

M16C v2.1

C CROSS-COMPILER USER'S GUIDE

A publication of
TASKING
Documentation Department
Copyright © 2001 TASKING, Inc.

All rights reserved. Reproduction in whole or part is prohibited
without the written consent of the copyright owner.

The following trademarks are acknowledged:

FLEXlm is a registered trademark of Globetrotter Software, Inc.
HP and HP-UX are trademarks of Hewlett-Packard Co.
Intel is a trademark of Intel Corporation.
Motorola is a registered trademark of Motorola, Inc.
MS-DOS and Windows are registered trademarks of Microsoft Corporation.
SUN is a trademark of Sun Microsystems, Inc.
UNIX is a registered trademark of X/Open Company, Ltd.

All other trademarks are property of their respective owners.

Data subject to alteration without notice.

E-mail: support@tasking.com
WWW: <http://www.tasking.com>

The information in this document has been carefully reviewed and is believed to be accurate and reliable. However, TASKING assumes no liabilities for inaccuracies in this document. Furthermore, the delivery of this information does not convey to the recipient any license to use or copy the software or documentation, except as provided in an executed license agreement covering the software and documentation.

TASKING reserves the right to change specifications embodied in this document without prior notice.

CONTENTS

TABLE OF CONTENTS



TASKING



CONTENTS

SOFTWARE INSTALLATION **1-1**

1.1	Introduction	1-3
1.2	Installation for Windows	1-3
1.3	Installation for Linux	1-4
1.3.1	RPM Installation	1-4
1.3.2	Tar.gz Installation	1-5
1.3.3	Setting the Environment	1-6
1.4	Installation for UNIX Hosts	1-7
1.4.1	Setting the Environment	1-9
1.5	Licensing TASKING Products	1-9
1.5.1	Obtaining License Information	1-10
1.5.2	Installing Node-Locked Licenses	1-10
1.5.3	Installing Floating Licenses	1-11
1.5.4	Starting the License Daemon	1-13
1.5.5	Setting Up the License Daemon to Run Automatically	1-14
1.5.6	Modifying the License File Location	1-16
1.5.7	How to Determine the Hostid	1-17
1.5.8	How to Determine the Hostname	1-18

OVERVIEW **2-1**

2.1	Introduction to M16C C Cross-Compiler	2-3
2.2	Product Definition	2-5
2.3	General Implementation	2-6
2.3.1	Compiler Phases	2-6
2.3.2	Frontend Optimizations	2-7
2.3.3	Backend Optimizations	2-10
2.4	Compiler Structure	2-11
2.5	Environment Variables	2-14
2.6	Sample Session	2-15
2.6.1	Using EDE	2-15
2.6.2	Using the Control Program	2-23
2.6.3	Using the Makefile	2-25



LANGUAGE IMPLEMENTATION		3-1
3.1	Introduction	3-3
3.2	Accessing Memory	3-4
3.2.1	Storage Types	3-5
3.2.1.1	Non-Volatile RAM	3-6
3.2.1.2	Storage and Section Relations	3-7
3.2.2	Memory Models	3-8
3.2.3	The _at() Attribute	3-9
3.3	Data Types	3-11
3.3.1	ANSI C Type Conversions	3-12
3.3.2	Character Arithmetic	3-15
3.3.3	The _bit Type	3-16
3.3.4	Special Function Registers	3-17
3.4	Function Parameters	3-18
3.5	Parameter Passing and Function Return	3-19
3.6	Automatic Variables	3-21
3.7	Initialized Variables	3-21
3.8	Type Qualifier volatile	3-21
3.9	Strings	3-22
3.10	Pointers	3-22
3.11	Inline C Functions	3-23
3.12	Inline Assembly	3-24
3.13	Calling Assembly Functions	3-25
3.14	Intrinsic Functions	3-27
3.15	Interrupts	3-31
3.16	Safer C	3-32
3.17	Structure Tags	3-33
3.18	Typedef	3-33
3.19	Switch Statement	3-34
3.20	Portable C Code	3-35
3.21	How to Program Smart	3-35
3.22	Some Examples of Complex Declarators	3-36

COMPILER USE **4-1**

4.1	Control Program	4-3
4.1.1	Detailed Description of the Control Program Options .	4-5
4.1.2	Environment Variables	4-9
4.2	Compiler	4-10
4.2.1	Detailed Description of the Compiler Options	4-14
4.3	Include Files	4-73
4.4	Pragmas	4-75
4.5	Alias	4-78
4.6	Compiler Limits	4-80

COMPILER DIAGNOSTICS **5-1**

5.1	Introduction	5-3
5.2	Return Values	5-4
5.3	Errors and Warnings	5-5

LIBRARIES **6-1**

6.1	Introduction	6-3
6.2	Header Files	6-3
6.3	C Libraries	6-4
6.3.1	Single Precision Floating Point	6-6
6.3.2	C Library Implementation Details	6-6
6.3.3	C Library Interface Description	6-12
6.3.4	Printf and Scanf Formatting Routines	6-62
6.4	Run-time Library	6-63

RUN-TIME ENVIRONMENT **7-1**

7.1	Startup Code	7-3
7.2	Register Usage	7-3
7.3	Section Usage	7-4
7.4	Stack	7-6
7.5	Heap	7-8

7.6 Floating Point Arithmetic 7-9

7.6.1 Special Floating Point Values 7-9

7.7 Interrupt Functions 7-10

7.8 Assembly Language Interfacing 7-12

FLEXIBLE LICENSE MANAGER (FLEXlm) A-1

1 Introduction A-3

2 License Administration A-3

2.1 Overview A-3

2.2 Providing For Uninterrupted FLEXlm Operation A-5

2.3 Daemon Options File A-7

3 License Administration Tools A-8

3.1 lmcksum A-10

3.2 lmdiag (Windows only) A-11

3.3 lmdown A-12

3.4 lmgrd A-13

3.5 lmhostid A-15

3.6 lmremove A-16

3.7 lmreread A-17

3.8 lmstat A-18

3.9 lmswitchr (Windows only) A-20

3.10 lmver A-21

3.11 License Administration Tools for Windows A-22

3.11.1 LMTOOLS for Windows A-22

3.11.2 FLEXlm License Manager for Windows A-23

4 The Daemon Log File A-25

4.1 Informational Messages A-26

4.2 Configuration Problem Messages A-29

4.3 Daemon Software Error Messages A-31

5 FLEXlm License Errors A-33

6 Frequently Asked Questions (FAQs) A-37

6.1 License File Questions A-37

6.2 FLEXlm Version A-37

6.3 Windows Questions A-38

6.4	TASKING Questions	A-39
6.5	Using FLEXlm for Floating Licenses	A-41

SAFER C

B-1

INDEX



CONTENTS

MANUAL PURPOSE AND STRUCTURE

PURPOSE

This manual is aimed at users of the TASKING M16C C Cross-Compiler. It assumes that you are familiar with the C language.

