OC.doc

COLLABORATORS			
	TITLE : OC.doc		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		December 8, 2024	

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: 5.4 $
    $Author: fjc $
      $Date: 1995/07/30 18:40:04 $

Copyright © 1994-1995, 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...
~Shell~~~~~~~~~~~~~~~   ...from the Shell
~Workbench~~~~~~~~~~~   ...from the Workbench
~FPE~~~~~~~~~~~~~~~~~   ...from the FPE utility
~Preferences~~~~~~~+~   Preferences settings

~Oberon-2~~~~~~~~~~~~   The programming language Oberon-2
~Extensions~~~~~~~~~~   Language extensions supported by the compiler
~Module~SYSTEM~~~~~~~   The pseudo-module SYSTEM
~Module~Kernel~~~~~+~   The run-time system
~Compiler~control~~~~   Controlling the compiler
~Garbage~collection~~   Using the garbage collector
~Run~time~errors~~~~~   Handling run-time errors
~Error~reports~~~~~~~   Error reports from the compiler
~Basic~types~~~~~~~~~   Implementation of basic types
~Compiler~limits~~~~~   Limits built in to the compiler

~The~Author~~~~~~~~~~   Contacting the author
~Bugs~&~Suggestions~~   Reporting bugs and suggestions
~Acknowledgements~~~~   Who did what and why
```

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:

Copyright © 1993-1995, Frank Copeland

Parts of OC are based on source code developed at ETH Zuerich. Permission to use, copy, modify and distribute this software is granted by ETH (see the file ETH-Copyright.txt).

See Oberon-A.doc for its conditions of use and distribution.

1.4 System requirements

OC requires an Amiga personal computer with at least 1 MB of RAM, running AmigaOS 2.04 or greater (Kickstart 37 or greater. Depending on the module being compiled, 500K or more of free RAM must be available to run the program.

Starting with with version 5, OC will no longer run under AmigaOS 1.3. In addition, programs compiled with it will not run under AmigaOS 1.3 either (this may well change in a future release). If this is a problem for you, please contact the~author. You should have said something earlier, when you had the chance.

1.5 Running OC from the Shell

```
Format:         [NS | NEWSYMFILE] [BATCH] [SETTINGS <filename>]
                {<filename>}

                (* This is temporary and will disappear eventually *)
```

```
                [FORCE]

Template:       NS=NEWSYMFILE/S,BATCH/S,SETTINGS/K,FILES/M,
                FORCE/S

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

Path:           Oberon-A/OC
```

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

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.

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 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.

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 an error file. If there are no errors, an object file containing machine code, data and relocation information is output. If the compiler cannot find a symbol file for the module, it will create one. 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 FORCE argument is for debugging purposes and is not documented. It will be removed at some point in the future.

OC has a number of preferences settings that affect its operations. The SETTINGS argument can be used to specify the name of a preferences file that is loaded before any other arguments are processed. See Preferences~Settings. If no SETTINGS argument is specified, the default preferences file is "OC.prefs". Preferences files are searched for first in the current directory, then in "PROGDIR:" (the directory containing OC), and finally in "ENV:OC".

Preferences files can be viewed and edited with the OCPrefs tool.

The Shell stack should be set to at least 12000 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 SETTINGS=OC.prefs [].mod
```

A module can then be compiled by typing:

```
OComp <module>
```

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 SETTINGS=OCLib.prefs DEBUG Intuition.mod
OC NS OCE.mod
```

1.6 Running OC from the Workbench

See Running~OC~from~the~Shell for a general description of the compiler's operation and the effect of the various arguments.

Double-clicking the compiler's icon will run it in interactive mode. You will be repeatedly prompted for the name of a file to be compiled. The compiler will halt when <enter> is pressed in response to the prompt.

One or more source or batch files can be passed as arguments to the compiler by extended selection. While holding down the shift key, select the files to be compiled, then double-click the compiler's icon. The compiler will then process the files in the order in which they were selected.

All the compiler arguments available when running the compiler from the Shell can be specified as tooltypes in the compiler's icon. The FILES argument is an exception; all source files must be specified by extended selection. The tooltypes can be edited by clicking the icon and selecting the "Information" item from the Workbench "Icons" menu.

For switch arguments like BATCH the name of the argument is entered as a tooltype. The standard WINDOW tooltype is also understood by OC. If it is omitted a default console window is opened.

*** WARNING ***

If the console window has a close gadget, DON'T CLICK IT. Clicking the close gadget may have unexpected results, including closing the console window without halting the program. Delete the CLOSE command from the console description in the WINDOW tooltype, otherwise exercise caution.

A typical list of tooltypes might look like this:

```
WINDOW=CON:0/0/640/200/Compiling...
SETTINGS=OC.prefs
NEWSYMFILE
(BATCH)
```

Enclosing the BATCH argument in parentheses disables it without the need to delete the entire tooltype. To enable it, edit the tooltype to remove the parentheses and save the icon.

If you often use two or more different preferences files and/or argument lists, it may become tedious to constantly edit the compiler's tooltypes to change the arguments. This can be solved by using the Shell command MakeLink to create a copy of the compiler and creating a seperate icon for it. See the OC-Lib icon in the Oberon-A directory for an example.

The default stack should be set to at least 12000 bytes.

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 12000 in the stack field.

For example:

