Open Systems Portability Checker

Reference Manual

Knowledge Software Ltd

November 1999

History

November 99: mcc v5.0, mcl v2.5, mce v2.5
May 98: mcc v4.3, mcl v2.5, mce v2.5
September 97: mcc v4.2, mcl v2.5, mce v2.5
April 97: APIdeduce v1.0
November 96: mcc v4.1, mcl v2.5, mce v2.5
April 96: mcc v4.0, mcl v2.5, mce v2.5
August 95: mcc v3.2, mcl v2.4, mce v2.4
December 94: mcc v3.1, mcl v2.4, mce v2.4
May 94: mcc v3.0, mcl v2.4, mce v2.4
December 93: mcc v2.3c, mcl v2.3, mce v2.3
July 93: mcc v2.3b, mcl v2.3, mce v2.3
December 92: mcc v2.3a, mcl v2.3, mce v2.3
July 92: mcc v2.3, mcl v2.3, mce v2.3
February 92: mcc v2.2, mcl v2.2, mce v2.2
September 91: mcc v2.1, mcl v2.1, mci v2.1
December 90: mcc v2.0, mcl v2.0, mci v2.0
August 90: mcc v1.0, mcl v1.0, mci v1.0

Support

Knowledge Software Ltd provides telephone and mail support for those users who have purchased their systems from Knowledge Software Ltd.

Disclaimer

This document and the software it describes are subject to change without notice. No warranty, express or implied, covers their use. Neither the manufacturer nor the seller is responsible or liable for any consequences of their use.

TradeMarks

Model Implementation C Checker, Open Systems Portability Checker, OSPC and APIdeduce are trademarks of Knowledge Software Ltd. Other brand and product names are trademarks or registered trademarks of their respective holders.

Knowledge Software Ltd. Farnborough, Hants, England. Tel: +44 (0) 1252-520667

e-mail: OSPC@knosof.co.uk

URL: http://www.knosof.co.uk

Table of Contents

Chapter 1

Introduction

Welcome to the Reference Manual for the Open Systems Portability Checker. The user guide gave a brief introduction to the services offered by the **OSPC** and how they might be used. This Reference Manual delves deeper into the standards checking abilities of **OSPC**. It also describes how **OSPC** can be tailored it to suite individual requirements.

First we will examine, in detail, how the standards checking information is stored, and how it might be configures to suite other standards and requirements. This is followed by a detailed description of the options provided by each component tool. This is followed by background information on platform profiles and how they can be managed.

1.1 Contents of Reference Manual

The following provides an overview of what the Reference Manual contains:

User interface and configuration. Describes the contents and format of each of the input files, and how they may be tailored for other requirements.

Source code checker. Lists all of the command line options for **mcc**.

Cross unit checker. Lists all of the options for **mcl**.

Platform profiles. Describes the concepts behind platform profiles. Also explains how they are processed and administered.

Creating Standards profiles. How to approach standards documents with a view to creating a new subprofile containing their requirements.

Understanding the output. Why does the checker complain about my apparently innocent code, and other gory stories.

Using makefiles.

Appendices

A. Headers in the ISO C and POSIX Standards

B. Syntax of C Language

1.2 Related documents

Installation guide

User Guide

Understanding Standards

ANSI C Standard. X3.159-1989

ISO C Standard. ISO/IEC 9899:1990

POSIX.1 ISO/IEC 9945-1 (system API)

P1003.16 Draft 2. Binding for ISO/IEC 9945-1

Realtime Extensions for Portable Operating System, P1003.4 Draft 11

Unix System V Release 4 Programmer's Guide: ANSI C and Programming Support Tools

C Language Interfaces, AT&T Data Systems Group, 1989. ISBN 0-13-109661-3

Systems Interfaces and Headers, Issue 4, X/Open. ISBN 1-872630-47-2

1.3 Conventions

References to the Standard, when a language is being discussed, should be taken to mean ISO 9899:1990 (was ANSI X3.159-1989).

The typographical conventions used follow those given in the POSIX standards.

Type of entry	Example
C-Language Data Type	*short int*
C-Language Error Number	[EINVAL]
C-Language Function	*printf()*
C-Language Argument	*stream*

Type of entry	Example
C-Language Global External	*errno*
C-Language Header	`<stdio.h>`
C-Language Keyword	`#undef`
Constants	MAX_UCHAR
Environment Variables	**MCEDITOR**
Example Input	`mcc myprog`
Example Output	**Hello world!**
File Name	`/usr/include`
Special Character	`<new-line>`
Utility Name	**mcc**
Utility Option	`-CFG`
Parameter	**[**<*platform type*>**]**

Constructs delimited by **[ ]** are optional. Items delimited by single quote characters ‘ are literals.

1.4 Reporting problems

Problems can be reported via electronic of paper mail. A bug report form can be found in the distributed software package in `doc/prob.txt`.

Our electronic address is:

support@knosof.co.uk

If you would prefer to write:

Knowledge Software Ltd, 62 Fernhill Road, Farnborough, Hants, GU14 9RZ, England

Suggestions for improvement are also welcome.

Papers and other information can also be found at http://www.knosof.co.uk

Note: These tools check the requirements given in standards documents. If you are unhappy with these requirements you should address your complaints to the relevant committee. Don't shoot the messenger.

Chapter 2

User interface and configuration

This chapter covers the details behind the user interface. It also describes the configuration files used and their structure. Given the information provided here it ought to be possible for users to reconfigure the tools to support new platforms and standards.

2.1 Organization of the checker

The **OSPC** tools read a large amount of information from configuration files. By default, this information is maintained within a single directory tree structure. This tree structure subdivides into information specific to each tool, information relating to each supported standard and information about each supported platform.

Synonyms are used to provide a shorthand notation for various, commonly referred to, directory names. These synonyms, or aliases, are expanded up to the full pathname when used within **OSPC**. They may also be used in standards, platform and `.mccrc` files.

ROOT The pathname of the distribution directory. This is used to locate `#include`, library and other files. It is determined by the contents of the **INFO**`/host/locate` file (see below).

INFO The directory containing all the configuration information.

PROG The directory containing specific information for the component tool of **OSPC** currently executing. This subdirectory lies within the **INFO** directory, with the same name as the component tool being executed.

PROFILE Root directory for the profile database files.

PLATFORM Contains all the platform profiles information.

Now we will take look at the structure and contents of the main distribution directory. The way files are located, at runtime, and what they are used for will also be described.

2.2 ROOT/bin

This directory contains the executable programs for all components of **OSPC** (**mcc**, **mcl**), and the supporting shell scripts (**profadm**, **c89**, **ccc**, **errorrange**, etc).

Executables and scripts in the **ROOT/bin** directory may be located by the system in one of two ways, depending on how the **OSPC** was installed.

1 Through the **PATH** environment variable containing and entry for **ROOT/bin**.

2 Via symbolic links in the `/usr/bin` directory, mapping to the files given above (assuming that `/usr/bin` is already on **PATH**).

If the **PATH** environment variable contains an entry ending in `checker/bin` the first method is used (see the Installation Guide for details).

2.3 INFO (ROOT/bin/checkinfo)

This directory contains all of the configuration information used by the **OSPC**. It is situated in the same directory as the executables, so the path name of the executing tool can be used to locate it. When **OSPC** is installed using symbolic links, a link is also created for this directory.

The **INFO** directory contains the following subdirectories:

mcc Static source checker specific data.
Includes default option settings, strings and the C language error file.

mcl Cross module checker specific data. Default option settings, strings, error file and the template file used in conjunction with host compiled units.

common Information common to more than one component tool. Includes date and time format, intermediate code format and extensions for file-names.

profile The platform and component profiles. Contains all of the information specific to the supported platforms profiles.

host Information about the host platform.

2.4 ROOT/lib

This directory contains the library files, in various forms, used by the **OSPC**:

- `lib.klc`. The **OSPC** library in .klc form.

Extra files are added to this directory if the dynamic checking portion of **OSPC** is purchased.

2.5 ROOT/include

This directory contains the `#include` headers specified by various standards. These may be used in place of the host system headers, if the host system headers are incomplete.

If the dynamic portion of **OSPC** has also been purchased there will be an additional directory, called `interp`. The contents of this directory only apply when dynamic checking is been performed.

2.6 Default options

Each tool reads in default values for each of the command line options. These are held in the file **PROG`/options`**. The format of this file is one option per line.

If the default option settings require modifying, the local options file, described below, may be a more appropriate method of achieving the desired result.

Example:

```
-Align double=8
-ARithrsh-
-BITLohi-
-BITSigned-
-REMark -Check-
-REMark -COnfig <strings etc.>=<filename>
-REMark -ECHO
-REMark -EXtensions { dos SVR4 languages }
-REMark -Forgetall-
-REMark -HDRsuppress+
-REMark -HCI+
-IDent+
-I /ksc/include
-INTErsperse-
-Listing-
-MAXErrors 99
-MAXWarnings 9999
-NAMelength 31
-REMark -Nomsg <error-number>
```

To change the name of the default options file, read on startup, use the option `-COnfig options=<filename>`

By convention the default configuration file contains an instance of each option. Those options whose values are not being modified occurring as `-REMark`s.

2.7 Local options file

A local options file contains a series of command line options. This local options file is read after the default options file has been read. Thus it overrides any settings made in the

default options file. In turn any command line options take precedence over the settings given in the local options file (since they are processed last).

The local options file is searched for in two places, in the order:

1 The current directory

2 The home directory

Its filename is generated by appending the letters 'rc' to the component tool name and prefixing a '.' character, e.g., `.mccrc`.

The format of the local options file follows that of the default options file, one command line argument per line.

2.7.1 Creating a local options file

The local options file is created using a text editor. It should contain one command line option per line. These options may add to, or override options contained in any other options file read previously. In turn these option settings may be modified by command line options.

```
-I /usr/me/myheads
-List+
```

In this example the `-I` option causes the path `/usr/me/myheads` to be added to the list of places to be searched when looking for header files. The default for the `-List` option may be either on or off. Here we are overriding the default setting to switch the listing on.

The `-Forgetall` option may be used to undo the effects of previous option settings.

```
-Forgetall nomsg
```

This line causes all previously suppressed messages (the effect of the `-Nomsg` option) to be reactivated, i.e., the behaviour is the same as if all previous `-Nomsg` options had never been given.

2.8 Locating the configuration files

When a component tool, of **OSPC** is executed it determines its own name, and the directory from which it was executed. The executing tool then searches in this located directory for a subdirectory (or a symbolic link to a directory) named `checkinfo`. All the information required by the tool is located in this subdirectory (whose name is also available via the alias **INFO**). A directory within **INFO**, with the same name as the tool being executed (whose alias is available through **PROG**), contains configuration information specific to that tool.

The tool name, prefixed with '.' and suffixed with 'rc', provides the name of the local resource file. So `.mccrc` is the resource file for **mcc**.

Thus if **mcl** is located on the path `/usr/bin`, the the directory `/usr/bin/checkinfo` is searched. The directory `/usr/bin/checkinfo/mcl` contains the tool specific information needed by **mcl**.

Renaming **mcl**, in the previous example, to `link_kic` would cause the common interface to look for the tool's specific information in `/usr/bin/checkinfo/link_kic` and use a local resource filename of `.link_kicrc`.

Under Unix the `which` command may be used to find out the full path used by the shell, in locating the component tool:

% which mcc

This will display the full path name of the copy of **mcc** that would be executed, if it were given as a command to the shell.

2.9 The string configuration files

In order to provide maximum flexibility, the strings generated as output by the tools are held in configuration files, they are not hard wired into the executables (except for the copyright message). Placing strings in configuration files, and having the tool read them on startup, gives the user the option of modifying them to change the format of the output.

The configuration strings are used as the second parameter in a call to *fprintf()*. It is thus possible to insert values, supplied by the tool, into some strings. However it is not possible to insert values that are not present in the original format string, or to change the order, or type of the specifiers. Care must be taken when modifying the values of these strings.

Some strings are common to several tools, whilst others are specific to one. The common strings are held in the **INFO/common** directory and include:

- Date and time format
- Intermediate code information
- File name extensions

Strings specific to each tool are defined in the file **PROG/strings**.

If, for some reason, any configuration strings file cannot be found, default values, stored internally by the tool, are used.

2.9.1 Layout of string configuration file

String files have a fixed layout. The tool will complain if it is given a string file that has incorrect layout. The term layout is probably too general a description for what is quite a simple format. A string file consists of comments (optional), headers and lines of text.

- Any line beginning with a star, '*', is treated as a comment.

- Lines containing a hash, '#', as the first character may be used to identify header names (not C header names, but configuration string file string header names). The hash symbol is followed by an identifier that denotes the name of the header. The headers group together similar strings, and are useful for locating layout mistakes in the configuration file.

  Headers must occur in a given order. The tools will complain if the headers are out of order. This can occur because of missing headers, missing text causing headers to be read as text lines, or simply the headers being out of order.

- The lines following a header form the text associated with that header. Each header expects to be followed by a predefined number of non-comment lines. These lines may contain any sequence of characters, including starting with a hash. Stars, however, still introduce comments, but can be obtained as the first character on a line by using the escape sequence `\0x28` (if using ASCII).

Example:

```
* Section from the file INFO/mcc/strings
*
* echoed input line (source line) by 'trace input' option
#ECHO
: %s\n
*
* #line message (lineno, filename)
#LINE
#line %ld "%s"\n
*
#INCLUDE
* #include message (filename) for " "
#include "%s"\n
* #include message (filename) for
#include <%s>\n
* string output at end of top level include
\n
```

2.9.2 Format lines

Once read in the text lines are interpreted as if they occurred as the second parameter of the C *fprintf()* library function.

- Escape sequences are converted as they would be by a C compiler.

- Lines ending in `\<new-line>` are spliced with the following line.

- Whitespace after the last non-whitespace character is significant.

- The maximum width of a text line is 254 characters (yes, the C Standard does specify that a physical text line may have up to 509 characters).

Note: Escape sequences that occur in *fprintf()* format string literals would normally be converted by a compiler. A format string read from a file and then passed to *fprintf()* would not have had its escape sequences converted. The **OSPC** configuration file handler converts escape sequences to mimic the behaviour of string literals contained in source code. Thus the *fprintf()* in a string that has had the escape sequences converted, just like a C compiler would.

2.9.3 Changing a configuration file

The best technique for modifying a string file is to make and test the changes on a copy of the original file. The `-CFG` option can be used to read this modified copy. Once the new file is complete and tested it can be renamed to cause it to become the new, default, configuration file. It may also be useful to remove the old string by prefixing the line with a '`*`', rather than deleting it. This has the advantage that the conversion specifiers (e.g. '`%s`') in the original are retained.

2.9.4 Trace option

If a configuration file has been modified and the component tool that reads it objects to the format, some method of localizing the problem is required. The `-TRace config` option causes the contents of the configuration files to be displayed, as they are being read. The tool displays the default string and the new string, read from the file, in the following form:

'<' default text '> := <' new text '>'

This information, coupled with complaints about out of sync headers, should be sufficient to solve the problem.

2.9.5 Common alterations

There are several strings in the string configuration files that are commonly modified. We shall now take a look at a few of them in turn:

Include messages (mcc).

The text output when **mcc** encounters an *include* directive follows the `#INCLUDE` header in the file **`INFO/mcc/strings`**.

- To stop the includes being displayed comment out all three strings, and insert three blank lines.

- To stop the extra blank line being output between the includes remove the '\n' from the third line.

Error message lead in (mcc).

This lead in can be useful if the system editor can use the error line information to position itself at the appropriate place in the C source code. The output of **mcc** can be made to match this format by changing the string following the #ERRORS header in **INFO/mcc/strings**. Note however, the filename and line number cannot be swapped around because the arguments are passed to *fprintf()* in a fixed order.

Hierarchy diagram text (mcl)

The strings output in the hierarchy diagram follow the #HIERARCHY header in the file **INFO/mcl/strings** including the characters used to create the lines. Under MSDOS graphics characters with a single line could be used instead of the double line characters.

2.9.6 Help information

The string file also contains the help text displayed by each tool. Each line of help text is preceded by a line containing a set of modifier characters that determine whether the option is displayed, or not.

The text line is displayed either, if the modifier line preceding it is empty (and that includes white-space characters), or if a modifier character preceding it is also specified in a -HELPMOD option. The -Detail option is equivalent to -HELPMOD D, hence options preceded by 'D' are displayed in the detailed help. The modifiers allow the help output to be tailored to the platform the checker is running on, by suppressing any irrelevant options. Uppercase modifiers are reserved for future **OSPC** internal use, whilst lowercase letters can be used for a users own groupings.

2.10 Error files

2.10.1 Introduction

The text of all warning and error messages output by **OSPC** are contained in text files. Internally within **OSPC** errors and warnings are represented by numbers. The error file provides a mapping from these numbers to a line of text. This method allows the severity level and content of the messages to be altered by the user. If no error file contains the sought after value the error number itself is given.

Here is a section of the error file that is used with the ISO C profile:

```
*
* ansic/errors  Lastmod 24 Oct 91  DJ
*               Created    Jul 87  SAC
*
#define info        0  very common, ignorable messages
```

```
#define warn        2
#define impldef     4  implementation defined behaviour
#define impundef    5  implicitly specified as undefined behaviour
#define expundef    5  explicitly specified as undefined behaviour
#define usercont    8  user error
#define recover     8  recoverable (semantic) error
#define constraint  9  standard constraint violations
*
* In the following case we must not generate code, or mce gets
* upset
*
#define expundef_err 9  explicitly undefined behaviour,but no code
#define fatal        9  general fatal error

* Errors caused by bad input from the user
  1  userfatal       Filename expected
  2  usercont        Warning previous output filename overridden
  5  usercont        Unknown directive
  6  recover         Argument out of range
  8  userfatal       Invalid filename
 12  usercont        Parameter too long - truncated

 29  fatal           Unexpected end of configuration file
* ...
234 recover          \\x... escape sequence has no hex digits -\
 zero value assumed
234 constraint       [C] \\x... escape sequence has no hex digits
235 expundef         [U] undefined escape code\\
 3.1.3.4 Character constants
244 constraint       [C] character constant contains too many\
 characters\\  3.1.3 Constants
245 constraint       [C] illegal character constant\\
 3.1.3.4 Character constants
249 constraint       [C] character constant exceeds source line
259 constraint       [C] string literal exceeds source line
261 recover          end of included file encountered\
 inside comment  - comment assumed to close
```

2.10.2 Who uses error files

Any standards or platform profile can refer to an error file. In fact, unless they do not refer to such a file they will not be able to generate meaningful messages. The convention is for messages associated with particular profiles to be held in error files in the same directory as that profile. An exception to this rule is the error file associated with C language problems. This is kept in the same directory as **mcc** specific files.

The `-ERRfile` option gives the full pathname of an error file. The message associated with a particular error number is found by searching all of the files given in `-ERRfile` options (the error handler builds a table of contents the first time the files are read; to speed up subsequent processing).

The two scripts **errorrange** and **errornumber** can be used to locate messages associated with particular error numbers and to display the error numbers currently being used.

A tool is not limited to using one error file. Any number of error files may be used. Separating the error messages out into several files provides several advantages:

- The error files can be kept to a manageable size.

- The tool only needs to read the relevant files.
  The errors associated with a given platform need only be used if the platform is referenced. This saves processing time, and enables the profiles to be modularized by placing the error files in the appropriate profile directory.

- Error messages are easier to maintain.
  Users may create their own error files without having to interact with those provided as part of **OSPC**. Thus when new updates are received it is simply a matter of copying over the new files without worrying what effect they will have on user created files (provided the error numbers used fall within the allowed limits).

It is a simple matter to change the text associated with any of the errors reported by **OSPC**. A new error file can be created, with an entry for each of the messages that needs to be changed (in the format given in the section above). By adding a `-ERRfile` option to a local options file, or the default configuration file, the new message text will replace the old.

The error files are processed in the order that they are specified on the command line (or in options files). Within a single file, the error message with the lowest reportable error (will depend on the presence of any `-SUppresslvl` option) will be selected. If the same error number occurs in a later file, the later entry will always take precedence, regardless of their relative error levels.

