XDS Family of Products

Native XDS-x86

for Linux Operating System

Version 2.51

User's Guide

http://www.excelsior-usa.com

Contents

Chapter 1

About XDS

1.1 Welcome to XDS

XDS™ is a family name for professional Modula-2/Oberon-2 programming systems for Intel x86-based PCs (Windows and Linux editions are available). XDS provides an uniform programming environment for the mentioned platforms and allows design and implementation of portable software.

The system contains both Modula-2 and Oberon-2 compilers. These languages are often called **"safe"** and **"modular"**. The principle innovation of the language Modula-2 was the module concept, information hiding and separate compilation.

Oberon-2 is an object-oriented programming (OOP) language based on Modula-2. With the introduction of object-oriented facilities, extensible project design became much easier. At the same time, Oberon-2 is quite simple and easy to learn and use, unlike other OOP languages, such as C++ or Smalltalk.

The XDS Modula-2 compiler implements ISO 10514 standard of Modula-2. The ISO standard library set is accessible from both Modula-2 and Oberon-2.

XDS is based on a platform-independent front-end for both source languages which performs all syntactic and semantic checks on the source program. The compiler builds an internal representation of the compilation unit in memory and performs platform-independent analysis and optimizations. After that the compiler emits output code. It can be either native code for the target platform or text in the ANSI C language. ANSI C code generation allows you to cross compile Modula-2/Oberon-2 for almost any platform.

Moving to a new language usually means throwing away or rewriting your existing library set which could have been the work of many years. XDS allows the

programmer to mix Modula-2, Oberon-2, C and Assembler modules and libraries in a single project.

XDS includes standard ISO and PIM libraries along with a set of utility libraries and an interface to the ANSI C library set.

XDS compilers produce highly optimized 32-bit code and debug information in the **stabs** format. Definition modules for the POSIX API and the entire X Window/Motif API are included in the XDS distribution package.

1.2 Conventions used in this manual

1.2.1 Language descriptions

Where formal descriptions for language syntax constructions appear, an extended Backus-Naur Formalism (EBNF) is used.

These descriptions are set in the `Courier` font.

```
Text = Text [ { Text } ] | Text.
```

In EBNF, brackets "`[`" and "`]`" denote optionality of the enclosed expression, braces "`{`" and "`}`" denote repetition (possibly 0 times), and the vertival line "`|`" separates mutually exclusive variants.

Non-terminal symbols start with an upper case letter (`Statement`). Terminal symbols either start with a lower case letter (`ident`), or are written in all upper case letters (`BEGIN`), or are enclosed within quotation marks (e.g. `":="`).

1.2.2 Source code fragments

When fragments of a source code are used for examples or appear within a text they are set in the `Courier` font.

```
MODULE Example;

IMPORT InOut;

BEGIN
  InOut.WriteString("This is an example");
  InOut.WriteLn;
END Example.
```

Chapter 2

Getting started

In this and following chapters we assume that XDS is properly installed and configured (See Chapter 3); the default file extensions are used.

Your XDS package contains a script file, xc`work`, which may be used to prepare a working directory. For more information, consult your `readme.1st` file from the XDS on-line documentation.

2.1 Using the Modula-2 compiler

In the working directory, use a text editor to create a file called **hello.mod**, containing the following text:

```
MODULE hello;

IMPORT InOut;

BEGIN
 InOut.WriteString("Hello World");
 InOut.WriteLn;
END hello.
```

Type

```
xc hello.mod
```

at the command prompt. `xc` will know that the Modula-2 compiler should be invoked for the source file with the extension **.mod**. The compiler heading line

will appear:

```
XDS Modula-2 version [code generator] "hello.mod"
```

showing which compiler has been invoked (including its version number), which code generator is being used (in square brackets) and what is its version, and finally the name of the source file `xc` has been asked to compile.

Assuming that you have correctly typed the source file, the compiler will then display something like

```
no errors, no warnings, lines   15, time  1.09
```

showing the number of errors, the number of source lines and the compilation time.

Note: The XDS compiler reports are user configurable. If the lines similar to the above do not appear, check that the **DECOR** equation value contains letters 'C' (compiler heading) and 'R' (report).

2.2 Using the Oberon-2 compiler

In our bilingual system the Modula-2 source code just shown is also perfectly valid as the Oberon-2 code. XDS allows you to use Modula-2 libraries when programming in Oberon-2 (in our case the `InOut` module).

As in Modula-2, this source code in Oberon-2 constitutes a *top-level module* or *program module*, but in Oberon-2, there is no syntactic distinction between a top-level module and any other module. The Oberon-2 compiler must be specifically told that this is a top-level module by using the option **MAIN**.

Copy the source file to the file **hello.ob2** and type:

```
xc hello.ob2 +MAIN
```

The same sequence of reports will occur as that of the Modula-2 compiler, but the Oberon-2 compiler will also report whether a new symbol file was generated or not. It is also possible to override the default source file extension using **M2** and **O2** options:

```
xc hello.mod +O2 +MAIN
```

In this case, the Oberon-2 compiler will be invoked regardless of the file extension.

2.3 Error reporting

If either compiler detects an error in your code, an error description will be displayed. In most cases a copy of the source line will also be shown with a dollar sign "$" placed directly before the point at which the error occurred. The format in which XDS reports errors is user configurable (See 5.5), by default it includes a file name, a position (line and column numbers) at which the error occurred, an error type indicator, which can be [E]rror, [W]arning or [F]ault, an error number, and an error message.

Example

```
* [bf.mod 26.03 W310]
* infinite loop
  $LOOP
```

2.4 Building a program

To have you program automatically linked, invoke the compiler in the MAKE mode (see 4.2.2):

```
xc =m hello.mod
```

In this mode, the compiler processes all modules which are imported by the module specified on the command line, compiling them if necessary. Then, if the specified module was a program module, the linker is invoked.

However, if your program consists of several modules, we recommend to write a project file (see 4.5). In the simplest case, it consists of a single line specifying a name of a main module:

```
!module hello.mod
```

but it may also contain various option settings (see 5). The following invocation

```
xc =p hello.prj
```

will compile modules constituting the project (if required) and then execute the linker.

Here is a more complex project file:

```
% debug ON
```

```
-gendebug+
-genhistory+
-lineno+
% optimize for Pentium
-cpu = pentium
% response file name
-mkfname = wlink
-mkfext  = lnk
% specify an alternate template file
-template = wlink.tem
% linker command line
-link = "wlink @%s",mkfname#mkfext;
% main module of the program
!module main.mod
% additional library
!module clib3s.lib
```

After successful compilation of the whole project the compiler creates a linker response file using the specified template file (see 4.8) and then executes a command line specified by the **LINK** equation.

2.5 Debugging a program

XDS compilers generate debug information in the stabs format and allow you to use any debugger compatible with that format (for example, GDB). However, the *postmorten history* feature of XDS run-time support may be used in many cases instead of debugger. To enable this feature, the option **LINENO** should be set on for all modules in the program and the option **GENHISTORY** for the main module of the program; the program also has to be linked with debug info included. If your program was built with the above settings, the run-time system dumps the stack of procedure calls on an abnormal termination into a file called `errinfo.$$$`. The HIS utility reads that file and outputs the call stack in terms of procedure names and line numbers using the debug info from your program's executable file.

Example

```
MODULE test;
```

```
PROCEDURE Div(a,b: INTEGER): INTEGER;
BEGIN
  RETURN a DIV b
END Div;

PROCEDURE Try;
  VAR res: INTEGER;
BEGIN
  res:=Div(1,0);
END Try;

BEGIN
  Try;
END test.
```

When this program is running, an exception is raised and the run-time system stores the exception location and a stack of procedure calls in a file `errinfo.$$$` and displays the following message:

```
#RTS: unhandled exception #6: zero or negative divisor

File errinfo.$$$ created.
```

The `errinfo.$$$` is not human readable. The HIS utility, once invoked, reads it along with the debug information from your program executable and outputs the call stack in a more usable form:

```
#RTS: unhandled exception #6: zero or negative divisor
------------------------------------------------------------
Source file                      LINE  OFFSET  PROCEDURE
------------------------------------------------------------
test.mod                            5  00000F  Div
test.mod                           11  000037  Try
test.mod                           15  000061  main
```

The exception was raised in line 5 of `test.mod`, the `Div` procedure was called from line 11, while the `Try` procedure was called from line 15 (module body).

Note: In some cases, the history may be inaccurate. See 9.2 for further details.

Chapter 3

Configuring the compiler

3.1 System search paths

In order for your operating system to know where to find the executable binary files which constitute the XDS package, you must set your operating system search paths appropriately. See the Read Me First file from your on-line documentation.

3.2 Working configuration

The core part of XDS is the `xc` utility, which combines the project subsystem with Modula-2 and Oberon-2 compilers, accompanied with a set of system files[1]:

xc.red Search path redirection file (see 3.5.1)
xc.cfg Configuration file (see 3.7)
xc.msg Texts of error messages (see 3.9)

Being invoked, `xc` tries to locate the `xc.red` file, first in the current directory and then in the directory where `xc` is placed (so called *master redirection file*).

Other system files are sought by paths defined in `xc.red`. If `xc.red` is not found, or it does not contain paths for a particular system file, that file is sought in the current directory and then in the directory where the `xc` utility resides.

A configuration file contains settings that are relevant for all projects. Project

[1] A name of a system file is constructed from the name of the compiler utility and the correspondent filename extension. If you rename the `xc` utility, you should also rename all system files.

specific settings are defined in project files (See 4.5). A so-called template file is used to automate the program build process (See 4.8).

A redirection file, a configuration file, and, optionally, a project file and a template file constitute a working environment for a single execution of the `xc` utility. The compiler preprocesses files of all these types as described in 4.4.

Portable software development is one of the main goals of XDS. To achieve that goal, not only the source texts should be portable between various platforms, but the environment also. XDS introduces a portable notation for file names that may be used in all system files and on the command line. The portable notation combines DOS-like and Unix-like notations (file names are case sensitive):

```
[ drive_letter ":" ] unix_file_name
```

Examples

```
c:/xds/bin
/mnt/users/alex/cur_pro
cur_pro/sources
```

Along with the *base directory* macro (See 4.4) this portable notation allows to write all environment files in a platform independent and location independent manner.

3.3 XDS memory usage

XDS compilers are written in Oberon-2[2]. As any other Oberon-2 program, a compiler uses garbage collector to deallocate memory. These days, most operating systems, including Windows and Linux, provide virtual memory. If an Oberon-2 program exceeds the amount of avaiable physical memory, the garbage collector becomes inefficient. Thus, it is important to restrict the amount of memory that can be used by an Oberon-2 program. As a rule, such restriction is set in the configuration or project file (See the **HEAPLIMIT** equation). You may also let the run-time system determine the proper heap size at run time by setting **HEAPLIMIT** to zero.

Similarly, the equation **COMPILERHEAP** should be used to control the amount of memory used by a compiler itself. That equation is set in the configuration file

[2]We use XDS in most of our developments.

(`xc.cfg`). We recommend to set it according to the amount of physical memory in your computer:

RAM in megabytes	COMPILERHEAP
32-64	16000000
64-128	48000000
more than 128	96000000

It may be necessary to increase **COMPILERHEAP** if you get the "`out of memory`" message (F950). It is very unlikely, if **COMPILERHEAP** is set to 16 megabytes or more. Your compilation unit should be very large to exceed this memory limit. **Note:** if you are using Win32 or X Window API definition modules, set **COMPILERHEAP** to at least 16 megabytes.

Vice versa, if you notice unusually intensive disk activity when compiling your program, it may indicate that the value of the **COMPILERHEAP** equation is too large for your system configuration.

Set **COMPILERHEAP** to zero if would prefer the compiler to dynamically adjust heap size in accordance with system load.

See 9.1 for more information on XDS memory management.

3.4 Directory hierarchies

XDS compilers give you complete freedom over where you store both your source code files and any files which compilers create for you. It is advisable to work in a project oriented fashion — i.e. to have a separate directory hierarchy for each independent project.

Due to the re-usable nature of modules written in Modula-2 or Oberon-2, it is wise to keep a separate directory for those files which are to be made available to several projects. We will call such files the *library* files.

We recommend you to have a separate working directory for each project. You can also create subdirectories to store symbol files and generated code files. We recommend to use the supplied script or its customized version to create all subdirectories and, optionally, a local redirection file or a project file. Refer to the "Read Me First" file for more information about that script.

3.5 XDS search paths

Upon activation, xc looks for a file called `xc.red` — a *redirection file*. That file defines paths by which all other files are sought. If a redirection file was not found in the current directory, the master redirection file is loaded from the directory where xc executable is placed.

3.5.1 Redirection file

A redirection file consists of several lines of the form[3]:

```
pattern = directory {";" directory}
```

`pattern` is a regular expression with which names of files xc has to open or create are compared. A pattern usually contains wildcard symbols '*' and '?', where

Symbol	Matches
*	any (possibly empty) string
?	any single character.

For a full description of regular expressions see 3.5.2.

It is also possible to have comment lines in a redirection file. A comment line should start with the "%" symbol.

A portable notation (see 3.2) is used for directory names or paths. A path may be absolute or relative, i.e. may consist of full names such as

```
/usr/myproj/def
```

or of names relative to the current directory, such as

```
src/common
```

denoting the directory `src/common` which is a subdirectory of the current directory. A single dot as a pathname represents the current directory, a double dot represents the parent, i.e. the directory which has the current directory as a subdirectory.

The base directory macro `$!` can be used in a directory name. It denotes the path to the redirection file. If the redirection file is placed in the `/usr/alex` directory then `$!/sym` denotes the `/usr/alex/sym` directory, whereas `$!/..` denotes the `/usr` directory.

[3] See also 4.4

For any file, its name is sequentially matched with a pattern of each line. If a match was found, the file is sought in the first of the directories listed on that line, then in the second directory, and so on until either the file is found, or there are no more directories to search or there are no more patterns to match.

If `xc` could not locate a file which is needed for correct operation, e.g. a necessary symbol file, it terminates with an appropriate error message.

When creating a file, `xc` also uses redirection, and its behavior is determined by the **OVERWRITE** option. If the option was set ON, `xc` first searches for the file it is about to create using redirection. Then, if the file was found, `xc` overwrites it. If no file of the same name as the one which `xc` needs to create was found or the **OVERWRITE** option was set OFF, then the file is be created in the directory which appears first in the search path list which pattern matched the filename.

If no pattern matching a given filename can be found in the `xc.red` file, then the file will be read from (or written to) the current working directory.

Note: If a pattern matching a given filename is found then `xc` will *not* look into the current directory, unless it is explicitly specified in the search path.

The following entry in `xc.red` would be appropriate for searching for the symbol files (provided that symbol files have the extension **.sym**).

```
*.sym=sym;/usr/xds/sym;.
```

Given the above redirection, the compiler will first search for symbol files in the directory `sym` which is a subdirectory of the current working directory; then in the directory storing the XDS library symbol files and then in the current directory.

Example of a redirection file:

```
xc.msg = /xds/bin
*.mod = mod
*.def = def
*.ob2 = oberon
*.sym = sym; /xds/sym/x86
```

3.5.2 Regular expression

A regular expression is a string containing certain special symbols:

Sequence	Denotes
*	an arbitrary sequence of any characters, possibly empty (equivalent to {\000-\377} expression)
?	any single character (equivalent to [\000-\377] expression)
[...]	one of the listed characters
{...}	an arbitrary sequence of the listed characters, possibly empty
\nnn	the ASCII character with octal code nnn, where n is [0-7]
&	the logical operation AND
|	the logical operation OR
^	the logical operation NOT
(...)	the priority of operations

A sequence of the form a-b used within either [] or {} brackets denotes all characters from a to b.

Examples

`*.def`
: all files which extension is .def

`project.*`
: files which name is project with an arbitrary extension

`*.def|*.mod`
: files which extension is either .def or .mod

{a-z}*X.def
: files starting with any sequence of letters, ending in one final "X" and having the extension .def.

3.6 Options

A rich set of xc options allows one to control the source language, code generation and internal limits and settings. We distinguish between boolean options (or just options) and equations. An *option* can be set ON (TRUE) or OFF (FALSE), while an *equation* value is a string. In this chapter we describe only the syntax of setup directive. The full list of xc options and equations is provided in the Chapter 5.

Options and equations may be set in a configuration file (see 3.7), on the command line (see 4.2), in a project file (see 4.5)), and in the source text (see 7.7).

The same syntax of a setup directive is used in configuration and project files and on the command line. The only difference is that arbitrary spaces are permitted in files, but not on the command line. Option and equation names are case independent.

```
SetupDirective   = SetOption
                 | SetEquation
                 | DeclareOption
                 | DeclareEquation
                 | DeclareSynonym
SetOption        = '-' name ('+'| '-')
SetEquation      = '-' name '=' [ value ]
DeclareOption    = '-' name ':' [ '+' | '-' ]
DeclareEquation  = '-' name ':=' [ value ]
DeclareSynonym   = '-' name '::' name
```

All options and equations used by `xc` are predeclared.

The `DeclareSynonym` directive allows one to use a different name (e.g. shorter name) for an option or equation.

The old version of `SetOption` is also supported for convenience:

```
OldSetOption   = '+' name | '-' name
```

Examples

Directive	Meaning
`-M2Extensions+`	**M2EXTENSION** is set ON
`-Oberon=o2`	**OBERON** is set to `"o2"`
`-debug:`	**DEBUG** is declared and set OFF
`-Demo:+`	**DEMO** is declared and set ON
`-Vers:=1.0`	**VERS** is declared and set to `"1.0"`
`-A::genasm`	**A** is declared as a synonym for **GENASM**
`+m2extensions`	**M2EXTENSIONS** is set OFF

3.7 Configuration file

A configuration file can be used to set the default values of options and equations (see Chapter 5) for all projects (or a set of projects). A non-empty line of a configuration file may contain a single compiler option or equation setup directive (see

3.6) or a comment. Arbitrary spaces are permitted. The "%" character indicates a comment; it causes the rest of a line to be discarded. **Note:** the comment character can not be used when setting an equation.

The *master configuration file*, placed along with the `xc` utility, usually contains default settings for the target platform and declarations of platform-specific options and equations, which may be used in project and template files.

```
% this is a comment line
% Set equation:
 - BSDef = df
% Set predeclared options:
 - RangeCheck -   % turn range checks off
 - M2EXTENSIONS + % allow Modula-2 extensions
% Declare new options:
 -iPentium:+
 -i80486:-
 -i80386:         % is equal to -i80386:-
% Declare synonym:
 -N :: checknil
 -N               % disallow NIL checks
% end of configuration file
```

Figure 3.1: A sample configuration file

3.8 Filename extensions

`xc` allows you to define what you want to be the standard extensions for each particular type of file. For instance, you may prefer your Oberon-2 source code texts to end in **.o2** instead of **.ob2**.

We recommend to either use the traditional extensions or at least the extensions which describe the kind of file they refer to, and keep same extensions across all your projects. For example, use **.def** and **.mod** for Modula-2 modules, **.ob2** for Oberon-2 modules, etc.

Certain other factors must also influence your decisions. Traditionally, Oberon-2 pseudo-definition modules (as created by a browser) are extended with a **.def**. With XDS, this may conflict with the extension used for Modula-2 definition modules. Therefore, the XDS browser (see 4.2.5) uses the extension **.odf** by default.

The following filename extensions are usually defined in the configuration file:

DEF	extension for Modula-2 definition modules
MOD	extension for Modula-2 implementation modules
OBERON	extension for Oberon-2 modules
BSDEF	extension for Oberon-2 pseudo definition modules
CODE	extension for generated code files
SYM	extension for symbol files

See Table 5.5 for the full list of file extensions.

Example (file extension entries in xc.cfg):

```
-def     = def
-mod     = mod
-oberon  = ob2
-sym     = sym
```

3.9 Customizing compiler messages

The file `xc.msg` contains texts of error messages in the form

```
number text
```

The following is an extract from xc.msg:

```
001 illegal character
002 comment not closed; started at line %d
...
042 incompatible assignment
...
```

Some messages contain format specifiers for additional arguments. In the above example, the message 002 contains a `%d` specifier used to print a line number.

To use a language other than English for compiler messages it is sufficient to translate xc.msg, preserving error numbers and the order of format specifiers.

3.10 XDS and your C compiler

XDS allows C object modules and libraries to be used in your projects. Different C compilers use different alignment, naming and calling conventions. **ATTENTION!** Since XDS libraries on linux are built through GCC compiler it is *absolutely neccesary* to configure XDS for GCC. See 10.5 for more details.

Chapter 4

Using the compiler

4.1 Invoking the compiler

The XDS Modula-2 and Oberon-2 compilers are combined together with the make subsystem and an Oberon-2 browser into a single utility, `xc`. When invoked without parameters, the utility outputs a brief help information.

`xc` is invoked from the command line of the following form

```
xc { mode | option | name }
```

where `name`, depending on the operation `mode` can be a module name, a source file name, or a project file name. See 4.2 for a full description of operation modes.

`option` is a compiler setup directive (See 3.6). All options are applied to all operands, notwithstanding their relative order on the command line. On some platforms, it may be necessary to enclose setup directives in quotation marks:

```
xc hello.mod '-checkindex+'
```

See Chapter 5 for the list of all compiler options and equations.

4.1.1 Precedence of compiler options

The `xc` utility receives its options in the following order:

1. from a configuration file **xc.cfg** (See 3.7)
2. from the command line (See 4.2)
3. from a project file (if present) (See 4.5)

4. from a source text (not all options can be used there) (See 7.7)

At any point during operation, the last value of an option is in effect. Thus, if the equation **OBERON** was set to **.ob2** in a configuration file, but then set to **.o2** on the command line, the compiler will use **.o2** as the default Oberon-2 extension.

4.2 XDS compilers operation modes

XDS Modula-2/Oberon-2 compilers have the following operation modes:

Mode	Meaning
COMPILE	Compile all modules given on the command line
PROJECT	Make all projects given on the command line
MAKE	Check dependencies and recompile
GEN	Generate makefile for all projects
BROWSE	Extract definitions from symbol files
HELP	Print help and terminate

Both the PROJECT and MAKE modes have two optional operation submodes: BATCH (see 4.2.7) and ALL (see 4.2.6). Two auxiliary operation submodes — options (see 4.2.8) and EQUATIONS (see 4.2.9) can be used to inspect the set of compiler options and equations and their values.

On the command line, the compiler mode is specified with the "=" symbol followed by a mode name. Mode names are not case sensitive, and specifying an unique portion of a mode name is sufficient, thus

`=PROJECT` is equivalent to `=p`
`=BROWSE` is equivalent to `=Bro`

Operation modes and options can be placed on the command line in arbitrary order, so the following two command lines are equivalent:

```
xc =make hello.mod =all -checknil+
xc -checknil+ =a =make hello.mod
```

4.2.1 COMPILE mode

```
xc [=compile] { FILENAME | OPTION }
```

COMPILE is the default mode, and can be invoked simply by supplying `xc` with a source module(s) to compile. If `xc` is invoked without a given mode, COMPILE mode is assumed. In order to determine which compiler should be used, `xc` looks

at the extensions of the given source files. The default mapping of extensions is given below :

.mod - Modula-2 implementation module
.def - Modula-2 definition module
.ob2 - Oberon-2 module

For example:

```
xc hello.mod
```

will invoke the Modula-2 compiler, whereas:

```
xc hello.ob2
```

will invoke the Oberon-2 compiler.

The user is able to reconfigure the extension mapping (See 3.8). It is also possible to override it from the command line using the options **M2** and **O2**:

```
xc hello.mod +o2  (* invokes O2 compiler *)
xc hello.ob2 +m2  (* invokes M2 compiler *)
```

Note: In the rest of this manual, the COMPILE mode also refers to any case in which the compiler *compiles* a source file, regardless of the actually specified mode (which can be COMPILE, MAKE, or PROJECT). For instance, an option or equation, which is stated to affect the compiler behaviour in the COMPILE mode, is relevant to MAKE and PROJECT modes as well.

4.2.2 MAKE mode

```
xc =make [=batch] [=all] { FILENAME | OPTION }
```

In the MAKE mode the compiler determines module dependencies using `IMPORT` clauses and then recompiles all necessary modules. Starting from the files on the command line, it tries to find an Oberon-2 module or a definition and implementaion module for each imported module. It then does the same for each of the imported modules until all modules are located. Note that a search is made for source files only. If a source file is not found, the imported modules will not be appended to the recompile list. See section 4.6 for more details.

When all modules are gathered, the compiler performs an action according to the operation submode. If the BATCH submode (see 4.2.7) was specified, it creates a batch file of all necessary compilations, rather than actually compiling the source code.

If the ALL submode (see 4.2.6) was specified, all gathered files are recompiled, otherwise XDS recompiles only the necessary files. The *smart recompilation* algorithm is described in 4.7.

Usually, a Modula-2 program module or an Oberon-2 top-level module is specified on the command line. In this case, if the **LINK** equation is set in either configuration file or `xc` command line, the linker will be invoked automatically in case of successful compilation. This feature allows you to build simple programs without creating project files.

4.2.3 PROJECT mode

```
xc =project [=batch] [=all] { PROJECTFILE | OPTION }
```

The PROJECT mode is essentially the same as the MAKE mode except that the modules to be 'made' are provided in a project file. A project file specifies a set of options and a list of modules. See 4.5 for further details. As in the MAKE mode, ALL (see 4.2.6) and BATCH (see 4.2.7) submodes can be used.

If a file extension of a project file is omitted, XDS will use an extension given by the equation **PRJEXT** (**.prj** by default).

It may be necessary to compile a single module in the environment specified in a project file. It can be accomplished in the COMPILE operation mode using with the **PRJ** equation:

```
xc -prj=myproject MyModule.mod
```

See also

- MAKE operation mode: 4.2.2
- Make strategy: 4.6
- Smart recompilation: 4.7

4.2.4 GEN mode

```
xc =gen { PROJECTFILE | OPTION }
```

The GEN operation mode allows one to generate a file containing information about your project. The most important usage is to generate a linker response file (See 2.4).

This operation mode can also be used to obtain additional information about your project, e.g. a list of all modules, import lists, etc.

A so-called template file, specified by the **TEMPLATE** equation, is used in this mode. A template file is a text file, some lines of which are marked with a certain symbol. All the lines which are not marked are copied to the output file verbatim. The marked lines are processed in a special way. See 4.8 for more information.

The compiler creates a file with a name specified by the equation **MKFNAME**. If the equation is empty, the project file name is used. A file name is then concatenated with the extension specified by the equation **MKFEXT**.

4.2.5 BROWSE mode

```
xc =browse { MODULENAME | OPTION }
```

The BROWSE operation mode allows one to generate a pseudo definition module for an Oberon-2 module. In this mode, the compiler reads a symbol file and produces a file which contains declarations of all objects exported from the Oberon-2 module, if a format resembling Modula-2 definition modules.

The configuration option **BSDEF** specifies the extension of a generated file. If this option is not set, then the default extension (**.odf**) will be used.

Options **BSCLOSURE** and **BSREDEFINE** can be used to control the form of a generated file. **Note:** the **BSTYLE** equation (described in 8.1.2) is ignored in this operation mode, and the browse style is always set to DEF.

The **MAKEDEF** option (See 8.1.2) provides an alternative method of producing pseudo definition modules, preserving so-called *exported* comments if necessary.

4.2.6 ALL submode

In both PROJECT and MAKE modes, the compiler checks the time stamps of the files concerned and recompiles only those files that are necessary (See 4.7). If the ALL submode was specified, the time stamps are ignored, and all files are compiled.

4.2.7 BATCH submode

In the BATCH submode, the compiler creates a batch file of all necessary compilations, rather than actually calling the compilers and compiling the source code.

A batch file is a sequence of lines beginning with the compiler name, followed by module names to recompile.

The compiler creates a batch file with a name determined by either:

1. The compiler option **BATNAME**
2. The project file name (if given)
3. The name **out** (if the name could not be determined by the above).

The name is then concatenated with the batch file extension specified by the equation **BATEXT** (**.bat** by default).

See also

- option **LONGNAME** (5.1)
- equation **BATWIDTH** (5.3)

4.2.8 OPTIONS submode

The OPTIONS submode allows you to inspect the values of options which are set in the configuration file, project file and on the command line. It can be used together with COMPILE (see 4.2.1), MAKE (see 4.2.2), and PROJECT (see 4.2.3) modes.

The following command line prints (to the standard output) the list of all defined options, including all pre-declared options, all options declared in the configuration file, in the project file `my.prj` and on the command line (`xyz` option):

```
xc =options -prj=my.prj -xyz:+
```

In the PROJECT mode options are listed for each project file given on the command line.

See also the EQUATIONS submode.