MANUAL STRUCTURE

Related Publications

Conventions Used In This Manual

1. Software Installation
Describes the installation of the C Cross-Compiler for the M16C.
2. Overview
Provides an overview of the TASKING M16C tool chain and gives you some familiarity with the different parts of it and their relationship. A sample session explains how to build an M16C application from your C file.
3. Language Implementation
Concentrates on the approach of the M16C architecture and describes the language implementation. The C language itself is not described in this document. We recommend: "The C Programming Language" (second edition) by B. Kernighan and D. Ritchie (1988, Prentice Hall).
4. Compiler Use
Deals with control program and C compiler invocation, command line options and pragmas.
5. Compiler Diagnostics
Describes the exit status and error/warning messages of the compilers.
6. Libraries
Contains the library functions supported by the compilers and describes their interface and 'header' files.
7. Run-time Environment
Describes the run-time environment for a C application. It deals with items like assembly language interfacing, C startup code and stack/heap size.

APPENDICES

A. Flexible License Manager (FLEXlm)

Contains a description of the Flexible License Manager.

B. Safer C

Supported and unsupported Safer C rules.

INDEX

RELATED PUBLICATIONS

- The C Programming Language (second edition) by B. Kernighan and D. Ritchie (1988, Prentice Hall)
- ANSI X3.159–1989 standard [ANSI]
- M16C C Cross-Assembler User's Guide [TASKING, MA012000]
- M16C CrossView Pro Debugger User's Guide [TASKING, MA012043]
- M16C/60 Series Software Manual, Mitsubishi 16-Bit Single-Chip Microcomputer M16C Family

CONVENTIONS USED IN THIS MANUAL

The notation used to describe the format of call lines is given below:

{ } Items shown inside curly braces enclose a list from which you must choose an item.

[] Items shown inside square brackets enclose items that are optional.

| The vertical bar separates items in a list. It can be read as OR.

italics Items shown in italic letters mean that you have to substitute the item. If italic items are inside square brackets, they are optional. For example:

filename

means: type the name of your file in place of the word *filename*.

... An ellipsis indicates that you can repeat the preceding item zero or more times.

screen font Represents input examples and screen output examples.

bold font Represents a command name, an option or a complete command line which you can enter.

For example

command [*option*]... *filename*

This line could be written in plain English as: execute the command *command* with the optional options *option* and with the file *filename*.

Illustrations

The following illustrations are used in this manual:



This is a note. It gives you extra information.



This is a warning. Read the information carefully.



This illustration indicates actions you can perform with the mouse.



This illustration indicates keyboard input.



This illustration can be read as “See also”. It contains a reference to another command, option or section.



MANUAL STRUCTURE

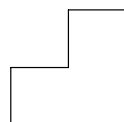
CHAPTER

1

SOFTWARE INSTALLATION



TASKING



1

CHAPTER

1.1 INTRODUCTION

This chapter describes how you can install the TASKING C Cross-Compiler for the M16C Family on Windows 95/98/NT/2000, Linux and several UNIX hosts.

1.2 INSTALLATION FOR WINDOWS

Step 1

Start Windows (95/98/NT/2000), if you have not already done so.

Step 2

Insert the CD-ROM into the CD-ROM drive.

If the TASKING Welcome dialog box appears, skip to Step 5. Otherwise, continue from Step 3.

Step 3

Select the Start button and select the Run . . . menu item.

Step 4

On the command line type:

d:\setup

(substitute the correct drive letter for your CD-ROM drive) and press the **<Return>** or **<Enter>** key or click on the OK button.

The TASKING Welcome dialog box appears.

Step 5

Select a product and click on Install.

Step 6

Follow the instructions that appear on your screen.



You can find your serial number on the *Certificate of Authenticity* or *Product Update Form*, delivered with the product.

Step 7

License the software product as explained in section 1.5, *Licensing TASKING Products*.

1.3 INSTALLATION FOR LINUX

Each product on the CD-ROM is available as an RPM package and as a gzipped tar file. For each product the following files are present:

```
SWproduct-version-RPMrelease.i386.rpm  
SWproduct-version.tar.gz
```

Both files contain exactly the same information. When your Linux distribution supports RPM packages, you can install the `.rpm` file. Otherwise, you can install the product from the `.tar.gz` file.

1.3.1 RPM INSTALLATION

Step 1

In most situations you have to be "root" to install RPM packages, so either login as "root", or use the **su** command.

Step 2

Insert the CD-ROM into the CD-ROM drive. Mount the CD-ROM on a directory, for example `/cdrom`. See the Linux manual pages about **mount** for details.

Step 3

Go to the directory on which the CD-ROM is mounted:

```
cd /cdrom
```

Step 4

To install or upgrade all products at once, issue the following command:

```
rpm -U SW*.rpm
```

This will install or upgrade all products in the default installation directory `/usr/local`. Every RPM package will create a single directory in the installation directory.

The RPM packages are 'relocatable', so it is possible to select a different installation directory with the **--prefix** option. For instance when you want to install the products in /opt, use the following command:

```
rpm -U --prefix /opt SW*.rpm
```



For Red Hat 6.0 users: The **--prefix** option does not work with RPM version 3.0, included in the Red Hat 6.0 distribution. Please upgrade to RPM version 3.0.3 or higher, or use the .tar.gz file installation described in the next section if you want to install in a non-standard directory.

1.3.2 TAR.GZ INSTALLATION

Step 1

Login as a user.

Be sure you have read, write and execute permissions in the installation directory. Otherwise, login as "root" or use the **su** command.

Step 2

Insert the CD-ROM into the CD-ROM drive. Mount the CD-ROM on a directory, for example /cdrom. See the Linux manual pages about **mount** for details.

Step 3

Go to the directory on which the CD-ROM is mounted:

```
cd /cdrom
```

Step 4

To install the products from the .tar.gz files in the directory /usr/local, issue the following command for each product:

```
tar xzf SWproduct-version.tar.gz -C /usr/local
```

Every .tar.gz file creates a single directory in the directory where it is extracted.

1.3.3 SETTING THE ENVIRONMENT

After you have installed the software, you can set some of the environment variables to make invocation of the tools easier (when invoking the tools from the command line). A list of all environment variables used by the toolchain is present in the section *Environment Variables* in the chapter *Overview*.

Make sure that your path is set to include all of the executables you have just installed.

The environment variable TMPDIR can be used to specify a directory where programs can place temporary files.

1.4 INSTALLATION FOR UNIX HOSTS

Step 1

Login as a user.

Be sure you have read, write and execute permissions in the installation directory. Otherwise, login as root or use the **su** command.

Step 2

If you are a first time user decide where you want to install the product (By default it will be installed in `/usr/local`).

Step 3

For CD-ROM install: insert the CD-ROM into the CD-ROM drive. Mount the CD-ROM on a directory, for example `/cdrom`. Be sure to use a ISO 9660 file system with Rock Ridge extensions enabled. See the UNIX manual pages about **mount** for details.