Each tool's default error message file is found in its specific (**PROG**) directory. The profile directories also contain error files for errors that are specifically associated with that profile (e.g., complaints about reserved names).

2.10.3 Format of the error file

An error file is made up of three different types of lines:

1 Comment lines. All lines containing a star, '*', as the first character are treated as comments. Blank lines are also ignored.

2 #define lines. This is a simple mechanism that allows a name to be associated with a number. A simplistic form of object like macros.

3 Message lines. These provide the actual mapping between error number and text messages. Within a message line there are rules for breaking up the text into various components.

2.10.4 Define line

A define line consists of four components, separated by whitespace:

1 `#define`. Introduces the directive.

2 An identifier. This can consist of alphabetic characters, and underscore or a period (a slight extension on the rules for C identifiers).

3 A single digit number. The number is substituted for the identifier in each of the message lines it occurs in.

4 An optional comment. Not necessarily preceded by a star, '*'

```
#define syntax 9      Highest error level available
```

These lines thus form a close approximation to the definition of object like macros in C.

2.10.5 Message lines

These lines have three main components:

1 The error or message number.
This may contain between one and four digits, and must occur in ascending order within each error file.

2 The error level.
May consist of a single digit, or the name of an identifier that previously appeared in a define line.

3 Characters making up the message.
The characters are treated as a *fprintf()* format string, and can extend over several lines by preceding new-line with a backslash '\'. A reference to a standard can be incorporated by preceding it with two back slashes, followed by newline. Since the messages are used as the format string for a call to *fprintf()* specifiers such as *%s* should only be included in an error message if they were also present in the original. Otherwise *fprintf()* will expect arguments that it haven't been passed, resulting in unpredictable behaviour. Note *%%* represents a percent character in standard C fashion.

```
429  constraint    [C] argument is not arithmetic\\
REFERENCE - ISO 6.3.2.2 Function calls
```

2.10.6 Error number ranges

The error numbers are split into logical chunks, with different ranges being used for different types of errors. The shell script **errorrange** can be used to list the error numbers used by each error file (use the `all` option to include the language errors in this listing. The following table summarizes the ranges into which various categories of problem fall:

Error Range	Use
1-199	Errors resulting from command line options
200-949	C language errors
950-980	System errors (e.g. disk full)
981-1000	Internal errors
1001-1199	C language extensions
1200-1299	Incorrect calls to the ISO C library
1300-1499	Coding standard issues
1500-1699	Special range of errors used by the conditional error mechanism (C++ and K&R related errors)
1700-1899	Lint like problems
1900-1999	Identifier usage problems
2000-5999	Reserved for current and future use by **OSPC**
6000-9999	User selectable error numbers

2.11 Location of host related files

The file **INFO/host/locate** specifies where other files, likely to be referenced by the users code, is located. It has the following entries, each on separate lines.

1 Directory containing the include files

2 Library directory

3 The name of a file containing command line options. This file is intended to hold host specific information, such as defines required by the host header files.

2.12 Hierarchy control files

When generating a hierarchy diagram from the input files, **mcl** displays all the functions that each function references. Having all the available information displayed can be overwhelming. It is not possible to see the wood for the trees.

The hierarchy control file provides a means of cutting out unwanted information. The user can specify:

- Files that should not appear in the hierarchy.
- Whether objects should appear.
- How the symbols should be displayed

The file consists of a series of a series of lines with the following options (only the first character in the option names is significant).

2.12.1 Control file options

The following is a list of the options that may occur in a hierarchy control file.

***** Lines starting with a star are treated as a comment.

All <unit> Display the unit and routine name for each of the functions called in `<unit>`, and trace through each of the routines they call. This is the default action for a unit.

Dots Put “etc ...” on the same line as the previously displayed function. The default is to place this string on the next line.

Follow <unit> Display the unit-less name of the function, and follow through the calls it makes.

Group Group object names together. Places all variable names in a comma separated list on one line. The default is to give one object name per line.

Ignore <unit> Ignore name. No calls to functions in this translation unit are displayed. Neither are any calls to functions called by functions in this unit.

Name <unit> Display the name of functions that are referenced in this unit, but ignore any functions that they reference.

Trace <unit> Trace references to functions outside of this unit, but don’t display the names of externals in this unit. This is used to stop *cmain()* being displayed, but ensure *main()* (and all the functions it calls) appear in the hierarchy.

Unit <unit> Display the name of the functions, prefixed by their unit name. Any functions called from this unit are ignored.

Vars When given this option removes the names of objects from the hierarchy diagram. The default is to display the names of objects.

The unit control options can be summarized in the following table:

	Display		
	Unit + Name	Name only	Nothing
Follow calls	All	Follow	Trace
Ignore calls	Unit	Name	Ignore

Chapter 3

Source code checker (mcc)

3.1 Introduction

This chapter takes a detailed look at all the **mcc** options. The standard help text does not list all the available options. A complete list can be obtained by using the `-DETail` option.

mcc -DETail

Most of the options considered to be 'details' relate to platform specific functionality. Thus the setting of these options would normally be controlled by the platform profiles.

3.1.1 Environment interface

mcc uses two environment variables, `HOME` and `MCCOUTPATH` (these names are held in the `mcc` strings file). The `HOME` variable, if set, is used to locate the .rc file used by **mcc**. The second environment variable, if set is used to override the setting of any `-OUTPUT-Path` options that may have occurred in any options files read in. It does not override the setting of an `-OUTPUTPath` that occurs on the command line.

At the end of processing **mcc** provides a return code to the host OS on the status of the work performed. The value of this code follows the conventions used by **cc** and has a simple zero/non-zero value.

The file `host/locate` contains two lines. The first is the path used to locate include files. The second is the root path used to locate standard includes and the library .klc files used by **mcl**.

3.2 Options

This section lists all of the options available in the compiler portion of **OSPC**.

Notation: In this section the minimum abbreviation for each of the options is given by the portion in capital letters. That is one or more of the lower case letters may be omitted.

Align Specify address byte boundary for objects of that type

SYNOPSIS

Align <type>=<bytes>

DESCRIPTION

Alignment restrictions, or lack of thereof, are brought about by limitations or efficiency requirements in the underlying target cpu. The three most common cases are (1) no restriction, (2) scalars larger than 1 byte must start on an even address boundary, and (3) scalars must start on an address that is an exact multiple of their size (RISC processors usually have this restriction).

<type> is one of *char, short, int, long, float, double,* ldouble, bitfield, dataptr, codeptr or param. This option specifies the byte boundary for each type when addresses are assigned to objects and offsets calculated for members within structs. Param gives the minimum alignment of parameters on the system stack (the actual alignment of parameters is taken to be the maximum of the parameter alignment and the alignment for that type).

The C standard imposes no requirements on the alignment of objects. Except that giving an alignment for a type will also cause any qualified or *unsigned* variants of that type to take the same alignment.

mcc assumes that the same alignment requirements apply to members of structures, parameters (subject to the minimum alignment specified by the param type) and individual objects.

Note: **mce** may have been built to expect a stricter alignment of datatypes than used by **mcc**. Thus specifying a less strict alignment may cause **mce** to flag an error.

EXAMPLE

mcc prog -A char=1 -A float=4

mcc prog -A int=5

PROFILE

This option is set by the cpu profile.

A**pi**usage Generate information on API usage

SYNOPSIS

APIusage±

DESCRIPTION

Switching this option on causes a `.api` file to be generated. Information on identifiers encountered in the source code, and specified in an API database, is written out to this file. References on all externally visible identifiers is also written out.

By combining the `.api` files from each translation unit making up a complete application it is possible to deduce the API's used by that application.

EXAMPLE

mcc prog -APIusage+

PROFILE

This option is set as part of the default startup information.

Arithrsh — Right shift (>>) to be arithmetic

SYNOPSIS

ARithrsh±

DESCRIPTION

The result of shifting a signed quantity right is specified as being implementation defined in the C standard.

This option affects **mcc** in two ways. It is used by the platform profile machinery to decide when arithmetic right shifts should be flagged. It also selects the type of code generated for use by the executor.

Switching this option off causes **mcc** to generate code that performs logical right shifts. That is, zeroes are shifted into the vacated top bits.

When this option is on arithmetic right shifts are performed. In this case one's are shifted into the vacated top bits if the top most bit was originally set.

EXAMPLE

mcc prog -AR+

PROFILE

This option is set by the compiler profile.

Assert

Perform #assert from the command line

SYNOPSIS

ASsert <identifier>(<identifier>)

ASsert <identifier>(<number>)

DESCRIPTION

One of the System V.4 extensions supported is the use of *#assert*. This option allows assertions to be predefined on the command line, similar to `D` for macro definitions. The arguments for an assertion specified on the command line is restricted to a single token.

If the argument is given on the command line, rather than from an options file, the second argument may need to be quoted to avoid interactions with the shell.

This option is not fully specified in the SVR4 documentation. The specification for the **mcc** implementation was obtained by experimentation.

EXAMPLE

mcc prog -ASSert cpu(68000)

mcc prog -ASSert “cpu(68000)”

The following two commands are equivalent. The first form is supported for compatibility with SVR4:

EXAMPLE

mcc prog -ASSert-

mcc prog -Forgetall ASSert

PROFILE

This option is set by the compiler profile.

Bigendian — Indicate memory storage ordering of multi-byte objects

SYNOPSIS

BIGendian±

DESCRIPTION

The ordering of bytes within multi-byte quantities is not defined by the C standard.

Switching this option off causes **mcc** to treat multi-byte quantities as being stored in little endian mode.

The setting of this option does not affect the generated code. Its purpose is to allow **mcc**, in conjunction with platform profile information, to flag potential problems caused by different byte orderings.

EXAMPLE

mcc prog -BIGendian+

PROFILE

This option is set by the cpu profile.

Bitlohi — Bit-fields allocated lo-bit to hi-bit

SYNOPSIS

BITLohi±

DESCRIPTION

The ordering of bit-fields within a unit of storage is given by the C standard as implementation defined. Allocation may be low to high or high to low within a storage unit.

This option allows the user to specify which ordering should be used. There is no platform profile interaction.

EXAMPLE

The following structure members will occupy different bits of the storage unit, depending on the order that bits are used.

```
struct mem_reg {
    signed int status : 4;
    unsigned int count : 6;
    }
```

mcc prog -BITL+

PROFILE

This option is set by the compiler profile.

Bitoverlap Can bit-fields occupy the same storage unit as a char

SYNOPSIS

BITOverlap±

DESCRIPTION

Switching this option on causes bit-fields to be allocated space, where possible, in the same storage unit as a *char*.

EXAMPLE

mcc prog -BITOverlap+

Bitsigned Plain bit-fields to be signed

SYNOPSIS

BITSigned±

DESCRIPTION

The signedness of a bit-field defined with just the *int* type specifier is given as implementation defined by the C standard.

Switching this option on causes such bit-fields to be treated as if they had been declared as *signed int*. When this option is off bit-fields are treated as if they had been declared *unsigned int*.

EXAMPLE

In the following structure the second member be a signed int or unsigned int.

```
struct tb {
    signed int f1 : 4;
    int f2 :2;
    unsigned int f3 :3;
   }
```

mcc prog -BITS+

PROFILE

This option is set by the compiler profile.

C**harc**onst Number and order of letters in character constant

SYNOPSIS

CHARConst <letter-sequence>

DESCRIPTION

This option is used to specify the number of characters in a character constant and the order in which these characters are held internally. The C standard specifies that if more than one character is present the behaviour is implementation defined.

The option works by assuming the character sequence abcdefg... (for as many characters as are supported by the compiler). The order given in the option provides the mapping to be used in reading from the character constant read to its internal value.

EXAMPLE

mcc prog -Charc xabcd

The character x is used to denote the end from which zero bytes are added for character constants containing less characters than the maximum supported. In the above case the character constant *'ab'* would have two bytes, containing zero, added before the *a* character. Had the option `-CHARC abcdx` been given the zero would have been added after the *b* character.

mcc prog -Charc xba

In the above case the characters are stored, in an *int* object in the reverse order to which they appear. Depending on more than one character being allowed in a character constant does reduce portability. This information is used in the source and target profile flagging to highlight places where behaviour will differ.

mcc prog -Charc x000d

Here there may be up to four characters in a character constant, but only the last character is used in the final value. The zeroes are needed to tell **mcc** that four characters may occur (a letter **d** on its own may have been caused by a typing error).

PROFILE

This option is set by the compiler profile.

C**harset** ASCII or EBCDIC character set

SYNOPSIS

CHARSet <letter-sequence>

DESCRIPTION

This option is used to specify the representation used to give numeric values to characters. Although the C standard requires that characters from ISO 646 be used it does not define their numeric value.

The two representations currently supported are ASCII and EBCDIC. The setting of this value will affect the values of characters in strings and character constants.

EXAMPLE

mcc prog -Charset ebcdic

PROFILE

This option is set by the compiler profile.

Check Perform syntax & semantic checks only

SYNOPSIS

CHECK±

DESCRIPTION

Switching this option on causes **mcc** to perform syntax and semantic checks but not to generate .kic files. This option may speed up the execution of **mcc** but because no .kic file is generated it will not be possible to perform cross unit checking.

EXAMPLE

mcc prog -C+

Checki**d** Specify an identifier checking filename

SYNOPSIS

CHECKId <filename>

CHK <filename>

DESCRIPTION

The <filename> is taken as the file containing information about reserved identifiers. Multiple identifier files can be specified, and identifiers will be checked against the information contained in each of them in turn.

If <filename> cannot be opened, or is not in the correct format an error message is displayed.

This option interacts with platform profiles in that if the same <filename> occurs in both the source and target profiles no checking is done using the contents of that file.

Note: The error reporting mechanism will only report a given error number once for each token. Thus if more than one checkid file flags a given identifier with the same error number only one of them will appear (since every error number should have the same text associated with it this should not cause a problem).

EXAMPLE

mcc prog -CHECKId PROFILE/standard/ansic/ident

PROFILE

This option is set by the standards and O/S profiles.

CODES

Switch on various coding standards checks

SYNOPSIS

CODES <letter-sequence>

DESCRIPTION

There are many checks that can be performed that do not, strictly speaking, fall under the category of lint like checks. Such constructs are usually described in company coding standards. The checking for such constructs is controlled by this option. `<letter-sequence>` may be one of the following.

`CAST` Flags those situations where implicit casts could cause a loss of information.

`COMMENT` Causes a warning to be generated if a *if*, *while*, *do*, *switch* statement, or the start of a function definition is not preceded by a comment. Note: Switching this option on implicitly switches on the checking of code layout.

`FLOW` Switches on extra checks involving the flow of control, i.e., multiple exits from a function and *break* appearing in an *if* statement.

`LAYOUT` Checks the layout of statements (consistent indentation and use of braces) and identifier declarations (only one per line).

`LOOPS` Perform checks involving the control variable in *for* and *while* statements.

EXAMPLE

mcc prog -CODES loop prog.c

PROFILE

This option is not set by any profile.

Conderr — Specify conditional error file

SYNOPSIS

CONDErr <filename>

CErr <filename>

DESCRIPTION

The conditional error mechanism is a method of flagging language constructs that are applicable to the source or target platform, but not both. Error numbers within the conditional error file are used, provided the error file either appear in the source platform profile, or the target platform profile, but not if it appears in both.

This is the mechanism used to flag language constructs that differ between K&R and ISO C.

EXAMPLE

mcc prog -CONDerr PROFILE/standard/kandr/conderrs

PROFILE

This option is set by the standard and O/S profiles.

Config — Specify a configuration filename

SYNOPSIS

COnfig <tag>=<filename>

CFG <tag>=<filename>

DESCRIPTION

Several configuration files are used by the tools. These files contain, amongst other things, strings output and default options. The tag may be one of:

- `datetime`. The names of the months and format for outputting the date and time.

- `extensions`. The extensions used for c, .kic, .klc files etc.
- `opcodes`. Information about the intermediate code (for display purposes).
- `strings`. The strings for all the output produced by the tool.
- `options`. The default option settings.
- `locate`. The location of the various special files, e.g. libraries.

If the file cannot be opened or is not in the correct format an error message is displayed and processing stops.

EXAMPLE

mcc prog -COnfig strings=/home/usr/fred/misc/newstrings

PROFILE

The tools always read files holding this information. This option provides a mechanism for replacing the files actually read.

D

Perform #define from the command line

SYNOPSIS

D <name>

D <name>=

D <name>=<string>

D <name>=<number>

DESCRIPTION

This option allows the user to define C macros from the command line. These macros are treated as if they formed the first lines of the source file being compiled.

There are three forms of this option:

`D ident` is equivalent to the C:

```
#define ident 1
```

and `D ident=string` is equivalent to the C:

```
#define ident string
```

Note: `D ident=` is equivalent to:

```
#define ident
```

Whitespace may not be used to separate the = character from the preceding identifier or the following sequence of characters.

PROFILE

This option is set by the compiler profile.

D**et**ail — Detailed rather than brief help

SYNOPSIS

DETail±

DESCRIPTION

Only those options commonly modified by users are given in the default help display. The other options are either better selected through the use of platform profiles, or are only applicable to certain hosts (e.g. DOS). This option causes these other options to be displayed, and is equivalent to `-Helpmod D`.

EXAMPLE

mcc -det+

E**cho** — Echo given text to standard output

SYNOPSIS

ECHO <text>

DESCRIPTION

The rest of the line, starting at the first non-whitespace character is echoed to the screen. A `-ECHO` on its own displays a newline.

This option may be used to display helpful information when options are being processed from a via file.

Note that it cannot be used on the command line.

EXAMPLE

If `abc.via` contains:

-ECHO Created on 9 Aug

-ECHO

then

mcc -via abc.via

causes the following lines to appear (the newline cannot be seen):

Created on 9 Aug

Errfile Specify error file name

SYNOPSIS

ERRfile <filename>

DESCRIPTION

When an error needs to be reported a list of files are searched by the error reporting machinery. This option adds a new filename to the list of files searched.

If <filename> does not exist a warning is given and processing continues. If an error file is not available any warning or error messages relating to that file will take the form of an error number.

EXAMPLE

mcc prog -ERR new.err

PROFILE

This option is set by most profiles.

E**rrn**umber Switch on output of error numbers

SYNOPSIS

ERRNUMBER <error-number>

DESCRIPTION

In order to switch off particular errors, with the `-NOmsg` option, it is necessary to know the error number associated with a particular message. Switching this option on causes the error number to be given along with the appropriate message.

EXAMPLE

mcc prog -ERRN+

E**v**alorder Specify order in which expressions are evaluated

SYNOPSIS

EValorder <letter-sequence>

DESCRIPTION

The C standard allows implementations the freedom to evaluate expressions containing side effects in any order. Code that relies on a given order of evaluation is very non-portable. The <letter-sequence> part of this option provides a method of indicating an order of evaluation. If the <letter-sequence> is the same on the source and target platform it is not necessary for **mcc** to flag expressions containing side effects.

A `-EValorder` option that is followed by a letter sequence of `?` is assumed to have an unknown order of evaluation (more probably an order that varies, depending on the complexity of the expression and contents of the cpu registers).

EXAMPLE

mcc prog -EValorder LR

PROFILE

This option is set by the compiler profile.

Extensions Enable extensions

SYNOPSIS

EXtensions <letter-sequence>

DESCRIPTION

Some language extensions have become so common on certain hosts that the majority of compilers on those hosts support them. **OSPC** supports the most common extensions. They have been broken down into various categories. The `<letter-sequence>` may be one of the following:

`DOS` Allow *near, far* and *huge* qualifiers

`C9X` Allow some C9X (Draft Revised C standard) features to be enabled.

`CPP` Allow C++ languages features commonly supported by C compilers, for instance `//` style comments.

`GCC` Support a variety of GCC extensions, including `typeof`, `attribute`, and some support for `long long`.

`JAVA` Flag C constructs that are not available, or behave differently, in Java.

`language`
Support *fortran* and *pascal* functions

`SVR4`
Support SVR4 extensions, including `#assert`ion's and `#ident`.