4.2.9 EQUATIONS submode

The EQUATIONS submode allows you to inspect the values of equations which are set in the configuration file, project file and on the command line. It can be

used together with COMPILE (see 4.2.1), MAKE (see 4.2.2), and PROJECT (see 4.2.3) modes.

See also the OPTIONS submode.

4.3 Files generated during compilation

When applied to a file which contains a module **name**, the compilers produce the following files.

4.3.1 Modula-2 compiler

When applied to a definition module, the Modula-2 compiler produces a *symbol file* (**name.sym**). The symbol file contains information required during compilation of a module which imports the module **name**.

When applied to an implementation module or a top level module, the Modula-2 compiler produces an object file (**name.o**).

4.3.2 Oberon-2 compiler

For all compiled modules, the Oberon-2 compiler produces a *symbol file* (**name.sym**) and an object file (**name.o**). The symbol file (**name.sym**) contains information required during compilation of a module which imports the module **name**. If the compiler needs to overwrite an existing symbol file, it will only do so if the **CHANGESYM** option is set ON.

Examples

Command line	Generated files
`xc Example.def`	`Example.sym`
`xc Example.mod`	`Example.o`
`xc Win.ob2 +CHANGESYM`	`Win.sym`
	`Win.o`

4.4 Control file preprocessing

An XDS compiler may read the following control files during execution:

- a redirection file (see 3.5.1)
- a configuration file (see 3.7)
- a project file (see 4.5)
- a template file (see 4.8)

All these files are preprocessed during read according to the following rules:

A control file is a plain text file containing a sequence of lines. The backslash character (`"\"`) at the end of a line denotes its continuation.

The following constructs are handled during control file preprocessing:

- macros of the kind `$(`*name*`)`. A macro expands to the value of the equation *name* or, if it does not exist, to the value of the environment variable *name*.
- the *base directory* macro (`$!`) This macro expands to the directory in which the file containing it resides.
- a set of directives, denoted by the exclamation mark (`"!"`) as a first non-whitespace character on a line.

A directive has the following syntax (all keywords are case independent):

```
Directive = "!" "NEW" SetOption | SetEquation
          | "!" "SET" SetOption | SetEquation
          | "!" "MESSAGE" Expression
          | "!" "IF" Expression "THEN"
          | "!" "ELSIF" Expression "THEN"
          | "!" "ELSE"
          | "!" "END".
SetOption   = name ( "+" | "-" ).
SetEquation = name "=" string.
```

The `NEW` directive declares a new option or equation. The `SET` directive changes the value of an existent option or equation. The `MESSAGE` directive prints `Expression` value to the standard output. The `IF` directive allows to process or skip portions of files according to the value of `Expression`. `IF` directives may be nested.

```
Expression  = Simple [ Relation Simple ].
Simple      = Term { "+" | OR Term }.
Relation    = "=" | "#" | "<" | ">".
Term        = Factor { AND Factor }.
Factor      = "(" Expression ")".
            | String
            | NOT Factor
            | DEFINED name
            | name.
String      = "'" { character } "'"
            | '"' { character } '"'.
```

An operand in an expression is either string, equation name, or option name. In the case of equation, the value of equation is used. In the case of option, a string `"TRUE"` or `"FALSE"` is used. The `"+"` operator denotes string concatenation. Relation operators perform case insensitive string comparison. The `NOT` operator may be applied to a string with value `"TRUE"` or `"FALSE"`. The `DEFINED` operator yields `"TRUE"` if an option or equation `name` is declared and `"FALSE"` otherwise.

See also section 5.6.

4.5 Project files

A project file has the following structure:

```
{SetupDirective}
{!module {FileName}}
```

Setup directives define options and equations that all modules which constitute the project should be compiled with. See also 3.6 and 4.4.

Every line in a project file can contain only one setup directive. The character "%" indicated a comment; it causes the rest of a line to be discarded. **Note:** the comment character can not be used in a string containing equation setting.

Each `FileName` is a name of a file which should be compiled, linked, or otherwise processed when a project is being built, e.g. a source file, an additional library, a resource file (on Windows), etc. The compiler processes only Modula-2 and Oberon-2 source files. The type of a file is determined by its extension (by default Modula-2/Oberon-2 source files extension is assumed). Files of other types are taken into account only when a template file is processed (see 4.8).

The compiler recursively scans import lists of all specified Modula-2/Oberon-2 source modules and builds the full list of modules used in the project. Thus, usually, a project file for an executable program would contain a single `!module` directive for the file which contains the main program module and, optionally, several `!module` directives for non-source files.

At least one `!module` directive should be specified in a project file.

A project file can contain several **LOOKUP** equations, which allow you to define additional search paths.

XDS compilers give you complete freedom over where to set options, equations and redirection directives. However, it is recommended to specify only those settings in the configuration and redirection files which are applied to all your projects, and use project files for all project-specific options and redirection directives.

```
-lookup = *.mod = mod
-lookup = *.sym = sym; $(XDSDIR)/sym/C
% check project mode
!if not defined mode then
  % by default use debug mode
  !new mode = debug
!end
% report the project mode
!message "Making project in the " + mode + " mode"
% set options according to the mode
!if mode = debug then
   - gendebug+
   - checkrange+
!else
   - gendebug-
!fi
% specify template file
- template = $!/templates/watcom.tem
!module hello
!module hello.res
```

Figure 4.1: A Sample Project File

Given the sample project file shown on Figure 4.1, the compiler will search for files with `.mod` and `.sym` extensions using search paths specified in the project file *before* paths specified in a redirection file.

A project file is specified explicitly in the PROJECT (see 4.2.3) and GEN (see 4.2.4) operation modes. In these modes, all options and equations are set and then the compiler proceeds through the module list to gather all modules constituting a project (See 4.6).

In the MAKE (see 4.2.2) and COMPILE (see 4.2.1) operation modes, a project file can be specified using the **PRJ** equation. In this case, the module list is ignored, but all options and equations from the project file are set.

The following command line forces the compiler to compile the module `hello.mod` using options and equations specified in the project file `hello.prj`:

```
xc hello.mod -prj=hello.prj
```

4.6 Make strategy

This section concerns MAKE (see 4.2.2), PROJECT (see 4.2.3), and GEN (see 4.2.4), operation modes. In these modes, an XDS compiler builds a set of all modules that constitute the project, starting from the modules specified in a project file (PROJECT and GEN) or on the command line (MAKE).

The MAKE mode is used in the following examples, but the comments also apply to the PROJECT and GEN modes.

First, the compiler tries to find all given modules according to the following strategy:

- If both filename extension and path are present, the compiler checks if the given file exists.

  ```
  xc =make mod/hello.mod
  ```

- If only an extension is specified, the compiler seeks the given file using search paths.

  ```
  xc =make hello.mod
  ```

- If no extension is specified, the compiler searches for files with the given name and the Oberon-2 module extension, Modula-2 implementation module extension, and Modula-2 definition module extension.

  ```
  xc =make hello
  ```

  An error is raised if more than one file was found, e.g. if both `hello.ob2` and `hello.mod` files exist.

Starting from the given files, the compiler tries to find an Oberon-2 source module or Modula-2 definition and implementation modules for each imported module. It then tries to do the same for each of the imported modules until all the possible modules are located. For each module, the compiler checks correspondence between the file name extension and the kind of the module.

4.7 Smart recompilation

In the MAKE (see 4.2.2) and PROJECT (see 4.2.3) modes, if the ALL (see 4.2.6) submode was not specified, an XDS compiler performs *smart recompilation* of modules which are inconsistent with the available source code files. The complier uses file modification time to determine which file has been changed. For each module the decision (to recompile or not) is made only after the decision is made for all modules on which it depends. A source file is (re)compiled if one or more of the following conditions is true:

Modula-2 definition module

- the symbol file is missing
- the symbol file is present but its modification date is earlier than that of the source file or one of the imported symbol files

Modula-2 implementation module

- the code file is missing
- the code file is present but the file modification date is earlier than that of the source file or one of the imported symbol files (including its own symbol file)

Modula-2 program module

- the code file is missing
- the code file is present but the file modification date is earlier than that of the source file or one of the imported symbol files

Oberon-2 module

- the symbol file is missing

- the symbol file is present but the modification date is earlier than that of one of the imported symbol files
- the code file is missing
- the code file is present but the file modification date is earlier than that of the source file or one of the imported symbol files

When the **VERBOSE** option is set ON, the compiler reports a reason for recompilation of each module. **Note:** if an error occured during compilation of a Modula-2 definition module or an Oberon-2 module, all its client modules are not compiled at all.

4.8 Template files

A *template file* is used to build a ”makefile” in the PROJECT (see 4.2.3) and GEN (see 4.2.4) operation modes, if the option **MAKEFILE** is ON[1].

The compiler copies lines from a template file into the output file verbatim, except lines marked as requiring further attention. A single character (attention mark) is specified by the equation **ATTENTION** (default is ’!’)

A template file is also subject to preprocessing (see 4.4).

A marked line (or template) has the following format[2]:

```
Template  = { Sentence }.
Sentence  = Item { "," Item } ";" | Iterator.
Item      = Atom | [ Atom | "^" ] "#" [ Extension ].
Atom      = String | name.
String    = '"' { character } '"'
          | "'" { character } "'".
Extension = [ ">" ] Atom.
Iterator  = "{" Set ":" { Sentence } "}".
Set       = { Keyword | String }
Keyword   = DEF | IMP | OBERON | MAIN
          | C | HEADER | ASM | OBJ.
```

`name` should be a name of an equation. Not more than three items may be used in a sentence. A first item in a sentence is a format string, while others are arguments.

[1] ”MAKEFILE” is a historical name; a linker or library manager response file may be built as well.

[2] The same syntax is used in the **LINK** equation.

The XDS distribution contains a template file `xc.tem` which can be used to produce a linker response file.

4.8.1 Using equation values

In the simplest form, a template line may be used to output a value of an equation. For example, if the template file contains the line

```
! "The current project is %s.\n",prj;
```

and the project `prj/test.prj` is processed, the output will contain the line

```
The current project is prj/test.prj.
```

Note: the line

```
! prj;
```

is valid, but may produce unexpected results under systems in which the backslash character ("\") is used as a directory names separator (e.g. OS/2 or Windows):

```
prj     est.prj
```

because `"\t"` in a format string is replaced with the tab character. Use the following form instead:

```
! "%s",prj;
```

4.8.2 File name construction

The `"#"` operator constructs a file name from a name and an extension, each specified as an equation name or literal string. A file is then searched for according to XDS search paths and the resulting name is substituted. For example, if the file `useful.lib` resides in the directory '../mylibs' and the redirection file contains the following line:

```
*.lib = /xds/lib;../mylibs
```

the line

```
! "useful"#"lib"
```

will produce

```
../mylibs/useful.lib
```

If the modifier `">"` is specified, the compiler assumes that the file being constructed is an output file and creates its name according to the strategy for output

files (See 3.5.1 and the **OVERWRITE** option).

The `"#"` operator is also used to represent the current value of an iterator (see 4.8.3). The form in which a name or extension is omitted can be used in an iterator only.

The form `"^#"` may be used in a second level iterator to represent the current value of the first level iterator.

4.8.3 Iterators

Iterators are used to generate some text for all modules from a given set. Sentences inside the first level of braces are repeated for all modules of the project, while sentences inside the second level are repeated for all modules imported into the module currently iterated at the first level. A set is a sequence of keywords and strings. Each string denotes a specific module, while a keyword denotes all modules of specific kind.

The meaning of keywords is as follows:

Keyword	Meaning
`DEF`	Modula-2 definition module
`IMP`	Modula-2 implementation module
`MAIN`	Modula-2 program module or Oberon-2 module marked as **MAIN**
`OBERON`	Oberon module
`ASM`	assembler source text
`OBJ`	object file

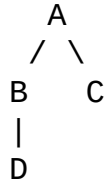
A keyword not listed above is treated as filename extension. Sentences are repeated for all files with that extension which are explicitly specified in the project file using `!module` directives (see 4.5). This allows, for instance, additional libraries to be specified in a project file:

```
sample.prj:

    -template = mytem.tem
    !module Sample.mod
    !module mylib.lib

mytem.tem:
      .  .  .
    ! "%s","libxds"#"lib"
```

```
    ! { lib: "+%s",#; }
    ! "\n"
      .  .  .

generated file:
      .  .  .
    d:\xds\lib\x86\libxds.lib+mylib.lib
      .  .  .
```

4.8.4 Examples

Consider a sample project which consists of a program module A, which imports modules B and C, and B, in turn, imports D (all modules are written in Modula-2):



The following examples illustrate template files usage:

This template line lists all project modules for which source files are available:

```
! { imp oberon main: "%s ",#; }
```

For the sample project, it would generate the following line:

```
A.mod B.mod C.mod D.mod
```

To output both definition and implementation modules, the following lines may be used:

```
! { def : "%s ",#; }
! { imp oberon main: "%s ",#; }
```

The output would be:

```
B.def C.def D.def A.mod B.mod C.mod D.mod
```

The last template line may be used to list all modules along with their import:

```
! { imp main: "%s\n",#; { def: "  %s\n",#; } }
```

The output:

```
A.mod
```

```
    B.def
    C.def
  B.mod
    D.def
  C.mod
  D.mod
```

Chapter 5

Compiler options and equations

A rich set of XDS compiler options allows you to control the source language, the generated code, and the internal limits and settings. We distinguish between boolean options (or just options) and equations. An option can be set ON (TRUE) or OFF (FALSE), while an equation value is a string.

5.1 Options

Options control the process of compilation, including language extensions, run-time checks and code generation. An option can be set ON (TRUE) or OFF (FALSE).

A compiler setup directive (See 3.6) is used to set the option value or to declare a new option.

Options may be set in a configuration file (see 3.7), on the command line (see 4.2), in a project file (see 4.5)). or in the source text (See 7.7). At any point of operation, the last value of an option is in effect.

Alphabetical list of all options along with their descriptions may be found in the section 5.2. See also tables 5.1 (page 36), 5.2 (page 36), 5.3 (page 37) and 5.4 (page 38).

Option	Meaning
ASSERT	enable ASSERT generation
CHECKDINDEX	check of dynamic array bounds
CHECKDIV	check for a positive divisor (DIV and MOD)
CHECKINDEX	check of static array bounds
CHECKNIL	NIL pointer dereference check
CHECKPROC	check of a formal procedure call
CHECKRANGE	range checks (range types and enumerations)
CHECKSET	range check of set operations
CHECKTYPE	dynamic type guards (Oberon-2 only)
COVERFLOW	cardinal overflow check
IOVERFLOW	integer overflow check

Table 5.1: Run-time checks

Option	Meaning
M2ADDTYPES	add SHORT and LONG types
M2BASE16	use 16-bits basic types in Modula-2
M2CMPSYM	compare symbol files in Modula-2
M2EXTENSIONS	enable Modula-2 extensions
O2ADDKWD	enable additonal keywords in Oberon-2
O2EXTENSIONS	enable Oberon-2 extensions
O2ISOPRAGMA	enable ISO Modula-2 pragmas in Oberon
O2NUMEXT	enable Oberon-2 scientific extensions
STORAGE	enable default memory management in Modula-2
TOPSPEED	enable Topspeed Modula-2-compatible extensions

Table 5.2: Source language control options

Option	Meaning
__GEN_C__	ANSI C code generation
__GEN_X86__	code generation for 386/486/Pentium/PentiumPro
DBGNESTEDPROC	generate information about procedure nesting
DBGQUALIDS	generate qualified identifiers in debug info
DEFLIBS	put the default library names into object files
DOREORDER	perform instruction scheduling
GENASM	generate assembly text
GENCPREF	generate underscore prefixes
GENDEBUG	generate code in the debug mode
GENFRAME	always generate a procedure frame
GENHISTORY	enable postmorten history
GENPTRINIT	generate a local pointer initialization
LINENO	generate line numbers in object files
NOHEADER	disable generation of a header file
NOOPTIMIZE	disable machine-independent opimizations
NOPTRALIAS	ignore pointer aliasing
ONECODESEG	generate one code segment
PROCINLINE	enable in-line procedure expansion
SPACE	favor code size over speed
VERSIONKEY	append version key to the module initialization
VOLATILE	declare variables as volatile

Table 5.3: Code generator control options

Option	Meaning
BSCLOSURE	browse control option
BSREDEFINE	browse control option
CHANGESYM	permission to change a symbol file
FATFS	limit file names to 8.3
GCAUTO	enables implicit call of the garbage collector
LONGNAME	use long names in batch files
M2	force the Modula-2 compiler
MAIN	mark an Oberon-2 main module
MAKEDEF	generate definition
MAKEFILE	generate makefile
O2	forces the Oberon-2 compiler
OVERWRITE	create a file, always overwrites the old one
VERBOSE	produce verbose messages
WERR	treat warnings as errors
WOFF	suppress warning messages
XCOMMENTS	preserve exported comments

Table 5.4: Miscellaneous options

5.2 Options reference

This section lists all options in alphabetical order. Those options that may be arbitrarily placed in the source code are marked as *inline* options (See also 7.7). There are also options which can be placed in a source file, but only in a module header (i.e. before any of the keywords `"DEFINITION"`, `"IMPLEMENTATION"`, and `"MODULE"`) These options are marked as *header*. If an option is not marked either as header or inline, then the result of setting it in the source text is undefined.

Operation modes in which an option has effect are listed in square brackets ([]) after the option name; the character '*' stands for all operation modes. For example, [browse] means that the option is used by the compiler in the BROWSE operation mode only.

Note: in the MAKE (see 4.2.2) and PROJECT (see 4.2.3) modes the compiler switches to the COMPILE (see 4.2.1) mode to compile each module.

Run-time check options are ON by default. If not explicitly specified, other options are OFF (FALSE) by default.

__**GEN_X86**__ [compile]

The compiler sets this option ON, if the code generation for 386/486/Pentium/PentiumPro is in operation.

The option can be used for compiling different text fragments for different targets. See also 7.7.2.

__**GEN_C**__ [compile]

The compiler sets this option ON, if the C code generation is in operation.

The option can be used for compiling different text fragments for different targets. See also 7.7.2.

ASSERT [compile] (*inline*)

If the option is OFF, the compiler ignores all calls of the standard procedure ASSERT.

Warning: Ensure that all ASSERT calls in your program do not have side effects (i.e. do not contain calls of other function procedures) before setting this option OFF.

The option is ON by default.

BSCLOSURE [browse]

Include all visible methods.

If the option is set ON, the browser includes all defined and inherited type-bound procedure declarations with all record declarations when creating a pseudo-definition module. See also 8.1.2.

BSREDEFINE [browse]

Include all redefined methods.

If the option is set ON, the browser includes original definitions of any overwritten type-bound procedures with record declarations. See also 8.1.2.

CHANGESYM [compile] (*header*)

Permission to change a module interface (a symbol file).

The Oberon-2 compiler creates a temporary symbol file every time an Oberon-2 module is compiled, compares this symbol file with the existing one and overwrites it with the new one if necessary. When the option is OFF (by default), the compiler reports an error if interface of a module (and, hence, its symbol file) has been changed and does not replace the old symbol file.

Note: if the **M2CMPSYM** option is set, the same is valid for compilation of a Modula-2 definition module, i.e., the **CHANGESYM** option should be set ON for the compilation to succeed if a module interface has been changed.

CHECKDINDEX [compile] (*inline*)

A check of dynamic array bounds.

If the option is set ON, the compiler generates index checks for dynamic arrays (`POINTER TO ARRAY OF T`).

The option is ON by default.

CHECKDIV [compile] (*inline*)

If the option is set ON, the compiler generates a check if a divisor is positive in `DIV` and `MOD` operators.

The option is ON by default.

CHECKINDEX [compile] (*inline*)

A check of static array bounds.

If the option is set ON, the compiler generates index checks for all arrays except dynamic (See the **CHECKDINDEX** option).

The option is ON by default.

CHECKNIL [compile] (*inline*)

If the option is set ON, the compiler generates NIL checks on all pointer dereferences.

The option is ON by default.

CHECKPROC [compile] (*inline*)

If the option is set ON, the compiler generates a NIL check when calling a procedure variable.

The option is ON by default.

CHECKRANGE [compile] (*inline*)

If the option is set ON, the compiler generates range checks for range types and enumerations.

The option is ON by default.

CHECKSET [compile] (*inline*)

If the option is set ON, the compiler generates range checks for set operations (`INCL`, `EXCL`, set aggregates).

The option is ON by default.

CHECKTYPE [compile, Oberon-2 only] (*inline*)

If the option is set ON, the compiler generates dynamic type guards.

The option is ON by default.

COVERFLOW [compile] (*inline*)

If the option is set ON, the compiler generates overflow checks for all cardinal (unsigned) arithmetic operators.

The option is ON by default.

DBGNESTEDPROC [compile]

If this option is set ON, the compiler includes procedure nesting data into debug information (CodeView and HLL formats only, see the **DBGFMT** equation).

This is a non-standard feature, so a third party debugger would not work correctly with an executable compiled with **DBGNESTEDPROC** ON. For instance, MSVC does not display local variables of nested procedures.

This option is OFF by default.

DBGQUALIDS [compile]

If the option is set ON, the compiler prefixes names of Modula-2/Oberon-2 global variable with the name of the respective module and underscore in debug information. This feature may help you distingushing identically named exported variables from different modules in third-party debuggers that do not support Modula-2/Oberon-2.

This option is OFF by default.

DEFLIBS [compile]

If the option is set ON, the compiler writes the default library names to the generated object files.

The option is ON by default.

DOREORDER [compile] (*header*)

Setting this option ON enables the *instruction scheduling* mechanism of the x86 code generator. It reorders CPU instructions so that independent instructions can be executed simultaneously whenever possible.

Note: this optimization significantly slows down the compiler, but results in a code perfomance gain of 5-15%.

FATFS [*]

Forces the compiler to limit file names to FAT ”8.3” convention.

GCAUTO [compile,*top-level module only*] (*header*)

Enables implicit calls of the garbage collector in the generated program. The option is ignored for all modules except the top-level module of the program. We recommend to set the option in the project or configuration file.

See also 9.1.

GENASM [compile] (*header*)

If this option is set ON, the compiler generates text in the assembly language instead of object files. The only assembler supported in the current version is GNU Assembler.

GENCPREF [compile] (*header*)

If the option is set ON, the compiler uses underscore as a prefix for all public names in object files.

GENDEBUG [compile] (*header*)

If the option is set ON, the compiler puts debug information into an object file.

In some cases, switching the option ON may reduce code quality.

See also the **DBGFMT** equation.

GENFRAME [compile] (*header*)

If the option is set ON, the compiler always generates a stack frame. It may be necessary to simplify debugging.

GENHISTORY [compile] (*header*)

If the option is set ON, the run-time system prints a stack of procedure calls (a file name and a line number) on abnormal termination of your program.

It should be set when compiling a main module of the program. In this case the required part of the run-time system will be added to the program. The option **LINENO** should be set for all modules in the program.

See 2.5 for an example.

Note: In some cases the printed list may contain incorrect lines, i.e. procedures that were not called in the given context (See 9.2).

GENPTRINIT [compile, Oberon-2 only] (*header*)

If the option is set ON, the compiler generates code for initialization of all local pointers, including variables, record fields and array elements. Values of all non-pointer record fields and array elements are undefined.

The option is ON by default.

IOVERFLOW [compile] (*inline*)

If the option is set ON, the compiler generates overflow checks of all integer (signed) arithmetic operators.

The option is ON by default.

LINENO [compile] (*header*)

If the option is set ON, the compiler inserts line number information into object files. This option should be set ON to get the postmortem history (See the **GENHISTORY** option) and for debugging.

LONGNAME [make,project]

Use long names.

If the option is set ON, the compiler uses full path as a prefix for all module names in the generated batch files. See also 4.2.7.

M2 [compile]

Force the Modula-2 compiler.

If the option is set ON, the Modula-2 compiler is invoked regardless of file extension. The option is ignored in MAKE and PROJECT modes.

M2ADDTYPES [compile,Modula-2 only] (*header*)

Add short and long modifications of whole types.

If the option is set ON, the compiler recognizes the types `SHORTINT`, `LONGINT`, `SHORTCARD` and `LONGCARD` as pervasive identifiers .

Warning: Usage of additional types may cause problems with the software portability to other compilers.

M2BASE16 [compile,Modula-2 only] (*header*)

If the option is set ON, the basic types `INTEGER`, `CARDINAL`, and `BITSET` are 16 bits wide in Modula-2. By default, they are 32 bits wide.

M2CMPSYM [compile,Modula-2 only]

If the option is set ON, the Modula-2 compiler compares the symbol file generated for a definition module with the old version exactly as the Oberon-2 compiler does. If the symbol files are equal, the old one is preserved, otherwise the compiler overwrites symbol file, but only if the **CHANGESYM** option is set ON.

M2EXTENSIONS [compile,Modula-2 only] (*header*)

If the option is set ON, the compiler allows XDS Modula-2 language extensions (see 7.6), such as line comment ("- -"), read-only parameters, etc., to be used in the source code.

Warning: Extensions usage may cause problems with porting to third-party compilers.

MAIN [compile, Oberon-2 only] (*header*)

Mark the Oberon-2 main module.

If the option is set ON, the compiler generates a program entry point ('main' function) for the Oberon-2 module (See 8.1.1). Recommended to be used in a module header.

MAKEDEF [compile,Oberon-2 only]

Forces the Oberon-2 compiler to generate a (pseudo-) definition module after successful compilation of an Oberon-2 module. The compiler preserves the so-called *exported* comments (i.e. comments started with '(**') if the **XCOMMENTS** option is set ON.

See 8.1.2.

MAKEFILE [project]

Forces the compiler to generate a makefile after successful compilation of a project. See also 4.2.4 and 4.8.

NOHEADER [compile,make,project] (*header*)

This option is used by translators to C. Native code compilers recognize but ignore it.

NOOPTIMIZE [compile] If this option is set OFF (default), the machine-independent optimizer is invoked before code generation. Setting it ON causes less optimized, but still not straightforward code to be produced.

NOPTRALIAS [compile] (*header*)

If the option is set ON, the compiler assumes that there is no pointer aliasing, i.e. there are no pointers bounded to non-structure variables. The only way to get a pointer to a variable is to use the low-level facilities from the module SYSTEM. We recommend to turn this option ON for all modules except low-level ones. **Note:** the code quality is better if the option is ON.

O2 [compile]

Force Oberon-2 compiler.

If the option is set ON, the Oberon-2 compiler is invoked regardless of the file extension. The option is ignored in MAKE and PROJECT modes.

O2ADDKWD [compile,Oberon-2 only] (*header*)

Allows Modula-2 exceptions (see 7.2.13) and finalization (see 7.2.12) to be used in Oberon-2 programs, adding keywords `EXCEPT`, `RETRY`, and `FINALLY`.

Warning: Usage of this extension will prevent your program from porting to other Oberon-2 compilers.

O2EXTENSIONS [compile,Oberon-2 only] (*header*)

If the option is set ON, the compiler allows Oberon-2 language extensions to be used (See 8.5).

Warning: Extensions usage will affect portability to third-party Oberon-2 compilers.

O2ISOPRAGMA [compile,Oberon-2 only]

If the option is set ON, the compiler allows the ISO Modula-2 style pragmas <* *> to be used in Oberon-2. See 8.2.3 and 7.7.

Warning: Usage of ISO Modula-2 pragmas may cause problems when porting source code to third-party Oberon-2 compilers.

O2NUMEXT [compile,Oberon-2 only] (*header*)

If the option is set ON, the compiler allows the Oberon-2 scientific language extensions to be used (See 8.5), including `COMPLEX` and `LONGCOMPLEX` types and the in-line exponentiation operator.

Warning: Usage of additional types may cause problems with portability to other compilers.

ONECODESEG [compile]

If the option is ON, the compiler produces only one code segment which contains all code of a module, otherwise it generates a separate code segment for each procedure.

Warning: Setting this option ON disables smart linking.

OVERWRITE [*]

The option changes the way the compiler selects a directory for output files. If the option is OFF, the compiler always creates a file in the directory which appears first in the search path list correspondent to a pattern matching the file name. Otherwise, the compiler overwrites the old file, if it does exist in any directory of that list. See also 3.5.1.

PROCINLINE [compile]

If the option is ON, the compiler tries to expand procedures in-line. In-line expansion of a procedure eliminates the overhead produced by a procedure call, parameter passing, register saving, etc. In addition, more optimizations become possible because the optimizer may process the actual parameters used in a particular call.

A procedure is not expanded in-line under the following circumstances:

- the procedure is deemed too complex or too large by the compiler.
- there are too many calls of the procedure.
- the procedure is recursive.

SPACE [compile]

If the option is set ON, the compiler performs optimizations to produce smaller code, otherwise (by default) to produce faster code.

STORAGE [compile, Modula-2 only] (*header*)

If the option is set ON, the compiler uses the default memory allocation and deallocation procedures for the standard procedures `NEW` and `DISPOSE`.