Or:

For tape install: insert the tape into the tape unit and create a directory where the contents of the tape can be copied to. Consider the created directory as a temporary workspace that can be deleted after installation has succeeded. For example:

```
mkdir /tmp/instdir
```

Step 4

For CD-ROM install: go to the directory on which the CD-ROM is mounted:

```
cd /cdrom
```

For tape install: copy the contents of the tape to the temporary workspace using the following commands:

```
cd /tmp/instdir
tar xvf /dev/tape
```

where *tape* is the name of your tape device.



If you have received a tape with more than one product, use the non-rewinding device for installing the products.

Step 5

Run the installation script:

```
sh install
```

and follow the instructions appearing on your screen.

First a question appears about where to install the software. The default answer is **/usr/local**. On certain sites you may want to select another location.

On some hosts the installation script asks if you want to install SW000098, the Flexible License Manager (FLEXlm). If you do not already have FLEXlm on your system, you must install it; otherwise the product will not work on those hosts. See section 1.5, *Licensing TASKING Products*.

If the script detects that the software has been installed before, the following messages appear on the screen:

```
*** WARNING ***
SWxxxx xxxx.xxxx already installed.
Do you want to REINSTALL? [y,n]
```

Answering **n** (no) to this question causes installation to abort and the following message being displayed:

```
=> Installation stopped on user request <=
```

Answering **y** (yes) to this question causes installation to continue. And the final message will be:

```
Installation of SWxxxx xxxx.xxxx completed.
```

For the M16C the directory **cm16c** will be created.

Step 6

For tape install: remove the temporary installation directory with the following commands:

```
cd /tmp
rm -rf instdir
```

Step 7

If you purchased a protected TASKING product, license the software product as explained in section 1.5, *Licensing TASKING Products*.

Step 8

Logout.

1.4.1 SETTING THE ENVIRONMENT

After you have installed the software, you can set some environment variables to make invocation of the tools easier. A list of all environment variables used by the toolchain is present in the section *Environment Variables* in the chapter *Overview*.

Make sure that your path is set to include all of the executables you have just installed.

The environment variable TMPDIR can be used to specify a directory where programs can place temporary files.

1.5 LICENSING TASKING PRODUCTS

TASKING products are protected with license management software (FLEXlm). To use a TASKING product, you must install the licensing information provided by TASKING for the type of license purchased.

You can run TASKING products with a node-locked license or with a floating license. When you order a TASKING product determine which type of license you need (UNIX products only have a floating license).

Node-locked license (PC only)

This license type locks the software to one specific PC so you can use the product on that particular PC only.

Floating license

This license type manages the use of TASKING product licenses among users at one site. This license type does not lock the software to one specific PC or workstation but it requires a network. The software can then be used on any computer in the network. The license specifies the number of users who can use the software simultaneously. A system allocating floating licenses is called a **license server**. A license manager running on the license server keeps track of the number of users.



See the *Flexible License Manager (FLEXlm)* appendix for detailed information on FLEXlm.

1.5.1 OBTAINING LICENSE INFORMATION

Before you can install a software license you must have a "License Information Form" containing the license information for your software product. If you have not received such a form follow the steps below to obtain one. Otherwise, you can install the license.

Node-locked license (PC only)

1. If you need a node-locked license, you must determine the hostid of the computer where you will be using the product. See section 1.5.7, *How to Determine the Hostid*.
2. When you order a TASKING product, provide the hostid to your local TASKING sales representative. The License Information Form which contains your license key information will be sent to you with the software product.

Floating license

1. If you need a floating license, you must determine the hostid and hostname of the computer where you want to use the license manager. Also decide how many users will be using the product. See section 1.5.7, *How to Determine the Hostid* and section 1.5.8, *How to Determine the Hostname*.
2. When you order a TASKING product, provide the hostid, hostname and number of users to your local TASKING sales representative. The License Information Form which contains your license key information will be sent to you with the software product.

1.5.2 INSTALLING NODE-LOCKED LICENSES

Keep your "License Information Form" ready. If you do not have such a form read section 1.5.1, *Obtaining License Information*, before continuing.

Step 1

Install the TASKING software product following the installation procedure described in section 1.2, *Installation for Windows*.

Step 2

Create a file called "license.dat" in the c:\flexlm directory, using an ASCII editor and insert the license information contained in the "License Information Form" in this file. This file is called the "license file". If the directory c:\flexlm does not exist, create the directory.



If you wish to install the license file in a different directory, see section 1.5.6, *Modifying the License File Location*.



If you already have a license file, add the license information to the existing license file. If the license file already contains any SERVER lines, you must use another license file. See section 1.5.6, *Modifying the License File Location*, for additional information.

The software product and license file are now properly installed.



See the *Flexible License Manager (FLEXlm)* appendix for more information on FLEXlm.

1.5.3 INSTALLING FLOATING LICENSES

Keep your "License Information Form" ready. If you do not have such a form read section 1.5.1, *Obtaining License Information*, before continuing.

Step 1

Install the TASKING software product following the installation procedure described earlier in this chapter on the computer or workstation where you will use the software product.

As a result of this installation two additional files for FLEXlm will be present in the flexlm subdirectory of the toolchain:

Tasking	The Tasking daemon (vendor daemon).
license.dat	A template license file.

Step 2

If you already have installed FLEXlm v6.1 or higher for Windows or v2.4 or higher for UNIX (for example as part of another product) you can skip this step and continue with step 3. Otherwise, install SW000098, the Flexible License Manager (FLEXlm), on the license server where you want to use the license manager.

The installation of the license manager on Windows also sets up the license daemon to run automatically whenever a license server reboots. On UNIX you have to perform the steps as described in section 1.5.5, *Setting Up the License Daemon to Run Automatically*.



It is not recommended to run a license manager on a Windows 95 or Windows 98 machine. Use Windows NT instead (or UNIX).

Step 3

If FLEXlm has already been installed as part of a non-TASKING product you have to make sure that the `bin` directory of the FLEXlm product contains a copy of the **Tasking** daemon (see step 1).

Step 4

Insert the license information contained in the "License Information Form" in the license file, which is being used by the license server. This file is usually called `license.dat`. The default location of the license file is in directory `c:\flexlm` for Windows and in `/usr/local/flexlm/licenses` for UNIX.



If you wish to install the license file in a different directory, see section 1.5.6, *Modifying the License File Location*.

If the license file does not exist, you have to create it using an ASCII editor. You can use the license file `license.dat` from the toolchain's `flexlm` subdirectory as a template.



If you already have a license file, add the license information to the existing license file. If the `SERVER` lines in the license file are the same as the `SERVER` lines in the License Information Form, you do not need to add this same information again. If the `SERVER` lines are not the same, you must use another license file. See section 1.5.6, *Modifying the License File Location*, for additional information.