`slashwhite`
Allow whitespace between \ and new-line.
The standard requires that the line splice character (\) be immediately followed by a new-line, if it is to have a splicing effect. Switching this option on relaxes this requirement by allowing whitespace to occur between it and a new-line.

EXAMPLE

mcc prog -EXtensions gcc

PROFILE

This option is set by the compiler profile.

F**namec**har Specify the characters that may occur in an include filename

SYNOPSIS

FNAMEChar <letter-sequence>

DESCRIPTION

The list of characters given are those that are allowed to occur in an include filename.

The configuration file holds the values allowed by ISO C. This option adds to this set.

EXAMPLE

mcc prog -FNAMEChar "$%!/"

PROFILE

This option is set by the standards and O/S profiles.

F**namel**en Maximum include filename length

SYNOPSIS

FNAMELen <length>

DESCRIPTION

Different operating systems have differing restrictions on the maximum number of characters allowed in the basename of a filename. This option specifies the maximum number of characters that may occur in an include filename.

Any include file longer than this length with be flagged.

EXAMPLE

mcc prog -FNAMELen 14

PROFILE

This option is set by the standards and O/S profiles.

Forgetall Forget all arguments of option given so far

SYNOPSIS

Forgetall <option>

DESCRIPTION

Some options do not have single values, they accumulate a list of values. For instance an `-I` option does not undo the effects of any prior `-I` option. It adds a new path to the list of existing paths. The `-Forgetall` option enables lists of values to be forgotten. It must be applied to an option that takes lists of values (it may also be applied to options that accept string arguments). When applied to the following options it has the specified effects:

`Align` Use default alignments.

`ASsert`

`CODES` Forget previously enabled coding standard checks.

`D` Forget previous definitions.

`Errorfile`

`EXtensions` Forget previously enabled extensions.

`I` Forget include paths given so far.

`LOGfile` Cancel request for logfile.

`NOmsg` Reinstate any previously suppressed error numbers.

`Output` Use default output filename.

`PAth` Cancel previous prefix.

`RETurn` Cancel information on identifiers that cause function termination.

`Size` Uses default sizes for scalars.

`STRUCT`

The main use of this option is in overriding any options given in the default or possibly the local options file.

EXAMPLE

H**cexept*** Treat header as being exceptions to host compiled

SYNOPSIS

HCexeptI <header>

DESCRIPTION

This option is of use when, during a particular compilation, all system headers have been marked as host compiled. It may be simpler to mark all system headers as being host compiled and have a few exceptions than list all the system headers that are host compiled.

H**ci*** Treat headers included from host compiled headers as host compiled

SYNOPSIS

HCI±

DESCRIPTION

Switching this option on causes **mcc** to regard header files included from within host compiled headers as also being host compiled.

Switching this option off causes **mcc** to not treat nested include files as being host compiled.

For more details on host compilation see the later chapter in this manual.

PROFILE

This option is set by the hostlib ‘misc’ profile.

H**cli**b* Is the system library to be host compiled

SYNOPSIS

HCLIb±

DESCRIPTION

Switching this option on causes **mcc** to regard the contents of system headers as host compiled.

PROFILE

This option is set by the hostlib 'misc' profile.

H**dr**suppress Suppress warnings while processing system headers

SYNOPSIS

HDRsuppress±

DESCRIPTION

Sometimes it may be necessary to use a header file other than that supplied with the **OSPC**. These alternative headers may not be strictly conforming. Declarations and definitions within these headers may be flagged by **mcc**. However, the user does not always want, or have the option of changing these headers.

Switching this option on causes **mcc** not to display any messages about the contents of systems headers.

Note: A constraint error within a system header will not be flagged.

EXAMPLE

mcc prog -HDR+

PROFILE

This option is set by the compiler and profiles.

H**ead**ers Specify a valid-header file

SYNOPSIS

HEADers <filename>

DESCRIPTION

The named file, in binary data base form, contains a list of valid header names, i.e., those names that can occur between chevrons (< >) in an

`#include` preprocessor directive without causing a warning message to be generated.

This options differs from the `-CHK` option in that it is source and target platform independent.

EXAMPLE

mcc prog -HEADers PROFILE/standard/ansic/headers

PROFILE

This option is set by many profiles.

H**elpm**odify Set modifiers for displayed help

SYNOPSIS

HELPMod <letter>

DESCRIPTION

In the default help display only the generic options are given. The output of each line of help text (contained in the config file) is controlled by a set of modifiers. Each help text line can either be always displayed, or whenever one of the modifiers associated with it is given in this option. The following modifiers are currently supported:

`D` Detailed help (display everything)

`H` Host compiled options

`M` Memory manager options (only relevant to MSDOS platforms)

EXAMPLE

mcc prog -HELPMod HDM -help

H**ostc**omp* Mark the contents of a header as being host-compiled

SYNOPSIS

HOSTComp <header-name>

HC <header-name>

DESCRIPTION

Host compilation allows routines compiled by the host compiler to be called from the executor. The header name need not include the '.h', a '.h' extension will be added by default.

More details on host compilation are provided in the manual on dynamic checking.

EXAMPLE

mcc prog -HOSTComp stdio

I

Specify directory to search for #include files

SYNOPSIS

I <filename>

DESCRIPTION

The specified directory is added to the list of directories used by the `#include` file search algorithm.

If more that one `-I` option is given then the paths are searched in the order in which they are encountered.

EXAMPLE

mcc prog -I lib/ourheader

Ident

Check declarations against reserved list

SYNOPSIS

IDent±

DESCRIPTION

Switching this option on causes **mcc** to check the declaration or definition of all identifiers against the list of names given as reserved in the platform profile being used. The list of reserved identifiers is read from the files specified by the `-CHECKId` option.

Note. Reserved identifiers that are declared in system header files are not flagged.

EXAMPLE

mcc prog -ID-

PROFILE

This option is set by the standards profile.

Idfollowchars Identifier extension characters

SYNOPSIS

IDFOLLOWChars <letter-sequence>

DESCRIPTION

Some platforms allow the set of characters that may occur within an identifier to go beyond that required by the C standard. This options allows these extended identifier characters to be specified.

Note: It is possible that making additions to the set of characters that may occur in identifiers can change the way that the input source is tokenize. For instance:

```
#define abc$
```

could define a macro named abc with body *$*, or *abc$* having no body.

EXAMPLE

mcc prog -IDFOLLOWChars "@\$"

PROFILE

This option is set by the compiler profile.

Idstartchars Characters that can start an identifier

SYNOPSIS

IDSTARTChars <letter-sequence>

DESCRIPTION

Some compilers allow the set of characters that may occur at the start of an identifier to go beyond that required by the C standard. This options allows these extended identifier characters to be specified.

Note: Because of environment variable expansion within via files the dollar, *$*, character should be quoted.

EXAMPLE

mcc prog -IDSTARTChars "\$"

PROFILE

This option is set by the compiler profile.

Intersperse Intersperse listing with generated code

SYNOPSIS

INTErsperse±

DESCRIPTION

As well as checking the source code and writing symbolic information to the .kic file **mcc** generates P-codes from the C source statements and initialisers. These P-codes are written, in compressed form, to the .kic file.

Switching this option on and also enabling the listing option causes a readable form of these P-codes to be written to the listing file. The P-codes are written in text form and appear close to the statements to which they refer.

This is not an option that is likely to be used by the majority of users. It exists because it was of use during the development of the **OSPC** and the design decision was taken not to have any 'magic' options for internal use.

EXAMPLE

mcc prog -INT+ -L+

INTrep Internal representation of integral quantities

SYNOPSIS

INTrep <letter-sequence>

DESCRIPTION

Two's compliment notation is not yet universal. Some processors use other methods of representing integral values. This option allows the the two most common alternative representations to be selected.

<letter-sequence> may be one of: `1cmp`, `2cmp` or `smag`. Representing 1's compliment, 2's compliment and signed magnitude notation, respectively.

EXAMPLE

mcc prog -INTRep 1cmp

PROFILE

This option is set by the cpu profile.

LIMits

Specify the values of certain limits

SYNOPSIS

LIMits <identifier>=<value>

DESCRIPTION

This option is used to specify the value of certain compile time limits. Numeric literal that exceed these limits will be flagged. Possible identifiers include:

`CALLARGS`

The number of arguments in a function call.

`CASELABELS`

The number of case labels in a *switch* statement.

`DBLMANMAX, DBLMANMIN, DBLEXPMAX, DBLEXPMIN`

`DBLSIGDIGIT, FLTSIGDIGIT, LDBLSIGDIGIT`

Number of siginifcant digits in a *double, float,* or *long double*.

`ENUMCONST`

The number of constants in an *enum* definition.

`EXACTFLTREP, EXACTDBLREP, EXACTLDBLREP`

`EXTERNALIDS`

The number of externally visible identifiers.

`FLTMANMAX, FLTMANMIN, FLTEXPMAX, FLTEXPMIN`

`FUNCPARAMS`

The number of parameters in a function declaration or definition.

`INCLUDEDEPTH`

The maximum nesting of *#include* directives.

`INVOKARGS`

The number of arguments in a macro invocation.

`LDBLMANMAX, LDBLMANMIN, LDBLEXPMAX, LDBLEXPMIN`

`MACRODEFS`

The number of macro definitions.

`MACROPARAMS`

The number of parameters in a macro definition.

`NESTCONDINCL`

The nesting of conditional compilation (*#if ... #endif*) directives.

`NESTDECLPAREN`

The number of parenthesis in a declaration.

`NESTDECLSPEC`

The number of specifiers in a declaration.

`NESTEXPRPAREN`

The nesting of parenthesis in an expression.

NESTSTRUCT

The level to which structures are defined within each other.

NESTSTMT

The nesting of compound statements.

OBJSIZE

The size, in bytes, of an object.

SIGCINTERNID

The number of significant characters in an identifier with internal linkage.

SOURCELINE

The length of a logical source line.

STRINGLEN

The number of characters in a string token.

EXAMPLE

mcc prog -LIMITs MACROBODY=33

PROFILE

This option is set by the compiler profile.

Lint — Perform lint like checks

SYNOPSIS

LINt±

DESCRIPTION

Switching this option on causes **OSPC** to perform lint like checking on the source. This checking will flag constructs, which although strictly conforming C code, are often unintended by the software developer, ie probable programmer errors.

This option also enables the checking of code layout.

EXAMPLE

mcc prog -LIN+

Listing — Generate a listing file

SYNOPSIS

Listing±

DESCRIPTION

Switching this option on causes a list file to be generated.

A listing file contains the preprocessed source of the input file plus any associated error or warning messages.

The source file name is used to create a listing file, by substituting a .lst for the .c suffix (or implied .c if none is given).

EXAMPLE

mcc prog -L+

Logfile — Specify a log file name

SYNOPSIS

LOGfile <filename>

DESCRIPTION

Causes all of the characters that are sent to standard output to be sent to the named file. If the named file consists of the plus, '+', character then the log filename is constructed from the name of the source file currently being processed.

EXAMPLE

mcc prog -LOG prog.log

MAPfile Generate a map file

SYNOPSIS

MAPfile±

DESCRIPTION

Switching this option on cause **mcc** to generate a `.map` file. This gives the sizes of all file scope objects and the total amount of local object storage used by the declarations within each function.

EXAMPLE

mcc prog -MAP+

Maxerrors Specify maximum number of errors

SYNOPSIS

MAXErrors <number>

DESCRIPTION

A run of **mcc** that results in large numbers of error messages usually occurs because of some large cascading problem. This option enables the maximum number of allowable errors to be specified. Once this limit is exceeded **mcc** stops processing the current source file.

EXAMPLE

mcc prog -MAXE 44

PROFILE

This option is set by the tool profile.

MaxwarningsSpecify maximum number of warnings

SYNOPSIS

MAXWarnings <number>

DESCRIPTION

As per `-MAXErrors` but applies to warning messages

EXAMPLE

mcc prog -MAXW 9999

PROFILE

This option is set by the tool profile.

Metrics — Generate a file containing metrics information

SYNOPSIS

METrics±

DESCRIPTION

Switching this option on causes **mcc** to generate a `.met` file. This file contains the raw information used by the **dispmet** program to generate software metrics.

EXAMPLE

mcc prog -MET+

Miscid — Create a miscellaneous identifier file

SYNOPSIS

MISCId±

DESCRIPTION

This is an ‘internal’ option which is used to create a new ‘platform specific identifier file’ for a new platform. If the `MISCId` option is selected, an identifier which doesn’t match any of the known reserved identifiers is output to a file. The filename of the destination file is given by replacing the `.c` extension of the input file with a `.mid` extension.

EXAMPLE

mcc aheader -MISCId+

Modsign Sign of % operator with negative operands

SYNOPSIS

MODSign±±±±

DESCRIPTION

The C standard does not define the sign of the result of dividing two negative values (a restriction is placed on the combined division and remainder values). There are four possibilities:

+ / + (plus divided by plus), + / -, - / + and - / -. This option allows the sign of the results of each of these divisions to be specified.

Note: This option is only used to flag possible differences in behaviour between the source and target platforms. The settings of this option do not affect the results of any constant folding that may occur.

EXAMPLE

mcc prog -MODSign +—+

PROFILE

This option is set by the compiler or cpu profile.

Namelength Internal name significance

SYNOPSIS

NAMelength <number>

DESCRIPTION

The C standard specifies that at least the first 31 characters of an identifier are significant. At link time only the first 6 characters need be significant.

This option specifies the number of characters in an internal identifier (that is with internal or no linkage) that will be treated as significant by **mcc**. A warning will be given about two identifiers that differ in non-significant characters, but they will still be treated as different identifiers by **mcc**,

EXAMPLE

mcc prog -NAMelen 8

PROFILE

This option is set by the compiler and standards profile.

N**amet**runc Internal name truncation

SYNOPSIS

NAMETrunc <number>

DESCRIPTION

Similar to `-NAMelength` except that two identifiers that differ in non-significant characters will be treated as representing the same identifier.

Note that the identifiers will also be written out in truncated form and that this option will also truncate language keywords.

EXAMPLE

mcc prog -NAMETrunc 16

PROFILE

This option is set by the compiler profile.

Nomsg Suppress a specific error number

SYNOPSIS

Nomsg <errnum>

DESCRIPTION

Rather than editing the error files to reduce the severity of an unwanted error, this option allows an error number to be disabled for the current invocation of **mcc**.

The `-ERRNumbers` option can be used to print the error number associated with each message. Alternatively the **errorrange** script will list which error number is held in which error file.

Note: Disabling constraint errors serves no useful purpose. Any messages given after a suppressed constraint error may be difficult to interpret without seeing the constraint message.

EXAMPLE

mcc prog -N 43 -N97

O**PT**imize* Enable intermediate code optimization

SYNOPSIS

OPTimize±

DESCRIPTION

This option is of use when performing dynamic checking for improving the quality of intermediate code generated. By default **mcc** generates code that follows the exact letter of the C standard. When this option is enabled some casts are not generated (they are known to be redundant on most processors and faster, special purpose, instructions are generated for common address calculations.

Note: The code generated when this option is switched on is not orthogonal. Hence it is not as easy to prove correctness of the generated code.

O**spcd**ir Specify the output path name for .kic files

SYNOPSIS

OSPCDir <path name>

DESCRIPTION

By default **mcc** puts the .kic files in creates in the current directory. This can cause the directory to become cluttered (even more so when the stubs for the dynamic checker are created). This option specifies a path name into which the .kic files and stub files (object code) should be placed.

Output Specify the output filename

SYNOPSIS

Output <filename>

DESCRIPTION

By default the name of the input source file (stripped of any associated path) is used as the basis of the generated output .kic filename. This option causes the given name to be used as the .kic filename.

EXAMPLE

mcc prog -O newfile.kic

P**p**list Produce preprocessed listing file

SYNOPSIS

PPlist±

DESCRIPTION

This option is similar to the `-Listing` option, but with the difference that the file produced is compilable (assuming that there were no constraint errors flagged in the original source). A .i is appended to the output filename.

EXAMPLE

mcc prog -PP+

P**rag**ma Specify compiler pragma

SYNOPSIS

PRAGma <letter-sequence>

DESCRIPTION

This option is used to tell **mcc** about pragmas that have a known effect. **mcc** does not know what the effect might be. Rather the information is used to flag unrecognized pragmas between source and target platforms.

EXAMPLE

mcc prog -PRAGma “pack”

PROFILE

This option is set by the compiler profile.

PREInclude Specify a preinclude file

SYNOPSIS

PREInclude <filename>

DESCRIPTION

A preincluded file is processed like an ordinary include file. Except that there is no explicit *#include* in the source code and the include happens before the file given on the command line is processed.

The main use of this type of include is to enable various system specific identifiers to be declared, prior to the processing of the users code. For instance to define function like macros, or objects and functions that may be predefined by a host compiler.

EXAMPLE

mcc prog -PREInclude linux.stuff

PROFILE

This option may be set in the defaults options file for a given host. It may also be set in a compiler profile.

Printfspec Specify printf conversions specifiers

SYNOPSIS

PRINTfspec <letter-sequence>

DESCRIPTION

This option is used to tell **mcc** which conversion specifiers may occur in the format string passed to calls to *printf*.

EXAMPLE

mcc prog -PRINTfspec “pWQ”

PROFILE

This option is set by the compiler profile.

P**si**d — Specify platform specific identifiers.

SYNOPSIS

PSId <filename>

DESCRIPTION

The <filename> contains information about platform specific identifiers. These are names of functions, variables or types that are supported by a platform in addition to the standards it purports to support. **OSPC** will warn about any psids that are present on the source platform but not on the target and which haven't been protected with a feature test macro (e.g., *sparc* for sun4).

This option only reads the source platform specific identifiers.

EXAMPLE

mcc prog -PSId ident_info

PROFILE

This option is set by the standards profile.

P**TRS**calar — Control flagging of pointer/scalar conversions

SYNOPSIS

PTRScalar±

DESCRIPTION

This option is used to control warnings generated as a result of pointer to integral type casts.

EXAMPLE

mcc prog -PTRScalar+

PROFILE

This option is set by the compiler profile.

Quiet — Quiet mode

SYNOPSIS

Quiet±

DESCRIPTION

While processing a source file **mcc** gives information on its progress. Switching this option on disables the generation of this progress reporting output, including the reporting of errors to standard output.

EXAMPLE

mcc prog -Q+

Range — Switch on pointer range checking

SYNOPSIS

RAnge±

DESCRIPTION

Pointer range checking is performed by **mce** and can be done in one of two ways: by using real pointers, or by extending the pointer data type. The first is slower but included checks for uninitialized data and can be used with host compiled routines. The latter requires extra information to be added to the .kic file generated by **mcc**. This information is not added by default because it would slow down the execution of programs that did not require this checking. The type of pointer checking that is used is determined when **mce** is compiled.

Note. In order to store the extra information needed to perform pointer range checking this option increases the size of pointers to objects from 4 to 12 bytes.

EXAMPLE

mcc prog -R+

References — Give standard reference on error messages

SYNOPSIS

REFerences±

DESCRIPTION

The messages reported by **mcc** are intended to be informative and easily understood. As such they do not necessarily use the terminology of the standard. Switching this option on causes the message reporting machinery to give a reference to the standard (provided one is present in the error file that contains the appropriate error number) with the text of the message.

EXAMPLE

mcc prog -REF+

Remark — Comment option

SYNOPSIS

REMark <text>

DESCRIPTION

The text following this option in an option file (not the command line) is treated as a comment. It has no effect on the behaviour of **mcc**.

If used as an option on the command line a warning will be given.

EXAMPLE

-REM Profile for BSD extensions to System V

Return — Name of identifier causing abnormal termination

SYNOPSIS

RETurn <identifier>

DESCRIPTION

The identifier following this option is the name of a function which, when executed, causes the current function to return (it may also cause other, outstanding, function invocations to be terminated, or even the entire program to terminate).

Calling functions that do not return to their point of call alters the flow of control. **mcc** uses this information to perform a more accurate analysis of the source code.

EXAMPLE

-RET exit

PROFILE

This option is set by various standards profile.