Warning: Usage of this option may cause problems with software portability to other compilers.

VERBOSE [make,project]

If the option is set ON, the compiler will report a reason for each module (re)compilation (See 4.7).

VERSIONKEY [compile]

This option may be used to perform version checks at link time. If the option is set ON, the compiler generates a name of a module body as composition of

- a module name
- a string "`_BEGIN_`"
- a time stamp

If a Modula-2 definition module or an Oberon-2 module imported by different compilation units has the same version, the same name is generated for each call of the module body. In all other cases unresolved references will be reported at link time.

If the option is OFF, the compiler generates module body names in a form: `<module_name>_BEGIN`.

Note: the option should be set when compiling a Modula-2 definition module or an Oberon-2 module.

VOLATILE [compile] (*inline*)

If this option appears to be switched ON during compilation of a variable definition, the compiler will assumes that references to that variable may have side effects or that the value of the variable may change in a way that can not be determined at compile time. As a result, the optimizer will not eliminate any operation involving that variable, and changes to the value of the variable will be stored immediately.

WERR [*] (*inline*)

When the option `WERRnnn` (e.g. `WERR301`) is set ON, the compiler treats the warning `nnn` (301 in the above example) as error. See the xc.msg file for warning texts and numbers.

`-WERR+` forces the compiler to treat all warnings as errors.

WOFF [*] (*inline*)

When the option `WOFFnnn` (e.g. `WOFF301`) is set ON, the compiler does not report the warning `nnn` (301 in the above example). See the xc.msg file for warning texts and numbers.

`-WOFF+` disables all warnings.

XCOMMENTS [compile,Oberon-2 only]

If the option is set ON, the browser includes so-called *exported* comments (i.e. comments which start with "`(**`") into a generated pseudo definition module.

See also 8.1.2.

5.3 Equations

An *equation* is a pair (`name`,`value`), where `value` is in general case an arbitrary string. Some equations have a limited set of valid values, some may not have the empty string as a value.

A compiler setup directive (See 3.6) is used to set an equation value or to declare a new equation.

Equations may be set in a configuration file (see 3.7), on the command line (see 4.2) and in a project file (see 4.5)). Some equations may be set in the source text, at an arbitrary position (marked as *inline* in the reference), or only in the module header (marked as *header*). At any point of operation, the most recent value of an equation is in effect.

Alphabetical list of all equations may be found in the section 5.4. See also tables 5.5 (page 48), 5.6 (page 49), 5.7 (page 50)

Name	Default	File type
BATEXT	`.bat`	recompilation batch file
BSDEF	`.odf`	pseudo-definition file created by browser
CODE	`.o`	object file
DEF	`.def`	Modula-2 definition module
MKFEXT	`.mkf`	makefile
MOD	`.mod`	Modula-2 implementation or main module
OBERON	`.ob2`	Oberon-2 module
OBJEXT	`.o`	object file
PRJEXT	`.prj`	project file
SYM	`.sym`	symbol file

Table 5.5: File extensions

Name	Default	Meaning
ALIGNMENT	4	data alignment (*please read details below*)
CC	GCC	C compiler compatibility
CODENAME	_TEXT	Code segment name
CPU	GENERIC	CPU to optimize for
DATANAME	_DATA	data segment name
DBGFMT	see desc.	debug information format
ENUMSIZE	4	default size of enumeration types
GCTHRESHOLD		garbage collector threshold (obsolete)
HEAPLIMIT	0	generated program heap limit
MINCPU	386	CPU required for execution
OBJFMT	ELF	object file format
SETSIZE	4	default size of small set types
STACKLIMIT	0	generated program stack limit

Table 5.6: Code generator equations

5.4 Equations reference

Operation modes in which an equation has effect are enclosed in square brackets ([]) after the equation name; the character '*' stands for all operation modes. For example [browse] means that the equation is used by the compiler in the BROWSE operation mode only. **Note:** the compiler switches from the MAKE and PROJECT mode to the COMPILE mode to compile a module.

ALIGNMENT [compile] (*inline*)

This equation sets the *data alignment*. Valid values are: 1,2,4, or 8. See 12.1.7 for further details.

Warning: Since XDS libraries are built through GCC which uses 4 byte alignment, you should always keep ALIGNMENT set to 4, unless you exactly know what you are doing. See 10.5 for more details.

ATTENTION [project,gen]

The equation defines an attention character which is used in template files ("!" by default). See 4.8.

BATEXT [make,project,*batch submode*]

Name	Default	Meaning
ATTENTION	`!`	attention character in template files
BATNAME	`out`	batch file name
BATWIDTH	`128`	maximum line width in a batch file
BSTYLE	`DEF`	browse style (See 8.1.2)
COMPILERHEAP		heap limit of the compiler
COMPILERTHRES		compiler's garbage collector threshold (obsolete)
DECOR	`hrtp`	control of compiler messages
ERRFMT	See 5.5	error message format
ERRLIM	`16`	maximum number of errors
FILE		name of the file being compiled
LINK		linker command line
LOOKUP		lookup directive
MKFNAME		makefile name
MODULE		name of the module being compiled
PRJ		project file name
PROJECT		project name
TABSTOP	`8`	tabulation alignment
TEMPLATE		template name (for makefile)

Table 5.7: Miscellaneous equations

Sets the file extension for recompilation batch files (by default **.bat**). See 4.2.7.

BATNAME [make,project,*batch submode*]

Sets the batch file name.

The name of the project file will be used if no batch file name is explicitly specified. See 4.2.7.

BATWIDTH [make,project,*batch submode*]

Sets the maximum width of a line in a generated batch file (by default 128). See 4.2.7.

BSDEF [browse]

Sets the file extension for pseudo-definition modules created by the browser (by default **.odf**). See 4.2.5.

BSTYLE [browse]

Sets the *style* of generated pseudo-definition modules. See 8.1.2.

CC [compile]

Sets the C compiler compatibility mode. The correspondent calling and naming conventions will be used for procedures and variables declared as `["C"]`.

Currently the only valid value on Linux is ”GCC”.

If the value of the equation is undefined, ”GCC” is assumed.

See 10.5 for more details.

CODE [*]

Sets the file extension for code files generated by the compiler (by default **.o**).

CODENAME [compile] (*header*)

Sets name for a code segment.

COMPILERTHRES [*]

This equation is left for compatibility; it is ignored by the compiler. In versions prior to 2.50, it was used to fine tune the compiler’s garbage collector.

See also 3.3.

COMPILERHEAP [*]

Sets the maximum amount of heap memory (in bytes), that can be used by the compiler. For systems with virtual memory, we recommend to use a value which is less than the amount of physical memory.

Setting this equation to zero forces adaptive compiler heap size adjustment according to system load.

CPU [compile]

Specifies on which Intel x86 family representative the resulting program will be executed optimally.

Valid values: ”386”, ”486”, ”PENTIUM”, and ”PENTIUMPRO”. The value must be ”greater of equal” than the value of the **MINCPU** equation.

There is also the special value ”GENERIC”, which means that the optimizer should not perform code transformations that may *significantly* reduce performance on a particular CPU.

DATANAME [compile] (*header*)

Sets name for a data segment.

DBGFMT [compile]

Sets debug information format for output object files. Valid values are ”CodeView” and ”HLL”.

DECOR [*]

The equation controls output of the xc utility. The value of equation is a string that contains any combination of letters ”h”, ”t”, ”r”, ”p” (capital letters are also allowed). Each character turns on output of

h header line, which contains the name and version of the compiler’s front-end and back-end

p progress messages

r compiler report: number of errors, lines, etc.

t the summary of compilation of multiple files

By default, the equation value is ”hrt”.

DEF [*]

Sets the file extension for Modula-2 definition modules (by default **.def**).

ENUMSIZE [compile](*inline*)

Sets the default size for enumeration types in bytes (1,2, or 4). If an enumeration type does not fit in the current default size, the smallest suitable size will be taken.

ENV_HOST [*]

A symbolic name of the host platform.

ENV_TARGET [*]

This equation is always set to a symbolic name of the target platform (CPU/operating system).

ERRFMT [*]

Sets the error message format. See 5.5 for details.

ERRLIM [*]

Sets the maximum number of errors allowed for one compilation unit (by default 16).

FILE [compile]

The compiler sets this equation to the name of the currently compiled file. See also the **MODULE** equation.

GCTHRESHOLD [compile,*top-level module only*]

This equation is left for compatibility; it is ignored by the compiler. In versions prior to 2.50, it was used to fine tune the garbage collector.

See also 9.1.

HEAPLIMIT [compile,*top-level module only*]

Sets the maximum amount of heap memory, that can be allocated by the generated program. The value is set in bytes.

Setting this equation to zero enables the run-time system to dynamically adjust heap size according to application's memory demands and system load.

The equation should be set when the top-level module of the program is compiled. We recommend to set it in a project file or the configuration file.

See also 9.1.

LINK [project]

Defines a command line, which will be executed after a successful completion of a project. As a rule, the equation is used for calling a linker or make utility.

See 2.4 for examples.

LOOKUP [*]

Syntax:

```
-LOOKUP = pattern = directory {";" directory }
```

The equation can be used for defining additional search paths that would complement those set in the redirection file. A configuration or project file may contain several **LOOKUP** equations; they are also permitted on the command line.

See also 3.5.1 and 4.5.

MINCPU [compile]

Specifies an Intel x86 family representative which (or higher) is requried for the resulting program to be executed.

Valid values: ”GENERIC”, ”386”, ”486”, ”PENTIUM”, and ”PENTIUMPRO”. For this equation, ”GENERIC” is equivalent to ”386”. The value of the **CPU** equation must be ”greater of equal” than the value of this equation.

MKFEXT [gen]

Sets the file extension for generated makefiles (by default **.mkf**). See 4.2.4.

MKFNAME [gen]

Sets the name for a generated makefile. See 4.2.4.

MOD [*]

Sets the file extension for Modula-2 implementation and program modules (by default **.mod**).

MODULE [compile]

The compiler sets this equation to the name of the currently compiled module. See also the **FILE** equation.

OBERON [*]

Sets the file extension for Oberon-2 modules (by default **.ob2**).

OBJEXT [*]

Sets the file extension for object files (by default **.o**).

OBJFMT [compile]

Sets format for output object files. Valid values are ”OMF”, ”COFF” and ”ELF”.

PRJ [compile,make,project]

In the COMPILE and MAKE operation modes, the equation defines a project file to read settings from. In the PROJECT mode, the compiler sets this equation to a project file name from the command line. See 4.2.3.

PRJEXT [compile,make,project]

Sets the file extension for project files (by default **.prj**). See 4.2.3.

PROJECT [compile,make,project]

If a project file name is defined, the compiler sets the equation to a project name without a file path and extension. For example, if the project file name is `prj/Work.prj`, the value of the equation is set to `Work`. The equation may be used, for instance, in a template file to set the name of the executable file.

SETSIZE [compile](*inline*)

Sets the default size for small (16 elements or less) set types in bytes (1,2, or 4). If a set type does not fit in the current default size, the smallest suitable size will be taken.

STACKLIMIT [compile,*top-level module only*]

Sets the maximum size of the stack in a generated program. The value is set in bytes.

The equation should be set when a top-level module of a program is compiled. We recommend to set the option in the project or configuration file.

Note: for some linkers the stack size should be also set as a linker option.

SYM [*]

Sets the file extension for symbol files (by default **.sym**). See 4.3.

TABSTOP [gen]

When reading text files, the compiler replaces the ASCII TAB character with the number of spaces required to align text (by default **TABSTOP** is

equal to 8). A wrong value may cause misplaced comments in a generated pseudo-definition module, incorrect error location in an error message, etc. We recommend to set this equation to the number used in your text editor.

TEMPLATE [gen]

Sets a name of a template file. See 4.8.

5.5 Error message format specification

The format in which XDS reports errors is user configurable through the **ERRFMT** equation. Its syntax is as follows:

```
{ string "," [ argument ] ";" }
```

Any format specification allowed in the C procedure `printf` can be used in `string`.

Argument	Type	Meaning
line	integer	position in a source text
column	integer	position in a source text
file	string	name of a source file
module	string	module name
errmsg	string	message text
errno	integer	error code
language	string	Oberon-2 or Modula-2
mode	string	ERROR or WARNING or FAULT
utility	string	name of an utility

Argument names are not case sensitive. By default, the error message format includes the following clauses:

`"(%s",file;`	—	a file name
`"%d",line;`	—	a line number
`",%d",column;`	—	a column number
`") [%.1s] ",mode;`	—	the first letter of an error mode
`"%s\n",errmsg;`	—	an error message

If a warning is reported for the file `test.mod` at line 5, column 6, the generated error message will look like this:

```
(test.mod 5,6) [W] variable declared but never used
```

5.6 The system module COMPILER

The system module `COMPILER` provides two procedures which allow you to use compile-time values of options and equations in your Modula-2 or Oberon-2 program:

```
PROCEDURE OPTION(<constant string>): BOOLEAN;
PROCEDURE EQUATION(<constant string>): <constant string>;
```

Both this procedures are evaluated at compile-time and may be used in constant expressions.

Note: The `COMPILER` module is non-standard.

Examples

```
Printf.printf("This program is optimized for the %s CPU\n",
              COMPILER.EQUATION("CPU"));

IF COMPILER.OPTION("__GEN_C__") THEN
  ...
END;
```

Chapter 6

Compiler messages

This chapter gives explanation for compiler diagnostics. For each error, an error number is provided along with a text of error message and an explanation. An error message can contain a format specifier in the form `%d` for a number or `%s` for a string. In this case, an argument (or arguments) is described on the next line.

In most cases the compiler prints a source line in which the error was found. The position of the error in the line is marked with a dollar sign placed directly before the point at which the error occurred.

6.1 Lexical errors

E001
`illegal character`

All characters within the Modula-2 or Oberon-2 character sets are acceptable. Control characters in the range `0C` to `37C` are ignored. All other characters, e.g. `%` are invalid.

E002
`comment not closed; started at line %d` (line number)

This error is reported if a closing comment bracket is omitted for a comment started at the given line.

E003
`illegal number`

This error is reported in the following cases:

- a numeric constant contains a character other than a digit (`0..7` for octal constants, `0..9` for decimal, `0..9,A..F` for hexadecimal).
- an exponent indicator is not followed by an integer
- an illegal suffix is used after a number, e.g. `"X"` in Modula-2 or `"C"` or `"B"` in Oberon-2.

E004
`string literal not closed or too long`

This error usually occurs if a closing quote is omitted or does not match the opening quote. Note that a string literal is limited to a single line and its size cannot exceed 256 characters. In Modula-2, string concatenation may be used to build long literal strings.

F005
`unexpected end of file`

Input file ends before end of a module.

E006
`identifier too long`

Length of an identifier exceeds compiler limit (127 characters).

F010
`source text read error`

A read error occurs while reading source text.

E012
`character constant too large (377C or 0FFX is maximum)`

The meaning of this message is obvious.

E171
`illegal structure of conditional compilation options`

This error is reported if a structure of conditional `IF` statements is broken, e.g. there is no `IF` for an `END`, `ELSE`, or `ELSIF` clause or there is no `END` for an `IF`.

E172
`conditional compilation option starts with incorrect symbol`

`IF`, `ELSIF`, `ELSE`, `END` or identifier expected.

F173
`pragma not closed; started at line %d` (line number)

This error is reported if a closing bracket `"*>"` is omitted for a pragma started at the given line.

F174
`unexpected end of file while skipping; see at %d` (line number)

Input file ended while the compiler was skipping source text according to the conditional compilation statement. It may be a result of a missed `<* END *>` clause. Check the pragma at the given line.

E175
`invalid pragma syntax`

Check the manual for the pragma syntax.

6.2 Syntax errors

E007
`identifier expected`

The compiler expects an identifier at the indicated position.

E008
`expected symbol %s` (symbol)

The compiler expects the given symbol at the indicated position. The symbol may be one of the following:

```
|       ;       :       .       [       ]       :=
(       )       {       }       ,       =       ..
DO      END     OF      THEN    TO      UNTIL   IMPORT
MODULE
```

E081
`expected start of factor`

The compiler expects start of *factor* at the indicated position, i.e. an identifier, literal value, aggregate, left parenthesis, etc. See the syntax of the language for more information.

E082
`expected start of declaration`

The compiler expects start of declaration at the indicated position, i.e. one of the keywords: `"CONST"`, `"VAR"`, `"TYPE"`, `"PROCEDURE"`, `"BEGIN"`, or `"END"`.

E083
`expected start of type`

The compiler expects start of a type at the indicated position. See the syntax of the language for more information.

E085
`expected expression`

The compiler expects expression at the indicated position.

E086
`expected start of statement`

The compiler expects start of a statement at the indicated position. See the syntax of the language for more information.

6.3 Semantic errors

E020
`undeclared identifier "%s"` (name)

The given identifier has no definition in the current scope.

E021
`type identifier "%s" shall not be used in declaring itself` (name)

An identifier being declared as a type shall not be used in declaring that type, unless that type is a new pointer type or a new procedure type. This error will be reported for the following example

```
TYPE
  Rec = RECORD
    next: POINTER TO Rec;
  END;
```

use the following declarations instead:

```
TYPE
```

```
Ptr = POINTER TO Rec;
Rec = RECORD
  next: Ptr;
END;
```

E022
`identifier "%s" was already defined at %s[%d.%d]`
(name,file name,line,column)
E028
`identifier "%s" was already defined in other module`
(name)

An identifier being declared is already known in the current context (the name used has some other meaning). If a file name and text position of previous definition are known, the compiler reports error 022 otherwise 028.

E023
`procedure with forward declaration cannot be a code procedure`

A forward declaration of a procedure is followed by a declaration of a code procedure.

E024
`recursive import not allowed`

A module imports itself. Example:

```
MODULE xyz;

IMPORT xyz;

END xyz.
```

E025
`unsatisfied exported object`

An object exported from a local object is not defined there. Example:

```
MODULE M; (* local module *)

EXPORT Foo;

END M;
```

E026
`identifier "%s" is used in its own declaration, see %s[%d.%d]`

An identfier cannot be used in its own declaration, like in:

```
CONST c = 1;
PROCEDURE proc;
  CONST c = c + 1;
END proc;
```

E027
`illegal usage of module identifier "%s"` (module name)

An identifier denoting module cannot be used at the indicated position.

E029
`incompatible types: "%s" "%s"`(type,type)
E030
`incompatible types`

The compiler reports this error in the following cases:

- operands in an expression are not expression compatible
- an expression is not compatible with the type of the variable in an assignment statement
- an actual parameter is not compatible with the type of the formal parameter in a procedure call

The compiler reports error 29 if it can display incompatible types and error 30 otherwise.

E031
`identifier does not denote a type`

An identifier denoting a type is expected at the indicated position.

E032
`scalar type expected`

The compiler expects a scalar type (real, integer, cardinal, range, enumeration, `CHAR`, or `BOOLEAN`).

E033
`ordinal type expected`

The compiler expects a value, variable, or type designator of ordinal type, i.e. CHAR, BOOLEAN, enumeration, integer, or cardinal type or a subrange of one of those types.

E034
`invalid combination of parameters in type conversion`

According to the language definition this combination of parameters in a call of the standard procedure VAL is not valid.

E035
`NEW: "%s" not known in this scope` (ALLOCATE or DYNALLOCATE)

A call to NEW is treated as a call to ALLOCATE (or DYNALLOCATE for open arrays). The required procedure is not visible in this scope. It must be either imported or implemented.

Note: In XDS, the default memory managemet routines may be enabled by setting the **STORAGE** option ON.

E036
`DISPOSE: "%s" not known in this scope` (DEALLOCATE or DYNDEALLOCATE)

A call to DISPOSE is treated as a call to DEALLOCATE (or DYNDEALLOCATE for open arrays). The required procedure is not visible in this scope. It must be either imported or implemented.

Note: In xds, the default memory managemet routines may be enabled by setting the **STORAGE** option ON.

E037
`procedure "%s" should be a proper procedure` (procedure name)

In Modula-2, calls of NEW and DISPOSE are substituted by calls of ALLOCATE and DEALLOCATE (for dynamic arrays by calls of DYNALLOCATE and DYNDEALLOCATE). The error is reported if one of those procedures is declared as a function.

E038
`illegal number of parameters "%s"` (procedure name)

In Modula-2, calls of NEW and DISPOSE are substituted by calls of ALLOCATE and DEALLOCATE (for dynamic arrays by calls of DYNALLOCATE and DYNDEALLOCATE). The error is reported if a number of parameters in the declaration of a substitution procedure is wrong.

E039
`procedure "%s": %s parameter expected for "%s"` (procedure name,"VAR" or "value",parameter name)

In Modula-2, calls of `NEW` and `DISPOSE` are substituted by calls of `ALLOCATE` and `DEALLOCATE` (for dynamic arrays by calls of `DYNALLOCATE` and `DYNDEALLOCATE`). The error is reported if the kind (variable or value) of the given parameter in the declaration of a substitution procedure is wrong.

E040
`procedure "%s": type of parameter "%s" mismatch`

In Modula-2, calls of `NEW` and `DISPOSE` are substituted by calls of `ALLOCATE` and `DEALLOCATE` (for dynamic arrays by calls of `DYNALLOCATE` and `DYNDEALLOCATE`). The error is reported if a type of the given parameter in the declaration of a substitution procedure is wrong.

E041
`guard or test type is not an extension of variable type`

In an Oberon-2 type test (`v IS T`) or type quard (`v(T)`), `T` should be an extension of the static type of `v`.

E043
`illegal result type of procedure`

A type cannot be a result type of a function procedure (language or implementation restriction).

E044
`incompatible result types`

A result type of a procedure does not match those of a forward definition or definition of an overriden method.

E046
`illegal usage of open array type`

Open arrays (`ARRAY OF`) usage is restricted to pointer base types, element types of open array types, and formal parameter types.

E047
`fewer actual than formal parameters`

The number of actual parameters in a procedure call is less than the number of formal parameters.

E048

`more actual than formal parameters`

The number of actual parameters in a procedure call is greater than the number of formal parameters.

E049
`sequence parameter should be of SYSTEM.BYTE or SYSTEM.LOC type`

The only valid types of a sequence parameter are `SYSTEM.BYTE` and `SYSTEM.LOC`.

E050
`object is not array`
E051
`object is not record`
E052
`object is not pointer`
E053
`object is not set`

The compiler expects an object of the given type at the indicated position.

E054
`object is not variable`

The compiler expects a variable (designator) at the indicated position.

E055
`object is not procedure:` `%s` (procedure name)

The compiler expects a procedure designator at the indicated position.

E057
`a call of super method is valid in method redifinition only`

A call of a super method (type-bound procedure bound to a base type) is valid only in a redifinition of that method:

```
PROCEDURE (p: P) Foo;
BEGIN
  p.Foo^
END Foo.
```

E058
`type-bound procedure is not defined for the base type`

In a call of a super method (type-bound procedure bound to a base type) `p.Foo^` either `Foo` is not defined for a base type of `p` or there is no base type.

E059
`object is neither a pointer nor a VAR-parameter record`

The Oberon-2 compiler reports this error in the following cases:

- in a type test `v IS T` or type guard `v(T)`, `v` should be a designator denoting either pointer or variable parameter of a record type; `T` should be a record or pointer type
- in a declaration of type-bound procedure a receiver may be either a variable parameter of a record type or a value parameter of a pointer type.

E060
`pointer not bound to record or array type`

In Oberon-2, a pointer base type must be an array or record type. For instance, the declaration `TYPE P = POINTER TO INTEGER` is invalid.

E061
`dimension too large or negative`

The second parameter of the LEN function is either negative or larger than the maximum dimension of the given array.

E062
`pointer not bound to record`

The Oberon-2 compiler reports this error in the following cases:

- in a type test `v IS T` or type guard `v(T)`, if `v` is a pointer it should be a pointer to record
- in a type-bound procedure declaration, if a receiver is a pointer, it should be a pointer to record

E064
`base type of open array aggregate should be a simple type`

A base type of an open array aggregate (`ARRAY OF T{}`) cannot be a record or array type.

E065
`the record type is from another module`

A procedure bound to a record type should be declared in the same module as the record type.

E067
`receiver type should be exported %s` (name of type)

A receiver type for an exported type-bound procedure should also be exported.

E068
`this type-bound procedure cannot be called from a record`

The receiver parameter of this type-bound procedure is of a pointer type, hence it cannot be called from a designator of a record type. Note that if a receiver parameter is of a record type, such type-bound procedure can be called from a designator of a pointer type as well.

E069
`wrong mode of receiver type`

A mode of receiver type in a type-bound procedure redefinition does not match the previous definition.

E071
`non-Oberon type cannot be used in specific Oberon-2 construct`

A (object of) non-Oberon type (imported from a non-Oberon module or declared with direct language specification) cannot be used in specific Oberon-2 constructs (type-bound procedures, type guards, etc).

E072
`illegal order of redefinition of type-bound procedures`

A type-bound procedure for an extended type is defined before a type-bound procedure with the same name for a base type.

E074
`redefined type-bound procedure should be exported`

A redefined type-bound procedure should be exported if both its receiver type and redefining procedure are exported.

E075
`function procedure without RETURN statement`

A function procedure has no `RETURN` statement and so cannot return a result.

E076

`value is required`

The compiler expects an expression at the indicated position.

E078
`SIZE (TSIZE) cannot be applied to an open array`

Standard functions `SIZE` and `TSIZE` cannot be used to evaluate size of an open array designator or type in the standard mode. If language extensions are enabled, the compiler allows to apply `SIZE` to an open array designator, but not type.

E087
`expression should be constant`

The compiler cannot evaluate this expression at compile time. It should be constant according to the language definition.

E088
`identifier does not match block name`

An identifier at the end of a procedure or module does not match the one in the procedure or module header. The error may occur as a result of incorrect pairing of `END`s with headers.

E089
`procedure not implemented "%s"`

An exported procedure or forward procedure is not declared. This error often occurs due to comment misplacement.

E090
`proper procedure is expected`

A function is called as a proper procedure. It must be called in an expression. A function result can be ignored for procedures defined as `"C"`, `"Pascal"`, `"StdCall"` or `"SysCall"` only. See 10.2.

E091
`call of proper procedure in expression`

A proper procedure is called in an expression.

E092
`code procedure is not allowed in definition module`

E093
`not allowed in definition module`

The error is reported for a language feature that can not be used in definition module, including:

- local modules
- elaboration of an opaque type
- forward declaration
- procedure or module body
- read-only parameters

E094
`allowed only in definition module`

The error is reported for a language feature that can be used in definition module only, i.e. read-only variables and record fields (extended Modula-2).

E095
`allowed only in global scope`

The error is reported for a language feature that can be used only in the global module scope, including:

- elaboration of an opaque type (Modula-2)
- export marks (Oberon-2)
- type-bound procedure definition (Oberon-2)

E096
`unsatisfied opaque type "%s"`

An opaque type declared in a definition module must be elaborated in the implementation module.

E097
`unsatisfied forward type "%s"`

A type `T` can be introduced in a declaration of a pointer type as in:

```
TYPE Foo = POINTER TO T;
```

This type `T` must then be declared at the same scope.

E098
`allowed only for value parameter`

The error is reported for a language feature that can be applied to value parameter only (not to `VAR` parameters), such as a read-only parameter mark (see 7.6.8).

E099
`RETURN allowed only in procedure body`

In Oberon-2, the `RETURN` statement is not allowed in a module body.

E100
`illegal order of declarations`

In Oberon-2. all constants, types and variables in one declaration sequence must be declared before any procedure declaration.

E102
`language extension is not allowed %s` (specification)

The error is reported for a language feature that can be used only if language extensions are switched on. See options **M2EXTENSIONS** and **O2EXTENSIONS**.

E107
`shall not have a value less than 0`

The error reported if a value of a (constant) expression cannot be negative, including:

- second operand of `DIV` and `MOD`
- repetition count in an array constructor (`expr BY count`)

E109
`forward type cannot be opaque`

A forward type `T` (declared as `TYPE Foo = POINTER TO T`) cannot be elaborated as an opaque type, i.e. declared as `TYPE T = <opaque type>`).

E110
`illegal length, %d was expected` (expected number of elements)

Wrong number of elements in an array constructor.

E111
`repetition counter must be an expression of a whole number type`

A repetition counter in an array constructor must be of a whole number type.

E112
`expression for field "%s" was expected` (field name)

The error is reported if a record constructor does not contain an expression for the given field.

E113
`no variant is associated with the value of the expression`

The error is reported if a record constructor for a record type with variant part does not have a variant for the given value of a record tag and the `ELSE` clause is omitted.

E114
`cannot declare type-bound procedure: "%s" is declared as a field`

A type-bound procedure has the same name as a field already declared in that type or one of its base types.

E116
`field "%s" is not exported` (field name)

The given field is not exported, put export mark into the declaration of the record type.

