trace memory accessible from GLOBAL pointer variables. LOCAL pointer variables inside procedures cannot be traced. If such local variables are still active, the memory allocated to them will be freed, almost certainly leading to a crash.

To avoid this, the programmer must ensure that the garbage collector is only called at a point in the program where it is guaranteed that there? are no active local pkinter variables. An ideal place for this would be in the program's main event loop (if it is a GUI program). A counter variable should be used to limit the frequency at which the collector is activated; activating it every cycle of the loop would bring the system to a halt.

Another danger comes from using the SYSTEM.DISPOSE procedure. If there is more than one reference to memory freed with this procedure, the garbage collector will be tricked into believing that the memory is still allocated, causing it to write all over memory it doesn't own. If you cannot guarantee that you know of all references to a dynamically allocated variable and have assigned NIL to all of them, DO NOT USE SYSTEM.DISPOSE. Assign NIL to any global pointer variable you are finished with, and trust the garbage collector to handle any other references. This kind of bug is very difficult to track down. When it happened to the compiler, it took almost a week to find (and 30 seconds to fix). Debuggers were useless, as they were being crashed by random memory writes. You have been warned.

A number of library modules distributed with Oberon-A allocate memory in their operations. For the reasons given above, most do not call SYSTEM.DISPOSE. Module Files is a notable example, allocating from one to four 1K buffers for every file opened. If you use such modules intensively, you are more or less obliged to call the garbage collector periodically to avoid running out of memory.

Garbage collection applies only to Oberon pointers. CPointer and BPointer variables are not traced and the garbage collector ignores them. If you use NEW or SYSTEM.NEW to allocate memory to such pointers, you should use SYSTEM.DISPOSE to free them. This is equivalent to using C's malloc() and free() functions.

You are not forced to use either the garbage collector or SYSTEM.DISPOSE. Any memory allocated by a program that is not freed explicitly (with SYSTEM.DISPOSE) or implicitly (with the garbage collector), will be automatically returned to the system when the program ends. This happens even if the program crashes due to a processor trap or is summarily terminated with HALT or ASSERT. It also applies to memory allocated to CPointer and BPointer variables with NEW and SYSTEM.NEW. IT DOES NOT APPLY TO MEMORY ALLOCATED WITH THE AMIGA MEMORY ALLOCATION FUNCTIONS. The run-time system cannot track such memory and if it is not explicitly freed it will remain allocated and cause a memory leak.

1.38 Handling run-time errors

The compiler generates code fragments to check for a number of ↩ errors that may occur at run-time. These include arithmetic overflows, failed type guards, array index errors, etc. They can be enabled and disabled with compiler switches; they are all enabled by default. Typically run-time errors produce a processor trap with a TRAP or TRAPV instruction. The run-time support code built into every Oberon-A

program contains a trap handler which intercepts all compiler-generated traps and several others such as divide-by-zero. The default trap handler has the same effect as a HALT statement, causing the program to terminate. Any cleanup procedures installed with SYSTEM.SETCLEANUP will be executed and all memory allocated with NEW or SYSTEM.NEW will be freed. The return code will be set to the trap number + 100.

Module Errors gives an example of a cleanup procedure which checks the return code and puts up a requester describing the error. This example should give you enough information to write your own replacement, or a supplementary procedure that catches return codes it doesn't understand. If you know what you are doing, you could install your own trap handler through the trapCode field in the program's Task structure. See the Amiga RKM for details.

Error codes +

1.39 Currently defined return codes and processor traps

Err #21 : Return code = 21

The run-time support code failed to open mathffp.library. As this is in ROM, there is something seriously wrong with your Amiga.

Err #22 : Return code = 22

A pointer passed to SYSTEM.DISPOSE() does not point to memory allocated by the program. SYSTEM.DISPOSE() must only be used to free memory allocated using NEW() or SYSTEM.NEW().

Err #23 : Return code = 23

An attempt has been made to divide a LONGINT value by zero.

Err #99 : Return codm = 99

By convention, procedures and methods (type-bound procedures) which are only stubs to be implemented later should contain the statement HALT (99) if they are not meant to be called.