SCANFspec Specify legal conversion specifiers

SYNOPSIS

SCANFset string

DESCRIPTION

This option is used to tell **mcc** which conversion specifiers may occur in the format string passed to calls to *scanf*.

EXAMPLE

mcc prog -SCANFset "WTE"

PROFILE

This option is set by the compiler profile.

Shend Execute string prior to termination

SYNOPSIS

SHEnd <string>

DESCRIPTION

Just before **mcc** returns to the host command line the specified string is used as the parameter to a call to the *system* function. Note: this call is protected by a call to *setgid*.

EXAMPLE

mcc prog -SHE !ls -l *.kic!

Shstart — Execute string before compilation

SYNOPSIS

SHStart <string>

DESCRIPTION

Just before **mcc** starts to check the program source, the specified string is used as the parameter to a call to the *system* function. Note: This call is protected by a call to *setgid*.

EXAMPLE

mcc prog -SHS !date!

Size — Specify type size in bits

SYNOPSIS

Size <type>=<bits>

DESCRIPTION

This option specifies the size in bits for an object declared to have the given scalar type.

<type> is one of *char, short, int, long, float, double*, ldouble, dataptr or codeptr.

The C standard imposes minimum limits on the range of values that integral types may take. Specifying a size smaller than that required to hold this range of values may result in a non-conforming program.

Interaction with alignment. Changing the size of scalar objects may affect the addressing of these objects through an interaction with the `-Alignment` option settings.

Interaction with **mce**. **Mce** will probably have been built to accept scalars of a given size. An error may result through incompatible sizes of scalars in **mcc** and **mce**.

EXAMPLE

mcc prog -S int=24

PROFILE

This option is set by the compiler profile.

Source — Select source platform

SYNOPSIS

SOurce <platform>

DESCRIPTION

Specify the platform on which the program is known to compile and execute correctly. The information on the source platform is used in conjunction with the target platform to give more specific information on likely porting problems.

If no source platform is given the `unknown` profile is used.

EXAMPLE

mcc prog -SOurce 88k

SQL — Enable embedded SQL

SYNOPSIS

SQL <sql_level>

DESCRIPTION

Specify that the source contains embedded SQL. The SQL/2 standard (ISO/IEC 9075:1992(E)) permits various levels of conformance. **OSPC** can be made to flag the use of constructs outside of a specific level by specifying one of:

`ENTry_sql` Check for entry level SQL

`INTermediate_sql` Check for intermediate level SQL

`SQL_2` Check against full SQL/3

`SQL_3` Check against the emerging SQL/3 draft standard

EXAMPLE

mcc prog -SQL SQL_2

SQLVendor Specify SQL vendor

SYNOPSIS

SQL <sql_vendor>

DESCRIPTION

Most vendors of SQL products have added their own extensions. This option can be used to tell **OSPC** which vendors dialect of SQL to expect. Valid SQL vendors include:

`ORACLE6` Oracle version 6

`ORACLE7` Oracle version 7

`INGRes` Ingres

`INFOrmix` Informix

`SYBAse` Sybase

`UNKNown` Vendor is unknown

To obtain an up todate list of those vendors supported in the current release use the `-DET` option, or read the latest **man** page.

EXAMPLE

mcc prog -SQL SQL_2 -SQLV ORACLE6

Srcprof Select additional source profiles

SYNOPSIS

SRCProf <profile>

DESCRIPTION

This option allows other standards to be specified in addition to those already supported through specifying a platform profile.

EXAMPLE

mcc prog -SRC sun4 -SRCProf Xwindows

This specifies the source platform as a sun4 that is using X-windows. This options removes the need to specify a platform profile for every possible system combination (e.g., sun4 under SunOS, X-windows, Motif, etc).

Standard Adhere rigidly to the ISO C standard

SYNOPSIS

STandard±

DESCRIPTION

This option does not cause **mcc** to perform any more checks. Rather it causes the messaging system to use the highest level of message available. The default behaviour is to attempt to recover from errors and use the lowest level message possible. When this option is switched on recovery still occurs, but the highest level message possible, is given. It also disables the use of extensions.

EXAMPLE

mcc prog -ST+

Stackdescend Generate code that uses a descending parameter stack

SYNOPSIS

STACKDescend±

DESCRIPTION

This option affects the way that **mcc** views the memory model used by the dynamic checker. Some cpus have a system stack that grows from low

memory to high memory. Other do the reverse. This option allows either type of behaviour to be mimicked.

EXAMPLE

mcc prog -STACKD+

Stdhdr — Use standard headers before system headers

SYNOPSIS

STDHdr±

DESCRIPTION

OSPC is shipped with a directory containing header files matching the specified contents of a variety of standards. When this option is enabled this directory is searched for a system header. Otherwise any host directories are searched.

Toggling this option on and off allows the user to select between the use of host supplied headers, which may not be fully conforming, and standards conforming headers. Without the need to reorganize the sequence of `-I` options.

EXAMPLE

mcc prog -STDHdr+

Struct — Specify structure or union member checking filename

SYNOPSIS

STRuct <filename>

DESCRIPTION

The named file contains the names of members that are required to be in *struct* and *union* types declared (either trough the use of tags or *typedef*s) in system headers. Multiple `-STRUCT` options can be given. If new members are specified for structures or unions declared in previous struct files the definitions are merged.

Using members of types not defined by standards constitutes implementation defined behaviour.

EXAMPLE

mcc prog -STRUCT PROFILES/standard/ansic/struct

PROFILE

This option is set by the standards profile.

Summary — Give a summary of messages given

SYNOPSIS

SUMmary <display-method>

DESCRIPTION

The summary is generated after all of the source has been processed. The output is sent to the same places as the messages it is summarizing. It is possible to break down the summary by where the warnings occur. It is also possible to obtain an additional summary of the suppressed warnings.

`BYFIle`
: Generate a summary for each file encountered.

`TOTAls`
: Generate one summary for all of the warnings generated.

`CFILes`
: Only generate a summary for warnings given in files ending in `.c`

`SUPPressed`
: Include a summary of warnings that did not appear because they had been suppressed.

EXAMPLE

mcc prog -SUMmary BYFIle -SUMmary SUPPressed

Suppresslvl — Suppress messages below given level

SYNOPSIS

SUppresslvl <number>

DESCRIPTION

The error message file associates one or more numeric levels with each error number it contains. The number given in this option acts as a cutoff. Messages with levels below this value are not given.

Thus if the highest level specified for a given message level is 5 and the cutoff is 6 no messages will ever appear for that error number. If messages at levels 5,6 and 7 are available for a given message and the cutoff is level 6 then the level 6 message acts as the minimum level available.

If this option occurs in the source platform information files then the <number> is associated with source platform error numbers only.

EXAMPLE

mcc prog -SU6

Suwrap — Are signed/unsigned conversions representable

SYNOPSIS

SUWrap±

DESCRIPTION

Assigning a signed quantity to an unsigned type of the same size may result in undefined behaviour (if the value cannot be represented in the target type). The same is also true of unsigned to unsigned conversions. The conversion may be ok for several reasons, for instance the value is known to always be in range or the integral representation is such that the desired result is obtained.

This option is used to specify that conversions involving signed and unsigned types of the same size are ok and should not be flagged.

EXAMPLE

mcc prog -SUWrap+

Tabwidth — Set the number of spaces indented by the tab character

SYNOPSIS

TABwidth <number>

DESCRIPTION

The tab character is treated the same as space character by C implementations. But one tab character is usually equivalent to more than one space character when the source code is displayed. When checking the indentation of statements it is necessary to know how many space characters appear when a tab character is used. This option allows the user to specify this value.

The default value is 8.

EXAMPLE

mcc prog -TABwidth 3 -lint+

T**a**rget — Select target platform

SYNOPSIS

TArget <platform>

DESCRIPTION

Select the platform to which the software is being ported. The information on the target platform is used in conjunction with the source platform (provided via the `-SOURCE` option) to give more specific information on likely problems.

If a target is not specified the `unknown` platform is used.

EXAMPLE

mcc prog -TArget cabstract

T**gtp**rof — Select additional target profile

SYNOPSIS

TGTProf <profile>

DESCRIPTION

This option is similar to `-SRCProf` except it applies to the target platform rather than the source.

EXAMPLE

Trace

Trace configuration files as they are read

SYNOPSIS

TRace <tag>

DESCRIPTION

This option specifies which of the following is to be traced, depending on the tags given:

- `config`
  The configuration file must follow a predefined format. Without any feedback, errors in the layout of this file can be difficult to track down. Using this option causes information on the configuration file to be given, as it is being read in. Information from this trace can be used to locate possible problems in the file layout.

- `include`
  Switching this option on causes the `#include` preprocessor directive line to be output. The full path of the included file is given.

- `input`
  Switching this option on causes lines read from the source file to be displayed on standard output and sent to the log file (if open). The displayed text corresponds to the line read before any preprocessing is performed.

  This option differs from the `-LOG` and `-Listing` options in that it displays the input file as it is read. At this stage no processing has taken place. So the contents match the text that would be seen using an editor.

- `memory`
  When processing very large source files it is possible to run out of memory. Switching this option on causes **mcc** to display the amount of memory remaining at various stages.

  Note: This information is not available on some host platforms.

- `options`
  Since the options are read from several files, each with a different

precedence, it can be hard to determine exactly where an option is being set. Selecting this option will display each line of an option file before it is interpreted.

- `profiles`
  Similar to `-TRACE options` except that the options are only displayed for the profiles that are read in (not the .rc and default files).

EXAMPLE

mcc prog -TRACE profiles

Unsignedchar Plain char to be unsigned

SYNOPSIS

Unsignedchar±

DESCRIPTION

The C standards allows a declaration that simply gives the *char* type specifier to be interpreted as being equivalent to *unsigned char* or *signed char*. The decision is implementation defined.

Switching this option on causes plain *char* to be treated as equivalent to *unsigned char*.

EXAMPLE

mcc prog -Unsigned-

PROFILE

This option is set by the compiler profile.

Via Specify control file to read further options from

SYNOPSIS

VIA <filename>

DESCRIPTION

Via files are text files, created by the user, that contain a list of options. Via files might be used to build a particular set of source files.

Options in a via file are given one per line. A via file may contain a reference to other via files (to an arbitrary depth). Each reference will be processed as it is encountered, and when the end of the file is reached processing will continue in the original file after the `-Via` option.

EXAMPLE

mcc prog -via proj.via

X**c**asesig Case fold external names

SYNOPSIS

XCasesig±

DESCRIPTION

Switching this option on causes a warning to be reported if two identifiers are the same except for their case, within their significant length. The identifiers are still treated as denoting separate objects.

EXAMPLE

mcc prog -XCases+

PROFILE

This option is set by the compiler profile.

X**n**amelength External name significance

SYNOPSIS

XNamelength <number>

XL <number>

DESCRIPTION

This option may be used to specify the number of significant characters in an external identifier. A warning will be given if identifiers are not significant after the given number of characters.

The C standard specifies that only the first six characters in an external identifier need be considered significant.

EXAMPLE

mcc prog -xn 23

PROFILE

This option is set by the compiler profile.

Chapter 4

Platform profiles

4.1 Introduction

This chapter describes the use, composition, and creation of platform profiles. At the top most level platform profiles provide a way of specifying all the attributes of a particular platform. This top level profile is created by merging the information from one or more, lower level subprofiles.

Not all developers are interested in 100% conformance to standards. Platform profiles offer a method of reducing the number of 'non interesting', that is non target specific, warnings.

4.2 The profile hierarchy

The lower level subprofiles contain information about each component that goes into making up a platform. These subprofiles include:

- processor used (cpu)
- application binary interface (abi)
- compiler (compiler)
- operating system (os)
- standards supported (e.g., ISO C, POSIX) (standards)
- platform specific information (misc)

Each of these subprofiles is a self contained entity (except perhaps for misc). One advantage of breaking down platform in this fashion is that information can be reused. For instance once a particular cpu profile has been created it can be reused as part of another platform profile. The subprofiles form a hierarchy in that they are processed in a given order, so a subsequent profile could override the settings of a previous profile. This topic is dealt with more fully later.

4.3 Reducing the output

As already mentioned the main purpose of platform profiles is to reduce the quantity of warnings generated by each tool. Rather than assuming the worst case (the unknown platform), **OSPC** reports the problems specifically associated with porting software from the named source platform to the named target. To this end it makes the assumption that the software will run on the source platform without any errors or warnings being generated.

Given the source and target platform it is possible to filter out warnings about those constructs that are unlikely to cause trouble on the target. Examples of warnings that may be filtered out include:

- Warnings about K&R style usage if both platform use a K&R compiler, or both do not.
- Possible memory alignment problems if both processors have the same alignment restrictions.
- External identifiers that are significant to at least as many characters on the target platform as the source platform.
- Use of identifiers reserved by standards.

4.4 How options obtain their values

With so many possible methods of setting the value of options, users might well become lost in trying to figure out what is going on. The following explanation is designed to give some background on the thinking behind the implementation. This is followed by a road map of what the tools actually do.

OSPC will probably be used by many developers targeting many different platforms using different standards. Although they may be using different source and target platforms they are all, probably, working on the same host. When checking software for portability three platform need to be considered:

1 The source platform.

2 The target platform.

3 The host platform, that is the platform on which the checking is being performed.

The source and target platforms are relatively independent. But there is some interconnection between the host and target platforms.

Each **OSPC** component tool obtains profile information from several sources. Thus it is necessary to know the order in which the files containing the options are processed. The subprofiles, local configuration file and command line options are processed in the following order:

1 Host, source and target are assigned default-default values.

2 The command line is read, and immediate options are processed. These immediate options include such things as the source and target platform profile.

3 The host configuration default file (**PROG/`hstopts`**) is read.

4 The `.mccrc` file is read from the home directory, and overrides host information.

5 The `.mccrc` file is read from the current directory, and overrides host information.

6 Source default file (**PROG/`srcopts`**) is read.

7 Target default file (**PROG/`tgtopts`**) is read.

8 Source platform profile is read.

9 Target platform profile read. Target platform profile options override current host option settings.

10 Command line options are fully processed and override current settings of host information.

4.4.1 Processing the component profiles

Platform profiles are composed of component profiles. Each of these subprofiles may give values to any option. Thus the order in which these subprofiles are processed can affect the final setting of the options. The component profiles are read in the order:

1 cpu

2 abi

3 standards

4 os

5 compiler

6 misc

This ordering is intended to reflect a natural order of precedence; compilers have ultimate control over the code that is produced, and how it interfaces to the O/S etc. A compiler can always generate code to overcome restrictions in the cpu and is not bound by the ABI or any standards. An ABI can be more specific than a cpu definition and may choose to relax hardware restrictions enforced by the cpu. For instance the Intel i860 ABI suggests that compiles might like to support unaligned data accesses, even though extra code would be needed to get around the hardware restrictions on such accesses. The O/S may choose not to implement completely, the standards it purports to support.

In practice most options are best controlled from a single subprofile. But be warned; there are some strange platforms out there.

4.5 Profile administration

The **OSPC** includes a tool `profadm` that provides a simple method of performing various operations on profiles (they can also be edited, by hand, but locating the relevant files can be time consuming). These operations include creating, deleting, and copying profiles, listing profiles and their contents, and determining which platforms use certain profiles. The syntax of the command is:

profadm <option> <profile-type> [<profile-name> [<new-profile-name>]]

where:

<option>:= create|copy|delete|find|list|update

<profile-type>:= abi|compiler|cpu|misc|os|platform|standard

4.5.1 Profadm options

It is not possible to abbreviate the options to **profadm**.

Create Create a component or platform profile

SYNOPSIS

create <profile-type> <profile-name>

DESCRIPTION

Platform profiles are usually created in two stages. First it is necessary to decide the component profiles that will go to make up the platform profile. It is then necessary to see which of the available component profiles can be

used in this new platform profile. If some components are not available they will need to be created.

EXAMPLE

To create a new cpu profile called SPARC

profadm create cpu SPARC

An editor screen (usually vi) screen will appear, with an outline profile in place. The profile consists of a series of options. The options should be edited to match the new processor, and then saved. The default the editor can be changed by setting the environment variable `MCEDITOR`.

Other component profiles can be created by executing a create command for each of the types and names. It is not necessary to have a component profile for each possible component, e.g., generally there will only be either a cpu or an abi profile, but not both.

After all the component profiles have been selected and created the platform profile can be created.

EXAMPLE

To create a platform profile for a cray1 use:

profadm create platform cray1

Copy

Make a copy of an existing profile

SYNOPSIS

copy <profile-type> <profile-name> <new-profile-name>

DESCRIPTION

This option is used for copying all the components of a platform profile.

EXAMPLE

profadm copy cpu i860_h i860_l

This creates a new cpu profile called i860_l that has the same specification as i860_h. This new profile (i860_l) can then be edited with the command:

profadm update cpu i860_l

to change those options that differ between the two profiles.

Delete — remove an old profile

SYNOPSIS

delete <profile-type> <profile-name>

DESCRIPTION

This option deletes an old component or cpu profile. Confirmation is requested before the delete occurs.

EXAMPLE

profadm delete cpu 8088

Find — Find all platforms referring to a component

SYNOPSIS

find <profile-type> <profile-name>

DESCRIPTION

This option searches through the platform profiles, and reports the names of any that reference the named component profile.

EXAMPLE

To discover which platforms reference the 8086:

profadm find cpu 8086

List — Display profile contents, or list of profiles

SYNOPSIS

list <profile-type>

list <profile-type> <profile-name>

DESCRIPTION

This command has two forms. If only <profile-type> is given all the possible names are listed out for that profile. If the profile name is given, then the contents of that profile is displayed.

EXAMPLE

To display all the platforms:

profadm list platform

or the operating systems:

profadm list os

To list the profile for the C standard use

profadm list standard ansic

Update — Edit an existing profile

SYNOPSIS

update <profile-type> <profile-name>

DESCRIPTION

The editor will be invoked on each of the file(s) that can make up the profile.

- Enter the new component name
- Leave the component as it is by entering `<return>` on its own.
- Delete the component entry by entering `delete`

After the components have been set given the option to edit the misc file (or create or delete it) is provided. The general philosophy behind the platforms is to make them as reusable as possible.

EXAMPLE

To implement an OS for system V with BSD extensions, three profiles should be created: SVR4 (the basic System V definition), BSD_ext (the BSD

extensions), and SVR4_BSD (both combined). SVR4_BSD consists of an options file with the following entries

-via PROFILE/os/SVR4/options

-via PROFILE/os/BSD_ext/options

This enables the base files (SVR4 and BSD_ext) to be used in other combinations.

4.6 Contents of profile files

Profile files take several forms:

- References to other files that contain the actual information (using the `-Via` option).
- A list of options, for instance sizes of datatypes.
- Standards information, for instance a list of reserved identifiers.

Because of their bulk it is worthwhile compressing some standards files into a binary form. This has the advantage of speeding up the reading of that information when the tool starts up. Information on the C and POSIX standards is shipped out in binary form (the C standard has nearly 1,000 reserved identifiers), as are the `PROFILE/misc` files (average of 22,000 identifiers per platform).

The layout of those files containing standards information is described earlier in this manual.

4.7 Restrictions

Filenames are not allowed in platform profiles. It is unlikely that a given source file needs to be processed on every invocation of **OSPC**, so such a request is likely to be the result of a bug in the profile information.

4.8 Directory structure

The information about the profiles is stored in the `profile` directory within `check-info`. Profiles of the same type are grouped within a subdirectory of the profile directory (see the diagram below).

profile

platform

<name-of-platform>

cpu abi os compiler standard misc

cpu

<name-of-cpu>

options

abi

<name-of-abi>

options

os

<name-of-os>

options

compiler

<name-of-compiler>

options

standard

<name-of-standard>

options ident errors struct headers assign params conderr psid

The information for the component profiles is stored in an options file. Any other files that are relevant to that component are also stored in the directory, e.g., a file for checking identifier names. They are accessed by specifying an appropriate option in the option file, e.g., `-CHECKId PROFILE/standard/ansic/ident`.