E118
`base type is not allowed for non-Oberon record`

A record type can be defined as an extension of another type, only if it is an Oberon-2 record type.

E119
`variant fields are not allowed in Oberon record`

A record with variant parts cannot be declared as an Oberon-2 record.

E120
`field of Oberon type is not allowed in non-Oberon record`

This is considered an error because garbage collector does not trace non-Oberon records and reference to an object may be lost.

E121
`illegal use of type designator "%s"`

A type designator cannot be used in a statement position.

E122
`expression out of bounds`

A value which can be checked at compile-time is out of range.

E123

`designator is read-only`

A designator marked as read-only cannot be used in a position where its value may be changed.

E124
`low bound greater than high bound`

A lower bound of a range is greater than high bound.

E125
`EXIT not within LOOP statement`

An `EXIT` statement specifies termination of the enclosing `LOOP` statement. This `EXIT` is not within any `LOOP`.

E126
`case label defined more then once`

In a `CASE` statement all labels must have different values. The label at the indicated position is alfeady used in this `CASE` statement.

E128
`FOR-loop control variable must be declared in the local scope`

A control variable of a `FOR` loop must be declared locally in the procedure or module which body contains the loop.

E129
`more expressions than fields in a record type`

In a record constructor there are more expressions than there are fields in the record type (or in this variant of a variant record type).

E131
`zero step in FOR statement`

In a `FOR` statement, the step cannot be equal to zero.

E132
`shall be an open array designator`

If language extensions are OFF, the standard procedure `HIGH` can be applied to open arrays only, otherwise to any array designator.

E133
`implementation limit exceeded for set base type (length > %d)`

The compiler restricts length of a base type of set (`MAX(base)-MIN(base)+1`). Note, that the limit does not depend on the low bound, so the following set types are valid:

```
SET OF [-256..-5]
SET OF [MAX(INTEGER)-512..MAX(INTEGER)]
```

E134
`must be value of unsigned type`

The compiler expects a parameter of this standard procedure to be a value of an unsigned type.

E135
`must be value of pointer type`

The compiler expects a parameter of this standard procedure to be a value of a pointer type. **Note:** the `SYSTEM.ADDRESS` type is defined as `POINTER TO LOC`.

E136
`must be type designator`

The compiler expects a parameter of this standard procedure to be a type designator.

E137
`numeric constant does not have a defined storage size`

The compiler must know the size of a value in the given context. A numeric constant cannot be used at the indicated position.

E139
`must be (qualified) identifier which denotes variable`

The ISO standard requires an ”entire designator” in this context, e.g. as a parameter of the `SIZE` function. It may be either a variable (which may be a formal parameter) or a field of a record variable within a `WITH` statement that applies to that variable.

E140
`interrupt procedures are not implemented yet`

Oberon compilers from ETH implements so-called interrupt procedures, marked by the symbol ”+”.

```
PROCEDURE + Foo;
```

In XDS, this feature is not implemented.

E141
`opaque type can not be defined as Oberon pointer`

A Modula-2 opaque type cannot be elaborated as an Oberon-2 pointer. See Chapter 10.

E143
`not allowed in Oberon`

The compiler reports this error for language features that are vaild in Modula-2 but not in Oberon-2, including:

- enumeration types
- range types
- local modules

E144
`pointer and record types are mixed in type test`

In an Oberon-2 type test `v IS T` or a type guard `v(T)`, both `v` and `T` must be either pointers or records.

E145
`control variable must not be a formal parameter`

According to ISO Modula-2, a control variable in a `FOR` statement cannot be a formal parameter (either `VAR` or value).

E146
`control variable cannot be exported`

A variable used as a control variable in a `FOR` statement or an Oberon-2 `WITH` statement cannot be exported.

E147
`control variable cannot be threatened`

A control variable of a `FOR` statement or an Oberon-2 `WITH` statement has been threatened inside the body of the statement, or in a nested procedure called from the body. Threatening actions include assignment and passing as a `VAR` parameter to a user-defined or standard procedure (`ADR`, `INC`, `DEC`, etc). The compiler also reports the error 158 to indicate the exact place of threatening.

E148
`finalization is allowed only in module block`

A procedure body can not contain a finalization part.

E149
`RETRY is allowed only in exceptional part of block`

This `RETRY` statement is outside an exceptional part of a block.

E150
`wrong value of direct language specification`

A value must be either one of the strings (`"Modula"`, `"Oberon"`, `"C"`,`"Pascal"`, `"SysCall"`, or `"StdCall"`) or the corresponding integer value. We recommend to use strings, integer values are preserved for backward compatibility.

E151
`must be of integer type`

The compiler expects a variable of an integer type.

E152
`incompatible calling conventions: "%s" "%s"`
E153
`incompatible calling conventions`

Two procedure types have different calling conventions. The error can reported in the following cases:

- a procedure is assigned to a procedure variable
- a procedure is passed as a parameter
- two procedure values are compared

The compiler reports error 152 if it can show incompatible types and error 153 otherwise.

E154
`procedure "%s" does not match previous definition: was: %s now: %s` (procedure name,proctype,proctype)
E155
`procedure "%s" does not match previous definition` (procedure name)

A procedure heading must have the same number of parameters, the same parameter modes (variable or value) and the same types as in the previous declaration. A previous declaration may be one of the following:

- procedure declaration in a definition module

- forward procedure declaration
- type-bound procedure declaration in a base type

The compiler reports error 154 if it can show incompatible types and error 155 otherwise.

E156
`procedure designator is expected`

A designator which appears to be called (e.g. `Foo(...)`) does not denote a procedure.

E158
`control variable "%s" is threatened here` (variable name)

A control variable of a `FOR` statement or an Oberon-2`WITH` statement is threatened at the indicated position. It means that its value may be changed. See also error 147.

E159
`type of aggregate is not set or array or record`

An object which appears to be an aggregate (e.g. `Foo{...}`) begins with an identifier which is not a set, record, or array type.

E160
`invalid parameter specification:  expected NIL`

Only one special kind of variable parameter is implemented: `VAR [NIL]`. It means that `NIL` may be passed to this parameter.

E161
`VAR [NIL] parameter expected`

A parameter of the `SYSTEM.VALID` function must be a `VAR [NIL]` parameter.

E162
`%s value should be in %{} (not "%s")` (equation,set of valid values,new value)

This error is reported for a wrong setting of `ALIGNMENT`, `ENUMSIZE`, or `SETSIZE` equation.

E163
`control variable cannot not be volatile`

A control variable of a `FOR` statement cannot be marked as volatile. See the **VOLATILE** option.

E200
`not yet implemented`

This language feature is not implemented yet.

E201
`real overflow or underflow in constant expression`

This error is to be reported if a real overflow (underflow) occurs during evaluation of a constant expression.

E202
`integer overflow in constant expression`

The compiler uses 64-bits (signed) arithmetics for whole numbers. The error is reported if an overflow occurs during evaluation of a constant expression. In the following example, an error will be reported for the assignment statement, while constant definition is valid.

```
MODULE Test;

CONST
  VeryBigConstant = MAX(CARDINAL)*2;            (* OK *)
  TooBigConstant = VeryBigConstant*VeryBigConstant;   (* OK *)

END Test.
```

E203
`division by zero`

The second operand of a `DIV`, `MOD`, `REM`, or `"/"` operator is zero.

E206
`array length is too large or less then zero`

The array length is either negative or exceeds implementation limit.

E208
`CASE statement always fails`

The error is reported if a case select expression can be evaluated at compile-time and there is no variant corresponding to its value, and the `ELSE` clause is omitted. If not constantly evaluated, this `CASE` statement would cause the `caseSelectException` exception at run-time.

E219
`too many nested open array types (implementation limit`

`%d)` (implementation limit)

The compiler (more precisely, run-time support) puts a limit on the number of nested open array types (or dimensions). Note, that there is no limit for arrays with specified length, because such arrays do not require special support in run-time system.

E220
`heirarchy of record extensions too high (implementation limit %d)` (implementation limit)

The run-time system puts a limit on the level of record extensions. It is required for efficient implementaion of type tests and type guards.

E221
`procedure declaration nesting limit (%d) has been exceeded` (implementation limit)

The compiler puts a limit on the number of procedures nested inside each other. When modules are nested inside procedures, only the level of procedure declarations is counted.

E281
`type-bound procedure is not valid as procedure value`

A type-bound procedure cannot be assigned to a variable of procedure type.

E282
`local procedure is not valid as procedure value "%s"` (procedure name)

A procedure local to another one cannot be assigned to a variable of procedure type.

E283
`code (or external) procedure is not valid as procedure value`

A code procedure and external procedure cannot be assigned to a variable of procedure type.

6.4 Symbol files read/write errors

F190
`incorrect header in symbol file "%s"` (module name)

A symbol file for the given module is corrupted. Recompile it.

F191
`incorrect version of symbol file "%s" (%d instead of %d)` (module name, symfile version, current version)

The given symbol file is generated by a different version of the compiler. Recompile the respecitve source or use compatible versions of the compiler and/or symbol file.

F192
`key inconsistency of imported module "%s"` (module name)

The error occurs if an interface of some module is changed but not all its clients (modules that imports from it) were recompiled. For example, let `A` imports from `B` and `M`; `B` in turn imports from `M`:

```
DEFINITION MODULE M;    DEFINITION MODULE B;      MODULE A;
                        IMPORT M;                 IMPORT M,B;
END M.                  END B.                    END A.
```

Let us recompile `M.def`, `B.def` and then `M.def` again. The error will be reported when compiling `A.mod`, because version keys of module `M` imported through `B` is not equal to the version key of `M` imported directly.

To fix the problem modules must be compiled in appropriate order. We recommend to use the XDS compiler make facility, i.e. to compile your program in the MAKE (see 4.2.2) or PROJECT (see 4.2.3) operation mode. If you always use the make facility this error will never be reported.

F193
`generation of new symbol file not allowed`

The Oberon-2 compiler creates a temporary symbol file every time a module is compiled, compares that symbol file with the existing one and overwrites it with the new one if necessary. When the **CHANGESYM** option is OFF (by default), the compiler reports an error if the symbol file (and hence the module interface) had been changed and does not replace the old symbol file.

Note: if the **M2CMPSYM** option is set ON, the same applies to compilation of a Modula-2 definition module, i.e., the **CHANGESYM** option should be set if the module interface has been changed.

F194
`module name does not match symbol file name "%s"` (module name)

A module name used in an `IMPORT` clause must be equal to the actual name of the module, written in the module heading.

F195
`cannot read symbol file "%s" generated by %s` (module name, compiler name)

The symbol file for the given module is generated by another XDS compiler. Native code compilers can read symbol files generated by **XDS-C** on the same platform, but not vice versa.

6.5 Internal errors

This section lists internal compiler errors. In some cases such a error may occur as a result of inadequate recovery from previous errors in your source text. In any case we recommend to provide us with a bug report, including:

- version of the compiler
- description of your environment (OS, CPU)
- minimal source text reproducing the error

F103
`INTERNAL ERROR(ME): value expected`
F104
`INTERNAL ERROR(ME): designator expected`
F105
`INTERNAL ERROR(ME): statement expected`
F106
`INTERNAL ERROR(ME): node type = NIL`
F142
`INTERNAL ERROR(ME): can not generate code`

F196
`INTERNAL ERROR: incorrect sym ident %d while reading symbol file "%s"`
F197
`INTERNAL ASSERT(%d) while reading symbol file "%s"`

6.6 Warnings

In many cases a warning may help you to find a bug or a serious drawback in your source text. We recommend not to switch warnings off and carefully check all of them. In many cases warnings have helped us to find and fix bugs very quickly (note that XDS compilers are written in XDS Oberon-2 and Modula-2).

Warnings described in this section are reported by both **XDS-C** and **Native XDS**. Each of these products may report additional warnings. Native XDS compilers fulfil more accurate analysis of the source code and report more warnings.

W300
`variable declared but never used`

This variable is of no use, it is not exported, assigned, passed as a parameter, or used in an expression. The compiler will not allocate space for it.

W301
`parameter is never used`

This parameter is not used in the procedure.

W302
`value was assigned but never used`

The current version of the compiler does not report this warning.

W303
`procedure declared but never used`

This procedure is not exported, called or assigned. The compiler will not generate it.

W304
`possibly used before definition "%s"` (variable name)

This warning is reported if a value of the variable may be undefined at the indicated position. Note, that it is just a warning. The compiler may be mistaken in complex contexts. In the following example, `"y"` will be assigned at the first iteration, however, the compiler will report a warning, because it does not trace execution of the `FOR` statement.

```
PROCEDURE Foo;
  VAR x,y: INTEGER;
BEGIN
  FOR x:=0 TO 2 DO
    IF x = 0 THEN y:=1
```

```
      ELSE INC(y)  (* warning is reported here *)
      END;
    END;
END Foo;
```

This warning is not reported for global variables.

W305
`constant declared but never used`

The current version of the compiler does not report this warning.

W310
`infinite loop`

Execution of this loop (`LOOP`, `WHILE` or `REPEAT`) will not terminate normally. It means that statements after the loop will never be executed and the compiler will not generate them. Check that the loop was intentionally made infinite.

W311
`unreachable code`

This code cannot be executed and the compiler will not generate it (dead code elimination). It may be statements after a `RETURN`, `ASSERT(FALSE)`, `HALT`, infinite loop, statements under constant `FALSE` condition (`IF FALSE THEN`), etc.

W312
`loop is executed exactly once`

It may be a loop like

```
FOR i:=1 TO 1 DO ... END;
```

or

```
LOOP ...; EXIT END;
```

Check that you wrote it intentionally.

W314
`variable "%s" has compile time defined value here`

The compiler was able to determine the run-time value of the given variable (due to constant propagation) and will use it instead of accessing the variable. For the following example

```
i:=5; IF i = 5 THEN S END;
```

the compiler will generate:

```
i:=5; S;
```

This warning is not reported for global variables.

W315
`NIL dereference`

The compiler knows that a value of a pointer variable is NIL (due to constant propagation), e.g:

```
p:=NIL;
pˆ.field:=1;
```

The code will be generated and will cause ”invalidLocation” exception at run-time.

This warning is not reported for global variables.

W316
`this SYSTEM procedure is not described in Modula-2 ISO standard`

This warning is reported in order to simplify porting your program to other Modula-2 compilers.

W317
`VAR parameter is used here, check that it is not threatened inside WITH`

A variable parameter of a pointer type is used as a control variable in an Oberon-2 `WITH` statement. The compiler cannot check that it is not changed inside `WITH`. In the the following example `"ptr"` and, hence, `"p"` becomes `NIL` inside `WITH`:

```
VAR ptr: P;

PROCEDURE proc(VAR p: P);
BEGIN
  WITH p: P1 DO
    ptr:=NIL;
    p.i:=1;
  END;
END proc;

BEGIN
  proc(ptr);
END
```

We recommend to avoid using variable parameters of pointer types in `WITH` statements.

W318
`redundant FOR statement`

The `FOR` statement is redundant (and not generated) if its low and high bounds can be evaluted at compile-time and it would be executed zero times, or if its body is empty.

6.7 Pragma warnings

W320
`undeclared option "%s"`

An undeclared option is used. Its value is assumed to be `FALSE`.

W322
`undeclared equation "%s"`

An undeclared equation is used. Its value is undefined.

W321
`option "%s" is already defined`
W323
`equation "%s" is already defined`

The option (equation) is already defined, second declaration is ignored.

W390
`obsolete pragma setting`

The syntax used is obsolete. The next release of the compiler will not understand it. We recommend to rewrite the clause using the new syntax.

6.8 Native XDS warnings

W900
`redundant code eliminated`

This warning is reported if a program fragment does not influence to the program execution, e.g:

```
i:=1;
```

```
i:=2;
```

The first assignemnt is redundant and will be deleted.

W901
`redundant code not eliminated - can raise exception`

The same as W900, but the redundant code is preserved because it can raise an exception, e.g.:

```
i:=a DIV b; (* raises exception if b <= 0 *)
i:=2;
```

W902
`constant condition eliminated`

The warning is reported if a boolean condition can be evaluated at run-time, e.g.

```
IF (i=1) & (i=1) THEN (* the second condition is TRUE *)
```

or

```
j:=2;
IF (i=1) OR (j#2) THEN (* the second condition is FALSE *)
```

W903
`function result is not used`

The compiler ignores function result, like in:

```
IF Foo() THEN END;
```

W910
`realValueException will be raised here`
W911
`wholeValueException will be raised here`
W912
`wholeDivException will be raised here`
W913
`indexException will be raised here`
W914
`rangeException will be raised here`
W915
`invalidLocation exception will be raised here`

A warning from this group is reported if the compiler determines that the exception will be raised in the code corresponding to this program fragment. In this case the fragment is omitted and the compiler generates a call of a run-time procedure which will raise this exception.

6.9 Native XDS errors

This section describes errors reported by a native code generator (back-end). The code generator is invoked only if no errors were found by a language parser.

F950
`out of memory`

The compiler cannot generate your module. Try to increase **COMPILERHEAP** or try to compile this module separately (not in the MAKE (see 4.2.2) or PROJECT (see 4.2.3) mode). Almost any module may be compiled if **COMPILERHEAP** is set to 16MB. Exceptions are very big modules or modules containing large procedures (more than 500 lines). Note that the amount of memory required for the code generator depends mostly on sizes of procedures, not of the module.

F951
`expression(s) too complex`

The compiler cannot generate this expression, it is too complex. Simplify the expression.

F952
`that type conversion is not implemented`

The compiler cannot generate this type conversion.

6.10 XDS-C warnings

W350
`non portable type cast:  size is undefined`

The compiler have to generate a type cast which may be unportable. Check that the generated code is correct or pay some attention to your C compiler warnings.

W351
`option NOHEADER is allowed only in C-modules`
W352
`option NOCODE is allowed only in C-modules`

Options **NOHEADER** and **NOCODE** have meaning only for modules defined as `"C"`, `"StdCall"` or `"SysCall"`. See 10.2

W353
`dependence cycle in C code`

The generated code contains a dependance cycle. It means that some declaration A depends on B and vice versa. It is not an error. The generated code may be valid.

Chapter 7

XDS Modula-2

This chapter covers details of the XDS implementation of the Modula-2 language. In the standard mode[1] XDS Modula-2 complies with ISO 10514 (See the statement of compliance and further details in 7.1). The compatibility rules are described in 7.4. The differences between ISO Modula-2 and the language described in the 4th edition of Wirth's "Programming in Modula-2" [PIM]are listed in 7.2. Language extensions are described in 7.6.

7.1 ISO Standard compliance

XDS Modula-2 partially complies with the requirements of ISO 10514. The details of non-conformities are as follows:

- Not all libraries are available in the current release.
- The current release may impose some restrictions on using new language features.

See Chapter A for further details.

7.1.1 Ordering of declarations

XDS Modula-2 is a so-called 'single-pass' implementation. It means that all identifiers must be declared before use. According to the International Standard this *declare-before-use* approach is perfectly valid. The alternative approach,

[1] When options **M2EXTENSIONS** and **M2ADDTYPES** are OFF

(*declare-before-use-in-declarations*), can be used in so-called ‘multi-pass’ implementations.

A forward declaration must be used to allow forward references to a procedure which actual declaration appears later in the source text.

Example

```
PROCEDURE a(x: INTEGER); FORWARD;
(* FORWARD declaration *)

PROCEDURE b(x: INTEGER);
BEGIN
  a(x-1);
END b;

PROCEDURE a(n: INTEGER);
(* proper procedure declaration *)
BEGIN
  b(n-1);
END a;
```

To provide source compatibility between ‘single-pass’ and ‘multi-pass’ implementations, the Standard requires that all conforming ‘multi-pass’ implementations accept and correctly process the `FORWARD` directive.

7.2 New language’s features

The language described in the International Standard varies in many details from the one described in Wirth’s “Programming in Modula-2”[PIM].

The most important innovations are

- complex numbers
- module finalization
- exception handling
- array and record constructors

- four new system modules
- standard library

Note: The system modules (except the module `SYSTEM`) are not embedded in the compiler and are implemented as separate modules.

7.2.1 Lexis

The ISO Modula-2 has some new keywords (table 7.1, page 93) and pervasive identifiers (table 7.2, page 94), and provides alternatives for some symbols (table 7.3, page 94). It also introduces the syntax for source code directives (or pragmas):

```
Pragma = "<*" pragma_body "*>"
```

The Standard does not specify a syntax of `pragma_body`. In XDS, source code directives are used for in-line option setting and for conditional compilation. See 7.7.1 for further details.

AND	ARRAY	BEGIN
BY	CASE	CONST
DEFINITION	DIV	DO
ELSE	ELSIF	END
EXIT	EXCEPT (see 7.2.13)	EXPORT
FINALLY (see 7.2.12)	FOR	FORWARD (see 7.1.1)
FROM	IF	IMPLEMENTATION
IMPORT	IN	LOOP
MOD	MODULE	NOT
OF	OR	PACKEDSET (see 7.2.3)
POINTER	PROCEDURE	QUALIFIED
RECORD	REM (see 7.2.9)	RETRY (see 7.2.13)
REPEAT	RETURN	SET
THEN	TO	TYPE
UNTIL	VAR	WHILE
WITH		

Table 7.1: Modula-2 keywords

7.2.2 Complex types

Types `COMPLEX` and `LONGCOMPLEX` can be used to represent complex numbers. These types differ in a the range and precision. The `COMPLEX` type is defined as

ABS	BITSET
BOOLEAN	CARDINAL
CAP	CHR
CHAR	COMPLEX (7.2.2)
CMPLX (7.2.2)	DEC
DISPOSE	EXCL
FALSE	FLOAT
HALT	HIGH
IM (7.2.2)	INC
INCL	INT (7.2.10)
INTERRUPTIBLE (7.2.18)	INTEGER
LENGTH (7.2.4)	LFLOAT (7.2.10)
LONGCOMPLEX (7.2.2)	LONGREAL
MAX	MIN
NEW	NIL
ODD	ORD
PROC	PROTECTION (7.2.18)
RE (7.2.2)	REAL
SIZE	TRUE
TRUNC	UNINTERRUPTIBLE (7.2.18)
VAL	

Table 7.2: Modula-2 pervasive identifiers

Symbol	Meaning	Alternative
[	left bracket	(!
]	right bracket	!)
{	left brace	(:
}	right brace	:)
|	case separator	!
^	dereference	@

Table 7.3: Modula-2 alternative symbols

a (REAL,REAL) pair, while LONGCOMPLEX consists of a pair of LONGREAL values.

There is no notation for a complex literal. A complex value can be obtained by applying the standard function CMPLX to two reals. If both CMPLX arguments are real constants the result is the complex constant.

```
CONST i = CMPLX(0.0,1.0);
```

If both expressions are of the REAL type, or if one is of the REAL type and the other is a real constant, the function returns a COMPLEX value. If both expressions are of the LONGREAL type, or if one is of the LONGREAL type and the other is a real constant the function returns a LONGCOMPLEX value. The following table summarizes the permitted types and the result type:

	REAL	LONGREAL	real constant
REAL	REAL	error	COMPLEX
LONGREAL	error	LONGCOMPLEX	LONGCOMPLEX
real constant	COMPLEX	LONGCOMPLEX	complex constant

Standard functions RE and IM can be used to obtain a real or imaginary part of a value of a complex type. Both functions have one formal parameter. If the actual parameter is of the COMPLEX type, both functions return a REAL value; if the parameter is of the LONGCOMPLEX type, functions return a LONGREAL value; otherwise the parameter should be a complex constant and functions return a real constant.

```
CONST one = IM(CMPLX(0.0,1.0));
```

There are four arithmetic binary operators for operands of a complex type: addition (+), subtraction (-), multiplication (*), and division (/). The following table indicates the result of an operation for permitted combinations:

	COMPLEX	LONGCOMPLEX	complex constant
COMPLEX	COMPLEX	error	COMPLEX
LONGCOMPLEX	error	LONGCOMPLEX	LONGCOMPLEX
complex constant	COMPLEX	LONGCOMPLEX	complex constant

There are two arithmetic unary operators that can be applied to the values of a complex type: identity (+) and negation (-). The result is of the operand's type.

Two complex comparison operators are provided for operands of complex type: equality (=) and inequality (<>).

Example

```
PROCEDURE abs(z: COMPLEX): REAL;
BEGIN
  RETURN RealMath.sqrt(RE(z)*RE(z)+IM(z)*IM(z))
END abs;
```

7.2.3 Sets and packedsets

A set or packedset[2] type defines a new elementary type whose set of values is the power set of an associated ordinal type called the *base type* of the set or packedset type.

```
SetType       = SET OF Type;
PackedsetType = PACKEDSET OF Type;
```

The International Standard does not require a specific representation for set types. Packedset types representation has to be mapped to the individual bits of a particular underlying architecture. The standard type `BITSET` is a predefined packedset type.

The current XDS implementation does not distinguish between set and packedset types. A set of at least 256 elements can be defined.

All set operators, namely union (`+`), difference (`-`), intersection (`*`), and symmetrical difference (`/`), can be applied to the values of both set and packedset types.

```
TYPE
  CharSet = SET OF CHAR;
  ByteSet = PACKEDSET OF [-127..128];

VAR
  letters, digits, alphanum: CharSet;
  neg, pos, zero : ByteSet;
 .  .  .
letters  := CharSet{'a'..'z','A'..'Z'};
digits   := CharSet{'0'..'9'};
alphanum := letters + digits;
```

[2] Packedset types are innovated in the Standard.

```
neg := ByteSet{-127..-1}; pos := ByteSet{1..127};
zero := ByteSet{-127..128}-neg-pos;
```

7.2.4 Strings

For operands of the string literal type, the string concatenation operation is defined, denoted by the symbol `"+"`. **Note:** a character number literal (e.g. `15C`) denotes a value of a literal string type of length 1. The empty string is compatible with the type `CHAR` and has a value equal to the string terminator (`0C`).

```
CONST
  CR = 15C;
  LF = 12C;
  LineEnd = CR + LF;
  Greeting = "hello " + "world" + LineEnd;
```

The new standard function `LENGTH` can be used to obtain the length of a string value.

```
PROCEDURE LENGTH(s: ARRAY OF CHAR): CARDINAL;
```

7.2.5 Value constructors

A value constructor is an expression denoting a value of an array type, a record type, or a set type. In case of array constructors and record constructors a list of values, known as *structure components*, is specified to define the values of components of an array value or the fields of a record value. In case of a set constructor, a list of members is specified, whose elements define the elements of the set value.

```
ValueConstructor = ArrayValue
                 | RecordValue
                 | SetValue.
ArrayValue  = TypeIdentifier "{"
              ArrayComponent { "," ArrayComponent }
              "}".
ArrayComponent = Component [ BY RepeatCount ].
Component   = Expression.
RepeatCount = ConstExpression.
```

```
RecordValue = TypeIdentifier "{"
              Component { "," Component }
              "}".
```

Set constructors are described in PIM.

The total number of components of an array constructor must be exactly the same as the number of array's elements (taking into account repetition factors). Each component must be assignment compatible with the array base type.

The number of components of a record constructor must be exactly the same as the number of fields. Each component must be an assignment compatible with the type of the field.

A special case is a record constructor for a record with variant parts. If the n-th field is the tag field the n-th component must be a constant expression. If there is no ELSE variant part associated with the tag field, then the variant associated with the value of expression should exist. If no variant is associated with the value, then the fields of the ELSE variant part should be included in the sequence of components.

The constructor's components may themselves contain lists of elements, and such nested constructs need not specify a type identifier. This relaxation is necessary for multi-dimensional arrays, where the types of the inner components may be anonymous.

Examples

```
TYPE
  String = ARRAY [0..15] OF CHAR;
  Person = RECORD
    name: String;
    age : CARDINAL;
  END;
  Vector = ARRAY [0..2] OF INTEGER;
  Matrix = ARRAY [0..2] OF Vector;

VAR
  string: String;
  person: Person;
  vector: Vector;
  matrix: Matrix;
```

```
   .  .  .
BEGIN
   .  .  .
  string:=String{" " BY 16};
  person:=Person{"Alex",32};
  vector:=Vector{1,2,3};
  matrix:=Matrix{vector,{4,5,6},Vector{7,8,9}};
  matrix:=Matrix{vector BY 3};
```

7.2.6 Multi-dimensional open arrays

According to the International Standard, parameters of a multi-dimensional open array type are allowed:

```
PROCEDURE Foo(VAR matrix: ARRAY OF ARRAY OF REAL);
  VAR i,j: CARDINAL;
BEGIN
  FOR i:=0 TO HIGH(matrix) DO
    FOR j:=0 TO HIGH(matrix[i]) DO
      ... matrix[i,j] ...
    END;
  END;
END Foo;

VAR a: ARRAY [0..2],[0..2] OF REAL;

BEGIN
  Foo(a);
END ...
```

7.2.7 Procedure type declarations

A procedure type identifier may be used in declaration of the type itself. This feature is used in the Standard Library. See, for example, modules `ConvTypes` and `WholeConv`.

```
TYPE
  Scan = PROCEDURE (CHAR; VAR Scan);
  Func = PROCEDURE (INTEGER): Func;
```

7.2.8 Procedure constants

A constant expression may contain values of procedure types, or structured values whose components are values of procedure types. Procedure constants may be used as a mechanism for procedure renaming. In a definition module it is possible to export a renamed version of the imported procedure.