Step 5

On each PC or workstation where you will use the TASKING software product the location of the license file must be known. If it differs from the default location (`c:\flexlm\license.dat` for Windows, `/usr/local/flexlm/licenses/license.dat` for UNIX), then you must set the environment variable **LM_LICENSE_FILE**. See section 1.5.6, *Modifying the License File Location*, for more information.

Step 6

Now all license information is entered, the license manager must be started (see section 1.5.4). Or, if it is already running you must notify the license manager that the license file has changed by entering the command (located in the flexlm bin directory):

lmreread

On Windows you can also use the graphical FLEXlm Tools (**lmtools**): Start **lmtools** (if you have used the defaults this can be done by selecting Start | Programs | TASKING FLEXlm | FLEXlm Tools), fill in the current license file location if this field is empty, click on the Reread button and then on OK. Another option is to reboot your PC.

The software product and license file are now properly installed.

Where to go from here?

The license manager (daemon) must always be up and running. Read section 1.5.4 on how to start the daemon and read section 1.5.5 for information how to set up the license daemon to run automatically.

If the license manager is running, you can now start using the TASKING product.



See the *Flexible License Manager (FLEXlm)* appendix for detailed information on FLEXlm.

1.5.4 STARTING THE LICENSE DAEMON

The license manager (daemon) must always be up and running. To start the daemon complete the following steps on each license server:

Windows

1. Start the license manager tool by (Start | Programs | TASKING FLEXlm | FLEXlm License Manager).
2. In the Control tab, click on the Start button.
3. Close the program by clicking on the OK button.

UNIX

1. Log in as the operating system administrator (usually root).
2. Change to the FLEXlm installation directory (default /usr/local/flexlm):

```
cd /usr/local/flexlm
```

3. For C shell users, start the license daemon by typing the following:

```
bin/lmgrd -2 -p -c licenses/license.dat >>& \
/var/tmp/license.log &
```

Or, for Bourne shell users, start the license daemon by typing the following:

```
bin/lmgrd -2 -p -c licenses/license.dat >> \
/var/tmp/license.log 2>&1 &
```

In these two commands, the **-2** and **-p** options restrict the use of the **lmdown** and **lmremove** license administration tools to the license administrator. You omit these options if you want. Refer to the usage of **lmgrd** in the *Flexible License Manager (FLEXlm)* appendix for more information.

1.5.5 SETTING UP THE LICENSE DAEMON TO RUN AUTOMATICALLY

To set up the license daemon so that it runs automatically whenever a license server reboots, follow the instructions below that are appropriate for your platform. steps on each license server:

Windows

1. Start the license manager tool by (Start | Programs | TASKING FLEXlm | FLEXlm License Manager).
2. In the Setup tab, enable the Start Server at Power-Up check box.
3. Close the program by clicking on the OK button. If a question appears, answer Yes to save your settings.

UNIX

In performing any of the procedures below, keep in mind the following:

- Before you edit any system file, make a backup copy.

HP-UX

1. Log in as the operating system administrator (usually root).
2. In the directory `/etc/rc.config.d` create a file named `rc.lmgrd` with the following contents. Replace *FLEXLMDIR* by the FLEXlm installation directory (default `/usr/local/flexlm`):

```
#!/sbin/sh
FLEXLMDIR/bin/lmgrd -2 -p -c FLEXLMDIR/licenses/license.dat >> \
/var/tmp/license.log 2>&1 &
```

After the `-c` option, you have to specify the correct location of the license file.

SunOS4

1. Log in as the operating system administrator (usually root).
2. Append the following lines to the file `/etc/rc.local`. Replace *FLEXLMDIR* by the FLEXlm installation directory (default `/usr/local/flexlm`):

```
FLEXLMDIR/bin/lmgrd -2 -p -c FLEXLMDIR/licenses/license.dat >> \
/var/tmp/license.log 2>&1 &
```

SunOS5 (Solaris 2)

1. Log in as the operating system administrator (usually root).
2. In the directory `/etc/init.d` create a file named `rc.lmgrd` with the following contents. Replace *FLEXLMDIR* by the FLEXlm installation directory (default `/usr/local/flexlm`):

```
#!/bin/sh
FLEXLMDIR/bin/lmgrd -2 -p -c FLEXLMDIR/licenses/license.dat >> \
/var/tmp/license.log 2>&1 &
```

3. Make it executable:

```
chmod u+x rc.lmgrd
```


4. Create an 'S' link in the /etc/rc3.d directory to this file and create 'K' links in the other /etc/rc?.d directories:

```
ln /etc/init.d/rc.lmgrd /etc/rc3.d/Snumrc.lmgrd
ln /etc/init.d/rc.lmgrd /etc/rc?.d/Knumrc.lmgrd
```

num must be an appropriate sequence number. Refer to your operating system documentation for more information.

1.5.6 MODIFYING THE LICENSE FILE LOCATION

The default location for the license file on Windows is:

```
c:\flexlm\license.dat
```

On UNIX this is:

```
/usr/local/flexlm/licenses/license.dat
```

If you want to use another name or directory for the license file, each user must define the environment variable **LM_LICENSE_FILE**. Do this in `autoexec.bat` (Windows 95/98), from the Control Panel -> System | Environment (Windows NT) or in a UNIX login script.

If you have more than one product using the FLEXlm license manager you can specify multiple license files to the **LM_LICENSE_FILE** environment variable by separating each pathname (*lppath*) with a ';' (on UNIX also ':'):

Example Windows:

```
set LM_LICENSE_FILE=c:\flexlm\license.dat;c:\license.txt
```

Example UNIX:

```
setenv LM_LICENSE_FILE
/usr/local/flexlm/licenses/license.dat:/myprod/license.txt
```

If the license file is not available on these hosts, you must set **LM_LICENSE_FILE** to *port@host*; where *host* is the host name of the system which runs the FLEXlm license manager and *port* is the TCP/IP port number on which the license manager listens.

To obtain the port number, look in the license file at *host* for a line starting with "SERVER". The fourth field on this line specifies the TCP/IP port number on which the license server listens. For example:

```
setenv LM_LICENSE_FILE 7594@elliott
```

 See the *Flexible License Manager (FLEXlm)* appendix for detailed information.

1.5.7 HOW TO DETERMINE THE HOSTID

The hostid depends on the platform of the machine. Please use one of the methods listed below to determine the hostid.

Platform	Tool to retrieve hostid	Example hostid
HP-UX	lanscan (use the station address without the leading '0x')	0000F0050185
SunOS/Solaris	hostid	170a3472
Windows	tkhostid (or use lmhostid)	0800200055327

Table 1-1: Determine the hostid

 If you do not have the program **tkhostid** you can download it from our Web site at: <http://www.tasking.com/support/flexlm/tkhostid.zip> . It is also on every product CD that includes FLEXlm.



1.5.8 HOW TO DETERMINE THE HOSTNAME

To retrieve the hostname of a machine, use one of the following methods.

Platform	Method
HP-UX	hostname
SunOS/Solaris	hostname
Windows 95/98	Go to the Control Panel, open "Network", click on "Identification". Look for "Computer name".
Windows NT	Go to the Control Panel, open "Network". In the "Identification" tab look for "Computer Name".