Most of the information contained in the profiles supplied with **OSPC** has been preprocessed into a binary file. This reduces the time required to read in the data, but means that the files cannot be modified by the user.

4.9 Creating new platform profiles

4.9.1 Obtaining the information

As a first approximation the general sales literature is a good place to start. This will probable tell you the cpu being used and the derivation of the O/S. It may even make some claims with regard to supporting various standards. It should be remembered that these claims are being made in sales literature. The next place to look is the C compiler documentation. If the compiler has been formally validated ask to see the documentation relating to the handling of implementation defined behaviours.

Included as part of the **OSPC** distribution is a directory called `deduce`. In this directory is a collection of programs and scripts intended to elicit characteristics of the host compiler. Their output, where possible, takes the form of **OSPC** command line options.

Unix comes with a lot of documentation. The manuals of interest will probably have the word programmer in it somewhere. Manuals with this word in their title tend to contain technical information.

4.9.2 What goes where

Within each subprofile subdirectory is a template file. This contains commented out settings for all options that we believe should be associated with that subprofile. Some options do have a single and obvious place where they belong. While other options could be, and sometimes are, given in several subprofiles.

As mentioned earlier the subprofiles are processed in a well defined order. If it is decided that options should be placed in different subprofiles, care should be taken that there is no interaction with existing platform profiles.

The directory `company` is where company specific profiles should be placed.

4.9.3 Checking it works

The first thing to do is to obtain a detailed help listing using the new platform profile as the target. This should cause the predicted default values to be given for the appropriate options.

Chapter 5

Creating Standards profiles

5.1 Introduction

Given a standards document this chapter describes how to go about creating a profile for the requirements it contains.

OSPC has been designed with a particular set of requirements in mind; those needed to check conformance to the ISO C and POSIX standards. Thus it assumes a particular view of the world. Bindings to new standards are likely to have a similar view. After all they will be providing an interface to software written in the same language.

If your standards document contains a conformance requirement that cannot be detected by **OSPC** please give Knowledge Software a call. We are always happy to discuss adding additional checks to the tool set.

5.2 When can the construct be detected?

The best time to flag any violation of a standards requirement is statically on the source code. For technical reasons this is not always possible. An application may rely on input received at runtime. Thus without knowing the possible input values it is not possible to checks all conformance issues statically.

Those constructs that are flagged when unit of code are linked together tend not to vary across standard bindings. In most cases the rules for cross unit checking given in the ISO C standard are used.

Runtime checks invariably involve the interface between the application program and the system services. The C language checks that the dynamic portion of **OSPC** performs have been built into the runtime checker. A set of interface checking routines have been provided for POSIX.1 and there is a mechanism for users to add their own interface checks.

5.3 Language

Bindings to standards rarely involve themselves in specifying implementation (at least for C, the POSIX.1 standard does suggest that language bindings might be done via language extensions and sites Fortran as a possible candidate). It is possible that an implementation of a particular binding might rely on language extensions. In this case the development

compiler would have to support such extensions and these would be enabled under the compiler subprofile.

Bindings are more likely to suggest restrictions rather than extensions. These might include the number of significant characters in an identifier and the case significance of external identifiers.

C language bindings invariably place declarations in header files. The names of these header files is specified, but their full pathname will depend on implementations. The characters that may occur in a pathname will depend on the host and will form part of the OS subprofile.

5.4 Service interface

5.4.1 Introduction

Before starting to create a new standards profile it is very important to read the appropriate document from beginning to end. The writers of standards do not always make the job of the person interested in conformance job easy (standards are usually written from the perspective of implementors). The most important part of the document to study are the conformance statements. These will often apply to implementations as well as applications using the services provided.

5.4.2 Identifiers

Names of functions, typedefs and objects are at the heart of any service interface. At the simplest level the name of an identifier may clash with an identifier already defined in the users application. So the a list of names defined for use by a given binding is needed. Bindings may also specify names beginning or ending in a given sequence of characters as being reserved for future use.

A binding may specify a convention that should be followed in naming identifiers in its interface, X11 identifiers start with X. It is possible for identifiers reserved by one standard to be used in another. For instance the C standard reserves all identifiers that begin with an upper case letter. This clashes with the X11 convention of starting external names with a capital X. The exceptions list provides a mechanism of reducing cross standards interference.

Standards that provide optional services often specify macro names to be used to test for the availability of these options (feature test macros). The names of these feature test macros become reserved, because of this usage.

Implementations sometimes also use predefined macro names. Vendors that ship software on a variety of platforms often choose to ship identical header files. The appropriate header contents being selected via conditional compilation, based on the definition, or non-definition, of macros. The use of these predefined macros is ‘known’ to the compiler and this information is best placed in the compiler subprofile.

5.4.3 Header files

Within **OSPC** the list of header names serves two purposes. Firstly usage of system headers not on the list can be flagged and secondly use of a system header on the list can be used as a method of suggesting the use of extra standard subprofiles.

Standards rarely specify the full path of headers. They either simply give the header file name and leave the location of this file up to the implementation, or they specify relative paths (in POSIX header files are assumed to exist in a system directory that may contain additional directories, ie *sys*). A standards subprofile should not give full pathnames. It is best to simply give the filename specified in the standard. If an implementation requires a pathname the `-I` option to **mcc** option should be used.

Again standards rarely use character outside of the ISO 646 character set. If they do the `-IDchars` option needs to be used to ensure that the additional characters are not flagged.

5.4.4 Restrictions on use of macro names

Macro names are sometimes more than constant expressions. A standard will usually specify whether a particular macro has to be implemented as a constant expression. Even though there may not be a requirement for a macro to be implemented as a constant expression implementations are free to do so. Relying on such behaviour is non portable and falls outside the bounds defined by the standard requirements.

5.4.5 Arguments in function calls

Calls to some services often required symbolic arguments. The reason for using symbolic arguments is that it allows particular implementations to use different values. The use of these symbolic constants may have to be tempered because of existing practice, ie the use of octal numbers instead of *mode_t* constants in calls to *open*.

5.5 Accredited standards

The POSIX and C standards define two types of conformance, 1) implementation conformance and 2) application (or source code) conformance. In this manual we are interested in the latter. Application conformance is broken down into various categories. The classification of these categories varies slightly between the two standards.

Other standards are not always as rigorous in specifying conformance requirements. Sometimes they may only specify implementation conformance, and sometimes they may not contain any conformance requirements at all.

5.6 Manufacturers standards

Most vendors do not simply implement the requirements of a standard. Many will add extensions and, depending on the age of the standard, some will only partially implement other features. Claims of conformance to standards should be taken with a pinch of salt, unless backed up by a formal validation certificate. Having read the standards document from cover to cover the various minor mentions of 'differences' in behaviour between the implementation and the published standard will have been located. The names of the functions that exhibit different behaviour should be noted.

5.6.1 Industry standards

The main problem with industry standards is that they are often not written down. This makes locating the conformance requirements difficult. If English documentation is not available another source of information is test suites and publicly available applications software (comments can be very revealing). Failing the availability of these sources the profile written has to fall back onto 'know-how'.

5.6.2 Derived or superset standards

Accredited standards documents often form the basis of manufacturer or industry standards. For instance many vendors supply implementations of the graphics standard GKS. These implementations often contain extensions and differences in behaviour from GKS, the base standard. In other cases the supplier of one 'industry standard' might claim conformance to an accredited standard, ie SVR4 compliance to POSIX.1.

The way to handle such derived, or superset standards is to create standards profile datafiles that contains the differences and additional behaviour. The subprofile for that standard then references the standards from which it is derived and the additional information specific to its own requirements (done using the `-Via` option in the subprofiles).

5.7 Interaction between standards

Standards may interact in ways known to the creators of the documents and ways unknown to them. When creating a base standard committees will often provide mechanisms for other standards to interface to the services provided. Providing the appropriate mechanisms for future use involves a certain amount of guess work. It also involves good will on the part of the committee using the interface. Sometimes the interface mechanisms are too strict and at other times to relaxed.

Many standards reserve identifiers for future use. **OSPC** has a mechanism for handling this process and it is described above.

Sometimes standards converge to a common base. The migration of SVID towards POSIX compliance is a good example. In these cases existing practice often has to be maintained

where possible. It is common for some system services to exhibit slightly different behaviour (but still being sufficiently compatible to pass a validation test). Different return types.

5.8 The error file

Each standard profile may have an error file associated with it. This file will contain error numbers and their respective messages. References to the appropriate standard may also appear in this file (see chapter 2 for details on the layout of this file).

If an error number cannot be found in any of the available error files **mcc** will simply display that number.

If an unexpected message is given check that the error numbers being used do not overlap with existing numbers. The error files are read in an unpredictable order. So if error numbers are not unique it is not possible to predict which message will appear.

5.9 Checking new profiles

As the information for a new profile is being gathered test cases should be written. These test cases should cause **OSPC** to both flag and not flag constructs (positive and negative testing). Test cases are not only used to show that the profile has been correctly specified but also help to crystallise thinking. Seeing constructs being flagged in a 'real life' context can be very helpful in understanding how developers will see the results.

In some cases there may be errors in the format of the options. A warning to this effect will be displayed when that profile is read by **mcc**.

The `-Det` option can also be used to view the settings of options. Note that most of the values displayed refer to the target platform. Remember that profiles are read in a given order. The `-Trace profile` option can be used to give a trace of the profiles, as they are being read in.

Chapter 6

The API identifier database

This chapter covers the identifier database used by **OSPC**. In particular the format of files read by the **iddb** utility.

The filename given in a `-CHK` option must have been generated by the **iddb** utility. **iddb** takes the information in one of more text files and creates a binary file containing that information, but which can be searched quickly.

The format of the input files is very simple. It consists of a title line followed by information lines, optionally interspersed with comment lines. Title lines start with a # character in the first column. As with the other files, blank lines are ignored, and lines starting with a star, `*`, are treated as comments.

Possible section title lines include:

1 #api

2 #assigns

3 #define

4 #duplicate services

5 #end

6 #errors

7 #exception

8 #feature test

9 #header

10 #literal

11 #not always const

12 #param

13 #path

14 #protect

15 #reserved

16 #sets

17 #status flags

18 #symbolic constants

Information on `struct`'s is held in a text file and does not form part of a binary database file. These files are input to **OSPC** using the `-STRUCT` option.

6.1 #api

This directive is followed by an identifier that names the API which this file is associated with.

```
#api posix_1
```

This is the API name is written out to the `.api` file when the `-API` option is switched on. All of the input files given on the command line to `iddb` must have the same API name.

6.2 #assigns

This section title introduces a list of identifiers. Information on which integer literal constants, or symbolic names, may be compared with and assigned to these identifiers occurs after each identifier.

The format of each line is:

<**name**> '=' <**modifiers**>

Where `<name>` is an API scalar object identifier. Functions may be specified by following the name with round brackets, **()**. Members of structures or unions may be specified using dot select notation.

`<modifiers>` is a list of the one or more of the following:

- <**number**> a valid particular number.
- **+ve** the routine may return any positive number.
- **-ve** the routine may return any negative number.

- **any** the routine may return a positive or negative number. If given on their own it means the numbers may be returned, but their values hold no significance.

- **{+ve} {-ve} {any}** if enclosed in brackets it means the numeric values have significance and may be compared against any number in that range.

- **{<number>,<number>}** a range of numeric values that the routine may return.

- **<identifier>** a symbolic name.

```
*
*      POSIX_1/assign     Lastmod  1 Aug 92  DJ
*                         Created  4 Jul 92  DJ
*
*********************************************************************

#assigns

errno = 0 E2BIG EACCES EAGAIN EBADF EBUSY ECHILD EDEADLK EDOM
      = EEXIST EFAULT EFBIG EINTR EINVAL EIO EISDIR EMFILE EMLINK
      = ENAMETOOLONG ENFILE ENODEV ENOENT ENOEXEC ENOLCK ENOMEM
      = ENOSPC ENOSYS ENOTDIR ENOTEMPTY ENOTTY ENXIO EPERM EPIPE
      = ERANGE EROFS ERRRNO ESPIPE ESRCH EXDEV

exec()         = -1
wait()         = -1 0 positive
sigismember()  = -1 0 1
alarm()        = 0 {positive}
pause()        = -1
getgroups()    = -1 {0, 32767}
getpgrp()      = {positive}
sysconf()      = -1 {positive}
chdir()        = -1 0
creat()        = -1 positive
fcntl()        = -1 {any}
cfgetospeed()  = B0 B50 B75 B110 B134 B150 B200 B300 B600 B1200
               = B1800 B2400 B4800 B9600 B19200 B38400
cfsetospeed()  = -1 0
cfgetispeed()  = B0 B50 B75 B110 B134 B150 B200 B300 B600 B1200
               = B1800 B2400 B4800 B9600 B19200 B38400
fileno()       = -1 STDIN_FILENO STDOUT_FILENO
               = STDERR_FILENO positive

stat.st_mode   = S_IRWXU | S_IRWXG | S_IRWXO | S_ISUID | S_ISGID

flock.l_type   = F_RDLCK F_WRLCK F_UNLCK
flock.l_whence = SEEK_SET SEEK_CUR SEEK_END

termios.c_iflag = BRKINT | ICRNL | IGNBRK | IGNCR | IGNPAR |
                = INLCR | INPCK | ISTRIP | IXOFF | IXON | PARMRK
termios.c_oflag = OPOST
termios.c_cflag = CLOCAL | CREAD | CSIZE | CS5 | CS6 | CS7 |
                = CS8 | CSTOPB | HUPCL | PARENB | PARODD
termios.c_lflag = ECHO | ECHOE | ECHOK | ECHONL | ICANON |
                = IEXTEN | ISIG | NOFLSH | TOSTOP

#end
```

6.3 #duplicate services

This section title is used to specify a list of API's that offer similar services (using an identical named external identifier). This can happen because of newer API's building on existing API's by adding additional functionality. For instance POSIX extends the definition of some of the services found in the ISO C standard.

By listing API's that offer duplicate services it is possible to reduce the number of false warnings concerning use of these services.

```
#duplicate services

ISO_C
POSIX_1

#end
```

6.4 #end

This section title is used to indicate the end of useful information within a text file. Files need not end in a `#end`, end of file also signals the end of useful information.

6.5 #error

This section title is used to change the error message associated with a reserved identifier after it has been *#undef*'d.

<initial-error> '->' [<final-error>]

Both `<initial-error>` and `<final-error>` must occur in the list of *#define*'s that occurred earlier in the file.

```
#errors

* A reserved macro cannot be undefed and then re-#defined
ps_res_macro    -> ps_no_def_macro

* Neither can a macro which might cover a function
ps_macro_id      -> ps_no_def_macro
```

6.6 #exception

If a matching reserved identifier is found, it is only reported if it doesn't also have a matching entry in the exception list. This section title allows exceptions to the reserved identifier list to be specified. The format of each line is the same as that in the reserved section, except no error number is given.

<regular-expression> [<specifier list>]

One use of this facility is to prevent the flagging of declarations that are reserved in one standard, but used in another. For instance the use of names beginning with _X, in X11 clashes with the ISO C requirements.

```
#exception

_POSIX_SOURCE              macro

#end
```

Note. Like identifiers in a `#reserved` section, identifiers in a `#exception` section need to match in the specified context.

6.7 #feature test

This section specifies a list of possible feature test macro identifiers. It occurs before the `#protect` section which uses those identifiers.

```
#feature test

_POSIX_JOB_CONTROL
_POSIX_SAVED_IDS

#end
```

6.8 #header

This section title associates header names with profile names.

<header name> <profile name list>

The database file generated by **iddb**, containing header information should be given to the `-HEADER` option.

```
#header

stdio.h ansic
stdlib.h ansic
string.h ansic
time.h ansic

#end
```

6.9 #literal

This section title marks the start of a list of integer literal constants. A context and usage may be associated with each literal.

The format of each line is:

<context> [<usage_list>] <error_number> '=' <modifiers>

where `<context>` may be one of:

1 **any**, a literal in any context.

2 **ppif**, a literal occuring in a *#if* expression.

3 **init_expr**, a literal occuring in the initialization expression of an objext definition.

4 **stmt_expr**, a literal occuring in a statement (excluding a controlling expression).

5 **ctl_expr**, a literal occuring in a control expression, ie *if* statement.

6 **dcl_expr**, a literal occuring in a declaration, ie array size, or bit-field width.

`<usage_list>` is an optional list of specifiers giving extra information on how the use of literals should be matched. If this list is empty, all three uses may match. The optional values are:

1 **typed_lit**, the literal appears in the source code, as typed in.

2 **macro_lit**, the literal does not visibly appear in the source code, but is used via a macro substitution.

3 **enum_lit**, the literal does not visibly appear in the source code, but is used via an enum literal substitution.

`<error_number>` is an error number or an identifier previously defined to represent such a number.

`<modifiers>` are the literal values. They are represented using the same notation as that specified for the field of the same name given above for `#assign` lines.

```
#literal

any 3000                 = {10,20}
ctl_expr 3001            = +ve
stmt_expr typed_lit 3002 = -ve

#end
```

6.10 #not always constant

This section title marks the start of a list of macro names that need not be constant. It is followed by a list of identifiers, one per line.

```
#not always const

CLK_TCK

#end
```

6.11 #param (symbolic parameters)

The information occurs in two main sections, the first defines the sets, and the second specifies the parameter of a function that requires a particular combination of symbolic arguments.

Note that this file does not contain any error numbers. A single error number is used, internally. The names of the allowable symbolic arguments being created from the contents of this file and merged into the error message.

The legal combinations of symbolic macros that may be passed in API calls is often complex. **OSPC** uses a set notation to describe which symbols are legal in each context. It is probably easier to get the guist of the format by looking at a few examples, and giving the full details later:

Example 1. The third argument to *lseek()*:

```
#sets

$lseek_args     = { SEEK_END, SEEK_CUR, SEEK_SET }

#param

lseek           param 3 in $lseek_args
```

This simple example could also have been specified with the entry:

```
#param
lseek           param 3 in { SEEK_END, SEEK_CUR, SEEK_SET }

#end
```

The convention is for set names to begin with a dollar, *$*, character, so that they are clearly differentiable from real identifier names.

Example 2. The second parameter to *access()*:

```
#sets

$access_args    = { F_OK } or { X_OK, R_OK, W_OK } +

#parameters
access          param 2 in $access_args

#end
```

Here, the legal values which can be passed to *access()* are either *F_OK*, or one or more of {*X_OK*, *R_OK*, *W_OK*} bitwise exclusive or'd together.

Example 3. The second parameter to *open()*:

```
#sets

$rw_type        = { O_RDONLY, O_WRONLY, O_RDWR }
$oflags         = { O_APPEND, O_CREATE, O_EXCL, ONOCTTY,
                    O_NONBLOCK, O_TRUNC } *
#parameters

open            param 2 in $rw_type and $oflags

#end
```

Here the second parameter must have one of the constants in the `$rw_type` set, and this may be inclusive OR'd with zero or more of the constants defined in `$oflags`.

6.11.1 #sets

Each line has the format:

<set-name> '=' <set-specifier>

where:

<set-specifier> ::= <simple-set> |
<simple-set> <set-op> <simple-set>

<simple-set> ::= <set-name> | <simple-defn>

<simple-defn> ::= '0' |
'{' <set-list> '}' <set-modifier>$_{opt}$

<set-list> ::= <macro-name> |
<set-list> ',' <macro-name>

<set-op> ::= 'and' | 'or'

<set-modifier> ::= '*' | '+' | '?'

Note that this grammar only supports one level of operand. Complicated expressions can be built up by defining and joining together many `<set-specifiers>`.

The `<set-modifier>`s use the same convention as **grep** to indicate how many `<macro-name>`s from a set can occur. No modifier implies one, '*****' means zero or more, '+' - one or more, and '**?**' means zero or one. The '**0**' is a useful way of indicating that a complicated set of names are optional, or for re-using a set definition:

$raise_args = 0 or $signal_names

6.11.2 #param

Each line has the format:

<function-name> 'param' <number> 'in' <set-specifier>

When a call to `<function-name>` occurs in the source code, argument `<number>` is checked against the set of allowable values for that parameter.