Examples

```
TYPE ProcTable = ARRAY [0..3] OF PROC;

CONST
  WS = STextIO.WriteString;
  Table = ProcTable{Up,Down,Left,Right};
```

7.2.9 Whole number division

Along with `DIV` and `MOD` the International Standard includes two additional operators for whole number division: '`/`' and `REM`.

Operators `DIV` and `MOD` are defined for positive divisors only, while '`/`' and `REM` can be used for both negative and positive divisors.

The language exception `wholeDivException` (See 7.2.13) is raised if:

- the second operand is zero (for all four operators)
- the second operand of `DIV` or `MOD` is negative.

For the given `lval` and `rval`

```
quotient  := lval / rval;
remainder := lval REM rval;
```

the following is true (for all non-zero values of `rval`):

- `lval = rval * quotient + remainder`
- the value of `remainder` is either zero, or an integer of the same sign as `lval` and of a smaller absolute value than `rval`.

For the given `lval` and `rval`

```
quotient := lval DIV rval;
modulus  := lval MOD rval;
```

the following is true (for all positive values of `rval`):

- `lval = rval * quotient + modules`
- the value of `modulus` is a non-negative integer less than `rval`.

Operations are exemplified in the following table:

op	31 *op* 10	31 *op* (-10)	(-31) *op* 10	(-31) *op* (-10)
`/`	3	-3	-3	3
`REM`	1	1	-1	-1
`DIV`	3	exception	-4	exception
`MOD`	1	exception	9	exception

7.2.10 Type conversions

The language includes the following type conversion functions: `CHR`, `FLOAT`, `INT`, `LFLOAT`, `ORD`, `TRUNC` and `VAL`. The functions `INT` and `LFLOAT` are not described in PIM.

All the type conversion functions (except `VAL`) have a single parameter and can be expressed in terms of the `VAL` function.

Function	**Parameter**	**Equals to**
`CHR(x)`	whole	`VAL(CHAR,x)`
`FLOAT(x)`	real or whole	`VAL(REAL,x)`
`INT(x)`	real or ordinal	`VAL(INTEGER,x)`
`LFLOAT(x)`	real or whole	`VAL(LONGREAL,x)`
`ORD(x)`	ordinal	`VAL(CARDINAL,x)`
`TRUNC(x)`	real	`VAL(CARDINAL,x)`

The function `VAL` can be used to obtain a value of the specified scalar type from an expression of a scalar type. The function has two parameters. The first parameter should be a type parameter that denotes a scalar type. If the type is a subrange type, the result of `VAL` has the host type of the subrange type, otherwise it has the type denoted by the type parameter.

The second parameter should be an expression of a scalar type and at least one of the restriction shall hold:

- the result type and the type of the expression are identical
- both the result type and the type of the expression are whole or real
- the result type or the type of the expression is a whole type

In the following table, $\surd$ denotes a valid combination of types and $\times$ denotes an invalid combination:

the type of expression	the type denoted by the type parameter				
	whole	real	CHAR	BOOLEAN	enumeration
whole type	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
real type	$\surd$	$\surd$	$\times$	$\times$	$\times$
CHAR	$\surd$	$\times$	$\surd$	$\times$	$\times$
BOOLEAN	$\surd$	$\times$	$\times$	$\surd$	$\times$
enumeration	$\surd$	$\times$	$\times$	$\times$	$\surd$

An exception is raised if the value of `x` is outside the range of the type `T` in the call `VAL(T,x)`. If `x` is of a real type, the calls `VAL(INTEGER,x)` and `VAL(CARDINAL,x)` both truncate the value of `x`.

7.2.11 NEW and DISPOSE

The standard procedures `NEW` and `DISPOSE` are back in the language. Calls of `NEW` and `DISPOSE` are substituted by calls of `ALLOCATE` and `DEALLOCATE` which should be visible at the current scope. The compiler checks compatibility of these substitution procedures with the expected formal type:

```
PROCEDURE ALLOCATE(VAR a: ADDRESS; size: CARDINAL);
PROCEDURE DEALLOCATE(VAR a: ADDRESS; size: CARDINAL);
```

As a rule, the procedures `ALLOCATE` and `DEALLOCATE` declared in the module `Storage` are used. These procedures are made visible by including the import list:

```
FROM Storage IMPORT ALLOCATE, DEALLOCATE;
```

When language extensions are enabled, the procedures `NEW` and `DISPOSE` can be applied to dynamic arrays. See 7.6.12 for further details.

See also the **STORAGE** option.

7.2.12 Finalization

A special mechanism called *finalization* is provided to perform certain operations during program termination.

A module declaration contains an optional finalization body, which is executed during program termination for static modules (See 7.2.16) or dynamic module finalization.

```
ModuleBody = [ BEGIN BlockBody
               [ FINALLY BlockBody ] ] END
BlockBody  = NormalPart
               [ EXCEPT ExceptionalPart ].
NormalPart = StatementSequence.
ExceptionalPart = StatementSequence.
```

Note: the `RETURN` statement can be used in a `BlockBody`.

Consider the following example:

```
MODULE Test;

 .  .  .

VAR cid: StreamFile.ChanId;

BEGIN
  StreamFile.Open(cid,"tmp",flags,res);
  Process(cid);
FINALLY
  StreamFile.Close(cid);
END Test
```

If the `Test` module is declared in a procedure block, then the initialization body will be executed on a call of the procedure, while the finalization body is executed upon return from the procedure.

If the `Test` module is a static module, its finalization will be executed during program termination.

In any case, finalization bodies are executed in reverse order with respect to their initializations.

In the following example, finalization of a local module is used to restore context:

```
VAR state: State;

PROCEDURE Foo;

  MODULE AutoSave;
    IMPORT state, State;
    VAR save: State;
  BEGIN
    save:=state; (* save state *)
    state:=fooState;
  FINALLY
    state:=save; (* restore state *)
  END AutoSave;

BEGIN
  ... process ...
END Foo;
```

The initialization part of the AutoSave module will be executed before any statement in the Foo body and finalization part will be executed directly before returning from a call of Foo.

7.2.13 Exceptions

An exception handling mechanism is now included in the language. Both user-defined exceptions and language exceptions can be handled. There is no special exception type; an exception is identified by a pair: exception source value and cardinal value. Two keywords (EXCEPT and RETRY) are added to the language. The essential part of exception handling is provided in two system modules: EXCEPTIONS and M2EXCEPTION.

The EXCEPTIONS module provides facilities for raising and identifying the user-defined exceptions, for reporting their occurrence, and for making enquiries concerning the execution state of the current coroutine.

The M2EXCEPTION module provides facilities for identifying language exceptions that have been raised.

A procedure body, an initialization or finalization part of a module body may contain an exceptional part.

```
BlockBody  = NormalPart [ EXCEPT ExceptionalPart ].
```

```
NormalPart = StatementSequence.
ExceptionalPart = StatementSequence.
```

Example:

```
PROCEDURE Div(a,b: INTEGER): INTEGER;
BEGIN
  RETURN a DIV b      (* try to divide *)
EXCEPT
  RETURN MAX(INTEGER) (* if exception *)
END Fly;
```

When an exception is raised (explicitly or implicitly) the 'nearest' (in terms of procedure calls) exceptional part in the current coroutine receives control. Each coroutine is executed initially in the normal state. If an exception is raised, the coroutine state switches to the exceptional state. If there is no exceptional part, raising of an exception is a termination event (See 7.2.16).

A procedure with an exceptional part is executed in the normal state. The state is restored after block execution. A procedure without an exceptional part is executed in the state of the caller.

If an exception is raised in the state of exceptional execution it is re-raised in the calling context. In this case finalization of local modules and restoring protection (See 7.2.18) will not take place.

An additional statement (`RETRY`) can be used in the exceptional part. Execution of the `RETRY` statement causes the normal part to be re-executed in the normal state.

Execution of the `RETURN` statement in the exceptional part causes switch to the normal state.

If neither `RETURN` nor `RETRY` was executed in the exceptional part, the exceptional completion will occur. In this case after finalization of local modules (if any) and restoring protection state (if necessary), the exception will be re-raised.

Example

```
PROCEDURE Foo;
BEGIN
  TryFoo(...);
EXCEPT
```

```
    IF CanBeRepaired() THEN
      Repair;
      RETRY;  (* re-execute the normal part *)
    ELSIF CanBeProcessed() THEN
      Process;
      RETURN; (* exception is handled *)
    ELSE
      (* exception will be automatically re-raised *)
    END;
  END Foo;
```

7.2.14 The system module EXCEPTIONS

The module `EXCEPTIONS` provides facilities for raising user's exceptions and for making enquiries concerning the current execution state.

User-defined exceptions are identified uniquely by a pair (exception source, number). When the source of a used-defined exception is a separate module, it prevents the defined exceptions of the module from being raised directly by other sources. See e.g. the module `Storage`.

```
TYPE ExceptionSource;
```

Values of the opaque type `ExceptionSource` are used to identify the source of exceptions raised; they should be allocated before usage.

```
TYPE ExceptionNumber = CARDINAL;
```

Values of the type `ExceptionNumber` are used to distinguish between different exceptions of one source.

```
PROCEDURE AllocateSource(VAR newSource: ExceptionSource);
```

The procedure allocates an unique value of the type `ExceptionSource`. The procedure is normally called during initialization of a module, and the resulting value is then used in all calls of `RAISE`. If an unique value cannot be allocated the language exception `exException` is raised (See 7.2.15).

```
PROCEDURE RAISE(source: ExceptionSource;
                number: ExceptionNumber;
               message: ARRAY OF CHAR);
```

A call to `RAISE` associates the given values of exception `source`, `number`, and `message` with the current context and raises an exception.

The function `CurrentNumber` can be used to obtain the exception number for the current exception.

```
PROCEDURE CurrentNumber
   (source: ExceptionSource): ExceptionNumber;
```

If the calling coroutine is in the exceptional execution state because of raising an exception from `source`, the procedure returns the corresponding number, and otherwise raises an exception.

The procedure `GetMessage` can be used to obtain the message passed when an exception was raised. This may give further information about the nature of the exception.

```
PROCEDURE GetMessage(VAR text: ARRAY OF CHAR);
```

If the calling coroutine is in the exceptional execution state, the procedure returns the (possibly truncated) string associated with the current context. Otherwise, in the normal execution state, it returns the empty string.

```
PROCEDURE IsCurrentSource
        (source: ExceptionSource): BOOLEAN;
```

If the current coroutine is in the exceptional execution state because of raising an exception from `source`, the procedure returns `TRUE`, and `FALSE` otherwise.

```
PROCEDURE IsExceptionalExecution (): BOOLEAN;
```

If the current coroutine is in the exceptional execution state because of raising an exception, the procedure returns `TRUE`, and `FALSE` otherwise.

The following example illustrates the recommended form of a library module and usage of procedures from `EXCEPTIONS`.

```
DEFINITION MODULE FooLib;

PROCEDURE Foo;
(* Raises Foo exception if necessary *)

PROCEDURE IsFooException(): BOOLEAN;
(* Returns TRUE, if the calling coroutine is in
   exceptional state because of the raising of
   an exception from Foo, and otherwise returns FALSE.
```

```
*)

END FooLib.

IMPLEMENTATION MODULE FooLib;

IMPORT EXCEPTIONS;

VAR source: EXCEPTIONS.ExceptionSource;

PROCEDURE Foo;
BEGIN
  TryFoo(...);
  IF NOT done THEN
    EXCEPTIONS.RAISE(source,0,"Foo exception");
  END;
END Foo;

PROCEDURE IsFooException(): BOOLEAN;
BEGIN
  RETURN EXCEPTIONS.IsCurrentSource(source)
END IsLibException;

BEGIN
  EXCEPTIONS.AllocateSource(source)
END FooLib.
```

If we want to distinguish the exceptions raised in the `FooLib` we will append an enumeration type and an additional enquiry procedure in the `FooLib` definition:

```
TYPE FooExceptions = (fault, problem);

PROCEDURE FooException(): FooExceptions;
```

The `FooException` procedure can be implemented as follows:

```
PROCEDURE FooException(): FooExceptions;
BEGIN
  RETURN VAL(FooExceptions,
             EXCEPTIONS.CurrentNumber(source))
END FooException;
```

The `Client` module illustrates the usage of the library module `FooLib`:

```
MODULE Client;

IMPORT  FooLib, EXCEPTIONS, STextIO;

PROCEDURE ReportException;
  VAR s: ARRAY [0..63] OF CHAR;
BEGIN
  EXCEPTIONS.GetMessage(s);
  STextIO.WriteString(s);
  STextIO.WriteLn;
END ReportException;

PROCEDURE TryFoo;
BEGIN
  FooLib.Foo;
EXCEPT
  IF FooLib.IsFooException() THEN
    ReportException;
    RETURN; (* exception is handled *)
  ELSE
    (* Exception will be re-raised *)
  END
END TryFoo;

END Client.
```

7.2.15 The system module M2EXCEPTION

The system module `M2EXCEPTION` provides language exceptions identification facilities. The language (which includes the system modules) is regarded as one source of exceptions.

The module exports the enumeration type `M2Exceptions`, used to distinguish language exceptions, and two enquiry functions.

```
TYPE
  M2Exceptions =
    (indexException,      rangeException,
```

```
    caseSelectException, invalidLocation,
    functionException,   wholeValueException,
    wholeDivException,   realValueException,
    realDivException,    complexValueException,
    complexDivException, protException,
    sysException,        coException,
    exException
  );
```

```
PROCEDURE IsM2Exception(): BOOLEAN;
```

If the current coroutine is in the exceptional execution state because of the raising of a language exception, the procedure returns `TRUE`, and `FALSE` otherwise.

```
PROCEDURE M2Exception(): M2Exceptions;
```

If the current coroutine is in the exceptional execution state because of the raising of a language exception, the procedure returns the corresponding enumeration value, and otherwise raises an exception.

The following description lists all language exceptions (in alphabetical order) along with the circumstances under which they are detected. **Note:** Compiler options can be used to control detection of some exceptions (See Chapter 5). Detection of some exceptions is not required by the Standard, however such exceptions can be detected on some platforms (See Chapter A).

caseSelectException

Case selector is out of range and the `ELSE` clause does not exist.

coException

The system module **COROUTINES** (see 7.2.17) exceptions:

- RETURN from a coroutine other than the main coroutine
- size of the supplied workspace is smaller than the minimum required (See description of the procedure `NEWCOROUTINE`)
- the caller is not attached to the source of interrupts (See description of the procedure `HANDLER`)
- coroutine workspace overflow

complexDivException

Divide by zero in a complex number expression.

complexValueException

Overflow in evaluation of a complex number expression.

exException

A system module **EXCEPTIONS** or **M2EXCEPTION** exception:

- exception identity is enquired in the normal execution state (See `CurrentNumber`)
- exception identity enquiry to a wrong source (See `CurrentNumber`)
- no further exception source values can be allocated (See `AllocateSource`)

functionException

No `RETURN` statement before the end of a function.

indexException

Array index out of range. See options **CHECKINDEX** and **CHECKDINDEX**.

invalidLocation

Attempt to dereference `NIL` or an uninitialized pointer. See the option **CHECKNIL**.

protException

The given protection is less restrictive than the current protection.

rangeException

Range exception (See the **CHECKRANGE** option):

- assignment value is out of range of the target's type
- structure component value is out of range
- expression cannot be converted to the new type
- value to be included/excluded is not of the base type of the set (See also the **CHECKSET** option)
- return value is out of range
- set value is out of range (See also the **CHECKSET** option)
- tag value is out of range (in a variant record).

realDivException

Divide by zero in a real number expression.

realValueException

Overflow in evaluation of a real number expression.

sysException

The system module **SYSTEM** exceptions. **Note:** All these exceptions are non-mandatory.

- invalid use of `ADDADR`, `SUBADR` or `DIFADR`
- the result of `MAKEADR` is out of the address range
- alignment problem with `CAST`
- the result of `CAST` is not a valid representation for the target type

wholeDivException

Whole division exception:

- divided by zero in evaluation of a whole number expression
- the second operand of `DIV` or `MOD` is negative (See the **CHECKDIV** option)

wholeValueException

Overflow in evaluation of a whole number expression.

An example of language exception handling

```
PROCEDURE Div(a,b: INTEGER): INTEGER;
BEGIN
  RETURN a DIV b
EXCEPT
  IF IsM2Exception() THEN
    IF M2Exception() = wholeDivException THEN
      IF a < 0 THEN RETURN MIN(INTEGER)
      ELSE          RETURN MAX(INTEGER)
      END;
    END;
  END;
END Div;
```

7.2.16 Termination

During the program termination, finalizations of those static modules that have started initialization are executed in reverse order with respect to their initializations (See also 7.2.12). The static modules are the program module, the implementation modules, and any local modules declared in the module blocks of these modules.

Program termination starts from the first occurrence of the following event:

1. end of the program module body is reached
2. a `RETURN` statement is executed in the program module body
3. the standard procedure `HALT` is called
4. an exception was raised and is not handled

The system module `TERMINATION` provides facilities for enquiries concerning the occurrence of termination events.

```
PROCEDURE IsTerminating(): BOOLEAN;
```

Returns `TRUE` if any coroutine has inititated program termination and `FALSE` otherwise.

```
PROCEDURE HasHalted(): BOOLEAN;
```

Returns `TRUE` if a call of `HALT` has been made and `FALSE` otherwise.

7.2.17 Coroutines

The system module `COROUTINES` provides facilities for coroutines creation, explicit control transfer between coroutines, and interrupts handling. **Note:** Some features can be unavailable in the current release. See Chapter A for details.

Values of the type `COROUTINE` are created dynamically by a call of `NEWCOROUTINE` and identify the coroutine in subsequent operations. A particular coroutine is identified by the same value of the coroutine type throughout the lifetime of that coroutine.

```
TYPE COROUTINE;
```

The correspondent type was called `PROCESS` in PIM. From the third edition of PIM, the `ADDRESS` type was used to identify a coroutine.

```
PROCEDURE NEWCOROUTINE(
                procBody: PROC;
                workspace: SYSTEM.ADDRESS;
                size: CARDINAL;
                VAR cr: COROUTINE
                [; initProtection: PROTECTION]);
```

Creates a new coroutine whose body is given by `procBody`, and returns the identity of the coroutine in `cr`. `workspace` is a pointer to the work space allocated to the coroutine; `size` specifies the size of that workspace in terms of `SYSTEM.LOC`. `initProtection` is an optional parameter that specifies the initial protection level of the coroutine.

An exception is raised (See `coException`) if the value of `size` is less than the minimum workspace size.

If the optional parameter is omitted, the initial protection of the coroutine is given by the current protection of the caller.

The created coroutine is initialized in such a way that when control is first transferred to that coroutine, the procedure given by `procBody` is called in a normal state. The exception (`coException`) is raised when the `procBody` procedure attempts to return to its caller. Since the caller has no exception handler, raising this exception is a termination event.

The procedure `TRANSFER` can be used to transfer control from one coroutine to another.

```
PROCEDURE TRANSFER (VAR from: COROUTINE; to: COROUTINE);
```

Returns the identity of the calling coroutine in `from` and transfers control to the coroutine specified by `to`.

```
PROCEDURE CURRENT (): COROUTINE;
```

Returns the identity of the calling coroutine.

Interrupt handling

The `INTERRUPTSOURCE` type is used to identify interrupts.

```
TYPE INTERRUPTSOURCE = INTEGER;
```

Programs that use the interrupt handling facilities may be non-portable since the type is implementation-defined.

```
PROCEDURE ATTACH(source: INTERRUPTSOURCE);
```

Associates the specified source of interrupts with the calling coroutine. More than one source of interrupts may be associated with a single coroutine.

```
PROCEDURE DETACH(source: INTERRUPTSOURCE);
```

Dissociates the specified source of interrupts from the calling coroutine. The call has no effect if the coroutine is not associated with source.

```
PROCEDURE IsATTACHED(source: INTERRUPTSOURCE): BOOLEAN;
```

Returns TRUE if and only if the specified source of interrupts is currently associated with a coroutine; otherwise returns FALSE.

```
PROCEDURE HANDLER(source: INTERRUPTSOURCE): COROUTINE;
```

Returns the coroutine, if any, that is associated with the source of interrupts. The result is undefined if there is no coroutine associated with the source.

```
PROCEDURE IOTRANSFER(VAR from: COROUTINE;
                         to: COROUTINE);
```

Returns the identity of the calling coroutine in `from` and transfers control to the coroutine specified by `to`. On occurrence of an interrupt, associated with the caller, control is transferred back to the caller, and `from` returns the identity of the interrupted coroutine. An exception is raised if the calling coroutine is not associated with a source of interrupts.

Protection

See section 7.2.18 for information about the type `PROTECTION`.

```
PROCEDURE LISTEN(prot: PROTECTION);
```

Momentarily changes protection of the calling coroutine to `prot`, usually lowering it so as to allow an interrupt request to be granted.

```
PROCEDURE PROT(): PROTECTION;
```

Returns protection of the calling coroutine.

7.2.18 Protection

A program module, implementation module or local module may specify, by including protection in its heading, that execution of the enclosed statement sequence is protected.

```
ModuleHeading = MODULE ident [ Protection ] ";".
Protection    = [ ConstExpression ].
```

A module with protection in its heading is called a directly protected module. A directly protected procedure is an exported procedure declared in a protected module.

Protection of a module is provided by surrounding the externally accessible procedures and module body by calls of access control procedures. The value of the protection expression is passed to the call of access control procedures as an actual parameter.

The protection expression should be of the `PROTECTION` type. The `PROTECTION` type is an elementary type with at least two values: `INTERRUPTIBLE` and `UNINTERRUPTIBLE`.

Operators `<`, `>`, `<=` and `>=` can be used to compare values of the `PROTECTION` type. If x is a value of `PROTECTION` type, then x satisfies the conditions:

$$\texttt{UNINTERRUPTIBLE} \leq x \leq \texttt{INTERRUPTIBLE}$$

7.3 Standard procedures

	Procedure	Meaning
√	`ASSERT(x[,n])`	Terminates the program if x≠`TRUE` (See 7.6.14)
√	`COPY(x,v)`	Copies a string: `v := x`
	`DEC(v[,n])`	`v := v - n`, default n=1
	`DISPOSE(v)`	Deallocates v^ (See 7.2.11)
	`EXCL(v,n)`	`v := v - {n}`
	`HALT`	Terminates program execution (See 7.6.13)
	`INC(v[,n])`	`v := v + n`, default n=1
	`INCL(v,n)`	`v := v + {n}`
	`NEW(v)`	Allocates v^ (See 7.2.11)
√	`NEW(v,`$x_0 \ldots x_n$`)`	Allocates v^ of length x_0...x_n (See 7.6.12)

Table 7.4: Modula-2 proper procedures

This section briefly describes the set of standard procedures and functions. Some of them are not defined in the International Standard and are available only if the

	Function	Meaning
	ABS(x)	Absolute value of x
√	ASH(x,n)	Arithmetic shift
	CAP(x)	Corresponding capital letter
	CHR(x)	Character with the ordinal number x
	CMPLX(x,y)	Complex number with real part x and imaginary part y
√	ENTIER(x)	Largest integer not greater than x
	FLOAT(x)	VAL(REAL,x)
	HIGH(v)	High bound of the index of v
	IM(x)	Imaginary part of a complex x
	INT(x)	VAL(INTEGER,x)
√	LEN(v[,n])	Length of an array in the dimension n (default=0)
	LENGTH(x)	String length
	LFLOAT(x)	VAL(LONGREAL,x)
	MAX(T)	Maximum value of type T
	MIN(T)	Minimum value of type T
	ODD(x)	x MOD 2 = 1
	ORD(x)	VAL(CARDINAL,x)
	RE(x)	Real part of a complex x
	SIZE(T)	The number of storage units, required by a variable of type T
	TRUNC(x)	Truncation to the integral part
	VAL(T,x)	Type conversion

Table 7.5: Modula-2 function procedures

option **M2EXTENSIONS** is set. The procedure HALT (see 7.6.13) may have an additional parameter, if the extensions are enabled .

In the tables (7.4 and 7.5) of predefined procedures, v stands for a designator, x, y and n — for expressions, T — for a type. Non-standard procedures are marked with √.

The procedure COPY and the functions ASH, ENTIER and LEN are described in *The Oberon-2 Report*.

7.4 Compatibility

This section describes compatibility between entities of different types. There are three forms of compatibility:

- expression compatibility (specifying the types that may be combined in expressions);
- assignment compatibility (specifying the type of a value that may be assigned to a variable);
- parameter compatibility (specifying the type of an actual parameter that may be passed to a formal parameter).

The rules for parameter compatibility are relaxed in the case where a formal parameter is of a system storage type. This variation is known as the system parameter compatibility.

In most cases the compatibility rules are the same as described in PIM. However, we suppose to explicitly list all the rules.

7.4.1 Expression compatibility

Two expressions a and b of types T_a and T_b are *expression compatible* if any of the following statement is true:

a. The types T_a and T_b are identical. **Note:** If a type is a subrange type, then only its host type matters, therefore values of subranges of the same host type are expression compatible with each other and with the host type.

b. A type of one expression is a complex type, and the other expression is a complex constant.

c. A type of one expression is a real type, and the other expression is a real constant.

d. A type of one expression is a whole type, and the other expression is a whole constant.

e. A type of one expression is character, and the other expression is a string literal of length 0 or 1. See also 7.2.4.

```
VAR
  char: CHAR;
  ...
  WHILE (char # '') & (char # ".") DO
  ...
```

7.4.2 Assignment compatibility

An expression e of type T_e is *assignment compatible* with the variable v of type T_v if one of the following conditions holds[3]:

a. T_v is identical to the type T_e, and the type is not an open array type.

b. T_v is a subrange of the type T_e.

c. T_v is the CARDINAL type or a subrange of the CARDINAL type and T_e is the INTEGER type or e is a whole constant.

d. T_v is the INTEGER type or a subrange of the INTEGER type and T_e is the CARDINAL type or e is a whole constant.

e. T_v is a real type and e is a real constant.

f. T_v is a complex type and e is a complex constant.

g. T_v is a pointer type and e is NIL.

h. T_v is a procedure type and e is the designator of a procedure which has the same structure as the procedure type T_v and which has been declared at level 0.

i. T_v is the character type or a subrange of the character type and e is a string literal of length 0 or 1.

j. T_v is an array type having the character type as its component type, and e is a string literal of length less then or equal to the number of components in arrays of type T_v[4].

k. T_v is the address type and T_e is a pointer type or T_e is the address type and T_v is a pointer type.

[3] For an expression of a subrange type only host type matters.

[4] A string literal is not assignment compatible with an array whose component's type is a subrange of the character type.

7.4.3 Value parameter compatibility

A formal type is *value parameter compatible* with an actual expression if any of the following statements is true:

a. The formal type is constructed from a system storage type and is system parameter compatible with the expression.

b. The formal parameter is an open array, the actual parameter is an array type and the component type of the formal type is value parameter compatible with the component type of the actual type [5].

c. The formal type is assignment compatible with the actual parameter.

7.4.4 Variable parameter compatibility

A formal type is *variable parameter compatible* with an actual variable if any of the following statements is true:

a. The formal type is constructed from a system storage type and is system parameter compatible with the expression.

b. The formal parameter is an open array, the actual parameter is an array type and the component's type of the formal type is variable parameter compatible with the component's type of the actual parameter type.

c. The formal type is identical to the actual parameter type.

7.4.5 System parameter compatibility

A formal type is *system parameter compatible* with an actual parameter if any of the following statements is true:

a. The formal parameter is of the SYSTEM.LOC type and the actual parameter is of any type T such that `SIZE(T)` is equal to 1.

b. The formal parameter is of the type

```
ARRAY [0..n-1] OF SYSTEM.LOC
```

and the actual parameter is of any type T such that `SIZE(T)` is equal to `n`.

[5] A formal array parameter with the component's type T is not parameter compatible with the actual parameter of type T.

c. The formal parameter is of the open array type

```
ARRAY OF SYSTEM.LOC
```

and the actual parameter is of any type but not numeric literal.

d. The formal parameter is of the multi-dimensional open array type

```
ARRAY OF ARRAY [0..n-1] OF SYSTEM.LOC
```

and the actual parameter is of any type T such that `SIZE(T)` is a multiple of `n`.