Err #100 : Return code = 100

By convention, if an Amiga shared code library _must_ be opened, and the attempt fails, the program should call HALT (100), or ASSERT (mumble, 100).

Trap #3 (Address Error) : Return code = 103

This is likely to mean that the program has attempted to dereference an un-initialised pointer. If it contains an odd address, trying to access a word or longword value will cause this trap to occur.

```
Trap #4 (Illegal Instruction) : Return code = 104
Trap #10 (Line 1010 emulator) : Return code = 110
Trap #11 (Line 1111 emulator) : Return code = 111

  The program has probably gone mad and is trying to execute random
  data as if it was code. This can happen if you try to execute an
  un-initialised procedure variable, or call an Amiga library function
  without opening the library first.

Trap #5 (Divide by zero) : Return code = 105

  An attempt has been made to divide a number by zero.

Trap #6 (CHK instruction) : Return code = 106

  If compiler range checking is on, this trap will occur if an attempt
  is made to use a value that is not in the legal range for the
  operation being attempted. For example, the expression "32 IN
  setVariable" will cause a trap because the maximum element in a set
  is 31.

  If compiler index checking is on, this trap will also occur if the
  index expression in an array access is out of range. For example,
  "arrayVariable [-1]" will cause a trap, as will "arrayVariable [LEN
  (arrayVariable)]".

Trap #7 (TRAPV instruction) : Return code = 107

  An overflow has occurred in an arithmetic expression.

Trap #32 : Return code = 132

  A compiler index check has failed. This is basically the same as Trap
  #6.

Trap #33 : Return code = 133

  A type guard statement has failed. For example "myNode(Exec.Node)"
  when myNode is only an Exec.MinNode.

Trap #34 : Return code = 134

  A call to NEW or SYSTEM.NEW has returned a NIL pointer.

Trap #35 : Return code = 135

  A case statement has been given a value not in its case label list,
  and it does not have an ELSE part.

Trap #36 : Return code = 136

  A function procedure has attempted to exit without executing a RETURN
  statement.

Trap #37 : Return code = 137

  A procedure has been called with insufficient stack remaining.
```

'Insufficient' means less than 1500 bytes. 1500 bytes might seem like a lot, but some dos.library functions require this much stack, and there is no way of knowing if such a function will be called by the procedure.

1.40 Error reports produced by the compiler

Any errors detected by the compiler are listed in the file "<module>.err" in the current directory.

By default this file is in a binary format, intended for use with an error lister utility like OEL. The first four bytes contain the tag "OAER", which is used to confirm that the file is indeed an error file. The file format, in EBNF, is:

```
ErrorFile = tag {error}
tag       = "OAER"
error     = line:2 col:2 errorCode:2
```

If the TEXTERR option is specified on the command line, the error file is written in human-readable form. Errors are listed one per line, with the following format:

"line <line#>, col <column#>: err = <error#>"

Lines and columns are numbered starting at 1. The meaning of each error number is listed in the file ErrorCodes.doc.

1.41 Implementation of basic types

The Oberon Report leaves the precise format and size of most basic types up to individual implementations. The relevant data for Oberon-A are:

Type	Size	MIN	MAX
SHORTINT	8 bits /1 byte	-128	127
INTEGER	16 bits/2 bytes	-32768	32767
LONGINT	32 bits/4 bytes	-2147483648	2147483647
REAL	32 bits/4 bytes	-9.22337177E18	9.22337177E18
LONGREAL	32 bits/4 bytes	-9.22337177E18	9.22337177E18
CHAR	8 bits /1 byte	0X	255X
BYTE	8 bits /1 byte	0	255
SYSTEM.BYTESET	8 bits /1 byte	0	7
SYSTEM.WORDSET	16 bits/2 bytes	0	15
SET	32 bits/4 bytes	0	31
Pointers	32 bits/4 bytes	N/A	N/A

Note that REAL and LONGREAL are identical in this implementation. They both conform to the Motorola Fast Floating Point standard. In a future version, LONGREAL will be re-implemented as an IEEE double-precision real. REAL may also be re-implemented as an IEEE single-precision real.