```
Command="Oberon-A:OC"
Arguments="SETTINGS=OC.prefs !F"
Console="CON:0/11/540/189/Compiling.../CLOSE/WAIT"
Stack=12000
```

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 Preferences settings for OC

Preferences settings are used to customize the operation of OC. OC loads its settings from a file, which can be specified on the command

line or in the tooltypes. The default settings file is "OC.prefs". When searching for settings files, OC looks first in the current directory, then in "PROGDIR:" (the directory containing OC), then in "ENV:OC". Settings files can be viewed and edited using the OCPrefs utility.

The settings are:

Search Paths

Directories to be searched for symbol files. These can be absolute paths, or relative paths. For example, "OLIB:" is an absolute path, and "Code" is a relative path, meaning the sub-directory "Code" in the current directory. Up to ten search paths can be specified. The current directory is searched first by default.

Output Paths

Directories in which to output symbol, object and error files. Again, these can be absolute or relative paths. The default is to output symbol and object files in the current directory, and error files in "T:".

Extensions

OC constructs file names by appending an extension to the module name. Extensions can be specified for symbol, object and error files. The defaults are ".sym", ".obj" and ".err" respectively.

Selectors

Pre-defined selectors to be used in conditional compilation commands. Selectors can be set (given a default value of TRUE) or cleared (given a default value of FALSE). The settings are strings, containing the names of zero or more selectors. Multiple names are seperated by spaces.

Setting a selector is equivalent to placing the following inline commands at the top of the source text:

```
<* NEW selector *>
<* selector+ *>
```

Clearing a selector is equivalent to placing the following inline commands at the top of the source text:

```
<* NEW selector *>
<* selector- *>
```

Buffer sizes

The number of bytes to be allocated for the code and constant buffers. The default size for both buffers is 32000 bytes, which should be more than enough for most modules.

Options, code models and pragmas

The default values to be used for compiler options, code models and

pragmas.

Miscellaneous

If the Verbose setting is TRUE, OC produces a long-winded description of the compilation, including the names of symbol files imported, the names of object and error files created, and the number of bytes of code, data and variables generated. If it is FALSE, the compiler simply outputs the name of the source text being compiled.

If the Make Icons setting is TRUE, OC creates icons for any symbol, object and error files it outputs, if they do not already have one. The default icons are expected to be in the "ENV:OC" directory, with the names "def_sym", "def_obj" and "def_err" respectively. If they cannot be found, the system default project icon is used instead.

1.9 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 STANDARD compiler option must be set to FALSE (see Compiler~Control).

~String~and~character~literals~~~~~
~Assignable~procedures~~~~~~~~~~~~~

External code interface

~System~flags~~~~~~~~~~~~~~~~~~~~~
~External~procedure~declarations~~~
~Amiga~library~functions~~~~~~~~~~~
~Register~parameters~~~~~~~~~~~~~~~
~Variable~length~parameter~lists~~~

~Examples~~~~~~~~~~~~~~~~~~~~~~~~~

1.10 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";
```

1.11 Assignable procedures

Procedures that are to be assigned to procedure variables must be marked with a "*" character, unless they are marked as exported. If the STANDARD option is ON, the compiler will report an error.

Example:

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

1.12 System Flags

System flags are used to notify the compiler that a particular declaration refers to an object that does not obey the same conventions as Oberon. They are a central feature of Oberon-A's external~code interface.

A system flag consists of an integer enclosed in square brackets. It is used to modify the meaning of certain keywords and is placed directly after them. The keywords affected are: MODULE, POINTER, PROCEDURE and RECORD.

The value of the system flag determines which language's conventions apply to the object. The following values are currently recognised:

```
  1 : Modula-2
  2 : C
```

```
  3 : BCPL
  4 : Assembly
```

One effect of a system flag is to determine if a pointer or record type is tagged or untagged. A tagged type is associated with a type descriptor which is used to implement some of Oberon's object-oriented features. An un-tagged type has no type descriptor, and cannot be used in operations that require one. These operations include type tests, type guards and declaring type-bound procedures. A pointer or record type declared with *any* system flag is considered to be untagged. The base type of a tagged pointer must also be tagged; the base type of an untagged pointer must be untagged.

A pointer declared with the BCPL system flag is treated as a longword pointer; that is, an address divided by four. The compiler will automatically perform any necessary shifts needed to convert it to and from a valid byte address. Such a pointer is compatible with the SYSTEM.BPTR type, but not with the SYSTEM.ADDRESS type. A pointer declared with any other system flag (Modula-2, C or Assembly) is compatible with SYSTEM.ADDRESS, but not with SYSTEM.BPTR.

A procedure declared with any system flag has no body, and will obey different calling conventions to a normal Oberon procedure. An Assembly procedure must have any parameters declared as register parameters.