7.5 The Modula-2 module SYSTEM

The module **SYSTEM** provides the low-level facilities for gaining an access to the address and underlying storage of variables, performing address arithmetic operations and manipulating the representation of values. Program that use these facilities may be non-portable.

This module does not exist in the same sense as other libraries but is hard-coded into the compiler itself. To use the facilities provided, however, identifiers must be imported in a usual way.

Some of the **SYSTEM** module procedures are generic procedures that cannot be explicitly declared, i.e. they apply to classes of operand types or have several possible forms of a parameter list .

The **SYSTEM** module is the only module specified in the International Standard that can be extended in the implementation. The XDS **SYSTEM** module provides additional types and procedures.

Note: The module **SYSTEM** is different in Oberon-2. See 8.6 for details.

```
DEFINITION MODULE SYSTEM;

CONST
  BITSPERLOC  = 8;
  LOCSPERWORD = 4;
  LOCSPERBYTE = 1;

TYPE
  LOC;
  ADDRESS = POINTER TO LOC;
  WORD = ARRAY [0 .. LOCSPERWORD-1] OF LOC;
```

```
  BYTE = LOC;

PROCEDURE ADDADR(addr: ADDRESS; offset: CARDINAL): ADDRESS;
PROCEDURE SUBADR(addr: ADDRESS; offset: CARDINAL): ADDRESS;
PROCEDURE DIFADR(addr1, addr2: ADDRESS): INTEGER;

PROCEDURE MAKEADR(val: <whole type>): ADDRESS;

PROCEDURE ADR(VAR v: <anytype>): ADDRESS;

PROCEDURE REF(VAR v: <anytype>): POINTER TO <type of the parameter>;

PROCEDURE ROTATE(val: <a packedset type>;
                 num: INTEGER): <type of the first parameter>;

PROCEDURE SHIFT(val: <a packedset type>;
                num: INTEGER): <type of the first parameter>;

PROCEDURE CAST(<targettype>;
               val: <anytype>): <targettype>;

PROCEDURE TSIZE (<type>; ... ): CARDINAL;

(*------------------------------------------------------- *)
(* -------------- non-standard features ----------------- *)

TYPE
  INT8   = <integer 8-bits type>;
  INT16  = <integer 16-bits type>;
  INT32  = <integer 32-bits type>;
  CARD8  = <cardinal 8-bits type>;
  CARD16 = <cardinal 16-bits type>;
  CARD32 = <cardinal 32-bits type>;
  BOOL8  = <boolean 8-bits type>;
  BOOL16 = <boolean 16-bits type>;
  BOOL32 = <boolean 32-bits type>;
  INDEX  = <type of index>
  DIFADR_TYPE = <type that DIFADR function returns>

TYPE (* for use in Oberon *)
  INT  = <Modula-2 INTEGER type>;
  CARD = <Modula-2 CARDINAL type>;
```

```
TYPE (* for interfacing to C *)
  int      = <C int type>;
  unsigned = <C unsigned type>;
  size_t   = <C size_t type>;
  void     = <C void type>;

PROCEDURE MOVE(src,dest: ADDRESS; size: CARDINAL);
PROCEDURE FILL(adr : ADDRESS; val : BYTE; size : CARDINAL;);

PROCEDURE GET(adr: ADDRESS; VAR var: SimpleType);
PROCEDURE PUT(adr: ADDRESS; var: SimpleType);

PROCEDURE CC(n: CARDINAL): BOOLEAN;

END SYSTEM.
```

7.5.1 System types

LOC

Values of the `LOC` type are the uninterpreted contents of the smallest addressable unit of a storage in implementation. The value of the call `TSIZE(LOC)` is therefore equal to one.

The type `LOC` was introduced as a mechanism to resolve the problems with `BYTE` and `WORD` types. Its introduction allows a consistent handling of both these types, and enables also `WORD`-like types to be further introduced, eg:

```
TYPE WORD16 = ARRAY [0..1] OF SYSTEM.LOC;
```

The only operation directly defined for the `LOC` type is an assignment. There are special rules affecting parameter compatibility for system storage types. See 7.4.5 for further details.

BYTE

`BYTE` is defined as `LOC` and has all the properties of the type `LOC`.

WORD

The type `WORD` is defined as

```
CONST LOCSPERWORD = 4;
TYPE WORD = ARRAY [0..LOCSPERWORD-1] OF LOC;
```

and the value of the call `TSIZE(WORD)` is equal to `LOCSPERWORD`.

The only operation directly defined for the WORD type is an assignment. There are special rules affecting parameter compatibility for system storage types. See 7.4.5 for further details.

ADDRESS

The type ADDRESS is defined as

```
TYPE ADDRESS = POINTER TO LOC;
```

The ADDRESS type is an assignment compatible with all pointer types and vice versa (See 7.4.2). A formal variable parameter of the ADDRESS type is a parameter compatible with an actual parameter of any pointer type.

Variables of type ADDRESS are no longer expression compatible with CARDINAL (as it was in PIM) and they cannot directly occur in expressions that include arithmetic operators. Functions ADDADR, SUBADR and DIFADR were introduced for address arithmetic.

Whole system types

Types INT8, CARD8, INT16, CARD16, INT32, CARD32 are guaranteed to contain 8, 16, or 32 bits respectively.

These types are introduced to simplify constructing the interfaces for foreign libraries (See Chapter 10). Types SHORTINT, LONGINT, SHORTCARD, LONGCARD are synonyms of INT8, INT32, CARD8, CARD32, respectively (See also the **M2ADDTYPES** option). Types INTEGER and CARDINAL are synonyms of INT16/INT32, CARD16/CARD32, depending on the target platform. See also the **M2BASE16** option.

These types are not described in the International Standard.

Boolean system types

Types BOOL8, BOOL16, and BOOL32 are guaranteed to contain 8,16 and 32 bits respectively. By default the compiler uses BOOL8 type for BOOLEAN. In some cases (e.g. in the interface to the Windows API) BOOL16 or BOOL32 should be used instead.

These types are not described in the International Standard.

Bitset system types

Types SET8, SET16, and SET32 are guaranteed to contain 8,16 and 32 bits respectively. The predefined type BITSET is a synonym for SYSTEM.SET16 or SYSTEM.SET32, depending on the target platform. See also the **M2BASE16** option.

These types are not described in the International Standard.

Modula-2 whole types

Types `INT` and `CARD` are equal to Modula-2 `INTEGER` and `CARDINAL` types, respectively. These types can be used in Oberon-2 in order to use Modula-2 procedures in a portable way. See 10.1 for further details.

These types are not described in the International Standard.

Interface to C

Types `int`, `unsigned`, `size_t` and `void` are introduced to simplify interfacing to C libraries. See 10.3 for further details.

7.5.2 System functions

```
PROCEDURE ADDADR(addr: ADDRESS;
                 offs: CARDINAL): ADDRESS;
```

Returns an address given by `(addr + offs)`. The subsequent use of the calculated address may raise an exception.

```
PROCEDURE SUBADR(addr: ADDRESS;
                 offs: CARDINAL): ADDRESS;
```

Returns an address given by `(addr - offs)`. The subsequent use of the calculated address may raise an exception.

```
PROCEDURE DIFADR(addr1,addr2: ADDRESS): INTEGER;
```

Returns the difference between addresses `(addr1 - addr2)`.

```
PROCEDURE MAKEADR(val: <whole type>): ADDRESS;
```

The function is used to construct a value of the ADDRESS type from the value of a whole type.

Note: The International Standard does not define the number and types of the parameters. Programs that use this procedure may be non-portable.

```
PROCEDURE ADR(VAR v: <any type>): ADDRESS;
```

Returns the address of the variable `v`.

```
PROCEDURE CAST(<type>; x: <any type>): <type>;
```

The function `CAST` can be used (as a type transfer function) to interpret a value

of any type other than a numeric literal value as a value of another type [6].

The value of the call `CAST(Type,val)` is an unchecked conversion of `val` to the type `Type`. If `SIZE(val) = TSIZE(Type)`, the bit pattern representation of the result is the same as the bit pattern representation of `val`; otherwise the result and the value of `val` have the same bit pattern representation for a size equal to the smaller of the numbers of storage units.

The given implementation may forbid some combinations of parameter types.

Note: In Oberon-2 module SYSTEM, the respective procedure is called `VAL`.

```
PROCEDURE TSIZE(Type; ... ): CARDINAL;
```

Returns the number of storage units (`LOC`s) used to store the value of the specified type. The extra parameters, if present, are used to distinguish variants in a variant record and must be constant expressions[7].

Example

```
TYPE
  R = RECORD
    CASE i: INTEGER OF
      |1: r: REAL;
      |2: b: BOOLEAN;
    END;
  END;

  ... TSIZE(R,1) ...
```

The value of `TSIZE(T)` is equal to `SIZE(T)`.

Packedset functions

Values of packedset types are represented as sequences of bits[8]. The bit number 0 is the least significant bit for a given platform. The following is true, where `v` is a variable of the type `CARDINAL`:

[6] The International Standard forbids the use of the PIM style type transfer, like `CARDINAL(x)`.
[7] Those constant expressions are ignored in the current release.
[8] The current implementation does not distinguish between set and packedset types.

```
CAST(CARDINAL,BITSET{0}) = VAL(CARDINAL,1)
SHIFT(CAST(BITSET,v),1)  = v * 2
SHIFT(CAST(BITSET,v),-1) = v DIV 2
```

Note: The functions `ROTATE` and `SHIFT` can be applied to a set with size less than or equal to the size of `BITSET`.

```
PROCEDURE ROTATE(x: T; n: integer): T;
```

Returns the value of `x` rotated `n` bits to the left (for positive `n`) or to the right (for negative `n`).

```
PROCEDURE SHIFT(x: T; n: integer): T;
```

Returns the value of `x` logically shifted `n` bits to the left (for positive `n`) or to the right (for negative `n`).

Warning: The result of `SHIFT(x,n)` , where `n` is greater than the number of elements in `T`, is undefined.

Non-standard functions

```
PROCEDURE CC(n: whole constant): BOOLEAN;
```

Returns TRUE if the corresponding condition flag is set. The function is not implemented in the current release.

```
PROCEDURE REF(VAR v: <anytype>):
              POINTER TO <type of the parameter>;
```

Returns the pointer to the variable `v`. See also 10.4.2.

```
PROCEDURE BIT(adr: T; bit: INTEGER): BOOLEAN;
```

Returns bit `n` of `Mem[adr]`. `T` is either ADDRESS or whole type.

7.5.3 System procedures

Note: all these procedures are non-standard.

```
PROCEDURE MOVE (src, dst: ADDRESS; size: CARDINAL);
```

Copies `size` bytes from the memory location specified by `src` to the memory location specified by `dst`.

Warning: No check for area overlap is performed. The behaviour of `SYSTEM.MOVE` in case of overlapping areas is undefined.

```
PROCEDURE FILL(adr : ADDRESS; val : BYTE; size : CARDINAL;);
```

Fills the memory block of size `size` starting from the memory location specified by `adr` with the value of `val` using a very efficient code.

```
PROCEDURE GET (adr: ADDRESS; VAR v: SimpleType);
PROCEDURE PUT (adr: ADDRESS;     x: SimpleType);
```

Gets/puts a value from/to address specified by `adr`. The second parameter cannot be of a record or array type.

```
VAR i: INTEGER;

  GET (128, i);   (* get system cell value *)
  i := i+20;      (* change it             *)
  PUT (128, i);   (* and put back          *)

PROCEDURE CODE(...);
```

The procedure is intended to embed a sequence of machine instructions directly into the generated code. The procedure is not implemented in the current release.

7.6 Language extensions

Warning: Using extensions may cause problems with software portability to other compilers.

In the standard mode the XDS Modula-2 compiler is ISO compliant (See 7.1). A set of language extensions may be enabled using the **M2EXTENSIONS** and **M2ADDTYPES** options.

The main purposes of supporting the language extensions are:

- to improve interfacing with other languages (See Chapter 10)
- to simplify migration from Modula-2 to Oberon-2
- to implement some useful features not found in ISO Modula-2
- to provide backward compatibility with previous releases

7.6.1 Lexical extensions

Comments

NOTE: Only valid when option **M2EXTENSIONS** is set.

As well as `(**)`, there is another valid format for comments in the source texts. The portion of a line from "`--`" to the end is considered as a comment.

```
  VAR i: INTEGER; -- this is a comment
--(*
  i:=0; (* this line will be compiled *)
--*)
```

Numeric constants

NOTE: Only valid when option **M2EXTENSIONS** is set.

Both Modula-2 and Oberon-2 syntax rules for the numeric and character representations may be used.

```
Number      = [ "+" | "-" ] Integer | Real.
Integer     = digit { digit }
            | octalDigit { octalDigit } "B"
            | digit { hexDigit } "X".
Real        = digit { digit } "." { digit } [ ScaleFactor ].
ScaleFactor = ( "E" | "D" ) [ "+" | "-" ] digit {digit}.

Character   = '"' char '"' | "'" char "'"
            | digit {hexDigit} "H"
            | octalDigit {octalDigit} "C".
```

Examples

```
1991        1991 (decimal)
0DH         13   (decimal)
15B         13   (decimal)
41X         "A"
101C        "A"
```

Note: the symbol "`D`" in a `ScaleFactor` denotes a `LONGREAL` value.

7.6.2 Additional numeric types

NOTE: Only valid when option **M2ADDTYPES** is set.

The compiler option **M2ADDTYPES** introduces the following additional numeric types:

1. **SHORTINT** integers between -128 and 127
2. **LONGINT** integers between -2^{31} and $2^{31}-1$
3. **SHORTCARD** unsigned integers between 0 and 255
4. **LONGCARD** unsigned integers between 0 and $2^{32}-1$

The following terms for groups of types will be used:

Real types for (`REAL`, `LONGREAL`)
Integer types for (`SHORTINT`, `INTEGER`, `LONGINT`)
Cardinal types for (`SHORTCARD`, `CARDINAL`, `LONGCARD`)
Whole types for *integer* and *cardinal types*
Numeric types for *whole* and *real types*

All integer types are implemented as subranges of internal compiler integer types. Therefore, according to the compatibility rules (See 7.4), the values of different integer types can be mixed in the expressions. The same holds for cardinal types. A mixture of integer and cardinal types is not allowed in expressions. As in Oberon-2, the numeric types form a hierarchy, and larger types include (i.e. can accept the values of) smaller types:

`LONGREAL` $\subseteq$ `REAL` $\subset$ *whole types*

Type compatibility in expressions is extended according to the following rules (See 7.4.1):

- The type of the result of an arithmetic or relation operation is the smallest type which includes the types of both operands.
- Before the operation, the values of both operands are converted to the result's type.

For instance, if the following variables are defined:

```
s: SHORTCARD;
c: CARDINAL;
i: INTEGER;
l: LONGINT;
r: REAL;
lr: LONGREAL;
```

then

Expression	Meaning	Result type
`s + c`	`VAL(CARDINAL,s) + c`	CARDINAL
`l * i`	`l * VAL(LONGINT,i)`	LONGINT
`r + 1`	`r + VAL(REAL,1)`	REAL
`r = s`	`r = VAL(REAL,s)`	BOOLEAN
`r + lr`	`VAL(LONGREAL,r) + lr`	LONGREAL
`c + i`	`not allowed`	

The assignment compatibility rules are also extended (See 7.4.2), so an expression e of type T_e is assignment compatible with a variable v of type T_v if T_e and T_v are of numeric types and T_v includes T_e. Cardinal types and integer types are assignment compatible. The compiler generates the range checks whenever necessary.

Examples (see declarations above):

Statement	Comment
`i:=c;`	`INTEGER` and `CARDINAL` are assignment compatible
`i:=s;`	`INTEGER` and `SHORTCARD` are assignment compatible
`l:=i;`	`LONGINT` and `INTEGER` are subranges of the same host type
`r:=i;`	`REAL`$\subset$ `INTEGER`
`r:=c;`	`REAL`$\subset$ `CARDINAL`
`lr:=r;`	`LONGREAL`$\subseteq$ `REAL`

7.6.3 Type casting

NOTE: Only valid when option **M2EXTENSIONS** is set.

In ISO Modula-2, the second parameter of the `SYSTEM.CAST` procedure can not be a numeric literal. XDS provides numeric literal casting as an extension:

```
VAR
  c: CARDINAL;
BEGIN
  (* Ok if M2EXTENSIONS is ON *)
  c := SYSTEM.CAST(CARDINAL,-1);
```

7.6.4 Assignment compatibility with BYTE

NOTE: Only valid when option **M2EXTENSIONS** is set.

An expression of type `CHAR`, `BOOLEAN`, `SHORTCARD`, `SHORTINT`, `SYSTEM.INT8`, or `SYSTEM.CARD8` can be assigned to a variable of type `BYTE` or passed as an actual parameters to a formal parameter of type `BYTE`.

7.6.5 Dynamic arrays

NOTE: Only valid when option **M2EXTENSIONS** is set.

XDS allows Oberon-2 style dynamic arrays to be used according to the Oberon-2 rules.

An open array is an array type with no lower and upper bound specified, i.e. ARRAY OF SomeType. Open arrays may be used only in procedure parameter lists or as a pointer base type.

```
TYPE String = POINTER TO ARRAY OF CHAR;
```

Neither variables nor record fields may be of open array type.

If the designator type is formally an open array, then the only operations allowed with it are indexing and passing it to a procedure.

The extended versions of standard procedures `NEW` and `DISPOSE` can be used to create and delete the dynamic arrays (See 7.6.12).

Example

```
TYPE
  VECTOR = ARRAY OF REAL;
    (* 1-dim open array *)
  Vector = POINTER TO VECTOR;
```

```
    (* pointer to open array *)
  MATRIX = ARRAY OF VECTOR;
    (* 2-dim open array *)
  Matrix = POINTER TO MATRIX;
    (* pointer to this *)

VAR
   v: Vector;
   m: Matrix;

PROCEDURE ClearVector(VAR v: VECTOR);
  VAR i: CARDINAL;
BEGIN
  FOR i := 0 TO HIGH (v) DO v[i] := 0 END;
END ClearVector;

PROCEDURE ClearMatrix(VAR m: Matrix);
  VAR i: CARDINAL;
BEGIN
  FOR i := 0 TO HIGH (m) DO ClearVector(m[i]) END;
END ClearMatrix;

PROCEDURE Test;
BEGIN
  NEW(v, 10);
  NEW(m, 10, 20);
  ClearVector(v^);
  ClearMatrix(m^);
  v^[0] := 1;
  m^[1][1] := 2;
  m^[2,2] := 1000;
  DISPOSE(v);
  DISPOSE(m);
END Test;
```

7.6.6 Constant array constructors

NOTE: Only valid when option **M2EXTENSIONS** is set.

XDS allows the declaration of constant arrays in the form

```
ARRAY OF QualIdent "{" ExprList "}"
```

`QualIdent` should refer to a basic type, range or enumeration type, and all expressions within `ExprList` should be of that type.

Note: structured types and non-constant expressions are not allowed.

The actual type of such a constant is `ARRAY [0..n] OF QualIdent`, where `n+1` is the number of expressions in `ExprList`.

```
CONST table = ARRAY OF INTEGER {1, 2+3, 3};
```

Constant arrays are subject to the same rules as all other constants, and may be read as a normal array.

In some cases constructors of this form are more convenient than ISO standard value constructors (See 7.2.5), because you do not need to declare a type and to calculate manually the number of expressions. However, to make your programs more portable, we recommend to use the standard features.

7.6.7 Set complement

NOTE: Only valid when option **M2EXTENSIONS** is set.

As in Oberon-2, an unary minus applied to a set denotes the complement of that set, i.e. $-x$ is the set of all values which are not the elements of x.

```
TYPE SmallSet = SET OF [0..5];
VAR x, y: SmallSet;
BEGIN
  x := SmallSet{1,3,5};
  y := -x;                    (* y = {0, 2, 4} *)
  y := SmallSet{0..5} - x;    (* y = {0, 2, 4} *)
END;
```

7.6.8 Read-only parameters

NOTE: Only valid when option **M2EXTENSIONS** is set.

In a formal parameter section, the symbol `"-"` may be placed after the name of a value parameter. Such a parameter is called *read-only*; its value can not be changed in the procedure body. Read-only parameters do not need to be copied

before procedure activation; this enables procedures with structured parameters to be more effective.

For ARRAY and RECORD read-only parameters, the array elements and record fields are protected. Read-only parameters cannot be used in definition modules.

We recommend to use read-only parameters with care. The compiler does not check that the read-only parameter is not modified via another parameter or a global variable.

Example

```
PROCEDURE Foo(VAR dest: ARRAY OF CHAR;
              source-: ARRAY OF CHAR);
BEGIN
  dest[0]:='a';
  dest[1]:=source[0];
END Foo;
```

The call `Foo(x,x)` would produce a wrong result, because the first `Foo` statement changes the value of `source[0]` (`source` is not copied and points to the same location as `dest`).

7.6.9 Variable number of parameters

NOTE: Only valid when option **M2EXTENSIONS** is set.

The last formal parameter of a procedure may be declared as a "sequence of bytes" (SEQ-parameter). In a procedure call, any (possibly empty) sequence of actual parameters of any types may be substituted in place of that parameter. Only the declaration

```
SEQ name: SYSTEM.BYTE
```

is allowed. A procedure may have only one SEQ parameter, and it must be the last element of the formal parameters list.

Within the procedure, sequence parameters are very similar to open array parameters. This means that :

- the `HIGH` function can be applied to the parameter;
- a `SEQ` actual parameter may be subsequently passed to another procedure

- the `i`-th byte of the sequence `s` can be accessed as `s[i]`, like an array element.

An array of bytes, which is passed to a procedure as a formal SEQ-parameter, is formed as follows:

- values of all actual parameters forming the sequence are represented as described below and concatenated into an array in their textual order
- integer values are converted to `LONGINT`
- `BOOLEAN`, `CHAR`, cardinal and enumeration values are converted to `LONGCARD`
- values of range types are converted according to their base types
- real values are converted to `LONGREAL`
- values of pointer, opaque and procedure types are converted to `ADDRESS`
- a structured value (record or array) is interpreted as an array of bytes and passed as a sequence of:
  - the address of the structure
  - a zero 32-bit word (reserved for future extensions)
  - size of the structure (in LOCs) minus one

See 12.2 for further information.

7.6.10 Read-only export

NOTE: Only valid when option **M2EXTENSIONS** is set.

The Oberon-2 read-only export symbol ”-”, being specified after a variable or field identifier in a definition module will define the identifier as read-only for any client. Only the module in which a read-only variable or field is declared may change its value.

The compiler will not allow the value of a read-only exported object to be changed explicitly (by an assignment) or implicitly (by passing it as a VAR parameter).

For read-only variables of an array or record type, both array elements and record fields are also read-only.

Example (an excerpt from a definition module):

```
TYPE Rec = RECORD
  n-: INTEGER;
  m : INTEGER;
END;

VAR
  in-: FILE;
  x-: Rec;
```

7.6.11 Renaming of imported modules

NOTE: Only valid when option **M2EXTENSIONS** is set.

An imported module can be renamed inside the importing module. The real name of the module becomes invisible.

```
Import = IMPORT [ Ident ":=" ] Ident
          { "," [ Ident ":=" ] ident } ";".
```

Example

```
MODULE test;
IMPORT vw := VirtualWorkstation;

VAR ws: vw.Station;

BEGIN
  ws := vw.open();
END test.
```

7.6.12 NEW and DISPOSE for dynamic arrays

Standard procedures `NEW` and `DISPOSE` can be applied to variables of a dynamic array type (See 7.6.5). Procedures `DYNALLOCATE` and `DYNDEALLOCATE` have to be visible in the calling context. Their headers and semantics are described below.

```
PROCEDURE DYNALLOCATE(VAR a: ADDRESS;
                          size: CARDINAL;
                           len: ARRAY OF CARDINAL);
```

The procedure must allocate a dynamic array and return its address in `a`. `size` is the size of the array base type (the size of an element) and `len[i]` is the length of the array in i-th dimension.

```
PROCEDURE DYNDEALLOCATE(VAR a: ADDRESS;
                        size,dim: CARDINAL);
```

The procedure must deallocate a dynamic array, where `size` is the size of an element and `dim` is the number of dimensions.

Note: In most cases, default implementation of these procedures may be used. The **STORAGE** option controls whether the default memory management should be enabled.

A dynamic array is represented as a pointer to a so-called *array descriptor* (See 12.1.8).

7.6.13 HALT

NOTE: Only valid when option **M2EXTENSIONS** is set.

An optional integer parameter is allowed for the `HALT` procedure.

```
PROCEDURE HALT ([code: INTEGER]);
```

`HALT` terminates the program execution with an optional return code. Consult your operating system/environment documentation for more details.

7.6.14 ASSERT

NOTE: Only valid when option **M2EXTENSIONS** is set.

The procedure `ASSERT` checks its boolean parameter and terminates the program if it is not `TRUE`. The second optional parameter denotes *task termination code*. If it is omitted, a standard value is assumed.

```
PROCEDURE ASSERT(cond: BOOLEAN [; code: INTEGER]);
```

A call `ASSERT(expr,code)` is equivalent to

```
IF NOT expr THEN HALT(code) END;
```

7.7 Source code directives

Source code directives (or pragmas) are used to set compilation options in the source text and to select specific pieces of the source text to be compiled (conditional compilation). The ISO Modula-2 standard does not describe pragma syntax. XDS supports source code directives in both Modula-2 and Oberon-2. The syntax described in *The Oakwood Guidelines for the Oberon-2 Compiler Developers* is used.

7.7.1 Inline options and equations

In some cases it is more desirable to set a compiler option or equation within the source text. Some compiler options, such as **MAIN**, are more meaningful in the source file before the module header, and some, such as run-time checks, even between statements.

XDS allows options to be changed in the source text by using standard ISO pseudo comments `<* ... *>`[9] Some options can only be placed in the source text before the module header (i.e. before keywords `IMPLEMENTATION`, `DEFINITION`, and `MODULE`). These options will be ignored if found elsewhere in the source text. See 5.2 for more details.

The format of an inline option or equation setting is described by the following syntax:

```
Pragma     = "<*" PragmaBody "*>"
PragmaBody = PUSH | POP | NewStyle | OldStyle
NewStyle   = [ NEW ] name [ "+" | "-" | "=" string ]
OldStyle   = ("+" | "-") name
```

`NewStyle` is proposed as the Oakwood standard for Oberon-2, `OldStyle` is the style used in the previous XDS releases. All option names are case-independent. If `OldStyle` is used, there should be no space between `<*` and

[9] The old pragma style `(*$..*)` is supported to provide backward compatibility, but the compiler reports the “obsolete syntax” warning.

`+` or `- OldStyle` does not allow to declare a new option or equation and to change an equation value.

In all cases, the symbol `+` sets the corresponding option ON, and the symbol `-` sets it OFF.

`PUSH` and `POP` keywords may be used to save and restore the whole state of options and equations.

Examples

```
PROCEDURE Length(VAR a: ARRAY OF CHAR): CARDINAL;
  VAR i: CARDINAL;
BEGIN
  <* PUSH *>              (* save state *)
  <* CHECKINDEX - *>      (* turn CHECKINDEX off *)
  i := 0;
  WHILE (i<=HIGH(a)) & (a[i]#0C) DO INC(i) END;
  <* POP *>               (* restore state *)
  RETURN i;
END Length;

<* ALIGNMENT = "2" *>
TYPE
  R = RECORD      (* This record is 6 bytes long *)
        f1: CHAR;
        f2: CARDINAL;
      END;
```

7.7.2 Conditional compilation

It is possible to use conditional compilation with Modula-2 and Oberon-2[10] compilers via the standard ISO pragma notation `<* *>`. Conditional compilation statements can be placed anywhere in the source code. The syntax of the conditional compilation `IF` statement follows:

```
IfStatement      = <* IF Expression THEN *> text
                 { <* ELSIF Expression THEN *> text }
                 [ <* ELSE *> text ]
```

[10] only if the **O2ISOPRAGMA** option is set ON

```
                    <* END *>
Expression        = SimpleExpression
                    [ ("=" | "#") SimpleExpression].
SimpleExpression  = Term { "OR" Term}.
Term              = Factor { "&" Factor}.
Factor            = Ident | string |
                    "DEFINED" "(" Ident ")" |
                    "(" Expression ")" |
                    "~" Factor | "NOT" Factor.
Ident             = option | equation.
```

An operand in an expression is either a name of an option or equation or a string literal. An option has the string value `"TRUE"`, if it is currently set ON and `"FALSE"`, if it is currently set off or was not defined at all. The compiler will report a warning if an undeclared option or equation is used as a conditional compilation identifier.

The comparison operators ”=” and ”#” are not case sensitive.

See also the section 5.6.