I will be guided in this by the compiler's users.

1.42 Limits built in to the compiler

* The compiler cannot evaluate constant expressions that contain REAL or LONGREAL values. You can still have real literals, just no arithmetic. The compiler will report an error if you attempt to do this.

* The code generated for a module cannot be greater than 32K in size. This limit is inherent in the addressing model used for generating machine code. Split large modules into smaller pieces.

* No more than 32K bytes of string literals or 64 type tags may be generated for a module. These limits can be changed by editing two constants in module OCC and recompiling it; the absolute maximums are 32K for each.

* There is a limit of about 80K on the size of the buffer used to store names. This is dynamically allocated and grows as needed, so there is no wasted RAM. The most I have seen used for this is about 60K.

* No more than 32K of local variables can be declared for a procedure. What do you mean you want more? Use dynamic allocation.

* The size of the parameters for a procedure cannot exceed 1500 bytes. This is necessary for the stack checking code to work. If anyone exceeds this limit, I would be very interested to know.

* There is no limit on the size of a module's global variables but variables more than 32K from the module's variable base will be less efficient to access.

* An array type may not have more than 32K elements (however, it may be larger than 32K bytes). This limit will disappear in a future version.

* Identifiers and string literals cannot be more than 255 characters long. This is primarily a limit imposed by module OCS, but increasing it may affect assumptions made in other parts of the compiler. The original limit for identifiers was 31 characters and this may still lurk in dark corners of the source code. Of course, if you need identifiers longer than this, stop writing variable names in German :-). Module names are limited to 26 characters. This limit is imposed by AmigaDOS.

* String literals longer than 1 character cannot be aliased if they are imported from another module. By this I mean, you cannot declare a constant such as:

```
CONST Alias = AnotherModule.StringConstant;
```

  where StringConstant is a string literal longer than 1 character. This limit will probably disappear in a future version. It will happen quicker if people complain :-).

* There are a number of arbitrary limits placed on the number of objects such as exported types, imported modules and the like. These limits allow the use of arrays for internal data structures, which are much more efficient than dynamically allocated lists. Most of these limits have been greatly increased from those in the Ceres compiler. If you still manage to exceed such a limit, a compiler error will be reported and you should easily be able to determine which constant to increase to get around it.

* The compiler needs at least 10000 bytes of stack and 500K or more of free RAM to run.

1.43 Who is responsible for THIS?

OC was ported to the Amiga by Frank Copeland. It is based on a compiler written by Niklaus Wirth.

For information on how to contact the author, see Oberon-A.doc.

1.44 Reporting bugs and suggestions

This version of OC is a beta-test version. That means that it is basically complete, but has not been rigorously tested. Bug fixes and suggestions from users of this version will be incorporated in future versions. You are encouraged to report any and all bugs you find, as well as any comments or suggestions for improvements you may have.

Before reporting a suspected bug, check the file ToDo.doc to see if it has already been noted. If it is a new insect, clearly describe its behaviour including the actions necessary to make it repeatable. Indicate in your report which version of OC you are using. Include an example of a program or short fragment of code that demonstrates the bug.

I am especially interested in the following areas:

* Compatibility with different versions of the Amiga hardware and operating system. So far OC has only operated on a stock A500 with AmigaDOS 2.05 and a 20MB hard disk.

* How good/useful/helpful/complete the documentation is.

* How suitable OC is for use by programmers with varying levels of experience, from beginners to hackers.

* Departures from the language specification.

* Extensions to the language supported by the compiler.

* Memory management. I am unable to use Enforcer or similar utilities on my A500, so I would like people who can to report any

Enforcer hits they get. I am also concerned about possibly excessive memory fragmentation caused by the run-time memory allocator.

1.45 Who did what and why

OC is a port of a compiler written for the Ceres workstation by Niklaus Wirth. The book "Project Oberon" written by Wirth and Jürg Gutknecht contains a description of this compiler and the full source code for it. The original source can also be obtained by anonymous ftp from neptune.inf.ethz.ch. Many thanks to Professor Wirth for making this source code available.