If a system flag follows the MODULE keyword, the value of the flag becomes the default for the module. *All* POINTER, RECORD and PROCEDURE declarations are treated as if they were declared with the same system flag. This can be over-ridden by using a system flag of 0, which will force the compiler to treat the declaration as a normal Oberon declaration.

1.13 External procedure declarations

OC provides a facility for using external code, that is, code generated by another translator such as a C compiler or an assembler. This involves the use of a special syntax to declare the external procedures, in conjunction with the use of system~flags.

The syntax for declaring an external procedure is:

```
  $ ExtProcDecl    = ExtProcHeading ";"
  $ ExtProcHeading = PROCEDURE [sysflag] identdef "[" string "]"
                       [FormalParameters | RegParameters].
```

If the sysflag is omitted, it is assumed to be the same as the module's system flag. If the module heading does not contain a system flag, it is assumed to be 4, meaning that the procedure is an Assembly procedure.

The string must be the linker label associated with the external procedure.

If the procedure's system flag is 4 (Assembly), then the procedures parameters, if any, must be register~parameters. Otherwise they are

declared as normal Oberon parameters. In either case, a pointer or record parameter cannot have a tagged type.

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

The object file containing the external procedure must be declared in the module heading, using the following syntax:

```
$ ModuleHeading = MODULE [sysflag] ident
                  ["[" string {"," string} "]"]
```

The strings must be the names of object files containing external procedures declared in the body of the module.

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

```
PROCEDURE [4] CoerceMethodA * ["_a_CoerceMethodA"]
  ( cl      [8] : I.IClassPtr;
    obj    [10] : I.ObjectPtr;
    VAR msg [9] : I.Msg );
```

1.14 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 library function declarations

1.15 Amiga library function declarations

```
The declaration of an Amiga library function must provide the following
information to the compiler:

  - The name of the library base variable
  - The library vector offset of the function
  - The registers in which individual parameters are passed

The syntax of a library function declaration is:

  $ LibCallDeclaration = LibCallHeading ";"
  $ LibCallHeading = PROCEDURE identdef "[" ident "," ["-"] integer "]"
      [RegParameters]

The ident must be the name of a variable, declared at the same scope
level as the library call, whose type occupies 4 bytes. This type will
usually be a pointer type, but LONGINT and LONGWORD are also
acceptable. It is the programmer's responsibility to ensure that the
variable is correctly initialised with the address of the library's
base structure.

The integer must be the function's library vector offset.

See Register~Parameters.
```

1.16 Register parameters

```
Amiga library functions and some external 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.
```

1.17 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
```

an Amiga~library~function or external~code~procedure, in a manner similar to C vararg parameters.

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.

1.18 Examples of declaring and using LIBCALLs

Amiga library function example:

```
PROCEDURE OpenLibrary* [base,-552]
  ( libName [9] : ARRAY OF CHAR;
    version [0] : Exec.ULONG )
  : Exec.APTR;
```

This defines the Amiga Exec library function OpenLibrary. It indicates that it is bound to the 'base' variable. 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 a call of OpenLibrary () might look like this:

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

VarArgs example:

```
...

PROCEDURE OpenWindowTagsA* [base,-606]
  ( newWindow [8]   : NewWindowPtr;
    tagList   [9].. : U.Tag )
  : WindowPtr;
...

VAR w : I.WindowPtr;

BEGIN
  ...

  w := I.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 ...
```

1.19 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
```

1.20 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 STANDARD compiler option must be set to OFF 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 anonymous pointer types are exported: PTR, ADDRESS and BPTR. Any Oberon pointer may be assigned to a variable or parameter of type PTR. Any C or Modula-2 pointer may be assigned to a variable or parameter of type ADDRESS. In addition, an ADDRESS value may be assigned to any variable or parameter of a C or Modula-2 pointer type. Any BCPL pointer 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 TYPETAG type is used to hold a type tag, which is a pointer to a type descriptor. The only operations allowed are comparisons with and assignments of other TYPETAG values and NIL.

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.

1.21 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.

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 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).

1.22 Memory access

The ADR procedure is used to find the run-time address of any variable or string constant. The result has a type of ADDRESS.

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.

1.23 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.

1.24 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.

1.25 Miscellaneous

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.

1.26 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	ADDRESS	address of variable v, or string constant v
AND(x, y)	x, y: basic type	larger type	bitwise AND
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
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
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
DISPOSE(v)	any pointer type	free memory allocated to v
GET(a, v)	a: LONGINT v: any basic type	v := Mem [a]
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
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.

1.27 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.

1.28 MODULE Kernel

Module~Kernel has a special status in the Oberon-A system. It is the run-time system, and is linked into every Oberon-A program even if it is not explicitly imported. It is closely related to module SYSTEM, and contains the code to implement the procedures in SYSTEM that are too large to be coded inline.

Certain assumptions about module Kernel are hard-coded into the compiler, and if you wish to make any modifications to it you must proceed with extreme care. See the comments in Kernel.mod for further information.