Examples

```
IMPORT lib :=
      <* IF  __GEN_X86__ THEN *> MyX86Lib;
      <* ELSIF __GEN_C__ THEN *> MyCLib;
      <* ELSE *> *** Unknown ***
      <* END *>

CONST Win = <* IF Windows THEN *> TRUE
            <* ELSE *> FALSE
            <* END *>;

<* IF DEFINED(Debug) & (DebugLevel = "2") THEN *>
   PrintDebugInformation;
<* END *>;

<* IF target_os = "OS2" THEN *>
  Strings.Capitalize(filename);
  <* IF NOT HPFS THEN *>
    TruncateFileName(filename);
```

```
    <* END *>
  <* END *>
```

Chapter 8

XDS Oberon-2

This chapter includes the details of the Oberon-2 language which are specific for this implementation. In the standard mode[1] XDS Oberon-2 is fully compatible with ETH compilers (See *The Oberon-2 Report*). The last changes to the language are described in 8.2.

To provide a smooth path from Modula-2 to Oberon-2 XDS allows all Modula-2 data types to be used in Oberon-2 modules (See 8.4).

Several language extensions are implemented in the language according to *The Oakwood Guidelines for the Oberon-2 Compiler Developers*[2] (See 8.3). Other language extensions are described in 8.5. As XDS is a truly multi-lingual system, special features were introduced to provide interfacing to foreign languages (See Chapter 10).

8.1 The Oberon environment

The Oberon-2 language was originally designed for use in an environment that provides *command activation, garbage collection,* and *dynamic loading* of the modules. Not being a part of the language, these features still contribute to the power of Oberon-2.

The garbage collector and command activation are implemented in the Oberon Run-Time Support and can be used in any program. The dynamic loader is not provided in the current release. See 9.3 for further information.

[1] When the options **O2EXTENSIONS** and **O2NUMEXT** are OFF.

[2] These guidelines have been produced by a group of Oberon-2 compiler developers, including ETH developers, after a meeting at the Oakwood Hotel in Croydon, UK in June 1993.

8.1.1 Program structure

In an Oberon-2 environment, any declared parameterless procedure can be considered as a main procedure and can be called by its name (a qualified identifier of the form `ModuleName.ProcName`).

Due to the nature of XDS, and its freedom from the Oberon system, a different approach had to be found to declare the 'top level' or program modules.

The module which contains the top level of your program must be compiled it with the **MAIN** option set. This will generate an entry point to your program. Only one module per program shall be compiled with the option set. It is recommended to set it in the module header:

```
<*+ MAIN *>
MODULE hello;

IMPORT InOut;

BEGIN
  InOut.WriteString ("Hello World!");
  InOut.WriteLn;
END hello.
```

8.1.2 Creating a definition

XDS provides two different ways to create a definition for an Oberon-2 module:

- the **BROWSE** operation mode (see 4.2.5) creates a definition module from a symbol file
- the **MAKEDEF** option forces the Oberon-2 compiler to generate a (pseudo) definition module after successful compilation of an Oberon-2 module.

The **MAKEDEF** option provides additional services: the compiler will preserve the so-called *exported* comments (i.e. comments which start with '(**') if the **XCOMMENTS** option is ON.

The generated pseudo-definition module contains all exported declarations in the order of their appearance in the source text. All exported comments are placed at the appropriate positions.

A definition can be generated in three *styles*. The **BSTYLE** equation can be used to choose one of the styles: **DEF** (default), **DOC** or **MOD**.

The DEF style

This produces an ETH-style definition module. All *type-bound procedures* (*methods*) and relative comments are shown as parts of the corresponding record types.

This is the only style for which the **BSREDEFINE** and **BSCLOSURE** options are applicable.

The DOC style

This produces a pseudo-definition module in which methods are shown as parts of the appropriate record types (ignoring comments) and at the positions at which they occur in the source text.

The MOD style

This attempts to produce a file which can be compiled as an Oberon-2 module after slight modification (i.e. the file will contain "`END procname`", etc.)

8.2 Last changes to the language

8.2.1 ASSERT

The procedure `ASSERT` checks its boolean parameter and terminates the program if it is not `TRUE`. The second optional parameter denotes a *task termination code*. If omitted, a standard value is assumed.

```
PROCEDURE ASSERT(cond: BOOLEAN [; code: INTEGER]);
```

A call `ASSERT(expr,code)` is equivalent to

```
IF NOT expr THEN HALT(code) END;
```

8.2.2 Underscores in identifiers

According to the *Oakwood Guidelines* an underscore ("_") may be used in identifiers (as a letter).

```
ident = ( letter | "_" ) { letter | digit | "_" }
```

We recommend to use underscores with care, as it may cause problems with software portability to other compilers. This feature may be important for interfacing to foreign languages (See Chapter 10).

8.2.3 Source code directives

Source code directives (or pragmas) are used to set compilation options in the source text and to select specific pieces of the source text to be compiled (conditional compilation). According to the *Oakwood Guidelines* all directives are contained in ISO Modula-2 style pseudo comments using angled brackets `<* ... *>`.

The additional language constructs should not be considered to be part of the Oberon-2 language. They define a separate compiler control language that coexist with Oberon-2. The option **O2ISOPRAGMA** allows pragmas to be used.

The syntax of the directives is the same for Modula-2 and Oberon-2. See 7.7 for further details.

8.3 Oakwood numeric extensions

XDS Oberon-2 supports two extensions which are of importance for scientific programming, namely

- complex numbers
- in-line exponentiation operator

The **O2NUMEXT** option should be set to use these extensions.

8.3.1 Complex numbers

NOTE: Only valid when option **O2NUMEXT** is set.

Two additional types are included in the type hierarchy if the **O2NUMEXT** option is set:

```
COMPLEX      defined as  (REAL,REAL)
LONGCOMPLEX  defined as  (LONGREAL,LONGREAL)
```

All numeric types form a (partial) hierarchy

$$\textit{whole types} \subset \text{REAL} \subseteq \begin{matrix}\text{COMPLEX}\\ \text{LONGREAL}\end{matrix} \subseteq \text{LONGCOMPLEX}$$

A common mathematical notation is used for complex number literals:

```
number = integer | real | complex
complex = real "i"
```

A literal of the form `5.0i` denotes a complex number with real part equal to zero and an imaginary part equal to 5.0. Complex constants with a non-zero real part can be described using arithmetic operators.

```
CONST
  i = 1.i;
  x = 1. + 1.i;
```

For the declarations

```
VAR
  c: COMPLEX;
  l: LONGCOMPLEX;
  r: REAL;
  x: INTEGER;
```

the following statements are valid:

```
  c:=i+r;
  l:=c;
  l:=c*r;
  l:=l*c;
```

New conversion functions `RE` and `IM` can be used to obtain a real or imaginary part of a value of a complex type. Both functions have one parameter. If the parameter is of the `COMPLEX` type, both functions return a `REAL` value; if the parameter is of the `LONGCOMPLEX` type, functions return a `LONGREAL` value; otherwise the parameter should be a complex constant and functions return a real constant.

A complex value can be formed by applying the standard function `CMPLX` to two reals. If both `CMPLX` arguments are real constants, the result is a complex constant.

```
CONST i = CMPLX(0.0,1.0);
```

If both expressions are of the `REAL` type, the function returns a `COMPLEX` value, otherwise it returns a `LONGCOMPLEX` value.

8.3.2 In-line exponentiation

NOTE: Only valid when option **O2NUMEXT** is set.

The exponentiation operator `**` provides a convenient notation for arithmetic expressions, which does not involve function calls. It is an arithmetic operator which has a higher precedence than multiplication operators.

```
Term     =  Exponent { MulOp Exponent }.
Exponent =  Factor { "**" Factor }.
```

Note: the operator is right-associated:

$a**b**c$ is evaluated as $a**(b**c)$

The left operand of the exponentiation (`a**b`) should be any numeric value (including complex), while the right operand should be of a real or integer type. The result type does not depend of the type of right operand and is defined by the table:

Left operand type	**Result type**
an integer type	`REAL`
`REAL`	`REAL`
`LONGREAL`	`LONGREAL`
`COMPLEX`	`COMPLEX`
`LONGCOMPLEX`	`LONGCOMPLEX`

8.4 Using Modula-2 features

All Modula-2 types and corresponding operations can be used in Oberon-2, including enumeration types, range types, records with variant parts, sets, etc.

Important Notes:

- It is not allowed to declare Modula-2 types in an Oberon-2 module.

- A module using Modula-2 features is likely to be non-portable to other compilers.

Example

```
(*MODULA-2*) DEFINITION MODULE UsefulTypes;

TYPE
  TranslationTable = ARRAY CHAR OF CHAR;
  Color  = (red,green,blue);
  Colors = SET OF Color;

END UsefulTypes.

(*OBERON-2*) MODULE UsingM2;

IMPORT UsefulTypes;

TYPE
  TranslationTable* = UsefulTypes.TranslationTable;

VAR colors*: UsefulTypes.Color;

BEGIN
  colors:=UsefulTypes.Colors{UsefulTypes.red};
END UsingM2.
```

8.5 Language extensions

Warning: Using extensions may cause problems with the software portability to other compilers.

In the standard mode, the XDS Oberon-2 compiler is fully compatible with ETH compilers (See also 8.2). The **O2EXTENSIONS** option enables some language extensions. The main purposes of language extensions are

- to improve interfacing to other languages (See Chapter 10).
- to provide backward compatibility with the previous versions of XDS.

See also

- Source language directives (8.2.3)
- Oakwood numeric extensions (8.3).

8.5.1 Comments

NOTE: Only valid when option **O2EXTENSIONS** is set.

As well as "`(**)`", there is another valid format for comments in source texts. The portion of a line from "`--`" to the end of line is considered a comment.

```
VAR j: INTEGER; -- this is a comment
```

8.5.2 String concatenation

NOTE: Only valid when option **O2EXTENSIONS** is set.

The symbol "+" can be used for constant string and characters concatenation. See 7.2.4 for more details.

8.5.3 VAL function

NOTE: Only valid when option **O2EXTENSIONS** is set.

The function `VAL` can be used to obtain a value of the specified scalar type from an expression of a scalar type. See 7.2.10 for more details.

```
PROCEDURE VAL(Type; expr: ScalarType): Type;
```

The function can be applied to any scalar types, including system fixed size types (See 8.6.2).

8.5.4 Read-only parameters

NOTE: Only valid when option **O2EXTENSIONS** is set.

In a formal parameter section, the symbol `"-"` may appear after a name of a value parameter. That parameter is called *read-only*; its value can not be changed in the procedure's body. Read-only parameters need not to be copied before the procedure activation; this enables procedures with structured parameters to be more effective. Read-only parameters can not be used in a procedure type declaration.

We recommend to use read-only parameters with care. The compiler does not check that the read-only parameter is not modified via another parameter or a global variable.

Example

```
PROCEDURE Foo(VAR dest: ARRAY OF CHAR;
                source-: ARRAY OF CHAR);
BEGIN
  dest[0]:='a';
  dest[1]:=source[0];
END Foo;
```

The call `Foo(x,x)` would produce a wrong result, because the first statement changes the value of `source[0]` (`source` is not copied and points to the same location as `dest`).

8.5.5 Variable number of parameters

NOTE: Only valid when option **O2EXTENSIONS** is set.

Everything contained in the section 7.6.9 is applicable to Oberon-2.

8.5.6 Value constructors

NOTE: Only valid when option **O2EXTENSIONS** is set.

Everything contained in the section 7.2.5 is applicable to Oberon-2.

8.6 The Oberon-2 module SYSTEM

Low-level facilities are provided by the module `SYSTEM`. This module does not exist in the same sense as other library modules; it is hard-coded into the compiler

itself. However, to use the provided facilities, it must be imported in the usual way.

Some procedures in the module SYSTEM are generic procedures that cannot be explicitly declared, i.e. they apply to classes of operand types.

XDS Oberon-2 compiler implements all system features described in *The Oberon-2 Report* (except GETREG, PUTREG, and CC) and allows one to access all features, described in the Modula-2 International Standard Modula-2 (See 7.5). In this section we describe only features specific for this implementation.

8.6.1 Compatibility with BYTE

Expressions of types CHAR, BOOLEAN, SHORTINT and SYSTEM.CARD8 can be assigned to variables of type BYTE or passed as actual parameters to formal parameters of type BYTE.

If a formal procedure parameter has type ARRAY OF BYTE, then the corresponding actual parameter may be of any type, except numeric literals.

8.6.2 Whole system types

Module SYSTEM contains the signed types INT8, INT16, INT32, and unsigned types CARD8, CARD16, CARD32, which are guaranteed to contain exactly 8, 16, or 32 bits respectively. These types were introduced to simplify consstructing the interfaces to foreign libraries (See Chapter 10). The basic types SHORTINT, INTEGER, LONGINT are synonyms of INT8, INT16, and INT32 respectively.

The unsigned types form a hierarchy whereby larger types include (the values of) smaller types.

$$\texttt{SYSTEM.CARD32} \supseteq \texttt{SYSTEM.CARD16} \supseteq \texttt{SYSTEM.CARD8}$$

The whole hierarchy of numeric types (See also 8.3.1):

$$\texttt{LONGREAL} \supseteq \texttt{REAL} \supseteq \begin{cases} \text{signed types} \\ \text{unsigned types} \end{cases}$$

8.6.3 NEW and DISPOSE

The procedure SYSTEM.NEW can be used to allocate the system memory, i.e. memory which is not the subject of garbage collection. SYSTEM.NEW is a generic

procedure, which is applied to pointer types and can be used in several ways, depending on pointer's base type.

```
PROCEDURE NEW(VAR p: AnyPointer [; x0,..xn: integer]);
```

Let type `P` be defined as `POINTER TO` T and p is of type `P`.

`NEW(`p`)`	T is a record or fixed length array type. The procedure allocates a storage block of `SIZE(`T`)` bytes and assigns its address to p.
`NEW(`p`,`n`)`	T is a record or fixed length array type. The procedure allocates a storage block of n bytes and assigns its address to p.
`NEW(`p`,`x_0`, ..`x_{n-1}`)`	T is an n-dimensional open array. The procedure allocates an open array of lengths given by the expressions x_0,..x_{n-1}

The procedure `SYSTEM.DISPOSE` can be used to free a block previously allocated by a call to `SYSTEM.NEW`. It does *not* immediately deallocate the block, but marks it as a free block. The block will be deallocated by the next call of the garbage collector.

```
PROCEDURE DISPOSE(VAR p: AnyPointer; [size: integer]);
```

`DISPOSE(`p`)`	T is a record or array type. The procedure deallocates the storage block p points to.
`DISPOSE(`p`,`n`)`	T is a record or fixed length array type. The procedure deallocates the storage block of n bytes p points to.

8.6.4 M2ADR

In Oberon-2, the `SYSTEM.ADR` procedure returns `LONGINT`, which is not always very convenient. The `SYSTEM.M2ADR` procedure behaves as Modula-2 `SYSTEM.ADR`, returning `SYSTEM.ADDRESS`:

```
PROCEDURE M2ADR(VAR x: any type): ADDRESS;
```

Chapter 9

Run-time support

Some language features are implemented in the run-time library, including:

- exceptions and finalization
- coroutines
- memory management
- garbage collection
- postmortem history

XDS provides an integrated Modula-2 and Oberon-2 run-time library, taking into account the possibility that modules written in both languages are used in one project. As a rule, if you do not use a particular feature, the part of RTS that implements that feature will not be added to your executable program. For example, if your program is written entirely in Modula-2, the Oberon-2 part of RTS (garbage collector, meta-language facilities) will not included.

The integrated memory manager is described in 9.1. The section 9.3 describes an interface to the Oberon-2 run-time support.

9.1 Memory management

The XDS integrated memory manager implements

- default memory allocation and deallocation procedures for Modula-2 (See the option **STORAGE**);

- memory allocation procedures for Oberon-2;
- system memory allocation procedures for Oberon-2 (See 8.6.3);
- the garbage collector.

The compiler provides the option **GCAUTO** and the equation **HEAPLIMIT** to control the memory management. They should be set when the top-level module of the program is compiled[1]. The compiler uses their values when generating the RTS initialization call.

The equation **HEAPLIMIT** specifies the maximum size of the heap in bytes. If that equation is set to zero, the run-time system automatically determines heap size at startup and dynamically adjusts it according to application's memory use and system load.

The option **GCAUTO** allows the garbage collector to be called implicitly. If the option is not set the garbage collector must be called explicitly (See 9.3). The garbage collector is called implicitly by the memory allocation procedure in the following cases:

- a memory block of the requested length cannot be allocated;
- the amount of busy memory exceeds the limit specified by the **HEAPLIMIT** equation (or the limit chosen by the run-ime system if **HEAPLIMIT** was set to zero during compilation);
- the amount of busy memory exceeds some limit set internally by the memory manager for optimum performance.

If the memory block still cannot be allocated after the call to the garbage collector, the exception `XEXCEPTIONS.noMemoryException` will be raised by the Oberon-2 memory allocation procedure[2].

Note: In a pure Modula-2 program, the garbage collector is never invoked, so you may set the **HEAPLIMIT** equation to a very large value.

9.2 Postmortem history

If the option **GENHISTORY** was set ON when your program was compiled, the run-time system dumps a procedure call stack into a file called `errinfo.$$$`,

[1] We recommend to set them in the configuration file or a project file.

[2] In Modula-2 it has to return `NIL` if failed to allocate a memory block.

which may then be read by the HIS utility to print each item with

- a file name
- a line number
- a program counter value
- a procedure name (sometimes)

Note: all modules constituting your program should be compiled with the option **LINENO** set ON.

To print the history, RTS scans the stack of the coroutine that caused an exception and tries to find procedure calls. This is not a trivial task because of the highly optimized code generated by the compiler. For example, not all procedures have a stack frame.

For each pointer to the code segment on the stack RTS checks the previous command. If this command is a call command, it assumes that this is a procedure call. It is unlikely that RTS misses a procedure call, but it can be cheated by something that looks like a procedure call. As a rule, it is caused by uninitialized local variables, especially character arrays.

The first line of the history is always correct. For each line, except the first one, we recommend to check that the procedure shown in the previous line is called from the given line.

From the other hand, if you turn the **GENFRAME** option on, the code will be a bit slower, but RTS will scan stack frames of the procedures and the history will show *absolutely* correct addresses and line numbers. Procedure names are almost always valid except the case of lack of debug information in some modules - probably compiled by foreign compilers or by XDS with not all debug flags set. So you should not rely on procedure names hard.

Turning the **GENHISTORY** option ON does *not* slow down your code, as it only adds an extra call to the initialization routine. It should be done when you compile the main module of your program, in its header, compiler command line, or project (we recommend the last approach).

The following example shows a sketch of a program and the procedure stack:

```
PROCEDURE P1;
  (* uninitialized variable: *)
  VAR x: ARRAY [0..50] OF INTEGER;
```

```
    BEGIN
      i:=i DIV j;   (* line 50 *)
    END P1;

    PROCEDURE P2;
    BEGIN
      i:=i DIV j;   (* line 100 *)
    END P2;

    PROCEDURE P3;
    BEGIN
      P1;           (* line 150 *)
    END P3;

#RTS: No exception handler #6: zero or negative divisor
-------------------------------------------------------------
Source file                          LINE  OFFSET  PROCEDURE
-------------------------------------------------------------
"test.mod"                            50 000000DE
"test.mod"                           100 0000024C
"test.mod"                           150 0000051D
```

It is obvious from the source text that the procedure `P1` cannot be called from `P2`. The second line is superfluous.

9.3 The oberonRTS module

The run-time support (RTS) is an integral part of the Oberon-2 language implementation. It includes command activation, memory allocation, garbage collection and meta-language facilities. The module **oberonRTS** (written in Modula-2) provides an interface to these features.

9.3.1 Types and variables

```
    TYPE
      Module;  (* run-time data structure for a module *)
      Type;    (* run-time data structure for a data type *)
      Command = PROC; (* parameterless procedure *)
      CARDINAL = SYSTEM.CARD32;
```

```
VAR
  nullModule: Module; (* Null value of type Module *)
  nullType: Type;     (* Null value of type Type *)
```

9.3.2 Garbage collection

Collect	*Garbage Collector*

```
PROCEDURE Collect;
```

Invokes the garbage collector.

GetInfo	*Get Memory Information*

```
PROCEDURE GetInfo(VAR objects, busymem: CARDINAL);
```

Returns the number of allocated objects and the total size of the allocated memory.

9.3.3 Object finalization

A system with garbage collection has some specific features. Its main difference from systems without garbage collection is that deallocation of any system resource must be postponed until garbage collection. For example, let some data structure contain descriptors of open files. To close a file (i.e. to destroy its descriptor), one needs to know that there are no references to that file. This information becomes known only in the course of garbage collection. The same argument also holds for other kinds of resources.

One immediate implication is that there must be some *finalization* mechanism: the ability to perform certain operations with an object when there are no more references to it.

XDS allows a finalization procedure to be attached to any dynamically allocated object.

Finalizer	*Type of a finalization procedure*

```
TYPE Finalizer = PROCEDURE (SYSTEM.ADDRESS);
```

InstallFinalizer *Set a finalizer to an object*

```
PROCEDURE InstallFinalizer(f: Finalizer;
                           obj: SYSTEM.ADDRESS);
```

The procedure sets the finalization procedure `f` for the object `obj`. That procedure will be called when the object becomes unreachable.

Note: a finalizer is called on the GC stack (stack size is limited).

Example

```
TYPE
  Obj = POINTER TO ObjDesc;
  ObjDesc = RECORD
    file: File; (* file handler *)
  END;

PROCEDURE Final(x: SYSTEM.ADDRESS);
  VAR o: Obj;
BEGIN
  o:=SYSTEM.CAST(Obj,x);
  IF o.file # NIL THEN Close(file) END;
END Final;

PROCEDURE Create(): Obj;
  VAR o: Obj;
BEGIN
  NEW(o);
  o.file:=NIL;
  oberonRTS.InstallFinalizer(Final,o);
  TryOpen(o.file);
END Create;
```

9.3.4 Meta-language facilities

The meta-programming operations can be used to retrieve the type of an object, to create an object of the given type, to get the name of a type and a type by its name, etc.

Search *Search a Module by its Name*

```
PROCEDURE Search(name: ARRAY OF CHAR): Module;
```

Returns a module by its `name` or `nullModule`.

NameOfModule *Name of Module*

```
PROCEDURE NameOfModule(m: Module;
                VAR name: ARRAY OF CHAR);
```

Returns the `name` of the `Module`.

ThisCommand *Get Command by its Name*

```
PROCEDURE ThisCommand(m: Module;
                  name: ARRAY OF CHAR;
                    ): Command;
```

Returns the command (parameterless procedure) named "`name`" in the module `m` or `NIL`, if the command does not exist.

ThisType *Get Type by its Name*

```
PROCEDURE ThisType(m: Module;
                name: ARRAY OF CHAR): Type;
```

Returns the type named "`name`" declared in the module `m` or `nullType`, if there is no such type.

SizeOf *Size of Type*

```
PROCEDURE SizeOf(t: Type): INTEGER;
```

Returns the size (in bytes) of an object of the type `t`.

BaseOf *Base of Type*

```
PROCEDURE BaseOf(t: Type; level: INTEGER): Type;
```

Returns the *level*-th base type of `t`.

LevelOf *Level of Type Extension*

```
PROCEDURE LevelOf(t: Type): INTEGER;
```

Returns a level of the type extension.

ModuleOf *Module of Type*

```
PROCEDURE ModuleOf(t: Type): Module;
```

Returns the module in which the type `t` was declared.

NameOfType *Name of Type*

```
PROCEDURE NameOfType(t: Type; VAR name: ARRAY OF CHAR);
```

Returns the `name` of the record type `t`.

TypeOf *Type of Object*

```
PROCEDURE TypeOf(obj: SYSTEM.ADDRESS): Type;
```

Returns the type of the object `obj`.

NewObj *Create Object*

```
PROCEDURE NewObj(type: Type): SYSTEM.ADDRESS;
```

Creates a new object of the type `type`.

9.3.5 Module iterators

The module `oberonRTS` provides procedures which can be used to iterate all loaded modules, all commands, and all object types (i.e., exported record types).

NameIterator *Iterator Type*

```
TYPE
  NameIterator = PROCEDURE (
                   (*context:*) SYSTEM.ADDRESS,
                   (*name:*) ARRAY OF CHAR
                 ): BOOLEAN;
```

A procedure of type `NameIterator` is called by an iterator on each iterated item. An iterator passes the name of the item along with the so-called *context* word. This allows some context information to be passed to the user-defined procedure (e.g., a file handler). If the procedure returns FALSE, the iteration is terminated.

IterModules *Iterate all Modules*

```
PROCEDURE IterModules(context: SYSTEM.ADDRESS;
                      iter: NameIterator);
```

The procedure iterates all Oberon-2 modules.

IterCommands *Iterate Commands*

```
PROCEDURE IterCommands(mod: Module;
                   context: SYSTEM.ADDRESS;
                      iter: NameIterator);
```

Iterates all commands implemented in the module `mod`.

IterTypes *Iterate Record Types*

```
PROCEDURE IterTypes(mod: Module;
                  context: SYSTEM.WORD;
                     iter: NameIterator);
```

Iterates all record types declared in the module `mod`.

Chapter 10

Multilanguage programming

XDS allows you to mix Modula-2, Oberon-2, C, and Assembler modules, libraries, and object files in one project.

10.1 Modula-2 and Oberon-2

It is not necessary to notify the compiler of using Modula-2 objects in Oberon-2 module and vice versa. The compiler will detect the language automatically when processing symbol files on `IMPORT` clause.

10.1.1 Basic types

In Oberon-2 the basic types have the same length on all platforms. In Modula-2 the size of types `INTEGER`, `CARDINAL` and `BITSET` may be different and depends on the value of the **M2BASE16** option. The following table summarizes the correspondence between the basic types.

Type	Size	Oberon-2	Modula-2	
			M2BASE16+	M2BASE16-
integer	8	SHORTINT	—	—
integer	16	INTEGER	INTEGER	—
integer	32	LONGINT	—	INTEGER
cardinal	8	—	—	—
cardinal	16	—	CARDINAL	—
cardinal	32	—	—	CARDINAL
bitset	16	—	BITSET	—
bitset	32	SET	—	BITSET

The system types `INT` and `CARD` correspond to Modula-2 `INTEGER` and `CARDINAL` types respectively. We recommend to use `INT` and `CARD` in Oberon-2 when importing Modula-2 modules. For example, if the procedure `Foo` is defined in the Modula-2 definition module `M` as

```
DEFINITION MODULE M;

PROCEDURE Foo(VAR x: INTEGER);

END M.
```

its portable usage in Oberon-2 is as follows:

```
VAR x: SYSTEM.INT;
   .  .  .
  M.Foo(x);
```

10.1.2 Data structures

XDS allows any Modula-2 data structures to be used in Oberon-2 modules, even those that can not be defined in Oberon-2 (e.g. variant records, range types, set types, enumerations, etc).

However, usage of Modula-2 types in Oberon-2 and vice versa is restricted. Whenever possible XDS tries to produce the correct code. If a correct translation is impossible, an error is reported:

- a Modula-2 record field type cannot be of an Oberon-2 pointer, record or array type;

- a Modula-2 pointer to an Oberon-2 record cannot be used in specific Oberon-2 constructs (type-bound procedures, type guards, etc);
- an opaque type can not be defined as an Oberon pointer.

Standard procedures `NEW` and `DISPOSE` are always applied according to the language of a parameter's type. For example, for the following declarations in an Oberon-2 module:

```
TYPE
  Rec = RECORD END;
  MP  = POINTER ["Modula"] TO Rec; (* Modula pointer *)
  OP  = POINTER TO Rec;     (* Oberon pointer *)
VAR
  m: MP;
  o: OP;
```

the call `NEW(m)` will be treated as a call to the Modula-2 default `ALLOCATE`, while `NEW(o)` will be treated as a call of the standard Oberon-2 run-time routine. See also 10.2.

Implicit memory deallocation (garbage collection) is applied to Oberon-2 objects only. If a variable of a Modula-2 pointer type is declared in an Oberon-2 module, it shall be deallocated explicitly.

Example: Using the Modula data type in Oberon

```
(* Modula-2*) DEFINITION MODULE m2;
TYPE
  Rec = RECORD  (* a record with variant parts *)
    CASE tag: BOOLEAN OF
      |TRUE:  i: INTEGER;
      |FALSE: r: REAL;
    END;
  END;
  Ptr = POINTER TO Rec;

VAR
  r: Rec;
  p: Ptr;
```

```
PROCEDURE Foo(VAR r: Rec);

END m2.

(* Oberon-2 *) MODULE o2;

IMPORT m2; (* import of a Modula-2 module *)

VAR
  r: m2.Rec;  (* using the Modula-2 record type *)
  p: m2.Ptr;  (* using the Modula-2 pointer type *)
  x: POINTER TO m2.Rec;

BEGIN
  NEW(p);     (* Modula-2 default ALLOCATE *)
  NEW(x);     (* Oberon-2 NEW *)
  m2.Foo(r);
  m2.Foo(p^);
  m2.Foo(x^);
END o2.
```

10.1.3 Garbage collection

It is important to remember that Modula-2 and Oberon-2 have different approaches to memory utilization. When a program contains both Modula-2 and Oberon-2 modules, garbage collection is used. See 9.1 for more information.