The machine code generator for early versions of the compiler was a port of part of Charlie Gibb's A68K assembler. This code is no longer part of the compiler, but it was extremely useful in the early stages of development and debugging.

Part of the run-time library (the 32 bit arithmetic) is taken from the Sozobon C compiler and is:

Copyright (c) 1988 by Sozobon, Limited. Author: Johann Ruegg

1.46 Release history

0.0 The initial port to the Amiga, written in Modula 2 and compiled by the Benchmark compiler. Implemented the Oberon dialect. Never released. Started in February 1993.

0.1 The initial conversion from Modula 2 to Oberon, compiled by the v0.0 compiler. Never released.

0.2 - 0.3 Bug fixes and upgrades. Never released.

1.0 Start of revision control. Upgrades and bug fixes. Never released.

2.0 First public release. Compiler upgraded to Oberon-2. Released in May 1994.

3.0
* Changed command line arguments:
  - Options now must come first;
  - Multiple filename arguments allowed.
* Batch compiles implemented.
* OLIB: is now the default symbol file search path.
* Error files are output in the current directory with the name "<module>.err".
* Compiles can be interrupted with CTRL-C.
* [bug] Enforcer hit caused when no DST parameter was specified
* [bug] Same error code (#228) used for different errors.

3.1
* [bug] Batch file was not closed when batch compile interrupted by CTRL-C.

3.2 * [bug] Numerous bugs in the translation of type-bound procedures, especially when forward declared. It was a wonder they worked at all.

3.3 * [bug] Bug in type-bound procedures caused a crash due to stack corruption if no parameter list was specified.
* Checks for RETURN statements in function procedures. Generates code for run-time check as well. $r switch added to turn this on and off.

3.4 * Error #5 (end of file in comment) now reports the position of the start of the offending comment.
* [bug] Quick fix of problem with UNION types and exported fields.

3.5 Removed all references to UNION types. They were more more trouble than they were worth.

4.0 Implemented varargs.

4.1 * Reorganised symbol table as a binary search tree.
* Changed symbol file format, using compressed integers.

4.2 Intermediate version.

4.3 Added new features to Module SYSTEM.

4.4 * Fixed bug causing address trap when calling type-bound procedures through a dereferenced CPointer.
* Passing an empty string to an ARRAY OF CHAR LIBCALL parameter now passes a NIL pointer to the LIBCALL. However, this change introduced a bug, meaning that strings longer than 1 character were not being passed at all.

4.5 Fixed string passing bug.

4.6 Fixed bug in parameter checking for SYSTEM.NEWTAG.

4.7 * The register involved in a LIBCALL parameter was being reserved too soon, causing register allocation errors in some cases where the actual parameters were expressions involving function procedures, or long integer or real arithmetic.
* Fixing the above bug uncovered another, in which the parameter register was being freed before it was reserved. This only happened when the actual parameter was a record field referenced through a pointer, or an array element.
* It was possible to dereference a function procedure that returned a pointer type as if it were a pointer variable, with unpredictable results.
* There was no check that forward declared procedures were actually implemented. The linker would have spotted this anyway.
* The stack offsets of procedure parameters were being written to the symbol file. The $L compiler switch changed these offsets, making the symbol file invalid.

4.8 * Added the $G compiler switch to suppress the generation of data

```
        for the garbage collector.
      * Changed the $Z switch from a module switch to a global switch.
      * Removed limitation preventing A4 being used in libcall
        parameters.
      * The parameters to SYSTEM.SETCLEANUP are now a single assignable
        procedure. There is no need for a variable to hold the old
        cleanup procedure, or a return code parameter.
      * Added SYSTEM.RC to return the current return code.
      * SYSTEM.NEW now has an optional parameter for passing memory
        requirements.
      * Changed code generated for HALT.

4.9   * No longer generates multiple error reports at the same
        location.
      * Implemented foreign procedures.
      * Implemented the $A compiler switch.
      * [bug] Changed code generated for ASSERT to match HALT.

4.10  * SYSTEM.LONGWORD variables can now be assigned any value whose
        type <= 32 bits. The same for SYSTEM.WORD when the type is <=
        16 bits. Integers are sign-extended, all other values are
        zero-extended.
      * Implemented NIL checking when dereferencing pointers, calling
        procedures from variables and executing type guards with
        pointers.

4.11  * Changed the way linker symbols are generated, to prepare the
        way for allowing underscores in identifiers.
      * Changed register parameter declarations to use square brackets
        instead of braces.

4.12  * Implemented stack checking.

4.13  * Added TEXTERR command line option.
      * Changed to output binary error file by default.
      * Implemented $s compiler switch.
      * Changed error numbers.
      * Extended the range of types that can be used with bit operations
        (SYSTEM.LSH, etc.)

4.14  [bug] Fixed problems with boolean comparisons.
```