Module Kernel provides a number of useful services that can be directly accessed by a program. These are grouped under the following headings:

1.29 MODULE Kernel: Memory management

```
PROCEDURE New
  ( VAR v: SYSTEM.PTR; type: SYSTEM.TYPETAG );
```

Allocates a record variable using a type tag obtained from the SYSTEM.TAG procedure. Given the declarations:

```
  TYPE T0 = RECORD ... END;
       T0Ptr = POINTER TO T0;
```

```
    VAR  v0 : T0Ptr;

  'Kernel.New (v0, SYSTEM.TAG(T0))' is equivalent to 'NEW (v0)'.

PROCEDURE Allocate
  ( VAR v: SYSTEM.ADDRESS; size: LONGINT; reqs: SET );

  Allocates a block of memory with a particular set of memory
  requirements. 'Kernel.Allocate (v, size, req)' is equivalent to
  'v := Exec.AllocMem (size, req)', except that the memory allocated is
  tracked by the Oberon-A runtime system.

PROCEDURE Dispose
  ( VAR adr: SYSTEM.ADDRESS );

  Frees a block of memory obtained by Allocate(). Directly equivalent
  to SYSTEM.DISPOSE (in fact, it *is* SYSTEM.DISPOSE).

PROCEDURE GC;

  Activates the garbage collector. The current implementation does not
  mark local procedure variables, and must be used with great care. See
  Garbage~Collection.
```

1.30 MODULE Kernel: Run-time error handling

```
Error location:

  VAR

    errCol - : INTEGER;
    errLine - : INTEGER;
    errModule - : ARRAY 32 OF CHAR;

If a run-time error occurs, these variables will contain the exact
location of the error. This information is used by module Errors to
generate a meaningful error report.


Trap handling:

  PROCEDURE InstallTrapHandler;
  PROCEDURE RemoveTrapHandler;

These procedures install and remove a trap handler that will
intercept any processor traps and branch to the finalization and
cleanup code. This ensures that the program exits gracefully instead
of causing a Guru.

If InstallTrapHandler() is ever called, RemoveTrapHandler() *must* be
called by the program before it exits. Failure to do so will result in
undesirable system behaviour ;-). The safest method is to make the call
to RemoveTrapHandler() in a cleanup procedure installed with
Kernel.SetCleanup().
```

1.31 MODULE Kernel: Finalization and cleanup

```
Program cleanup:

  TYPE

    CleanupProc * = PROCEDURE ( VAR rc : LONGINT );

  PROCEDURE SetCleanup
    ( proc : CleanupProc );

    Installs a procedure in a list of procedures that are to be
    executed when the program exits for any reason. A cleanup procedure
    will typically return one or more previously allocated resources.
    The rc parameter will contain the return code set by a HALT or
    ASSERT procedure, or corresponding to a particular run-time~error.


Finalizing objects:

  TYPE

    Finalizer * = PROCEDURE ( obj : SYSTEM.PTR );
    StructFinalizer * = PROCEDURE ( str : SYSTEM.ADDRESS );

  PROCEDURE RegisterObject
    ( obj: SYSTEM.PTR; fin: Finalizer );
  PROCEDURE RegisterStruct
    ( str: SYSTEM.ADDRESS; fin: StructFinalizer );

Not implemented yet. In a future release, these will be used to
implement a finalization system for individual objects, integrated
with the garbage collector.
```

1.32 MODULE Kernel: Registration of modules, types and command

```
Registration data structures:

  TYPE

    RegNode * = POINTER [1] TO RegisterDesc;
    RegisterDesc = RECORD [1]
      next - : RegNode;
      name - : ARRAY 32 OF CHAR;
    END;

    Command * = POINTER [1] TO CommandDesc;
    CommandDesc * = RECORD [1] (RegisterDesc)
      proc - : CommandProc;
    END;
    CommandProc * = PROCEDURE;

    Module * = POINTER [1] TO ModuleDesc;
    ModuleDesc * = RECORD [1] (RegisterDesc)
```

```
      types - : RegNode;
      commands - : RegNode;
    END;

    Type * = POINTER [1] TO TypeDesc;
    TypeDesc * = RECORD [1] (RegisterDesc)
      tag - : SYSTEM.TYPETAG;
    END;

  VAR

    modules - : RegNode;
```

These declarations are exported in their current form for debugging purposes only. Do not rely on the declarations above, but instead assume the following declarations:

```
  TYPE

    Command * = POINTER [1] TO CommandDesc;
    CommandDesc * = RECORD [1] (RegisterDesc)
      proc - : CommandProc;
    END;
    CommandProc * = PROCEDURE;

    Module * = POINTER [1] TO ModuleDesc;
    ModuleDesc * = RECORD [1] (RegisterDesc)
    END;

    Type * = POINTER [1] TO TypeDesc;
    TypeDesc * = RECORD [1] (RegisterDesc)
      tag - : SYSTEM.TYPETAG;
    END;
```