Table 1-2: Determine the hostname

CHAPTER

2

OVERVIEW



2

CHAPTER

2.1 INTRODUCTION TO M16C C CROSS-COMPILER

This manual provides a functional description of the TASKING M16C C Cross-Compiler. This manual uses **cm16** (the name of the binary) as a shorthand notation for "TASKING M16C C Compiler".

The TASKING M16C C compiler accepts source programs written in ANSI C and translates these into M16C assembly source code files. The compiler accepts language extensions to improve code performance and to allow the use of typical M16C architectural provisions efficiently at the C level. The compiler is ANSI C compatible and consists of three major parts; the *preprocessor*, the C *frontend* and the associated M16C *backend* or code generator. These are all integrated into a single program to avoid the need of intermediate files, thus speeding up the compilation process. It also simplifies the implementation of joint frontend-backend optimization strategies and preprocessor pragmas. This effectively makes the compiler a one pass compiler, with minimum file I/O overhead.

The compiler processes one C function at a time, until the entire source module has been read. The function is parsed, checked on semantic correctness and then transformed into an intermediate code tree that is stored in memory. Code optimizations are performed during the construction of the intermediate code, and are also applied when the complete function has been processed. The latter are often referred to as *global* optimizations.

cm16 generates assembly source code using the M16C assembly language specification, you must assemble this code with the TASKING M16C Cross-Assembler. This manual uses **asm16** as a shorthand notation for "TASKING M16C Cross-Assembler".

You can link the generated object with other objects and libraries using the TASKING **lkm16** M16C linker. In this manual we use **lkm16** as a shorthand notation for "TASKING **lkm16** M16C linker". You can locate the linked object to a complete application using the TASKING **lcm16** M16C locator. In this manual we use **lcm16** as a shorthand notation for "TASKING **lcm16** M16C locator".

The program **ccm16** is a control program. The control program facilitates the invocation of various components of the M16C toolchain. **ccm16** recognizes several filename extensions. C source files (.c) are passed to the compiler. Assembly sources (.asm) are preprocessed and passed to the assembler. Relocatable object files (.obj) and libraries (.a) are recognized as linker input files. Files with extension .out and .dsc are treated as locator input files. The control program supports options to stop at any stage in the compilation process and has options to produce and retain intermediate files.

You can debug the software written in C with the TASKING CrossView Pro high-level language debugger. This manual uses XVW as a shorthand notation for "TASKING CrossView Pro high-level language debugger". A list of supported platforms and emulators is available from TASKING.

2.2 PRODUCT DEFINITION

Name:

TASKING M16C C Cross-Compiler

Ordering Code:

TK499-002-05

Target Assembler:

TASKING M16C Cross-Assembler

TK499-000-05 (included in TK499-002-05)

Target Debugger:

TASKING M16C CrossView Pro debugger Simulator (TK499-043-05)

TASKING M16C CrossView Pro debugger ROM Monitor (TK499-041-05)

Target Processors:

All M16C derivatives. Special function registers can be accessed by means of a user-definable register file.

2.3 GENERAL IMPLEMENTATION

This section describes the different phases of the compiler and the target independent optimizations.

2.3.1 COMPILER PHASES

During the compilation of a C program, a number of phases can be identified. These phases are divided into two groups, referred to as *front end* and *backend*.

frontend:

The preprocessor phase:

File inclusion and macro substitution are done by the preprocessor before parsing of the C program starts. The syntax of the macro preprocessor is independent of the C syntax, but also described in the ANSI X3.159-1989 standard.

The scanner phase:

The scanner converts the preprocessor output to a stream of tokens.

The parser phase:

The tokens are fed to a parser for the C grammar. The parser performs a syntactic and semantic analysis of the program, and generates an intermediate representation of the program.

The frontend optimization phase:

Target processor independent optimization is performed by transforming the intermediate code. The next section discusses the frontend optimizations.

backend:

The backend optimization phase:

Performs target processor specific optimizations. Very often this means another transformation of the intermediate code and actions like register allocation techniques for variables, expression evaluation and the best usage of the addressing modes. The chapter *Language Implementation* discusses this item in more detail.

The code generator phase:

This phase converts the intermediate code to an internal instruction code, representing the M16C assembly instructions.

The peephole optimizer:

This phase uses pattern matching techniques to perform peephole optimizations on the internal code. The peephole optimizer translates the internal instruction code into assembly code for **asm16**.

All phases (of both frontend and backend) of the compiler are combined into one program. The compiler does not use intermediate files for communication between the different phases of compilation. The back end part is not called for each C statement, but starts after a complete C function has been processed by the frontend (in memory), thus allowing more optimization. The compiler only requires one pass over the input file, resulting in relatively fast compilation.

2.3.2 FRONTEND OPTIMIZATIONS

The command line option **-O** controls the amount of optimization applied on the C source. Within a source file, the pragma `#pragma optimize` sets the optimization level of the compiler. Using the pragma, certain optimizations can be switched on or off for a particular part of the program. Several optimizations cannot be controlled individually. e.g., constant folding will always be done.

The compiler performs the following optimizations on the intermediate code. They are independent of the target processor and the code generation strategy:

Constant folding

Expressions only involving constants are replaced by their result.

Expression rearrangement

Expressions are rearranged to allow more constant folding. E.g. $1 + (x - 3)$ is transformed into $x + (1 - 3)$, which can be folded.

Expression simplification

Multiplication by 0 or 1 and additions or subtractions of 0 are removed. Such useless expressions may be introduced by macros, or by the compiler itself (e.g., array subscription).

Logical expression optimization

Expressions involving '&&', '|', and '!' are interpreted and translated into a series of conditional jumps.

Loop rotation

With `for` and `while` loops, the expression is evaluated once at the 'top' and then at the 'bottom' of the loop. This optimization does not save code, but speeds up execution.

Switch optimization

A number of optimizations of a switch statement are performed, such as the deletion of redundant case labels or even the deletion of the switch.

Control flow optimization

By reversing jump conditions and moving code, the number of jump instructions is minimized. This reduces both the code size and the execution time.

Jump chaining

A conditional or unconditional jump to a label which is immediately followed by an unconditional jump may be replaced by a jump to the destination label of the second jump. This optimization does not save code, but speeds up execution.

Remove useless jumps

An unconditional jump to a label directly following the jump is removed. A conditional jump to such a label is replaced by an evaluation of the jump condition. The evaluation is necessary because it may have side effects.

Conditional jump reversal

A conditional jump over an unconditional jump is transformed into one conditional jump with the jump condition reversed. This reduces both the code size and the execution time.

Cross jumping and branch tail merging

Identical code sequences in two different execution paths are merged when this is possible without adding extra instructions. This transformation decreases code size rather than execution time, but under certain circumstances it avoids the execution of one jump.

Constant/copy propagation

A reference to a variable with known contents is replaced by those contents.