6.12 #path

One difference associated with matching the contents of strings verses matching identifiers is that the latter requires an complete match. That is to say it is not possible to match on the first few characters or the last few characters, ignoring the rest of the characters. In the case of checking string literals it is very likely that the required pattern will be surrounded by other characters. For this reason matching against string literals is based on the longest possible match.

Given the set of patterns:

```
*
#define e_ref_usr_bin      1234
#define e_ref_vi_ed        1235
#define e_ref_vital_file   1236
#define e_music_to_my_ears 1237
*
#path

/usr/bin         e_ref_usr_bin
/usr/bin/vi      e_ref_vi_ed
/usr/bin/vital   e_ref_vital_file
/etc/b[io]ng     e_music_to_my_ears

#end
```

The string *"/usr/bin was here"* will only match against the first pattern. The string literal *"The editor is in /usr/bin/vi"* will match against the first two patterns, but the second is longer; so its error number will be given. The literal *"The file /usr/bin/vital must not be lost"* will match on the first three patterns, with the third being the longest match.

It is possible to specify regular expressions. For details see the `#reserved` section.

6.13 #protect

The `#protect` section takes the same form as the `#reserved` section. The file is split into three parts:

1 **#define** lines. Give mnemonics to error numbers.

2 **#feature test** section. List of feature test macros.

3 **#protect** section. List of 'protected' identifiers.

The `#define` lines have exactly the same format as those that occur in a `#reserved` section.

Lines in a `#protect` section specify identifiers whose use should be protected with a feature test macro. They have a format similar to lines in a `#reserved` section.

<ftm-name> <ident-name> <error-mnemonic> [<specifiers>]

The `<ftm-name>` should be the name of a feature test macro listed in a previous `#feature test` section.

The rest of the line has the same format as a line in a `#reserved` section, with the following exceptions:

- `<ident-name>` cannot be a regular expression.
- The specifiers `define` and `declare` are not allowed (since we are dealing with the point of usage).
- The `macro` specifier is used to check for the use of a macro, and the other 'pseudo-namespace' specifiers have no meaning.

```
*
*          POSIX1/ftms  Lastmod    11 Aug 92  DJ
*                       Created    27 Mar 92  GH
*
*****************************************************************
* Same format as an ident file - in fact it is an ident file!
*

#define ps_need_protection         1420     Needs to be protected
#define ps_protect_action          1421     Action changes ...

#feature test

_POSIX_JOB_CONTROL
_POSIX_SAVED_IDS

* Table B-1 (in language specific definition)

_V7
_BSD
_BSD4_2
_BSD4_3
_SYSIII
_SYSV
_SYSV3
_SYSV4
_XPG1
_USR_GROUP

*
```

```
* These identifiers need to be protected by themselves since they
* might not always be defined.
*
ARG_MAX
CHILD_MAX
LINK_MAX
MAX_CANON
MAX_INPUT
NAME_MAX
OPEN_MAX
PATH_MAX
PIPE_BUF
STREAM_MAX

#protect
*
* Protecting ftm    Name       Error              specification
* P1003.1 3.2.1.2
_POSIX_JOB_CONTROL WUNTRACED    ps_need_protection    macro
_POSIX_JOB_CONTROL WIFSTOPPED   ps_need_protection    macro
* P1003.1 3.3.1.1
_POSIX_JOB_CONTROL SIGCHLD      ps_need_protection    macro
_POSIX_JOB_CONTROL SIGCONT      ps_need_protection    macro
_POSIX_SAVED_IDS   setuid  ps_protect_action   external identifier

#end
```

6.14 #reserved

This section title introduces a list of reserved identifier names. A context may be associated with each identifier.

The information within a `#reserved` section is split into three sections:

1 **#define**. Optionally specified mnemonics for error numbers.

2 **#errors**. Specifies the new error number, to be used if a symbol is `#undef`'d (in the applications source code).

3 **#reserved**. List of reserved identifiers, along with the context in which they are reserved.

6.14.1 Error number mnemonics

The lines in this section take the form:

'#define' <error-mnemonic> <error-number> [<comment>]

This format is very similar to the `#define`'s found in error files. The difference is that in `#reserved` sections they are used to associate a mnemonic with an error number, rather than with an error level.

6.14.2 Reserved identifiers (#reserved)

These lines contain the bulk of the information about reserved identifiers. Each entry gives the name of the identifier, its name space, scope, any headers that must be included, and the error number to be reported if all these contexts match. Lines have the format:

<regular-expression> <error-mnemonic> [<specifier list>]

The regular expressions may consist entirely of ordinary characters (e.g., `strcmp`), or may contain special characters (e.g., `E:a.*`). Special characters recognised include:

- **x** Any ordinary character matches that character.
- \ Backslash quotes any character (e.g. \:, \\ or \[)
- **.** Matches any single character
- **:A** Matches any upper case character
- **:a** Matches any lower case character
- **:d** Matches any decimal digit
- **:n** Matches any digit or lower case character
- **:N** Matches any digit or upper case character
- ***** An expression followed by an asterisk matches zero or more occurrences of that expression: fo* matches f, fo, foo etc.
- **+** An expression followed by an asterisk matches one or more occurrences of that expression: fo+ matches fo, foo, fooo etc.
- **[]** A string enclosed in square brackets matches any character in that string. If a circumflex (^) is the first character in the string, it causes the expression to match any character that isn't in the string. E.g., [xyz] matches 'x' and 'z', whilst [^xyz] matches a, but not y. A range of characters may be specified by separating two characters by a - (e.g., [a-z]).

The `<error-mnemonic>` must have been given in a previous `#define` line in the errors section. The error number should also have a corresponding entry in an associated error file.

The optional list of specifiers provide a method of narrowing down the context in which the name is reserved. For instance they allow the name to be reserved as an identifier, but not as a macro. The specifiers can occur in any order and cover the following identifier attributes:

Namespace `identifier`, *`label`*, *`tag`*, *`macro`*, *`header`*, *`include`*, *`field`*, *`no_def_macro`*, *`res_macro`* or `macro_ident`.
As well as the namespaces defined by the C standard, there are also pseudo namespaces which are used for handling `#undef`'s correctly. These pseudo namespaces are explained in detail in the section dealing with `#undef`.

Linkage `external`, `internal` or `none`. See the C standard for details.

Scope `file`, `block`, `function`, or `prototype` scope.

Definition `define`, `declare`, `default`. A definition is a declaration that also causes storage to be allocated. The specifier `default` only applies to identifiers that are default declared, *extern int ident()*.

Included header `included header.h`, `included !header.h`
If a header is specified, the entry only matches if the header has been included. Prefixing the header name with a *!* causes a match to occur if the given header hasn't been included. The headers must have been specified in a previous 'valid header' file. If a header name is omitted, then no checks are made against the list of headers that have been included.

Identifier usage `function_id`, `object_id`, or `typedef_id`. When an `Identifier` namespace is given these specifiers may be used to narrow down the matching of the sort of identifier being declared.

If a specifier for an attribute isn't given, the entry matches against an identifier having any of the options available for that attribute.

Any of the namespaces that indicate a reserved macro name (`macro, res_macro` or `macro_ident`) also match against any of the other name spaces. This reflects the action of a compiler, where, because macros are substituted first, an identifier in any namespace will clash with a macro name.

The specifiers `any`, `scope`, `namespace` and `linkage` may also be given in the specifier list. They are ignored, but can help to make the file more readable. The use of `any` simply makes it explicit that a match may occur in any context, the default if attributes are omitted.

```
*
*        POSIX1/ident        Lastmod     27 Mar 93  DJ
*                            Created      3 Feb 92  DJ
*
*****************************************************************

* A macro which is marked as reserved
#define ps_res_macro       1400

*A macro reserved, but can be used for any use if undef'd first
#define ps_fut_macro       1401

* A reserved external identifier
#define ps_ident   1402

* A identifier reserved for future use
#define ps_fut_ident       1403

* An identifier which may be covered by a macro definition
#define ps_macro_id        1404

* An future reserved identifier which might be covered by a macro
#define ps_fut_macro_id    1405

* A reserved file scope identifier/tag (e.g. a typedef)
```

```
#define ps_file_scope      1406

* A reserved tag name for a structure
#define ps_tag_name        1407

* Cannot redefine this macro even after an #undef (used in the
* error transitions)
#define ps_no_def_macro    1408

* Reserved for any uses....
#define ps_any_use 1409

* This doesn't have to be defined as a function
#define ps_maybe_not_id    1410

* This library function needs a type from the appropriate header
#define func_ref_hd        1415

#errors

* A reserved macro cannot be undefed and then re-#defined
ps_res_macro    -> ps_no_def_macro

* Neither can a macro which might cover a function
ps_macro_id     -> ps_no_def_macro

* P1003.1 Table 2-2 Says that POSIX.1 only reserves identifiers
* for future use in two ways:
*  1. As macros which can be undefined & then used freely
*  2. Reserving the symbols for all uses.

#reserved
*
* expression           err_num        specifiers
*
d_.*         ps_any_use     any          included dirent.h
l_.*         ps_any_use     any          included fcntl.h
F_.*         ps_fut_macro   macro        included fcntl.h
S_.*         ps_fut_macro   macro        included fcntl.h
gr_.*        ps_any_use     any          included grp.h
.*_MAX       ps_any_use     any          included limits.h
pw_.*        ps_any_use     any          included pwd.h
sa_.*        ps_any_use     any          included signal.h
SA_.*        ps_fut_macro   macro        included signal.h
st_.*        ps_any_use     any          included sys/stat.h
S_.*         ps_fut_macro   macro        included sys/stat.h
tms_.*       ps_any_use     any          included sys/times.h
c_.*         ps_any_use     any          included termios.h
V.*          ps_fut_macro   macro        included termios.h
B:d.*        ps_fut_macro   macro        included termios.h
*
* LC_:A.*  defined in the ansic ident file.
*
.*_t                ps_any_use     any

*
*  dirent.h
*

dirent    ps_tag_name    file scope tag       included dirent.h
DIR       ps_file_scope  file scope identifier   included dirent.h

*
* We still need 3 cases even though the first and last seem  to
* be sufficient, in order to cover the case
* #include <dirent.h>
* #undef opendir
* int opendir;
*
```

```
opendir     ps_macro_id    macro_ident            included dirent.h
opendir     ps_ident       define external identifier
opendir     func_ref_hd    external identifier    included !dirent.h

readdir     ps_macro_id    macro_ident            included dirent.h
readdir     ps_ident       define external identifier
readdir     func_ref_hd    external identifier    included !dirent.h

rewinddir   ps_macro_id    macro_ident            included dirent.h
rewinddir   ps_ident       define external identifier
closedir    ps_macro_id    macro_ident            included dirent.h
closedir    ps_ident       define external identifier

*
*  errno.h
*

E2BIG       ps_res_macro   res_macro       included errno.h
EACCES      ps_res_macro   res_macro       included errno.h
EDEADLK     ps_res_macro   res_macro       included errno.h
*EDOM - See ansic
EEXIST      ps_res_macro   res_macro       included errno.h
EFAULT      ps_res_macro   res_macro       included errno.h
EINVAL      ps_res_macro   res_macro       included errno.h
*ERANGE - See ansic
EXDEV       ps_res_macro   res_macro       included errno.h

*
*  fcntl.h
*

flock       ps_tag_name    file scope tag       included fcntl.h
*
* P1003.1 6.5.2.2
*
FD_CLOEXEC  ps_res_macro   res_macro       included fcntl.h
F_DUPFD     ps_res_macro   res_macro       included fcntl.h
F_GETFD     ps_res_macro   res_macro       included fcntl.h

#end
```

6.15 #status flags

The format of lines in a `#status flags` section is very similar to that of the `#assigns` section. They have one of the forms:

<function id>'()' 'sets' <id> <modifiers>

or

<function id>'()' 'checks' <id> <modifiers>

In the first case a call to `<function id>` sets `<id>` to one of the values given in `<modifiers>`. In the second case a call to the function checks that the `<id>` has one of the values given in `<modifiers>`.

```
*
*            statechk Lastmod  10 Apr 96  DJ
*                     Created  23 Jul 92  GH
*
```

```
**********************************************************************

#status flags

call1() sets flag {1,3}
call2() sets flag 11 22

call3() sets flag3 {1,6}
call3() sets flag3_1 {4,5}
call3() sets flag3_2 {6,7}

call4() sets flag4 {99,100}
call5() checks flag4

call6() sets flag6 2

#end
```

6.16 Structure files

A struct file consists of a list of the names of `struct`, `unions` or `typedef`s, followed by a list of the members that they contain. The indentation within the structure is determined by the number of whitespace characters on the start of the line (either spaces or tabs).

Note: Structure files are not processed by **iddb**. The information is read in directly, by **mcc**, from the text file.

To indicate a symbol is a `typedef` the modifier `:typedef` is appended to the name. The default action is to assume the name represents a `struct` or `union`. This can be explicitly specified by appending `:union` or `:struct`.

Multiple struct files can be specified, and the definitions from all of the files will be merged.

Comment lines, as for other files, are indicated by starting a line with the star, `*`, character.

Example: The entry for *struct dirent* in the POSIX.1 structure file has the specification:

```
dirent:struct
        d_name
```

and that for the XPG structure file contains:

```
dirent:struct
        d_ino
```

Since XPG is a superset of POSIX its platform profile includes the standard profile information for POSIX.1, so only the extra fields need be specified - the two definitions will be merged. Thus under XPG both *d_name* and *d_ino* are valid member references for objects of type *struct dirent*.

Referencing the POSIX platform profile, rather than filling in all of the available fields in the XPG struct file gives important information. It specifies how XPG differs from POSIX.

Example:

The *div_t* type defined in `<stdlib.h>` has the following entry in the ISO C structure file:

```
div_t:typedef
        quot
        rem
```

Note the indentation of the members, this must be one whitespace character. Here it is a tab character, (not one tab position's worth of whitespace).

The structures defined can be fully hierarchical - with members of structures containing members of structures, containing and so on ad infinitum (actually 15 levels are supported).

Example:

The format of the structure file should reflect the way in which structures are defined. As an example consider information that might be held about a person:

```
struct person_info {
    char *     name;
    int        age;
    struct {
        char *     street;
        char *     town;
        char *     county;
        char       postcode[PostCodeLen];
        char *     country;
    }  address;
};
```

This might have a corresponding structure definition:

```
person_info:struct
        name
        age
        address
                street
                town
                county
                postcode
```

So in:

```
struct person_info joe;

joe.address.town = "London";
joe.address.country = "England";
```

the use of country as a field of address will be flagged.

Chapter 7

Identifier checking

7.1 Introduction

The type of identifier checks carried out usually varies according to context; declaration or use. Declarations may clash with identifiers declared by API's. Referenced identifiers may be platform specific or rely on properties that go beyond those specified by an API.

7.2 Declaration/definition checks

Both the ISO C and POSIX Standards reserve certain identifier names. These include the names of functions and types that are already defined, and also those that may be added to the libraries in the future. Using one of these reserved names may cause unforeseen problems either when the program is ported to a new environment, or when the C compiler (and its libraries) are upgraded. Problems can arise from several different sources:

- Name clashing of externals.
  Although it is unlikely that a program will contain functions called *printf()* or *strcmp()*, names such as *step, advance, compile* or *timezone* (which are all defined by XPG) are not so obvious and may slip through the net, possibly causing problems when the program is ported.

- Differences in implementation.
  setjmp() is a good example of this. The C standard does not specify whether *setjmp()* is implemented as a macro, or as a function call. On some systems a program that does not include the file `<setjmp.h>`, but declares the function *setjmp()* instead, may work fine (if it has been implemented as a function). However problems will occur if this program is then moved to a system that implements it as a macro.

- Clashes with macros.
  Both the C and POSIX standards specify that any of the library functions may be implemented via a macro (a function must also be provided in these cases). This means that inside a system header an implementation may have defined a macro for each function the header declares.

  #define strcmp __strcmp

  One reason for this usage might be to generate inline code for the comparison. The problem that this usage causes arises because macros are substituted before any syntax analysis takes place. Thus if a member of a structure is called strcmp (or

step, remembering the previous example) it would also be replaced by the macro body.

OSPC checks the name of every identifier that is declared or defined, against a list of reserved names for the target platform profile. These names may be reserved for all occurrences, or for particular scopes, namespaces, whether a particular header has been included etc. The names that are checked can be added to, enabling other standards to be supported.

7.3 Checking algorithm

The following algorithm is used to check the declaration/definition of identifiers:

1 Does the identifier match any of the reserved identifiers whose regular-expressions consist of ‘ordinary‘ characters? If so continue at step 4.

2 Does the identifier match a reserved identifier containing one or more wild cards? If not, the identifier is not flagged.

3 Does the identifier match any of the reserved ‘ordinary’ character names, regardless of namespace etc? If so exit checking without flagging the identifier. This situation occurs when an ‘ordinary’ character reserved name has been `#undef`’ed, hence it is not caught at step 1.

4 Does the identifier match any of the exception regular expressions (whether this involves character only matching or wild cards is not relevant). If so exit checking without flagging the identifier.

5 The identifier has matched and is not in the exceptions list, so issue an error number.

7.3.1 Action on #undef

The ISO C rules regarding the processing of an `#undef` can be quite involved. If a symbol is only reserved as an identifier, or as a tag then a `#undef` has no effect on the entry. However if the object is a macro the behaviour is rather more complicated. The following pseudo namespaces deal with the various conditions:

macro This is the namespace that is used for a normal macro that is reserved but can be used by the program after it has been `#undef`’d. Some of the macros defined by POSIX.1 (e.g those with prefixes of `S_`, `V`) fall into this category.

res_macro This corresponds to the C Standard idea of a reserved macro. If a reserved macro of this type (e.g. `offsetof()`) is `#undef`’ed it should still not be defined within the program as a macro name. If a macro of this type is `#undef`’d the entry type is changed to

`no_def_macro`. So that future attempts to `#define` the identifier will be flagged.

no_def_macro Entries in the reserved list with this namespace indicate names of macros that cannot be *#define*'d. These entries are normally created after a identifier in the reserved list has been `#undef`'d.

macro_ident The C and POSIX standards state that a library function may additionally be implemented as a macro. The `macro_ident` namespace marks a identifier that is reserved as a potentially macro covered identifier (e.g *putc*, *printf*).

When an entry is `#undef`'d as well as changing the reserved identifier's namespace, the error associated with the entry may be changed. The change is specified in the `#errors` section of ident checking file.#undef[undef]

7.3.2 Programming style example

The identifier checking can also be used to check against certain company standards. The following example illustrates how to add new identifier checks:

```
*
* Bloggs Company internal identifier standards file
* First created 3 March 1992
*
* First define some error numbers
#define  err_name_too_short       8000
#define  err_not_meaningful       8001

* Now come the reserved names ....
#reserved

* pattern         error message          scope/linkage modifiers
*
* Complain about one, two or three letter identifiers
*
.                 err_name_too_short     any
..                err_name_too_short     any
...               err_name_too_short     any

* Complain about some silly variable names as well
* 'Temp' & case variations
[tT][eE][mM][pP]  err_not_meaningful     any
* 'buff'
[bB][uU][fF][fF]  err_not_meaningful     any

#exception
* Allow a sensible exception to the three letter rule
end                                      field
*
```

The associated error file:

```
*
* Error file for the bloggs internal name convention
*
#define expundef   5

8000    expundef          The name '%s' is too short\\
```

```
Bloggs standards Sec2.1.3.4.5.6a Variable names.
8001    expundef          '%s' is an unhelpful variable name
Bloggs standards Sec4.5.12.3   Hangable offenses.
*
```

7.4 Feature test macros

There are many different POSIX standards. POSIX.1 is the base standard. The other standards (most still in draft form) specify extensions to this base. POSIX.4, defines standard extensions which are designed to support 'real time' operation. For many applications, however, the base POSIX services will provide all the functionality required to write an application; other non-POSIX standards may however still be used.