Registration procedures:

```
  PROCEDURE RegisterModule
    ( name: ARRAY OF CHAR ): Module;
  PROCEDURE RegisterType
    ( module: Module; tag: SYSTEM.TYPETAG ): Type;
  PROCEDURE RegisterCommand
    ( module: Module; name: ARRAY OF CHAR; proc: CommandProc ): Command;
```

If the REGISTER compiler option is on, the compiler automatically generates calls to these procedures to register the module and any eligible types and commands. If you decide to call these procedures directly, wrap the calls in a conditional compilation block:

```
  <* IF ~REGISTER THEN *>
    mod := Kernel.RegisterModule (...);
    typ := Kernel.RegisterType (mod, ...);
    cmd := Kernel.RegisterCommand (mod, ...);
  <* END *>
```

Search procedures:

```
  PROCEDURE FindModule
```

```
    ( name: ARRAY OF CHAR ) : Module;
  PROCEDURE FindType
    ( module: Module; name: ARRAY OF CHAR ) : Type;
  PROCEDURE FindCommand
    ( module: Module; name: ARRAY OF CHAR ) : Command;
```

1.33 MODULE Kernel: Type descriptor handling

```
The extension level of a type is the number of base types it is
extended from. A type that has no base type has a level of 0, a type
extended from it has a level of 1, and so on. The maximum extension
level is currently 15.

PROCEDURE BaseOf
  ( type: SYSTEM.TYPETAG; level: INTEGER ) : SYSTEM.TYPETAG;

  Returns the base type of an extended type with a given extension
  level.

PROCEDURE LevelOf
  ( type: SYSTEM.TYPETAG ) : INTEGER;

  Returns the extension level of a type.

PROCEDURE Name
  ( type: SYSTEM.TYPETAG; VAR buf: ARRAY OF CHAR );

  Returns the name of a type, in the form "module.type".

PROCEDURE Size
  ( type: SYSTEM.TYPETAG ) : LONGINT;

  Returns the size in bytes of a type. Equivalent to SIZE (type).
```

1.34 MODULE Kernel: Command-line handling

```
VAR

  fromWorkbench - : BOOLEAN;

    TRUE if the program was started from the Workbench, FALSE if
    started from a Shell.

  dosCmdBuf - : SYSTEM.ADDRESS;

    If started from a Shell, contains a pointer to the actual command
    line used to run the program.

  dosCmdLen - : LONGINT;

    If started from a Shell, the number of characters pointed to by
    dosCmdBuf.
```

```
WBenchMsg - : SYSTEM.ADDRESS;

  If started from the Workbench, contains a pointer to the startup
  message sent by Workbench. DO NOT ATTEMPT TO MODIFY OR RETURN THIS
  MESSAGE. This is handled automatically by the runtime system.
```

1.35 MODULE Kernel: Miscellaneous

```
Task.userData:

  TYPE

    UserData * = RECORD [1]
      userData : SYSTEM.ADDRESS;
      dataSegment : SYSTEM.ADDRESS;
    END;
    UserDataPtr * = POINTER [1] TO UserData;

A pointer to this data structure is placed in the tasks userData field
by the startup code. If the SMALLCODE or RESIDENT options are being
used, UserData.dataSegment contains the contents of the A4 register
used to access global variables and constants. UserData.userData can be
used by the program.

PROCEDURE GetDataSegment;

  Loads the contents of UserData.dataSegment into the A4 register if
  the SMALLCODE or RESIDENT options are being used. This is necessary
  to access constants and global variables from inside call-back
  procedures. This must only be called for procedures that are
  executing in the program's own context. Do NOT use it in hook
  procedures that are being executed in another task's context.
```

1.36 Controlling the compiler

```
The behaviour of the compiler is to some extent under programmer
control. This control is exercised through compiler options and
pragmas. Options are used to affect the compilation of an entire
module. Pragmas are used to affect the compilation of specific blocks
of code.

Oberon-A also allows for the conditional compilation of blocks of
source code through the use of programmer-defined selectors.

Options, pragmas and source control commands are embedded in ISO-style
pseudo comments. These are similar to Oberon comments, but use the
tokens "<*" and "*>". They can be embedded inside Oberon comments, but
not vice versa.
```

```
~Compiler~options~~~~~~~~
~Code~models~~~~~~~~~~~~~
~Pragmas~~~~~~~~~~~~~~~~~
~Conditional~compilation~
```

1.37 Compiler options

Compiler options affect the translation of an entire module. The default values are determined by the preferences settings in use, and can be over-ridden by inline commands in the source text.

The currently supported options are:

Option	Meaning
STANDARD	The module follows the Oberon-2 Report standard exactly, with no language extensions allowed.
INITIALISE	All variables are initialised to zero.
MAIN	A program entry point is generated. The module may be used as the main module of a stand-alone program.
WARNINGS	Questionable usage generates warnings.
REGISTER	Code is automatically generated to register the module and any eligible types and commands (parameterless exported procedures).