Common subexpression elimination

The compiler has the ability to detect repeated uses of the same (sub-) expression. Such a "common" expression may be temporarily saved to avoid recomputation. This method is called *common subexpression elimination*, abbreviated CSE.

Dead code elimination

Unreachable code can be removed from the intermediate code without affecting the program. However, the compiler generates a warning message, because the unreachable code may be the result of a coding error.

Loop optimization

Invariant expressions may be moved out of a loop and expressions involving an index variable may be reduced in strength.

Loop unrolling

Eliminate short loops by replacing them with a number of copies.

Sharing of string literals and floating point constants

String literals and floating point constants are put in ROM memory. The compiler overlays identical strings (within the same module) and let them share the same space, thus saving ROM space. Likewise identical floating point constants are overlaid and allocated only once.

2.3.3 BACKEND OPTIMIZATIONS

The following optimizations are target dependent and are therefore performed by the backend.

Allocation graph

Variables, parameters, intermediate results and common subexpressions are represented in allocation units. Per function, the compiler builds a graph of allocation units which indicates which units are needed and when. This allows the register allocator to get the most efficient occupation of the available registers. The compiler uses the allocation graph to generate the assembly code.

Peephole optimizations

The generated assembly code is improved by replacing instruction sequences by equivalent but faster and/or shorter sequences, or by deleting unnecessary instructions.

Leaf function handling

Leaf functions (function not calling other functions), are handled specially with respect to stack frame building.

Dead store elimination

Expressions from which the result is never used are eliminated.

Interrupt frame optimizations

Only resources required by the interrupt function are saved.

Tail recursion elimination

Replace a recursion statement to branch to the beginning of the statement.

Jump table optimizations

Three ways of code generation for a switch statement are supported: a jump chain (linear switch), a jump table, or a binary search table.

2.4 COMPILER STRUCTURE

If you want to build an M16C application you need to invoke the following programs directly, or via the control program:

- The C compiler (**cm16**), which generates an assembly source file from the file with suffix `.c`. The suffix of the compiler output file is `.src`. However, you can direct the output to `stdout` with the **-n** option, or to another file with the **-o** option. C source lines can be intermixed with the generated assembly statements with the **-s** option. High level language debugging information can be generated with the **-g** option. You are advised not to use the **-g** option when inspecting the generated assembly source code, because it contains a lot of 'unreadable' high level language debug directives. The C compilers make only one pass on every file. This pass checks the syntax, generates the code and performs code optimization.
- The corresponding cross-assembler (**asm16**), which processes the generated assembly source file into a relocatable object file with suffix `.obj`. A full assembly listing with suffix `.lst` is available after this stage.
- The **lkm16** linker, which links the generated relocatable object files and C-libraries. The result is a relocatable object file with suffix `.out`. A linker map file with suffix `.lnl` is available after this stage.
- The **lcm16** locator, which locates the generated relocatable object files. The result is an absolute loadable file with suffix `.abs`. A full application map file with suffix `.map` is available after this stage.

You can directly load the output file of the locator with extension `.abs` into the CrossView Pro debugger.

The next figure explains the relationship between the different parts of the TASKING M16C toolchain:

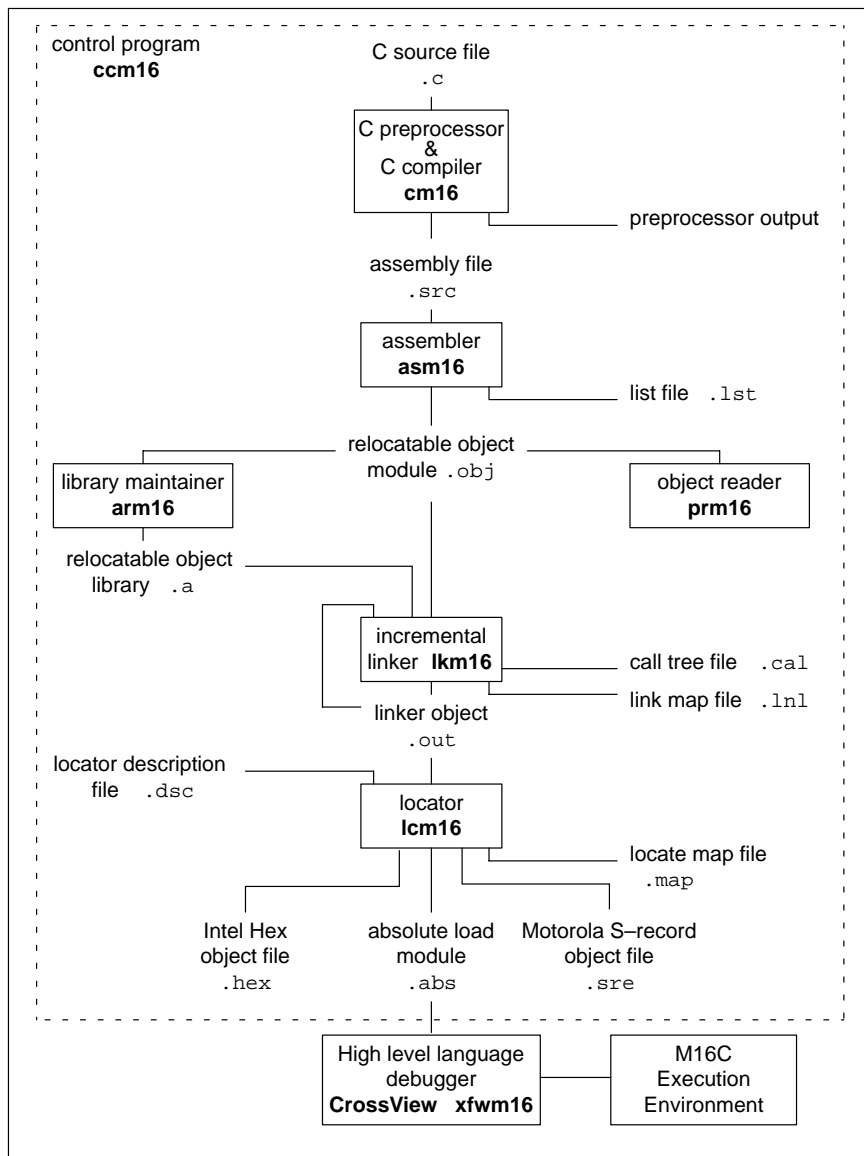


Figure 2-1: M16C development flow

The program **ccm16** is a so-called control program, which facilitates the invocation of various components of the M16C toolchain. C source programs are compiled by the compiler, assembly source files are passed to the assembler. A C preprocessor program is available as an integrated part of the C compiler. The control program recognizes the file extensions `.a` and `.obj` as input files for the linker. The control program passes files with extensions `.out` and `.dsc` to the locator. All other files are considered to be object files and are passed to the linker. The control program has options to suppress the locating stage (**-cl**), the linker stage (**-c**) or the assembler stage (**-cs**).