Because the extensions are not needed by most applications, a POSIX compliant platform is under no obligation (except customer pressure) to implement them. Indeed a platform may support some of the functionality defined in POSIX.4, but not all of it (by not defining certain feature test macros whole swathes of the standard can be omitted). In order that a application may have some method of knowing whether these facilities are available, the standard requires that certain 'feature test macros' be defined, in the header `<unistd.h>`, for each area supported. An application can then check for the existence of this feature test macro, before using any of the functionality it refers to.

Example: (POSIX.4 Memory sharing)

```
#ifdef _POSIX_MEMORY_SHARING
/* Create a new shared memory file ... */
file = mkshm(shm_file, SHM_PERSIST, shm_size);
#else
.....
#endif
```

Here *_POSIX_MEMORY_SHARING* is the feature test macro, and *mkshm()* (and *SHM_PERSIST*) are the identifiers which are being protected.

7.5 Identifier usage

7.5.1 Symbolic parameters

Several of the functions defined by POSIX require symbolic parameters (in the form of pre-defined macros, in system headers) to be passed to them. e.g., the third parameter of *lseek()* should be one of the macros *SEEK_SET*, *SEEK_CUR*, or *SEEK_END*. Historically, rather than using these macros, the values 0, 1 and 2, respectively, have been used. POSIX, however, by defining the action of *lseek()* in terms of these macros leaves the implementation free to give them any (integer) values. For a program to be fully POSIX conformant it must use the macro names defined in the headers. Often problems will be caused by using the numbers traditionally used under Unix, instead of the macros, when porting to a proprietary, yet POSIX conformant, platform.

7.5.2 Assignment to and comparison with

Many system service routines only return a small set of values. Sometimes they do not even return explicit values, but rather properties, ie a positive value. The assign section of a database file contains information on the values that may be assigned to, or compared against when referring to an object or function of a specific name.

7.5.3 Optionally defined macros

Some of the macros that POSIX.1 specifies as being in the header `<limits.h>` need not be defined on some systems. So an application that uses these macro names should first check that they have been defined, ie they are their own feature test macros. Checking for this can be done using the feature test macro mechanism.

An example of one of the POSIX.1 optional macros in `<limits.h>` is:

```
#feature test macro
ARG_MAX
#protect
ARG_MAX      ARG_MAX     e_should_check_exist     macro
#end
```

OSPC will then complain if `ARG_MAX` is used without a `#ifdef` checking that it exists.

Chapter 8

Cross unit checker (mcl)

8.1 Introduction

This chapter takes a detailed look at all of the **mcl** options. The standard help text only lists those options which a user might wish to change with any frequency. A complete list can be obtained by using the `-DETail` option.

mcl -DETail

Most of the options considered to be 'details' relate to platform specific functionality. Thus the setting of these options would normally be controlled by the platform profiles.

Traditionally, linking is the process of joining two or more separately compiled source files, to create an executable program. The **OSPC** cross unit checker performs this and other tasks.

mcl performs three main functions:

1 It carries out type checking of declarations and definitions between translation units. This is its main checking role within the **OSPC**.

2 It reorganizes .kic files into a form suitable for execution; the traditional linkers job.

3 It provides a method of joining multiple .kic and .klc files into one file, deleting and replacing translation units within a .klc file. This job might normally be carried out by a separate utility, a librarian, on other systems.

8.2 Options

This section lists all of the options available in the cross unit checking portion of **OSPC**.

Notation: In this section the minimum abbreviation for each of the options is given by the portion in capital letters. That is one or more of the lower case letters may be omitted.

ATP Purge Audit Trail from file

SYNOPSIS

Atp±

DESCRIPTION

Although the word purge is used the effect is not to copy any audit trails from the input files to the output. It also has the effect of not creating an audit trail for the current cross unit check. The main use for this option is to slightly reduce the size of the .klc file.

EXAMPLE

mcl prog -ATP+

Atv View Audit Trail

SYNOPSIS

ATV±

DESCRIPTION

Lists the audit trails contained in the input files being checked, together with the audit trails for the current invocation of **mcl** to standard output.

EXAMPLE

mcl prog -ATV+

Body Do we check macro bodies between files

SYNOPSIS

Body±

DESCRIPTION

When switched on and more than one file is being checked this option causes **mcl** to check macros for consistency. This checking involves comparing the bodies, and in the case of function like macros, the parameters, of macros with the same names. The checking rules used are the same as for multiple macro definitions within the same translation unit.

The standard doesn't mandate that these checks need to be done. However, inconsistencies between macro definitions in different source files can be the cause of hard to locate problems (also see the `-MAcro` option).

EXAMPLE

mcl prog -Body-

Buildmce*

Build a new executor

SYNOPSIS

BUILDmce±

DESCRIPTION

Switching this option on causes a new executor to be built. When host compiled units, or system libraries, are being used it is necessary to build a new executor to run the program. This new executor contains the 'glue' necessary to call the routines from interpreted code.

If units are being checked together to form a .klc library, this option should be switched off. It should only be switched on when a final program is being produced.

To run the program type **mce.<prog> <prog> <arguments>**. It may be necessary to type **unhash** to inform the shell about the new program.

EXAMPLE

mcl prog -BUILDmce-

Chklib*

Link in the host compiled startup libraries

SYNOPSIS

CHKLIB±

DESCRIPTION

Switching this option on causes the host compiled startup library, `hclib.klc` to be linked into the .klc file.

EXAMPLE

mcl prog -CHKLIB+

Config — Specify a configuration filename

SYNOPSIS

COnfig <tag>=<filename>

CFG <tag>=<filename>

DESCRIPTION

Several configuration files are used by the tools, containing strings and default options. The relevant tags are

- `datetime`
  The names of the months and format for outputting the date and time.
- `extensions`
  The extensions used for c, .kic, .klc files etc.
- `strings`
  The strings for all the output produced by the tool.
- `options`
  The default option settings.
- `locate`
  The location of the various special files e.g. libraries.

If the file cannot be opened or is not in the correct format an error message is displayed and processing stops.

EXAMPLE

mcl prog -COnfig strings=/home/usr/fred/misc/newstrings

Delete — Delete the given file from the input file

SYNOPSIS

Delete <filename>

DESCRIPTION

The input file is assumed to have had the named file linked into it at some time. The named file is deleted from the .klc file. This deletion involves removing all code and symbol table information associated with that file. Macro names unique to the named file are not deleted.

The deletion is performed by making a new version of the .klc file (minus the deleted file), deleting the old file and renaming the new version with the old name (unless a new output file is given).

EXAMPLE

mcl prog -Delete strutil

D**et**ail — Detailed rather than brief help

SYNOPSIS

DETail±

DESCRIPTION

Only the main options are displayed on the normal help display. The other options are used by the platform profiles, or are only applicable on certain hosts (e.g. DOS). This option causes these other options to be displayed, and is equivalent to `-helpmod D`.

EXAMPLE

mcl -help det

E**cho** — Echo given text to standard output

SYNOPSIS

ECHO <text>

DESCRIPTION

The rest of the line, starting at the first non-whitespace character is echoed to the screen. `-ECHO` on its own displays a newline. It may be used to display helpful information when options are being processed from a via file.

Note: This option cannot be used on the command line.

EXAMPLE

-ECHO Created on 29 Feb 1988

Errfile — Specify error file name

SYNOPSIS

ERRfile <filename>

DESCRIPTION

The named file is searched by the error reporting mechanism if any errors or warnings may have to be given. If more than one error file is specified, the files are searched in reverse order. Error messages may be changed by creating a new error file and giving the appropriate `-ERRfile` option. The new error file will take precedence over the rest and will cause the new messages to be reported.

If `<filename>` does not exist a warning is given and processing continues. If no error files could be opened, or no entry can be found for a particular error, only the error number will be reported. Also, since the severity of an error cannot be determined they are treated as warnings.

EXAMPLE

mcl prog -ERR new.err

Exe* — Create an executable

SYNOPSIS

Exe±

DESCRIPTION

An executable differs from a .klc file in that it contains a function called main and does not contain any uncalled functions (any functions that are not explicitly called or assigned to a pointer to function). The generated file has the suffix .kec rather than .klc.

When using this option one of the .kic or .klc files must contain a function called main. The purpose of this option is to remove unused functions from the executable. It can also be used in conjunction with the `-HIER` and `-HC` options to produce a hierarchy diagram.

EXAMPLE

mcl prog -Exe+

F**old** Fold filenames before comparing

SYNOPSIS

FOLD±

DESCRIPTION

The names of the linked files are stored in the .klc file. Some operating systems support mixed case alphabetic characters in filenames, while others do not. This option causes all characters in filenames to be forced to upper case before searching or comparing to other filenames. This option is only relevant when using the `-DElete` or `-REplace` options.

EXAMPLE

mcl prog -FOLD+

Forgetall Forget all arguments of option given so far

SYNOPSIS

Forgetall <option>

DESCRIPTION

Some options do not have single values, they accumulate a list of values. This option enables this list to be forgotten. It must be applied to an option that takes lists of values or a single string. In particular, when applied to the following options it has the specified effects:

`LOGfile` Cancel request for logfile.

`NOmsg` Reinstate any previously suppressed error numbers.

`Output` Use default output filename.

`PAth` Cancel previous prefix.

The main use of this option is in overriding any options given in the configure file.

mcl prog -f hostinclude

F**u**lltype — Perform full cross translation unit checking

SYNOPSIS

FUlltype±

DESCRIPTION

Full type checking across all objects and functions declared and defined in all files can be a slow process. **mcl** provides the option of 'quick' checking. Quick checking is performed using a 32 bit checksum, rather than comparing information in the symbol table.

Switching this option on causes the type checking to be as per the standard.

Note: Quick checking is not always correct. Sometimes compatible types are flagged as being incompatible and incompatible types are not flagged.

EXAMPLE

mcl prog -FU-

G**l**ue* — Combine the new executor with the .klc file

SYNOPSIS

GLue±

DESCRIPTION

This option provides a simple method of making sure the .klc file and executor match each other. The program can be run by typing the name of the program which is created.

See the chapter on host-compiled units in the dynamic reference manual for more details.

EXAMPLE

mcl prog -GLUE+

Graphics

Use graphics in the hierarchy report

Some terminals and printers support graphics characters that enable more readable diagrams to be produced.

The configuration file supports two sets of characters for drawing the hierarchy diagrams. This option is used to choose which of these sets is used. By convention the first set in the configuration files are the graphics characters and the second are the standard ASCII.

EXAMPLE

mcl prog -Graphics+

Hcontrol*

Specify the hierarchy control file

SYNOPSIS

HControl <filename>

DESCRIPTION

If the file is not found or does not have the expected layout a message is then given. A full hierarchy is then produced. This option only has any effect when the `-HIerarchy` option is switched on. Full details of the control file formats are given in chapter 2.

EXAMPLE

mcl prog -HC hier.ctl

Hierarchy

Create a hierarchy file

SYNOPSIS

HIerarchy±

DESCRIPTION

Switching this option on causes a file containing a hierarchy diagram to be generated.

Note: This option can currently only be used in conjunction with linking to produce an executable (`-Exe` option).

EXAMPLE

mcl prog -HI+

He

Entries in external hash table

SYNOPSIS

HE <number>

DESCRIPTION

This option controls the size of an internal **mcl** table. Reducing it will save space at the expense of performance. While increasing it may speed up **mcl** at the expense of using more memory. Unless storage space us very tight, or there are ten thousand plus externals, it is best left alone.

If this number is changes it is recommended that a prime number be used.

EXAMPLE

mcl prog -HE 211

Hm

Entries in the macro hash table

SYNOPSIS

HM <number>

DESCRIPTION

This value follows the same idea as the one above except that it refers to macros.

Decreasing the default value can have a very large effect on performance. Like `-HE` this option should only be changed to a prime number.

EXAMPLE

mcl prog -HM 23

H**elpm**od Set modifiers for displayed help

SYNOPSIS

HELPMod <letter>

DESCRIPTION

The output of each line of the help text is controlled by a set of modifiers. Each line can be displayed either always or whenever one of the modifiers associated with it is given by a `-helpmod` or `-DETail` option. The following modifiers are currently used:

`D` Detailed help (display everything)

`H` Host compiled options

`M` Memory manager options (only relevant to MS-DOS Platforms)

EXAMPLE

mcl prog -helpmod HDM -help

H**ost**include* Link the object or library file into a new executor

SYNOPSIS

HOSTinclude <filename>

DESCRIPTION

When using host compiled libraries (or miscellaneous units), this option allows the name of a library, or object file, to be specified. This library will then be linked into the new executor.

EXAMPLE

mcl prog -HOSTinclude /usr/lib/xlib.a

Inputpath — Specify prefix path for input filenames

SYNOPSIS

INPUTPAth <path>

DESCRIPTION

The parameter to this option is used to change the default path for opening files. All filenames following this option will be prefixed by the specified path.

EXAMPLE

mcl prog1 prog2 -INPUTPA osdir

Keeptemp* — Keep temporary files

SYNOPSIS

Keeptemp±

DESCRIPTION

While creating a new executor several temporary files are produced. This option stops these temporary files from being removed so they can then be examined. This option is thus a useful for debugging on new systems.

Note: the temporary files should be removed as soon as possible to prevent the temporary partition filling up. (Most systems place them in `/usr/tmp`.)

EXAMPLE

mcl prog -Keep+

Lib — Link in the standard libraries

SYNOPSIS

Lib±

DESCRIPTION

It can be tiresome always having to specify the full pathname of the standard library. This option can be used to cause the .klc file (presumably a linked form of the standard library) named in the configure file to be always linked into the users program.

The library shipped with **OSPC** only provides the functionality required by the C standard. If anything extra is required the system (host compiled) libraries must be used instead. For more details on this subject see the Dynamic Reference Guide.

EXAMPLE

mcl prog -L+

L**og**file — Create a log file

SYNOPSIS

LOGfile <filename>

DESCRIPTION

The named file is opened and all characters sent to standard output are sent to it (standard output still receives the characters sent to it).

EXAMPLE

mcl prog -LOG hist.log

M**a**cro — Macro checking between files

SYNOPSIS

MAcro±

DESCRIPTION

Switching this option off stops macro definitions from being copied to the output file. If the option is on and there are multiple definitions of macros with the same name then only one of the definitions is copied. Also see `-Body` option.

EXAMPLE

mcl prog -MA-

Mapfunc* Change the mapping for a function

SYNOPSIS

MAPFunc <func-name>[:<check-prefix>:[<call-prefix>]]

DESCRIPTION

This option allows a call to a host compiled function to be remapped, allowing interface checking to be added, or the routines to be replaced. `<check-prefix>` is prefixed onto the name of the function being mapped to form the name of a new name. A function with this name is called before the original function. If the field is omitted then no function will be called before hand.

If the `<call-prefix>` is specified then the function, whose name is given by prefixing the <call-prefix> onto the original function name, is called instead of the original function. If omitted the original function is called.

See the chapter on hostcompiled units for more details.

EXAMPLE

mcl prog -MAPFunc printf::MCE_

Mapunit* Re-map the functions in a unit

SYNOPSIS

MAPunit <header>[:<new-name>[:<check-prefix>:[<call-prefix>]]]

DESCRIPTION

This has a similar effect to `-MAPFunc` except it works on a unit, rather than a function basis.

`<new-name>` gives the name of an object file containing the interface checks or replacements for the functions in this unit. The file is linked into the new executor when it is built. The option also implies a `-MAPFunc` option for each of the functions defined in the unit, although a real `-MAP-Func` option always takes precedence.

Any unit that does not have a mapunit command associated with it, is assumed to have no interface checking, and have a file <basename>.o associated with it.

EXAMPLE

-rem Contents of a re-mapping file.
-MAPUnit stdio:::MCE_
-REM stdio does not have an associated object file, but
-REM direct all calls to functions it contains via functions
-REM with the prefix MCE_.

mcl prog -MAPUnit stdarg
-REM No object file associated with <stdarg.h>
-rem and don't remap the functions in int.

Mcerts*

Set path of the mce run time system

SYNOPSIS

MCErts <path>

DESCRIPTION

Add a directory to the path the linker searches for the executor library.

EXAMPLE

mcl prog -MCErts /home/fred/lib/mce.a

Min

Use the least memory possible

SYNOPSIS

Min <number>

DESCRIPTION

On some systems where available memory is very tight it may be necessary to use this option when checking large numbers of files. Generally it should be left off unless absolutely necessary since **mcl**'s performance may be significantly degraded.

EXAMPLE

mcl prog -Min+

Mm Memory the memory manager can use

SYNOPSIS

MM <number>

DESCRIPTION

The available memory does not limit **mcl**. A memory management package is used to allow a hard disc to be used as temporary storage. The memory manager needs to know in advance the maximum amount of real memory that might be used (to perform its job). The value of this variable is proportional to maximum memory usage.

mcl uses the memory manager to store the most voluminous data items. Smaller, infrequently used data items have memory permanently allocated to them Since the memory manager can swap data out to disc it can run with very little real memory. However, disc I/O performance is a bottleneck.

If you are linking large files and the disc is very active then increasing the amount of real memory available to the memory manager may improve performance. This is only applicable under MS-DOS.

EXAMPLE

mcl prog -MM 40000

Mn Number of nodes for the memory manager

SYNOPSIS

MN <number>

DESCRIPTION

If **mcl** gives the message ‘Out of node in memory manager’ then use this option to increase the memory manager disc space. See the `-MM` option for more details.

EXAMPLE

mcl prog -MN max

Nomsg — Suppress a specific error number

Rather than editing the error files to reduce the severity of an unwanted error, this option allows an error number to be disabled for the current invocation of **mcl**. The error file can be obtained by looking in the appropriate error file.

Note: Disabling fatal or internal errors serves no useful purpose. The result of continuing after such an error is unpredictable.

EXAMPLE

mcl prog -N99 -N 211

Outbuf — size of output buffer

SYNOPSIS

OUTBuf <number>

OB <number>

DESCRIPTION

On some systems increasing the size of the output buffer used by **mcl** will speed up disc I/O. Generally the best performance will be obtained if the output buffer is a multiple of the block size used on the disks.

EXAMPLE

mcl prog -OB 10240

Objpath* — Path to find object files along

SYNOPSIS

OBJpath <filename>

DESCRIPTION

This path is used by **mcl** to locate the object files specified in the `-HOSTInclude` and `-MAPUnit` options.

EXAMPLE

mcl prog -objpath /lib -objpath/usr/lib

Output — Send output to the given file

SYNOPSIS

Output <filename>

DESCRIPTION

The named file is used as the output destination rather than the name derived from the first input file encountered on the command line.

EXAMPLE

mcl prog -O xyz.klc

Op — Specify output path for all output files

SYNOPSIS

OP <path>

OUTPUTPath <path>

DESCRIPTION

This option provides a method of specifying a directory to which the checked file should be sent.

EXAMPLE

mcl prog -OB 10240

Ospcdir — Specify directory containing interface stubs

SYNOPSIS

OSPCdir <directory>

DESCRIPTION

This option specifies a directory that function interface stubs can be placed (along with their respective .kic files). This option is used to speed up producing a new executor. It also has the advantage of hiding all the .kic files in a different directory to the .kic's. When this option is used **mcl** only rebuilds the interface stub if the .kic file is older than the '.o' stub, or the stub doesn't exist.

The normal convention is to place the line:

-OSPCDir OSPC

in both the .mccrc and the .mclrc files. A directory called `OSPC` should also be created in each of the source directories. The name `OSPC` is assumed by the `ccc` front end.

EXAMPLE

mcl prog -OSPCDir OSPC

Quiet — Stop displaying messages on standard output

SYNOPSIS

Quiet±

DESCRIPTION

Switching this option on causes output to standard output to stop. Output may be resumed by switching this option off.

EXAMPLE

mcl prog -Q+

References — Display references to the standard

SYNOPSIS

REFerences±

DESCRIPTION

Most of the warning and error messages given by **mcl** arise as a result of wording in the C standard. The priority in wording these messages was ease

of comprehension rather than using strict standards terminology. This option causes the appropriate standard reference to be given with these messages.