When placed in the source text, an option takes the following form:

```
"<*" option "+"|"-" "*>"
```

Example:

```
<* STANDARD- *> <* INITIALISE- *> <* MAIN- *>
```

1.38 Code models

Oberon-A supports two code and three data models, which cause the compiler to generate different code for calling procedures and accessing constants and global variables. The actual models used are determined by the current preferences settings.

The code models supported are: large and small. The data models supported are: large, small and resident.

When using the large code model, the compiler generates BSR instructions when calling procedures in the same module, and JSR instructions for all other procedure calls. In the small code model, the compiler generates BSR instructions for all procedure calls. The

small code model relies on the linker having the ability to detect branches longer than 32K and automatically insert skip lists into the final executable to convert any illegal BSRs into JSRs. If the linker being used does not have this ability, the small code model is restricted to programs with 32K of code or less.

When using the large data model, each module has seperate hunks for constants and variables. Constants are accessed through absolute 32 bit addressing. Variables are accessed through the A4 register, which is initialised to point to the module's variables at the start of every exported and assignable procedure.

In the small data model the constants and variables for all modules are merged into a single hunk and accessed through the A4 register. The A4 register is initialised to point to this hunk in special startup code in the main program module. The maximum size of the combined constants and variables is 32K.

In the resident data model constants are merged with the module's code and accessed through absolute 32 bit addressing. Space for variables is dynamically allocated by special startup code in the main program module and accessed through the A4 register. The maximum size for global variables is 32K.

Code generated using the large and small data models is not pure, but such programs are 're-executable' and can still be made resident. However, only one process can execute the code at any one time. If more than one process tries to execute the same code, the second and subsequent processes will immediately exit with a return code of 25.

Code generated using the resident data model is pure and can executed by multiple processes simultaneously.

There are three pre-defined selectors that can be used in conditional compilation expressions, which have their values determined by the code and data models in use. The selectors are SMALLCODE, SMALLDATA and RESIDENT, and they take on the following values:

	SMALLCODE	SMALLDATA	RESIDENT
Large code model	FALSE	--	--
Small code model	TRUE	--	--
Large data model	--	FALSE	FALSE
Small data model	--	TRUE	FALSE
Resident data model	--	FALSE	TRUE

1.39 Run-time checks and pragmas

Pragmas are used to enable and disable runtime checks and control other aspects of the code being generated. The default values of some pragmas are determined by the preferences settings in use. The current values can be modified by inline commands placed in the source text.

The following pragmas have their default values determined by the

current preferences settings:

Pragma	Meaning
TypeChk	Controls the generation of code for type checks.
OvflChk	Controls the generation of code for detecting overflows in arithmetic expressions.
IndexChk	Controls the generation of code to check array indexes against array bounds.
RangeChk	Controls the generation of code to ensure that values assigned to variables are within the legal range.
CaseChk	Controls the generation of code for checking the arguments to case statements.
NilChk	Controls the generation of code for checking that de-referenced pointers are valid.
ReturnChk	Controls the generation of code to check that function procedures exit through a valid RETURN statement.
StackChk	Controls the generation of code to check the amount of stack remaining on procedure entry.
AssertChk	Controls the generation of code to perform ASSERT statements.
LongVars	When On, global variables are accessed with 32-bit absolute addressing instead of with register indirect addressing through A4. This saves space when a procedure does not access any global variables. If a module has no global variables at all, this pragma has a default value of On. Only applies when *both* the SMALLDATA and RESIDENT options are off.
ClearVars	Controls the generation of code to zero all variables. This has the same effect as the INITIALISE compiler option, but only affects a block instead of the entire module. It is ingored if the INITIALISE option is on.

The following pragmas are not affected by the preferences settings. They can only be changed by an inline command, which only affects the procedure or module body immediately following the inline command:

Pragma	Default	Meaning
CopyArrays	On	Controls the generation of code to copy the contents of value open array parameters. Use <*CopyArrays-*> to suppress the copying of parameters when you are certain that the parameter will not be written to.
SaveRegs	Off	Controls the generation of code to save and restore all registers (except the scratch

		registers D0,D1,A0 & A1) on procedure entry and exit.
SaveAllRegs	Off	Controls the generation of code to save and restore *all* registers (including the scratch registers) on procedure entry and exit.
DeallocPars	On	Controls the generation of code to deallocate the procedure's parameters when it exits. Use <*DeallocPars-*> *only* when the procedure is to be used as a call-back by system software that assumes it follows C procedure call conventions.
EntryExitCode	On	Controls the generation of code on entry to and exit from a procedure. It overrides the effects of the StackChk, ClearVars, CopyArrays, SaveRegs, SaveAllRegs and DeallocPars pragmas, and the INITIALISE compiler option. A procedure affected by this pragma must *not* contain formal parameters or local variables, but may have a return type. Global variables may only be accessed if the LongVars pragma is on. This pragma also suppresses the generation of the RTS instruction normally used to exit from the procedure.

At the end of every procedure body, all these pragmas are reset to their default values.