Optionally the locator, **lcm16** produces output files in Motorola S-record format or Intel Hex format. The default output format is IEEE-695.

Normally, the control program removes intermediate compilation results, as soon as the next phase completes successfully. If you want to retain all intermediate files, the option **-tmp** prevents removal of these files.

For a description of all utilities available and the possible output formats of the locator, see the M16C Cross-Assembler and Utilities User's Guide.

The name of the M16C CrossView Pro Debugger is **xfwm16**. For more information check the *M16C CrossView Pro Debugger User's Guide*.



2.5 ENVIRONMENT VARIABLES

This section contains an overview of the environment variables used by the M16C toolchain.

Environment Variable	Description
ASM16CINC	Specifies an alternative path for include files for the assembler.
CM16CINC	Specifies an alternative path for #include files for the C compiler cm16 .
CM16CLIB	Specifies a path to search for library files used by the linker lkm16 .
CCM16CBIN	When this variable is set, the control program, ccm16 , prepends the directory specified by this variable to the names of the tools invoked.
CCM16COPT	Specifies extra options and/or arguments to each invocation of ccm16 . The control program processes the arguments from this variable before the command line arguments.
PATH	Specifies the search path for your executables.
TMPDIR	Specifies an alternative directory where programs can create temporary files. Used by cm16 , ccm16 , asm16 , lkm16 , lcm16 , arm16 .

Table 2-1: Environment variables

2.6 SAMPLE SESSION

The subdirectory `dhry` in the `examples` subdirectory contains a demo program for the M16C toolchain.

In order to debug your programs, you will have to compile, assemble, link and locate them for debugging using the TASKING M16C tools. You can do this with one call to the control program or you can use EDE, the Embedded Development Environment (which uses a project file and a makefile) or you can call the makefile from the command line.

2.6.1 USING EDE

EDE stands for "Embedded Development Environment" and is the MS-Windows oriented Integrated Development Environment you can use with your TASKING toolchain to design and develop your application.

To use EDE on the `dhry` demo program in the subdirectory `dhry` in the `examples` subdirectory of the M16C product tree follow the steps below. This procedure is outlined as a guide for you to build your own executables for debugging.



The dialog boxes shown in this manual serve as an example. They may slightly differ from the ones in your product.

How to Start EDE

You can launch EDE by double-clicking on the EDE shortcut on your desktop.



The EDE screen provides you with a menu bar, a toolbar (command buttons) and one or more windows (for example, for source files), a status bar and numerous dialog boxes.

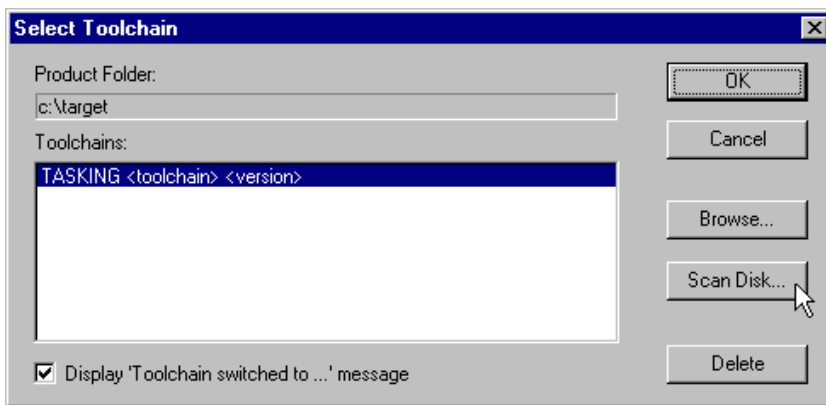


How to Select a Toolchain

EDE supports all the TASKING toolchains. When you first start EDE, the toolchain of the product you purchased is selected and displayed in the title of the EDE desktop window.

If you selected the wrong toolchain or if you want to change toolchains do the following:

1. Access the EDE menu and select the `Select Toolchain...` menu item. This opens the `Select Toolchain` dialog.
2. Select the toolchain you want. You can do this by clicking on a toolchain in the `Toolchains` list box and press OK.



If no toolchains are present, use the `Browse...` or `Scan Disk...` button to search for a toolchain directory. Use the `Browse...` button if you know the installation directory of another TASKING product. Use the `Scan Disk...` button to search for all TASKING products present on a specific drive. Then return to step 2.

How to Open an Existing Project

Follow these steps to open an existing project:

1. Access the `Project` menu and select `Set Current...`
2. Click the project file to open. For the `dhry` demo program select the file `dhry.pjt` in the subdirectory `dhry` in the `examples` subdirectory of the `M16C` product tree. If you have used the defaults, the file `dhry.pjt` is in the directory `c:\cm16c\examples\dhry`.

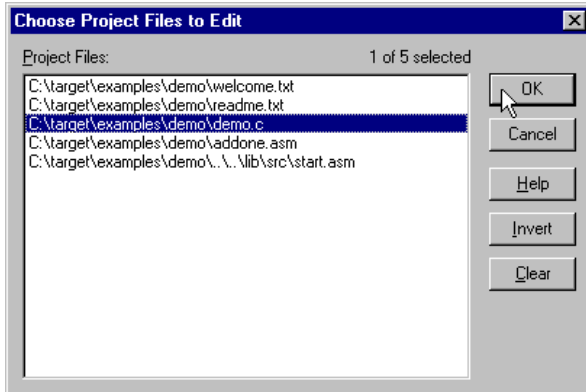
How to Load/Open Files

The next two steps are not needed for the demo program because the files `dhry_1.c` and `dhry_2.c` are already open. To load the file you want to look at.

1. In the Project menu click on Load files....

This opens the Choose Project Files to Edit dialog.

2. Choose the file(s) you want to open by clicking on it. You can select multiple files by pressing the <Ctrl> or <Shift> key while you click on a file. With the <Ctrl> key you can make single selections and with the <Shift> key you can select everything from the first selected file to the file you click on. Then press the OK button.



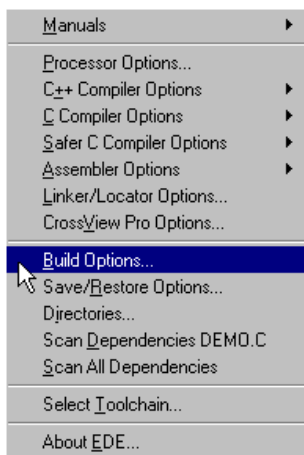
This launches the file(s) so you can edit it (them).