EXAMPLE

mcl prog -REF+

Remark — Treat delimited text as a comment

SYNOPSIS

REMark <text>

DESCRIPTION

The option follows the same rules as the `-ECHO` option with the difference that the string is not displayed. It is treated as a comment.

EXAMPLE

-REM This is a comment

Replace — Replace the given file in the linked file

SYNOPSIS

Replace <filename>

DESCRIPTION

This is one of the librarian facilities provided by **mcl**. The `-Replace` option allows individual .kic files within a .klc to be replaced (the other files still remain).

NOTE: This option cannot be used in conjunction with adding other files to the .klc (including linking in a library with `-LIB` or `-CHKLIB`). The two operations should be done one after another.

EXAMPLE

mcc mylib.klc -r strutil

Search — Search path for .klc files

SYNOPSIS

Search <filename>

DESCRIPTION

Adds the path to the search path in the .klc file (for use by the executor). This currently has no effect on the executor's behaviour.

Suppresslvl — Suppress messages below given level

SYNOPSIS

SUppresslvl <number>

DESCRIPTION

The error message file associates one or more numeric levels with each error number it contains. The number given in this option acts as a cutoff. Messages with levels below this value do not appear in the output.

Thus if the highest level specified for a given message level is 5 and the cutoff is 6 no messages will ever appear for that error number. If messages at levels 5, 6 and 7 are available for a given message and the cutoff is level 6 then the level 6 message acts as the minimum level available.

EXAMPLE

mcl prog -SU 4

Tracecfg — Trace Configuration being read in

SYNOPSIS

TRACECfg±

TRC±

DESCRIPTION

Switching this option on causes a trace of the configuration file to be given, as it is being read in. The configuration file must be in a given format, and without any feedback, errors in the layout of this file can be difficult to track down. Information from this trace can be used to locate possible problems in the layout.

EXAMPLE

mcl prog -TRC+

Userlib — Add path to userlib in .klc file

SYNOPSIS

Userlib <filename>

DESCRIPTION

Specify the name of the userlib for the executor to read when dynamically linking.

Verbose — Talkative mcl

SYNOPSIS

Verbose±

DESCRIPTION

Switching this option on cases **mcl** to display the name of every translation unit read in from a .kic or .klc file. If this name is a member of a library file it will be enclosed in *()*'s. When on, together with full type checking, this option causes the full type of an object, or function to be displayed when incompatible types are found.

EXAMPLE

mcl prog -V-

Via — Take options from the given file

SYNOPSIS

VIA <filename>

DESCRIPTION

Via files are text files, created by the user, that contain frequently used command lines. Options in a via file are given one per line. A via file may contain a reference to another via file, provided that it is the last entry in the via file.

EXAMPLE

mcl prog -VIA lib.lnk

X**casef**old Case fold external names

SYNOPSIS

XCASEFold±

DESCRIPTION

Switching this option on causes external identifiers which only differ in their case to be flagged. The identifiers will be treated as referring to the same object and the externals output to the .klc file will all have their case folded.

The standard specifies that the case of external identifiers need not be considered significant. This option is primarily available to provide backwards compatibility with older linkers. The `-XCASESIG` option, described below, will generally be more useful.

X**cases**ig Number of significant characters in external names

SYNOPSIS

XCASESig <number>

DESCRIPTION

Switching this option on causes a warning to be reported if two identifiers are the same except for their case, within their significant length. The identifiers are still treated as separate symbols.

X**n**amelength External name significance

This option may be used to specify the number of significant characters in an external identifier. A warning will be given if identifiers aren't significant after the given number of characters.

The C standard specifies that only the first six characters in an external identifier need be considered significant.

X**namet**runc External name significance

SYNOPSIS

XNAMETrunc±

DESCRIPTION

This option may be used to specify the number of significant characters in an external identifier. Identifiers not significant after the given number of characters will be treated as referring to the same object. As in the `-XNAME-FOLD` option, the use of this option is primarily for backward compatibility

The C standard specifies that only the first six characters in an external identifier need be considered significant.

The minimum number of characters any component .kic of .klc files have been truncated to, is stored in the .klc file. On a subsequent link, this is taken as the maximum number of significant characters for the link. A smaller value may be given via the `-XNAMELength` option. Thus when performing cross unit checks on a file previously truncated to six characters, with the option `-XN 8`, **mcl** will flag any clashes to within six characters.

EXAMPLE

mcl prog -NT 10

X**tr**act Set record to delete from an executable

SYNOPSIS

XTRact <letter>

DESCRIPTION

The .kic and .klc files contain a large amount of information (only comments from the original source are not present). It is possible to reduce the size of .klc files by deleting unneeded information.

What is needed is open to debate. For instance deleting macros will prevent any future links checking against them.

See Chapter 10 for a fuller description of these records and the use to which that are put.

`A` All record

`D` Type **D**efinitions (local and global)

`F` **F**unction dictionary

`G` Ta**G**s (local and global)

`J` Internal function declarations and definitions.

`L` **L**ine numbers

`M` **M**acros

`O` Local **O**bject definitions

`R` Global **R**eference lists

`T` **T**ype records

`X` E**X**ternal record

8.3 The Hierarchy diagram

When linking to form an executable, **mcl** also provides the option of producing a call graph (the hierarchy diagram). Full details are given in the `-HEIR` and `-HC` option information, together with the details of the control file format in chapter 2 of this guide.

Chapter 9

The Internet

9.1 Introduction

The list of standards related information available on the Internet continues to grow rapidly. The information given here can only act as a snapshot of what is available.

9.2 Web pages

The doc directory, on the distribution tape contains various web pages that have been downloaded. See directory for details.

http://www.knosof.co.uk	The Knowledge Software home page.
http://www.knosof.co.uk/posix.html	A collection of useful POSIX URL's.
http://www.xopen.org	The X/Open home page.
http://www.iso.ch	The ISO home page.
http://www.itl.nist.gov	The US National Institute of Standards and Technology.
http://www.ecma.ch	The European Computer Manufacturers Association home page.
http://www.dkuug.dk/JTC1/SC22/WG14/	ISO C working group home page.
http://www.dkuug.dk/JTC1/SC22/WG15/	ISO POSIX working group home page.
http://www.jcc.com/sql_stnd.html	A SQL standards home page by a committee member.
http://stdsbbs.ieee.org/	The IEEE home page
http://www.maths.warwick.ac.uk/c++	A C++ standards home page
http://www.qucis.queensu.ca/home/dalamb/info.html	The comp.software-eng FAQ.

9.3 Newsgroups

Newsgroups offer a public forum for debate and asking technical questions. Some groups seem to attract questioners and not answers. The following groups have been found to be worth keeping an eye on.

comp.databases	Mostly people asking simple questions.

comp.risks	A moderated group, reporting on its readers experiences of computer rated problems, disasters and inconveniences in their lives.
comp.doc.techreports	A moderated group that is posted to sporadically. A good source for references to technical papers.
comp.software-eng	General discussion on software engineering issues. Usually contains several interesting posts per week.
comp.software.measurement	This is a new newsgroup that has yet to attract many contributors.
comp.software.testing	General discussion on software testing. Usually contains several interesting posts per week.

9.4 Other sources

There's an on-line paper on Hungarian notation at:

gopher://wiretap.spies.com:70/11/Library/Techdoc/Language

Chapter 10

Summary of .kic and .klc contents

10.1 Introduction

The .kic files are generated as output from **mcc** and used as input to **mcl**. The main difference between a .kic and .klc file is that the latter has been reorganised by **mcl**. This reorganisation is necessary to speed up the loading of the users program by **mce**. Of course running more than one .kic file through **mcl** also allows type checking across translation units to be performed.

The .kic and .klc files consist of a series of records. The contents of some of these records are described below.

10.2 Audit trails

When a C file is processed by **mcc** the date, time and command line options are stored in the kic file. Similarly when a .kic file or .klc file is processed by **mcl** the date, time and command line options are stored. This information is known as an audit trail.

When two or more files are processed by **mcl** their combined audit trails are written to the output file in addition to the information on the current link.

This audit information may be displayed using the `-ATV` (view audit trail) option. The information may be deleted by using the `-ATP` (purge audit trail) option.

10.3 Externals

Symbol table information on all the externals visible in the .klc file are kept together. The external record for each object and function contains a list of files in which they were declared and the file in which they were defined, if any.

10.4 Header

This contains various pieces of miscellaneous information such as, number of significant characters, search paths and limits that is specific to each translation unit.

10.4.1 File information

The internal information for each translation unit is grouped together. Klc files containing more than one file have a list of these file records.

The internal information for a translation unit consists of:

- File information header. Information specific to that translation unit. Includes the filename, compile time options, time and date.
- Startup information. Various items used by the executor when loading this file.
- Function block header. This header is followed by information for each function within the file. This includes compiled code, literals, line numbers and symbol table information.
- Static information, ie global initialisation.

10.4.2 Line numbers

This record contains the line numbers for each line of code in the original source and the associated offset of the generated code (from the start of the function).

10.4.3 Literal area

This record contains the literals for each function. The literal area has a separate section reserved for floating point numbers.

10.4.4 Typeinfo

This record contains the type information associated with every object or function used in the original source file.

Chapter 11

Syntax of the C language

escape-sequence:
simple-escape-sequence
octal-escape-sequence
hexadecimal-escape-sequence

simple-escape-sequence:
one of
**\’ \" \? **
\a \b \f \n \r \t \v

octal-escape-sequence:
\ octal-digit
\ octal-digit octal-digit
\ octal-digit octal-digit octal-digit

hexadecimal-escape-sequence:
\x hexadecimal-digit
hexadecimal-escape-sequence hexadecimal-digit

token:
keyword
identifier
constant
string-literal
operator
punctuator

identifier:
non-digit
identifier non-digit
identifier digit

non-digit: one of
_ a b c d e f g h i j k l m
n o p q r s t u v w x y z
A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z
$ (in non-strict C only)

digit: one of
0 1 2 3 4 5 6 7 8 9

constant:
floating-constant
integer-constant
enumeration-constant
character-constant

floating-constant:
fractional-constant exponent-part$_{opt}$ **floating-suffix**$_{opt}$
digit-sequence exponent-part floating-suffix$_{opt}$

fractional-constant:
digit-sequence$_{opt}$ **. digit-sequence**
digit-sequence .

exponent-part:
e sign$_{opt}$ **digit-sequence**
E sign$_{opt}$ **digit-sequence**

sign:
+
-

digit-sequence:
digit
digit-sequence digit

floating-suffix:
f
l
F
L

integer-constant:
decimal-constant integer-suffix$_{opt}$
octal-constant integer-suffix$_{opt}$
hexadecimal-constant integer-suffix$_{opt}$

decimal-constant:
non-zero-digit
decimal-constant digit

octal-constant:
0
octal-constant octal-digit

hexadecimal-constant:
0x hexadecimal-digit
0X hexadecimal-digit
hexadecimal-constant hexadecimal-digit

non-zero-digit: one of
1 2 3 4 5 6 7 8 9

octal-digit: one of
0 1 2 3 4 5 6 7

hexadecimal-digit: one of
0 1 2 3 4 5 6 7 8 9
a b c d e f
A B C D E F

integer-suffix:
unsigned-suffix long-suffix$_{opt}$
long-suffix unsigned-suffix$_{opt}$

unsigned-suffix:
u
U

long-suffix:
l
L

enumeration-constant:
identifier

character-constant:
‘c-char-sequence’
L ‘c-char-sequence’

c-char-sequence:
c-char
c-char-sequence c-char

c-char:
any character in the source character set except the single-quote ‘, backslash \, or new-line character
escape-sequence

string-literal:
“s-char-sequence$_{opt}$**”**
L “s-char-sequence$_{opt}$**”**

s-char-sequence:
s-char
s-char-sequence s-char

s-char:
any character in the source character set except The double-quote “, backslash \, or new-line
escape-sequence

operator: one of
[ ] ( ) . - + - ~ ! / % ^ |
? : = , # sizeof
++ — & *
< > = == != && ||
***= /= %= += -= <= >= &= ^= |=**
##

punctuator: one of
[ ] ( ) { } * , : = ; ... #

preprocessing-file:
group$_{opt}$

group:
group-part
group group-part

group-part:
pp-tokens$_{opt}$ **new-line**
if-section
control-line

if-section:
if-group elif-groups$_{opt}$ **else-group**$_{opt}$ **endif-line**

if-group:
 # if constant-ex new-line group$_{opt}$
 # ifdef identifier new-line group$_{opt}$
 # ifndef identifier new-line group$_{opt}$

elif-groups:
elif-group
elif-groups elif-group

elif-group:
 # elif constant-ex new-line group$_{opt}$

else-group:
 # else new-line group$_{opt}$

endif-line:
 # endif new-line

control-line:
 # include pp-tokens new-line
 # define identifier replacement-list new-line
 # define ident lparen ident-list$_{opt}$ **) replace-list new-line**
 # undef identifier new-line
 # line pp-tokens new-line
 # error pp-tokens$_{opt}$ **new-line**
 # pragma pp-tokens$_{opt}$ **new-line**
 # new-line

lparen:
the left-parenthesis character without preceding white-space

replacement-list:
pp-tokens$_{opt}$

pp-tokens:
preprocessing-token
pp-tokens preprocessing-token

preprocessing-token:
header-name (only within a #include directive)
identifier (no keyword distinction)
pp-number
character-constant
string-literal

operator
punctuator
each non-white-space character that cannot be one of the above

header-name:
h-char-sequence
“q-char-sequence”

h-char-sequence:
h-char
h-char-sequence h-char

h-char:
any character in the source character set except
the new-line character and >

q-char-sequence:
q-char
q-char-sequence q-char

q-char:
any character in the source character set except
the new-line character and “

new-line:
the new-line character

pp-number:
digit
. digit
pp-number digit
pp-number nondigit
pp-number e sign
pp-number E sign
pp-number .

declaration:
declaration-specifiers init-declarator-list$_{opt}$**;**

declaration-specifiers:
storage-class-specifier declaration-specifiers$_{opt}$
type-specifier declaration-specifiers$_{opt}$
type-qualifier declaration-specifiers$_{opt}$

init-declarator-list:
init-declarator
init-declarator-list , init-declarator

init-declarator:
declarator
declarator = initializer

storage-class-specifier:
typedef

extern
static
auto
register

type-specifier:
void
char
short
int
long
float
double
signed
unsigned
struct-or-union-specifier
enum-specifier
typedef-name

struct-or-union-specifier:
struct-or-union identifier$_{opt}$ **{ struct-declaration-list }**
struct-or-union identifier

struct-or-union:
struct
union

struct-declaration-list:
struct-declaration
struct-declaration-list struct-declaration

struct-declaration:
specifier-qualifier-list struct-declarator-list ;

specifier-qualifier-list:
type-specifier
type-qualifier specifier-qualifier-list

struct-declarator-list:
struct-declarator
struct-declarator-list , struct-declarator

struct-declarator:
declarator
declarator$_{opt}$**: constant-expression**

enum-specifier:
enum identifier$_{opt}$ **{ enumeration-list }**
enum identifier

enumeration-list:
enumeration
enumeration-list , enumeration

enumeration:
enumeration-constant

enumeration-constant = constant-expression

declarator:
pointer$_{opt}$ **direct-declarator**

direct-declarator:
identifier
(declarator)
direct-declarator [constant-expression$_{opt}$ **]**
direct-declarator (parameter-type-list)
direct-declarator (identifier-list$_{opt}$ **)**

pointer:
*** type-qualifier-list**$_{opt}$
*** type-qualifier-list**$_{opt}$ **pointer**

type-qualifier-list:
type-qualifier
type-qualifier-list type-qualifier

parameter-type-list:
parameter-list
parameter-list , ...

parameter-list:
parameter-declaration
parameter-list , parameter-declaration

parameter-declaration:
declaration-specifiers declarator
declaration-specifiers abstract-declarator$_{opt}$

identifier-list:
identifier
identifier-list , identifier

type-name:
type-specifier-list abstract-declarator$_{opt}$

abstract-declarator:
pointer
pointer$_{opt}$ **direct-abstract-declarator**

direct-abstract-declarator:
(abstract-declarator)
direct-abstract-declarator$_{opt}$ **[constant-expression**$_{opt}$ **]**
direct-abstract-declarator$_{opt}$ **(parameter-type-list**$_{opt}$ **)**

typedef-name:
identifier

initializer:
assignment-expression
{ initializer-list }
{ initializer-list , }

initializer-list:
    initializer
    initializer-list , initializer

translation-unit:
    external-definition
    translation-unit external-definition

external-definition:
    function-definition
    declaration

primary:
    identifier
    constant
    string-literal
    (expression)

postfix-ex:
    primary-ex
    postfix-ex [expression]
    postfix-ex (argument-expression-list$_{opt}$)
    postfix-ex . identifier
    postfix-ex - identifier
    postfix-ex ++
    postfix-ex —

argument-expression-list:
    assignment-ex
    argument-expression-list , assignment-ex

unary-ex:
    postfix-ex
    ++ unary-ex
    — unary-ex
    unary-operator cast-ex
    sizeof unary-ex
    sizeof (type-name)

unary-operator: one of
    & * + - ~ !

cast-ex:
    unary-ex
    (type-name) cast-ex

multiplicative-ex:
    cast-ex
    multiplicative-ex * cast-ex
    multiplicative-ex / cast-ex
    multiplicative-ex % cast-ex

shift-ex:
    additive-ex
    shift-ex < additive-ex
    shift-ex > additive-ex

relational-ex:
    shift-ex
    relational-ex < shift-ex
    relational-ex > shift-ex
    relational-ex <= shift-ex
    relational-ex >= shift-ex

equality-ex:
    relational-ex
    equality-ex == relational-ex
    equality-ex != relational-ex

AND-ex:
    equality-ex
    AND-ex & equality-ex

exclusive-OR-ex:
    AND-ex
    exclusive-OR-ex ^ AND-ex

inclusive-OR-ex:
    exclusive-OR-ex
    inclusive-OR-ex | exclusive-OR-ex

logical-AND-ex:
    inclusive-OR-ex
    logical-AND-ex && inclusive-OR-ex

logical-OR-ex:
    logical-AND-ex
    logical-OR-ex || logical-AND-ex

conditional-ex:
    logical-OR-ex
    logical-OR-ex ? ex : conditional-ex

assignment-ex:
    conditional-ex
    unary-ex assignment-operator assignment-ex

assignment-operator: one of
    = *= /= %= += -= <= >= &= ^= |=

expression:
    assignment-ex
    expression , assignment-ex

constant-expression:
    conditional-expression

statement:
    labelled-statement
    compound-statement
    expression-statement
    jump-statement
    selection-statement

iteration-statement

labelled-statement:
identifier : statement
case constant-ex : statement
default : statement

compound-statement:
{ declaration-list$_{opt}$ **statement-list**$_{opt}$**}**

declaration-list:
declaration
declaration-list declaration

statement-list:
statement
statement-list statement

expression-statement:
expression$_{opt}$**;**

jump-statement:
goto identifier ;
continue ;
break ;
return expression$_{opt}$**;**

selection-statement:
if (expression) statement
if (expression) statement else statement
switch (expression) statement

iteration-statement:
while (expression) statement
do statement while (expression) ;
for (expression$_{opt}$**; expression**$_{opt}$**; expression**$_{opt}$**) statement**

11.1 Precedence of operators

primary expressions

16	literals names	simple tokens
16	a[i]	subscripting
16	f()	function call
16	.	direct selection
16	->	indirect selection

unary expressions

15	++ —	postfix increment/decrement
14	++ —	prefix increment/decrement
14	sizeof	size
14	(type-name)	cast
14	~	bitwise not
14	!	logical not
14	-	arithmetic negation
14	&	adress of
14	*	contents of

binary operators

13L	* / %	multiplicative
12L	+ -	additive
11L	<< >>	shift
10L	=	inequality
9L	= = !=	equality
8L	&	bitwise and
7L	^	bitwise xor
6L	||	bitwise or
5L	&&	logical and
4L	|	logical or
3R	?:	conditional
2R	= += -= *= /= %= <<= >>= &= ^= |=	assignment
1L	,	comma

L, indicates left associative operators; R, right associative operators.

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X