When placed in the source text, a pragma takes the following form:

```
"<*$" {modifier} "*>"
```

where modifier is:

pragma+	set pragma ON, enable.
pragma-	set pragma OFF, disable.
<	push the current pragma state onto a stack.
>	pop a pragma state of the stack and make it the current state.
!	revert to the pragma state defined by the original preferences settings.

Example:

```
PROCEDURE Copy (from : ARRAY OF CHAR; VAR to : ARRAY OF CHAR);

<*$CopyArrays-*>
BEGIN
...
END Copy.
```

1.40 Compiler source control

The compiler can selectively compile blocks of source text based on the value of compiler options and programmer-defined selectors. The syntax for selecting the source text to be compiled is:

```
<* IF condition THEN *>
<* ELSIF condition THEN *>
<* ELSE *>
<* END *>
```

The conditional expression consists of programmer defined selectors which can be combined as an Oberon-like boolean expression which can contain the operators ~, & and OR. Compiler options are in effect predefined selectors and can be used with the condition part. The following options may be used:

```
STANDARD  MAIN      INITIALISE WARNINGS
SMALLCODE SMALLDATA RESIDENT   REGISTER
```

To define a new selector, which has the default value of FALSE:

```
<* NEW SelectorName *>
```

To give a selector a value:

```
<* SelectorName+ *>   to set it to TRUE
<* SelectorName- *>   to set it to FALSE
```

There is one additional pre-defined selector named 'OberonA', which has a default value of TRUE.

Examples:

```
<* IF ~MAIN THEN *> ...

<* IF M68000 & WARNINGS THEN *>
IMPORT CG68000;
<* ELSE *>
IMPORT CG80x86;
<* END *>
```

1.41 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 module Kernel. 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 pointer 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. C, Modula-2 and BCPL pointer 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 non-Oberon pointer 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. If you want such memory to be tracked, allocate it

with NEW, SYSTEM.NEW, or Kernel.Allocate.

1.42 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 (module Kernel) contains a trap handler which can intercept all compiler-generated traps and several others such as divide-by-zero. This trap handler must be explicitly installed using the procedure Kernel.InstallTrapHandler(). It can be removed if necessary by calling Kernel.RemoveTrapHandler(). The trap handler has the same effect as a HALT statement, causing the program to terminate. Any cleanup procedures installed with Kernel.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. The name of the module in which the error occurred is placed in the variable Kernel.errModule, and the position in the module's source text is placed in the variables Kernel.errLine and Kernel.errCol.

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~

1.43 Currently defined return codes and processor traps

The following error codes are suggested as conventions. All the library modules make use of them. They are declared as constants in module Errors, which will produce an appropriate error message if it detects them.

Err #95 : Return code = 95 (Errors.outOfMemory)

If an attempt to allocate memory fails and is detected by the program, it should be halted with either a HALT (95) or an ASSERT (mumble, 95) statement.

Err #96 : Return code = 96 (Errors.invariant)

An invariant is a condition that must always be true, typically a variable that must contain a certain value or range of values. If a program detects an invariant violation, it should be halted with either a HALT (96) or ASSERT (mumble , 96).

Err #97 : Return code = 97 (Errors.preCondition)

If the parameters passed to a procedure do not match some formally defined pre-condition(s), the program should be halted with either a HALT (97) or an ASSERT (mumble, 97) statement.

Err #98 : Return code = 98 (Errors.postCondition)

If the result produced by a procedure does not match some formally defined post-condition, the program should be halted with either a HALT (98) or an ASSERT (mumble, 98) statement.

Err #99 : Return code = 99 (Errors.notImplemented)

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 (Errors.noLibrary)

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

The following error codes are produced as the result of run-time errors.

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 index checking is on, this trap will 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)]".

```
    If compiler range checking is on, this trap will also 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.

  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

    An attempt has been made to de-reference 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.

  Trap #38 : Return code = 138

    If range checking is on, this trap will occur if the parameter of a
    SHORT() statement is too large for the result type.

    If overflow checking is on, this trap will also occur if the result
    of an integer multiplication is too large for the result type.
```

1.44 Error reports produced by the compiler

```
By default, any errors detected by the compiler are listed in the file
"<module>.err" in the current directory.
```