10.2 Direct language specification

The compiler must know the implementation language of a module to take into account different semantics of different languages and to produce correct code.

In some cases, it is necessary for a procedure or data type to be implemented according to the rules of a language other than that of the whole module. In XDS, it is possible to explicitly specify the language of a type or object. *Direct language specification (DLS)* is allowed either if language extensions are enabled or if the module `SYSTEM` is imported.

In a record, pointer, or procedure type declaration, or in a procedure declaration,

the desired language (or, more precisely, the way in which that declaration is treated by the compiler) can be specified as `"[" language "]"` immediately following the keyword `RECORD`, `POINTER`, or `PROCEDURE`. `language` can be a string or integer constant expression[1]:

Convention	**String**	**Integer**
Oberon-2	`"Oberon"`	`0`
Modula-2	`"Modula"`	`1`
C	`"C"`	`2`
Pascal	`"Pascal"`	`5`
Win32 API	`"StdCall"`	`7`
OS/2 API	`"SysCall"`	`8`

Examples:

```
TYPE
  UntracedPtr = POINTER ["Modula"] TO Rec;
```

Here `UntracedPtr` is defined as a Modula-2 pointer, hence all variables of that type will not be traced by garbage collector.

```
PROCEDURE ["C"] sig_handler (id : SYSTEM.int);
 .  .  .
  signal.signal(signal.SYSSEGV, sig_handler);
```

Here `sig_handler` has C calling and naming conventions, so it can be installed as a signal handler into C run-time support.

A direct language specification clause placed after a name of a field, constant, type, or variable points out that the name of the object will be treated according to the rules of the specified language.

```
TYPE
  Rec ["C"] = RECORD
    name ["C"]: INTEGER;
  END;

CONST pi ["C"] = 3.14159;

VAR buffer[]["C"]: POINTER TO INTEGER;
```

[1] We recommend to use strings, integer values are preserved for backward compatibility.

Note: In ISO Modula-2, an absolute address may be specified for a variable after its name in square brackets, so the empty brackets are required in the last line.

A procedure name is treated according to the language of its declaration, so in the following declaration:

```
PROCEDURE ["C"] Foo;
```

both the procedure type and the procedure name are treated according to the C language rules. **Note:** If you are using a C++ compiler, the `Foo` function should be declared with C name mangling style. Consult your C++ manuals for further information.

10.3 Interfacing to C

Special efforts were made in XDS to provide convenient interface to other languages, primarily to the C language. The main goal is to allow direct usage of existing C libraries and APIs in Modula-2/Oberon-2 programs.

10.3.1 Foreign definition module

A direct language specification (see 10.2) clause may appear immediately after keywords `DEFINITION MODULE`. The effect is that all objects defined in that module are translated according to the specified language rules, thus making unnecessary direct language specifications for each object.

Several options are often used in foreign definition modules.

Example

```
<*+ M2EXTENSIONS *>
<*+ CSTDLIB *>      (* C standard library *)
<*+ NOHEADER *>     (* we already have header file *)
DEFINITION MODULE ["C"] string;

IMPORT SYSTEM;

PROCEDURE strlen(s: ARRAY OF CHAR): SYSTEM.size_t;
PROCEDURE strcmp(s1: ARRAY OF CHAR;
                 s2: ARRAY OF CHAR): SYSTEM.int;
```

```
END string.
```

Take the following considerations into account when designing your own foreign definition module:

- If you are developing an interface to an existing header file, use the **NOHEADER** option to disable generation of the header file. This option is meaningful for translators only.
- If the header file is a standard header file, use the **CSTDLIB** option. This option is meaningful for the translators only.
- Use the special `SYSTEM` types `int`, `unsigned`, `size_t`, and `void` for corresponding C types.
- XDS compilers use relaxed type compatibility rules for foreign entities. See 10.4 for more information.

10.3.2 External procedures specification

In some cases, it may be desirable not to write a foreign definition module but to use some C or API functions directly. XDS compilers allow a function to be declared as external.

The declaration of an external procedure consists of a procedure header only. The procedure name in the header is prefixed by the symbol `"/"`.

```
PROCEDURE ["C"] / putchar(ch: SYSTEM.int): SYSTEM.int;
```

10.4 Relaxation of compatibility rules

The compiler performs all semantic checks for an object or type according to its language specification. Any object declared as that of Modula-2 or Oberon-2 is subject to Modula-2 or Oberon-2 compatibility rules respectively. The compiler uses relaxed compatibility rules for objects and types declared as `"C"`, `"Pascal"`, `"StdCall"`, and `"SysCall"`.

10.4.1 Assignment compatibility

Two pointer type objects are considered assignment compatible, if

- they are of the same Modula-2 or Oberon-2 type.
- at least one of their types is declared as `"C"`, `"Pascal"`, `"StdCall"`, or `"SysCall"`, and their *base types* are the same.

```
VAR
  x: POINTER TO T;
  y: POINTER TO T;
  z: POINTER ["C"] TO T;
BEGIN
  x := y;       -- error
  y := z;       -- ok
  z := y;       -- ok
```

10.4.2 Parameter compatibility

For procedures declared as `"C"`, `"Pascal"`, `"StdCall"`, or `"SysCall"`, the type compatibility rules for parameters are significantly relaxed:

If a formal value parameter is of the type declared as `POINTER TO T`, the actual parameter can be of any of the following types:

- the same type (the only case for regular Modula-2/Oberon-2 procedures);
- another type declared as `POINTER TO T`.
- any array type which elements are of type `T`. In this case the address of the first array element is passed, as it is done in C.
- the type `T` itself, if `T` is a record type. In this case the address of the actual parameter is passed.

If a formal parameter is an open array of type `T`, the actual parameter can be of any of the following types:

- an (open) array of type `T` (the only case for regular Modula-2/Oberon-2 procedures);
- type ́verb'T' itself (if **M2EXTENSIONS**option is set ON);
- any type declared as `POINTER TO T`.

This relaxation, in conjunction with the `SYSTEM.REF` function procedure (see 7.5.2), simplifies Modula-2/Oberon-2 calls to C libraries and the target operating system API, preserving the advantages of the type checking mechanism provided by that languages.

Example

```
TYPE
  Str = POINTER TO CHAR;
  Rec = RECORD ... END;
  Ptr = POINTER TO Rec;

PROCEDURE ["C"] Foo(s: Str); ... END Foo;
PROCEDURE ["C"] Bar(p: Ptr);  ... END Bar;
PROCEDURE ["C"] FooBar(a: ARRAY OF CHAR);  ... END FooBar;

VAR
  s: Str;
  a: ARRAY [0..5] OF CHAR;
  p: POINTER TO ARRAY OF CHAR;
  R: Rec;
  A: ARRAY [0..20] OF REC;
  P: POINTER TO REC;

  Foo(s);    (* allowed - the same type *)
  Foo(a);    (* allowed for the "C" procedure *)
  Foo(p^);   (* allowed for the "C" procedure *)
  Bar(R);    (* the same as Bar(SYSTEM.REF(R)); *)
  Bar(A);    (* allowed for the "C" procedure *)
  Bar(P);    (* allowed for the "C" procedure *)
  FooBar(s); (* allowed for the "C" procedure *)
```

10.4.3 Ignoring function result

It is a standard practice in C programming to ignore the result of a function call. Some standard library functions are designed taking that practice into account. E.g. the string copy function accepts the destination string as a variable parameter (in terms of Modula-2) and returns a pointer to it:

```
extern char *strcpy(char *, const char *);
```

In many cases, the result of the `strcpy` function call is ignored.

In XDS, it is possible to ignore results of functions defined as `"C"`, `"Pascal"`, `"StdCall"`, or `"SysCall"`. Thus, the function `strcpy` defined in the `string.def` foreign definition module as

```
PROCEDURE ["C"] strcpy(VAR d: ARRAY OF CHAR;
                           s: ARRAY OF CHAR): ADDRESS;
```

can be used as a proper procedure or as function procedure:

```
strcpy(d,s);
ptr:=strcpy(d,s);
```

10.5 Configuring XDS for a C Compiler

Different C compilers have different naming and calling conventions. If you use C functions or libraries in your projects, you have to specify your C compiler using the **CC** equation in order to have all C functions to be called in a way compatible with that compiler. The compiler also sets the default values of some other options and equations according to the value of the **CC** equation.

For Linux XDS supports the GCC (ELF) compiler. Therefore, the **CC** equation has to be set to `"GCC"`, written in any case. If the equation value is not set, `"GCC"` is assumed by default.

Alignment of data structures is controlled by the **ALIGNMENT** equation.

ATTENTION! Libraries included in XDS distribution are built via GCC. Since GCC usually produces aligned code, the **ALIGNMENT** equation has to be set to 4. Setting it to other values may cause unpredictable results. Don't change it unless you exactly know what you are doing!

Names in an object file produced by a C compiler may have leading underscore. If you are going to use C modules and libraries, you have to force XDS to use the same naming rules. To do this, turn the **GENCPREF** option ON in the foreign definition modules:

```
<* +GENCPREF *>
DEFINTION MODULE ["C"] stdio;
```

Since GCC (ELF) produces no underscore prefixes you should not turn this option ON.

10.5.1 Possible problems

To use a C function or a data type from Modula-2 or Oberon-2 you have to express its type in one of these languages. Usually it is done in a foreign definition module (See 10.3). The current version of XDS does not support all calling conventions, so direct usage of some functions is not possible, namely:

- functions with a parameter of a structured type, passed by value, e.g.:

```
void foo(struct MyStruct s);
```

- functions that return structured types, e.g.:

```
struct MyStruct foo(void)
```

- C functions with Pascal calling convention that return a real type.

Both Modula-2 and C/C++ have exception handling and finalization facilities. Unpredictable results may occur if you try to utilize that facilities from both languages in one program.

Chapter 11

Optimizing a program

It sometimes happens with almost all compilers that the unoptimized version of a program works properly, but the optimized one does not or vice versa. If the compiler has a dozen of optimization control options it may be extremely difficult to test the compiler itself. The compiler manufacturer has to check all possible combinations of options. Fortunately, this is not the case with XDS.

Unlike many other compilers, XDS performs optimizations by default. Most of them may be turned off by setting the **NOOPTIMIZE** option ON. However, the code generator always performs some low-level optimizations[1]. Instruction scheduling can be turned on or off using the **DOREORDER** option. The last option that implicitly disables some of optimizations is the **GENDEBUG** option.

There are still several ways to control the generated code. First of all, you have to choose what is more important for you: performance or compactness. By default, the option **SPACE** is set OFF, forcing the compiler to favor the code effeciency.

To get the maximum performance, do the following:

- turn **GENFRAME** off
- turn **SPACE** off
- turn **GENDEBUG** off
- turn **NOOPTIMIZE** off
- turn **DOREORDER** on
- set **CPU** and **MINCPU** equations according to your target

[1] As a result, disabling optimizations optimizer significantly *slows down* the complier

- turn run-time checks and overflow checks off

It is possible not to turn run-time checks off in the product versions of your programs, because the code generator usually removes redundant checks. A typical program runs only 10-15% faster with all run-time checks turned off (but the code size is usually significantly smaller).

Two options should be used with care:

- the **PROCINLINE** option allows the compiler to expand procedures inline. As a rule, switching the option ON leads to faster but bigger code. However, the effect of this option depends on your programming style (size of procedures, etc).
- the **NOPTRALIAS** option allows the compiler to assume that there is no pointer aliasing, i.e. there are no pointers bounded to non-structure variables. The code quality is better if the option is ON.

Example of project file for maximum performance

```
-alignment=4          % is unnecessary under Linux
-noptralias+
-procinline+
-space
-doreorder+
-cpu=486
-genframe

-checkindex
-checkrange
-checknil
-ioverflow
-coverflow

-gendebug
-genhistory
-lineno
!module Foo.mod
```

In some cases, it may be better to set different options for different modules in your program. See `dry.mod` from XDS samples.

Chapter 12

Low-level programming

12.1 Data representation

The internal representation of values of Modula-2 and Oberon-2 basic types is described in the tables 12.1 and 12.2. In the table 12.3 the representation of system types is described.

12.1.1 Modula-2 INTEGER and CARDINAL types

If the option **M2BASE16** is OFF, objects of types `INTEGER` and `CARDINAL` are 4 bytes (32 bits) long, otherwise they are 2 bytes (16 bits) long.

12.1.2 Modula-2 BOOLEAN type

A value of the type `BOOLEAN` occupies 1 byte of memory.

12.1.3 Modula-2 enumeration types

Representation of enumeration type values depends on the current **ENUMSIZE** equation setting. Values of an enumeration type which fits the specified size (1, 2, or 4 bytes) occupy exactly that number of bytes; otherwise the smallest suitable size from that list is taken.

Modula-2 type	Bits	Representation
SHORTINT	8	signed
INTEGER	16/32	signed (See 12.1.1)
LONGINT	32	signed
SHORTCARD	8	unsigned
CARDINAL	16/32	unsigned (See 12.1.1)
LONGCARD	32	unsigned
CHAR	8	unsigned
BOOLEAN	8/32	unsigned (See 12.1.2)
		0 for FALSE, 1 for TRUE
subranges		according to the base type
REAL	32	80x87 single-precision data format
LONGREAL	64	80x87 double-precision data format
LONGLONGREAL	80	80x87 extended-precision data format

Table 12.1: Representation of Modula-2 basic types

Oberon-2 type	Bits	Representation
SHORTINT	8	signed
INTEGER	16	signed
LONGINT	32	signed
CHAR	8	unsigned
BOOLEAN	8	unsigned byte
		0 for FALSE, 1 for TRUE
REAL	32	80x87 single-precision data format
LONGREAL	64	80x87 double-precision data format
LONGLONGREAL	80	80x87 extended-precision data format
SET	32	packed set

Table 12.2: Representation of Oberon-2 basic types

System type	Bits	Representation
ADDRESS	32	unsigned
BOOL8	8	unsigned
BOOL16	16	unsigned
BOOL32	32	unsigned
BYTE	8	unsigned
CARD8	8	unsigned
CARD16	16	unsigned
CARD32	32	unsigned
INT8	8	signed
INT16	16	signed
INT32	32	signed
LOC	8	unsigned
WORD	32	ARRAY [0..3] OF LOC

Table 12.3: Representation of SYSTEM types

12.1.4 Modula-2 set types

Sete are represented as bit arrays. The **SETSIZE** equation specifies the default size for small sets (1, 2, or 4 bytes).

If the option **M2BASE16** is OFF, the type BITSET is represented by 32 bits, otherwise by 16 bits.

12.1.5 Pointer, address, and opaque types

The XDS compiler allocates 4 bytes of storage for a value of a pointer, address, or opaque type. Address arithmetic is implemented as 32-bit unsigned arithmetic without overflow checks.

12.1.6 Procedure types

Procedure types are represented by 4 bytes which hold an address of a procedure entry point in the task code segment.

12.1.7 Record types

Records are represented by a continuous memory segment containing all record components (fields) in a representation corresponding to their types. The compiler aligns each field according to its size and the current alignment (1,2,4, or 8), which may be set with the **ALIGNMENT** equation. Fields, which sizes, being rounded to the nearest power of 2, are less or equal to the current alignment, are placed at offsets which are multiple of their (rounded) sizes. Offsets of all other fields are multiples of the current alignment. Variant parts are aligned at the largest alignment of variant fields. Size of a record is rounded so that size of an array of such records is a multiple of the record size and the number of elements in the array, and each record in the array is correctly aligned.

```
TYPE
  R1 = RECORD                  (* ALIGNMENT   1   2   4  *)
         f1: CHAR;             (* f1 offset   0   0   0  *)
         f2: SYSTEM.CARD16;    (* f2 offset   1   2   2  *)
         f3: SYSTEM.CARD16;    (* f3 offset   3   4   4  *)
         f4: CARDINAL;         (* f4 offset   5   6   8  *)
         f5: CHAR;             (* f1 offset   9   10  12 *)
       END;                    (* SIZE(R1)    10  12  16 *)
```

12.1.8 Array types

An array is represented by a continuous memory segment containing all array elements in a representation corresponding to their type.

Note that elements within an array can be aligned, so in general for

```
TYPE A = ARRAY [0..N-1] OF T;
```

`SIZE(A)` may be not equal to `SIZE(T) * N`.

Open arrays, as well as procedure formal parameters of type `ARRAY OF ... ARRAY OF T`, are represented by an open array descriptor. For an N-dimensional open array, the descriptor is an array of $2N$ 32-bit elements, which are:

- the first element is the address of the array data
- the second is the highest dimension

- for each of $N - 1$ higher dimensions, the descriptor contains the size in bytes of a next dimension array and a number of elements in the dimension

Let `A` be a dynamic 3-dimensional array of `INTEGER` (`SIZE(INTEGER)=2` in Oberon-2) created as

```
NEW(A,4,3,6)
```

then its descriptor is a 6-element array containing:

```
#0:  Address of array itself
#1:   6
#2:  12   (6*2)
#3:   3
#4:  36   (12*3)
#5:   4
```

12.2 Sequence parameters

The array of bytes which is passed to a procedure in place of a formal SEQ-parameter is formed as follows:

- values of all actual parameters forming the sequence are represented as described below and concatenated in the array in their textual order
- integer values are converted to `LONGINT`
- `BOOLEAN`, `CHAR`, cardinal, and enumeration values are converted to `LONGCARD`
- range type values are converted according to their base type
- real values are converted to `LONGREAL`
- pointer, address, opaque, and procedure type values are converted to `ADDRESS`
- a structured value (record or array) is interpreted as a one-dimensional array of bytes and is represented by a 3-element descriptor:
  - the address of the structure
  - a zero 32-bit word (reserved for future extensions)
  - size of the structure (in LOCs) minus one

Example

```
PROCEDURE write(SEQ args: SYSTEM.BYTE);
BEGIN
END write;

VAR i: INTEGER;
    c: SYSTEM.CARD8;
    r: LONGREAL;
    S: RECORD a: LONGINT; c: CHAR END;
    p: POINTER TO ARRAY OF CHAR;
 .  .  .

write(i,c,S,r,p^);
```

For this call the actual byte array passed to `write` will contain:

- 4 bytes of the sign-extended value of `i`
- 4 bytes of the zero-extended value of `c`
- 12 bytes of the array descriptor
  - 4 bytes containing the address of `S`
  - 4 bytes containing 0
  - 4 bytes containing 4 (`SIZE(S)-1`)
- 8 bytes value of `r` in the double-precision 80387 format
- 12 bytes of the array descriptor
  - 4 bytes containing the address of the `P` data
  - 4 bytes containing the value 0
  - 4 bytes containing `SIZE(p^)-1`

12.3 Calling and naming conventions

The calling and naming conventions for Modula-2, Oberon-2, and foreign procedures are described in this section.

12.3.1 General considerations

All parameters are always passed on the stack. The number of bytes occupied by a parameter is a multiple of 4. High-order bytes of parameters which are of shorter types (e.g. `CHAR`, `SYSTEM.CARD16`) are undefined.

Value parameters of scalar types (boolean, character, enumeration, whole, range, real, pointer, opaque, and procedure) and sets of size not greater than 32 bit are placed onto the stack. A complex type value parameter is passed as a pair of real.

Value parameters of all other types (even an array of a single `CHAR`) are passed by reference. A procedure is responsible for copying its non-scalar value parameter onto the stack, unless it is marked as read-only.

Warning: In C, a *caller* should copy value parameters of structure type onto the stack. You should provide a wrapper C function which receives these parameters by reference. Fortunately, this is a very rare case.

Note: The number of 4-byte words pushed onto the stack is passed in the AL register to a `"SysCall"` foreign procedure.

12.3.2 Open arrays

For an N-dimensional open array parameter $N+1$ parameters are actually passed — the address of the array and its sizes in all dimensions from left to right. This is true for Modula-2 and Oberon-2 procedures only. In case of a foreign procedure, only the address is passed.

12.3.3 Oberon-2 records

To a formal VAR-parameter which type is an Oberon-2 record type, the address of the actual parameter and the address of its dynamic type descriptor are passed.

12.3.4 Result parameter

If a function procedure result type is not scalar, it receives one extra parameter — the address of a temporary variable in which the procedure should store the reslut. **Note:** This may be incompatible with C.

A complex result is returned as a record with two real fields.

12.3.5 Nested procedures

A nested Modula-2 or Oberon-2 procedure, which access scopes of outer procedures, receives their *bases* as extra parameters. More precisely, the procedure P receives bases of all outer procedures which scopes are accessed by P or any procedure nested in P.

The base of a procedure is the address at which the procedure's return address resides on the stack.

12.3.6 Oberon-2 receivers

An extra parameter — receiver — is passed to an Oberon-2 type bound procedure. A reference to its dynamic type descriptor is also passed if the receiver is declared as a VAR-parameter.

12.3.7 Sequence parameters

Sequence parameters for a Modula-2/Oberon-2 procedure are collected into a temporary variable, which is then passed as an `ARRAY OF BYTE` (i.e. its address and size are passed). For foreign procedures, a C-compatible approach is used — parameters are pushed onto the stack. In either case, all ordinal type parameters are extended to 4 bytes, `REAL`s to `LONGREAL`s, non-scalar type parameters are passed by reference.

12.3.8 Order of parameters

The abstract order of parameters (all categories are optional):

- address of a temporary variable to store the result (see 12.3.4)
- bases of outer procedures (see 12.3.5)
- receiver (see 12.3.6) (Oberon-2 procedures only)
- regular parameters
- sequence parameter (see 12.3.7)

Actual order, in all cases except "Pascal" foreign procedures, is from-right-to-left, i.e. the last sequence parameter is pushed onto the stack first, the result parameter is pushed last.

12.3.9 Stack cleanup

The stack space allocated for parameters has to be freed upon return from a procedure. Depending on the language of the procedure, it is performed by the caller ("C" and "SysCall") or the procedure itself (Modula-2, Oberon-2, "StdCall", "Pascal").

12.3.10 Register usage

A procedure must preserve reisters ESI, EDI, EBP, and EBX registers, keep ES=DS, and clear the D flag.

The FPU stack must be empty before a call to a procedure and upon return from it. Exceptions are procedures which return REAL or LONGREAL. In this case, the result is placed in ST(0).

Note: If the **CC** equation is set to either "WATCOM" or "SYMANTEC", foreign procedures declared as "C" are considered to return REAL results in EAX, and LONGREAL results in EAX (low order bytes) and EDX (high order bytes).

12.3.11 Naming conventions

External names of exported procedures in object modules are built accroding to the following rules:

Convention	Name is	As in
"Modula"	prepended with the module name and "_"	Module_Proc
"Oberon"	ditto	Module_Proc
"C"	prepended with "_" (see note)	_Proc
"Pascal"	capitalized	PROC
"StdCall"	unchanged	Proc
"SysCall"	unchanged	Proc

Note: If the **CC** equation is set to "WATCOM", external names of "C" foreign procedures are not prepended with an underscore character.

Chapter 13

Inline assembler

This chapter contains a very brief description of the inline x86 assembler.

13.1 Implemented features

The following features are implemented in the current version:

- Base instruction set up to and including Pentium Pro (see 13.5).
- Floating point instructions up to and including Pentium Pro.
- Pentium MMX instructions.
- Labels and their usage in branch and call instructions.
- Modula-2/Oberon-2 procedure calls and variable access (see 13.4).

13.2 Basic syntax

The assembler uses the same scanner as the Modula-2/Oberon-2 front-end, so it is possible to use conditional compilation (see 7.7.2) and comments.

Language extensions have to be enabled using the **M2EXTENSIONS** and **O2EXTENSIONS** options in order to use inline assembly facilities.

The keyword `ASM` denotes the beginning of inline assembly code; the keyword `END` denotes its end.

Each line in a piece of the assembly code may contain not more than one instruction. It is not possible to continue an instruction on the next string.

Keywords and names of instructions and registers are not case sensitive.

There are instructions one of which arguments is fixed (`DIV`, `FCOMI`). These instructions are differently denoted in different assemblers. We use the syntax described in Intel's documentation.

If size or an operand may not be determined based on instruction semantics and/or size of another operand, it is necessary to explicitly specify it. The following size specifiers are recognized:

Specifier	Size
`BYTE PTR`	1
`WORD PTR`	2
`DWORD PTR`	4
`FWORD PTR`	6
`QWORD PTR`	8
`TBYTE PTR`	10

Examples:

`MOV WORD PTR [EBX],1`	here size specifier is obligatory
`MOV [EBX],AX`	here size is determined automatically

13.3 Labels

An instruction may be prepended with a label, delimited with a colon character:

```
Save: PUSH EAX
```

Labels may not match instruction names. It is also not recommended to use assembly keywords (`DWORD`, `EAX`) as labels.

In the current version, labels may only be used in branch and call instructions.

13.4 Accessing Modula-2/Oberon-2 objects

It is possible to reference Modula-2/Oberon-2 entities from within the inline assembly code, namely whole constants, variables, and procedures (in `JMP` and `CALL` instructions only).

Example:

```
MOV j,10
```

In this example, the type of `j` is used to choose between byte, word, and double-word `MOV` instructions.

Note: In the first pre-release versions which included inline assembler, it was necessary to specify the base register `EBP` to access local variables:

```
MOV j[EBP],10
```

It is *no longer required*.

In case of nested procedures, code to access a variable from an outer procedure scope has to be written by hand.

Record field access is not supported yet. There are also no operators to denote attributes of Modula-2/Oberon-2 entities.

The `OFFSET` operator returns the offset of its operand, which has to be a variable:

```
MOV EAX, OFFSET j
```

13.5 Known problems

1. Instruction prefixes (`REP`, `LOCK`, etc.) are not supported yet.
2. Segment overriding (`DS:`, etc.) is not supported yet.
3. Error position precision is $+/-$ 1-2 tokens.
4. Error 3029 incorrectly positioned.
5. 16-bit addressing modes (e.g. `[BX+SI]`) are not supported and unlikely to be supported in the future.
6. Modula-2/Oberon-2 entities access facilities are limited.
7. It is possible to use commands like `MOVSB`, `MOVSW`, `MOVSD`, but not as in `MOVS DWORD PTR [ESI]`.
8. Modula-2/Oberon-2 variables usage is poorly checked for correctness.
9. It is possible to use only one `OFFSET` operator in a constant expression.

13.6 Potential problems

1. Not all instructions were tested with all addressing modes.
2. Error diagnostics and recovery were not tested.

Appendix A

Limitations and restrictions

There are some limitations and restrictions in implementation of both Modula-2 and Oberon-2 compilers.

Length of identifiers

The length of an identifier is at most 127 characters.

Length of literal strings

The length of a literal string is at most 256 characters. Longer strings may be constructed using the string concatenation operator (See 7.2.4).

Record extension hierarchy

The depth of a record extension hierarchy is at most 15 extensions.

Unimplemented ISO libraries

Unimplemented Oakwood libraries

The following Oberon-2 Oakwood library modules are not available in the current release:

Input	Keyboard and pointer device access
Files	File input/output, riders
XYPlane	Elementary pixel plotting

Coroutines

The current release provides a restricted implementation of the system module **COROUTINES**: the interrupt requests are not detected.

Dynamic loader

The Oberon-2 dynamic loading facility is not provided in the current release.

Bibliography

[MöWi91] H.Mössenböck, N.Wirth. The Programming Language Oberon-2. Structured Programming,1991, 12, 179-195.

[PIM] N.Wirth. Programming in Modula-2. 4th edition. Springer-Verlag, 1988. ISBN 0-387-50150-9.

[Wirth88] N.Wirth. From Modula-2 to Oberon. Software, Practice and Experience 18:7(1988), 661-670.

[ReWi92] M.Reiser, N.Wirth. Programming in Oberon - Steps Beyond Pascal and Modula. ACM Press, Addison Wessley, 1992. ISBN 0-201-56543-9

[Mö93] H.Mössenböck. Object Oriented Programming in Oberon-2. Springer-Verlag, 1993. ISBN 3-540-56411-X

Index

This page had been intentionally left blank.

Excelsior, LLC
6 Lavrenteva Ave. Suite 441
Novosibirsk 630090 Russia
Tel: +7 (3832) 39 78 24
Fax: +1 (509) 271 5205
Email: info@excelsior-usa.com
Web: http://www.excelsior-usa.com