1.47 Foreign code interface

```
                OC provides a limited facility for using foreign code, that is,  ↩
                    code
generated by another translator such as a C compiler or an assembler.
At present the facility is very limited in the type of interface it
supports, but this will be improved when a more complete system is
implemented in a future release.

At present, the foreign code interface can only be used with procedures
or functions that are paramaterless, or which pass their parameters in
registers. This usually means code written directly in assembly
language.
```

```
Any foreign code procedure called by an Oberon-A program must preserve
registers A4 and A5, and return any results in register D0.

To create an interface to a foreign code procedure, the programmer must
declare a procedure heading for it. The syntax is:

$  FCodeDecl    = FCodeHeading ";"
$  FCodeHeading = PROCEDURE identdef "[" string "]" [RegParameters].

The string must contain the linker label associated with the
procedure's entry point. The RegParameters are the same as those used
in a
              LibCallDeclaration
              .

Example (see module BoopsiUtil, and Classface.asm):

  PROCEDURE CoerceMethodA * ["_a_CoerceMethodA"]
    ( cl      [8] : I.IClassPtr;
      obj    [10] : I.ObjectPtr;
      VAR msg [9] : I.Msg );
```

1.48 Register parameters

```
                LIBCALL and foreign code procedures are passed their parameters  ↩
                   in CPU
registers instead of on the stack. The formal parameter list of such
procedures must therefore be declared with a modified syntax, in which
the registers used are indicated in square brackets. The syntax is:

$ RegParameters = "(" [RegParSection {";" RegParSection}] ")"
    [":" qualident].
$ RegParSection = [VAR] ident RegSpec {"," ident RegSpec } ":"
    FormalType.
$ RegSpec = "[" integer "]" [".."]

The integer in a RegSpec must be in the range 0 .. 15 and it
represents a CPU register number. The data registers D0 .. D7 are
numbered 0 .. 7; the address registers A0 .. A7 are numbered 8 .. 15.
It is used to indicate which register the corresponding parameter is
to be passed in. The ".." symbol indicates that the parameter is to be
treated as a
              VarArg
              .
```

1.49 Case checking

```
(* $C+ enable case checking *)
(* $C- disable case checking *)
```

$C+ is the default. When TRUE, the compiler generates code that will cause a processor trap if a CASE statement without an ELSE part does not execute any of the listed cases. This is a global switch.

1.50 Saving all registers

```
(* $A- Do not save registers *)
(* $A+ Save all registers *)
```

$A- is the default. When TRUE, this switch causes the compiler to generate code to save ALL registers on entry to a procedure, and restore them before exiting. The only register not save is A7. This is a procedure switch.

1.51 Saving non-scratch registers

```
                (* $s- Save all non-scratch registers *)
(* $s+ Do not save any registers *)
```

$s- is the default. This is similar to the
$A
switch, except that the
D0, D1, A0 and A1 registers are not saved. This is a procedure switch.

1.52 Stack checking

```
(* $S+ Enable stack checking *)
(* $S- Disable stack checking *)
```

$S+ is the default. When TRUE, this switch causes the compiler to generate code to call a stack checking routine on entry to a procedure. The stack checking code will cause a processor trap if there is less than 1500 bytes of stack remaining after allocating local variables for the procedure. This allows for possible stack usage for the parameters of procedures called in the body of the procedure, and stack required by system calls. This is a global switch.