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
```

Lines and columns are numbered starting at 1. The meaning of each error number is listed in the file ErrorCodes.doc. The error messages are also available in a catalog file named ErrorMessages.catalog (this is currently available only in English and Italian). The catalog description file is ErrorMessages.cd.

1.45 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	-3.4E+38	3.4E+38
LONGREAL	32 bits/4 bytes	-3.4E+38	3.4E+38
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 IEEE Single Precision Floating Point standard. In a future version, LONGREAL will be re-implemented as an IEEE double precision real.

1.46 Limits built in to the compiler

* The buffers for holding code and constants can now be any size, and are controlled by preferences settings. However, the maximum size for any branch is still 32K, which may place a practical limit on the size of a module.

* 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. In the large data model, variables more than 32K from the module's variable base will be less efficient to access. In the small data and resident models, the *total* size of global variables in *all* modules cannot exceed 32K. The linker will report an error if this occurs.

* Identifiers and string literals cannot be more than 255 characters long. This is primarily a limit imposed by module OCS. 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 12000 bytes of stack and 500K or more of free RAM to run.

1.47 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.

1.48 Reporting bugs and suggestions

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. OC has now been shown to operate successfully on a wide range of machines and configurations.

* 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.

1.49 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

1.50 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 re|urn 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 fkreign 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.

4.15 Checks for the existence of SYM and DST directories.

4.16 [bug] Using the same name twice in a formal parameter list caused an endless loop.

4.17 [bug] Calling type-bound procedures from arrays of objects caused register allocation errors.

5.1 Replaced the old compiler switches with Oakwood-style pragmas and options.

5.2 Updated the source code to use the new pragmas and options.

5.3 * [bug] Dereferencing a pointer in an array caused register allocation problems when NIL checking was enabled.

5.4 * Implemented source code control (conditional compilation).
* Added SET and CLEAR arguments.
* Now uses Kickstart 2.04+ ReadArgs() for argument parsing.

5.5 Re-implemented Amiga library calls as normal procedures instead of type-bound procedures.

5.6 * Removed CPOINTER, BPOINTER and LIBCALL keywords.
* Implemented system flags for MODULE, POINTER, RECORD and PROCEDURE declarations.
* Implemented calling conventions for Modula-2 and C procedures.
* Rationalised compatibility rules for pointers.
* Changed symbol file format to reflect new object modes and system flags. Names of external modules are now exported for the benefit of OL.

5.7 * Further work on pointer assignments.
* The INITIALISE option now works for pointers in arrays and record variables.

5.8 Minor modifications.

5.9 Minor modifications (OK, I wrote this long after the event and I forget).

5.10 Simple re-link to use bug-fixed garbage collector.

5.11 Modified to use new interface to module Strings.

5.12 Modified to use modules In and Out for console IO.

5.13 * [bug] Implemented ABS for reals.
* Added SYSTEM.CC.
* SYSTEM.PTR is no longer compatible with POINTER TO ARRAY OF ...
* Braces can now be used instead of square brackets in most of the external code interface.

5.14 Minor modifications.

5.15 [bug] If type-bound procedures were not declared in a particular order, they could be allocted to the wrong slots in the type descriptor.

5.16 The code generated for run-time checks now includes a pointer to the module's name and the current position in the source text.

5.17 * [bug] The potential existed for the use of the wrong addressing mode when accessing array elements.
* Added support for preferences settings and preferences files.
* Added Workbench interface.

5.18 Version included in Release 1.5.

5.19 Uses OberonClock instead of Oberon.

5.20 [bug] Didn't properly handle code buffer overflows.

5.21 [bug] Comparing a value ARRAY OF CHAR procedure parameter with an empty string generated invalid code.

5.22 * [bug] Multiplication of SHORTINTs was completely broken.
* Implemented overflow checking for multiplication of integers.
* Implemented range checking when calling the standard procedure SHORT.

5.23 [bug] OCStrings.OpenCatalog() used the wrong tag (u.skip instead of u.ignore).

5.24 * Implemented SMALLCODE option.
* Improved the code generated by remembering the contents of address registers in some circumstances.

5.25 * Implemented SMALLDATA option. It didn't actually work, but that's life I suppose :-).
* Added CODESIZE and CONSTSIZE arguments to set the size of the code and constant buffers.

5.26 Further work on SMALLDATA option.

5.27 Minor adjustments.

5.28 * Implemented a GUI and preferences dialog using EAGUI.
* Restructured code generation to allow code to exceed 32K.

5.29 * Deleted the GUI and transferred the preferences dialog to OCPrefs.
* Now allows floating point constant expressions.

5.30 Minor adjustments.

5.31 * Greatly simplified command line arguments by removing arguments that over-rode preferences settings.
* Properly implemented SMALLDATA option and added RESIDENT option.
* Increased maximum depth of type extensions from 7 to 15.
* Implemented AssertChk pragma.
* Implemented REGISTER option.

5.32 * [bug] An invalid value was loaded into A6 for library calls

where the A4 register was used for a parameter. Only applied to the SMALLDATA and RESIDENT models.

5.33 * Makes better use of the A6 register.
* [bug] Wasn't freeing registers properly when processing type tests.
* [bug] The RESIDENT keyword wasn't in the table of pre-defined selectors.
* [bug] YARAB (Yet Another Register Allocation Bug) in the handling of type-bound procedure calls.

5.34 * Removed code that was trying to second-guess the garbage collector.
* [bug] Remembering pointers (introduced in 5.24) caused a bug in the handling of pointers to open arrays.

5.35 * Code generated for SYSTEM.ADR ("string literal") under the small data model changed to avoid running out of registers in VarArg parameter lists.
* AssertChk pragma temporarily disabled while I work out why it doesn't work.

5.36 * [bug] Modules compiled with the small data model were *not* re-executable. This was only a problem if you attempted to make small data model programs resident.

5.37 * [bug] Catalogs were not being opened under AmigaOS 2.1. Changed the catalog version number to 0.