How to Build the Demo Application

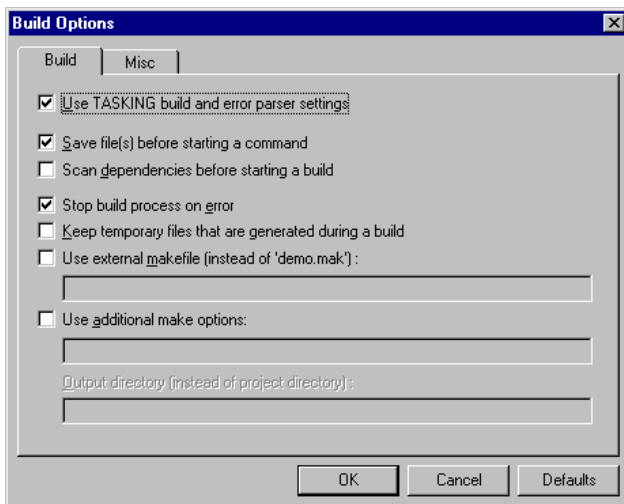
The next step is to compile the file(s) together with its dependent files so you can debug the application.

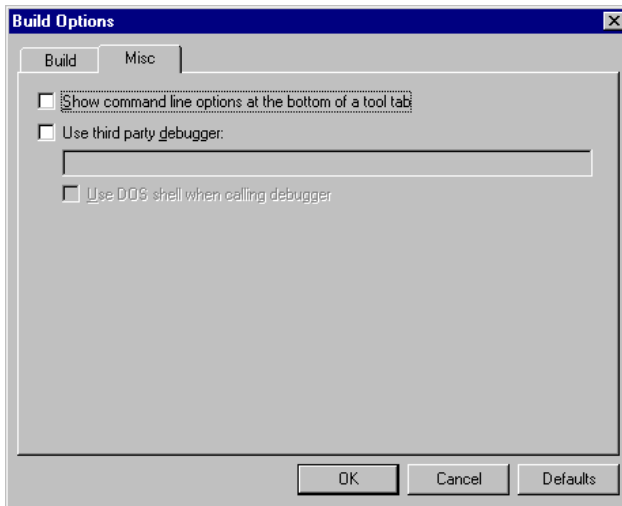
Steps 1 and 2 are optional. Follow these steps if you want to specify additional build options such as to stop the build process on errors and to select a command to be executed as foreground or background process.

1. Access the EDE menu and select the Build Options... menu item.

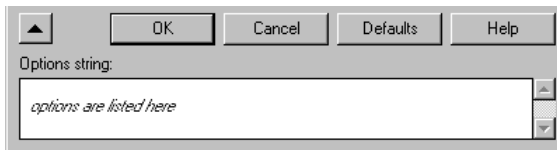


This opens the Build Options dialog.

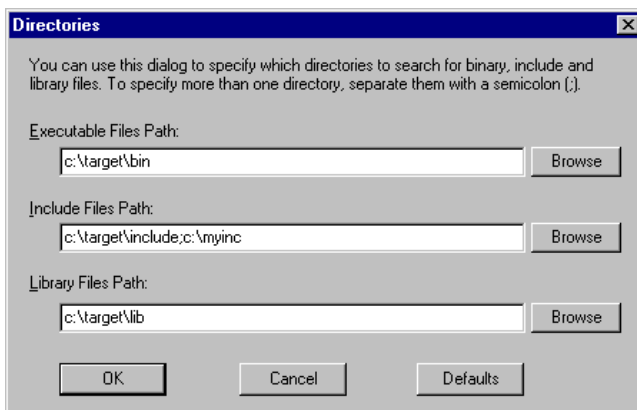




If you set the Show command line options at the bottom of a tool tab check box EDE shows the command line equivalent of the selected tool option. You can also click on the arrow button (left of the OK button) in a tool options dialog.



2. Make your changes and press the OK button.
3. Select the EDE | Directories menu item and check the directory paths for programs, include files and libraries. You can add your own directories here, separated by semicolons.



4. Access the EDE menu and select the Scan All Dependencies menu item.
5. Click on the Execute 'Make' command button. The following button is the execute Make button which is located in the toolbar.



If there are any unsaved files, EDE will ask you in a separate dialog if you want to save them before starting the build.

How to View the Results of a Build

Once the files have been processed you can inspect the generated messages in the Build tab:

```
TASKING program builder vx.y rz          SN00000001-020 (c) year TASKING, Inc.
Compiling "dhry_2.c"
Assembling "dhry_2.src"
Compiling "dhry_1.c"
Assembling "dhry_1.src"
Linking to "dhry.out"
Locating "dhry.out" to "dhry.abs" (IEEE-695)
```

How to Start the CrossView Pro Debugger

Once the files have been compiled, assembled, linked, located and formatted they can be executed by CrossView Pro.

To execute CrossView Pro:

1. Click on the `Debug` application button. The following button is the `Debug` application button which is located in the toolbar.



CrossView Pro is launched. CrossView Pro will automatically download the compiled file for debugging.

How to Load an Application

You must tell CrossView Pro which program you want to debug. To do this:

1. Click on `File` in the menu bar and select the `Load Symbolic Debug Info...` item. This opens up the `Load Symbolic Debug Info` dialog box.
2. Click `Load`.

How to View and Execute an Application

To view your source while debugging, the `Source Window` must be open. To open this window,

1. Click on `View` in the menu bar and select the `Source->Source lines` item.

Before starting execution you have to reset the target system to its initial state. The program counter, stack pointer and any other registers must be set to their initial value. The easiest way to do this is:

2. Click on `Run` in the menu bar and select the `Program Reset` item.
3. Again click on `Run` in the menu bar and now select the `Animate` item.

The program `dhry_1.abs` is now stepping through the high level language statements. Using the `Accelerator bar` or the menu bar you can set breakpoints, monitor data, display registers, simulate I/O and much more. See the *CrossView Pro Debugger User's Guide* for more information.

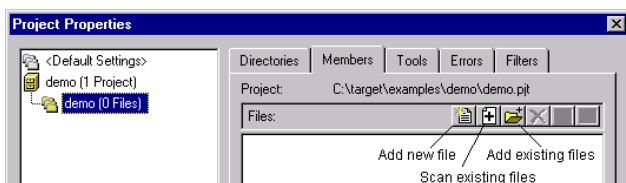
How to Start a New Project

When you first use EDE you need to setup a project space and add a new project:

1. Access the Project menu and select Project Space | New...
2. Give your project space a name and then click OK.
3. Click on the Add new project to project space button.
4. Give your project a name and then click OK.

The Project Properties dialog box then appears for you to identify the files to be added.

5. Add all the files you want to be part of your project. Then press the OK button. To add files, use one of the 3 methods described below.



- If you do not have any source files yet, click on the Add new file to project button in the Project Properties dialog. Enter a new filename and click OK.
- To add existing files to a project by specifying a file pattern click on the Scan existing files into project button in the Project Properties dialog. Select the directory that contains the files you want to add to your project. Enter one or more file patterns separated by semicolons. The button next to the Pattern field contains some predefined patterns. Next click OK.
- To add existing files to a project by selecting individual files click on the Add existing files to project button in the Project Properties dialog. Select the directory that contains the files you want to add to your project. Add the applicable files by double-clicking on them or by selecting them and pressing the Open button.

The new project is now open.

6. Click Project | Load Files to open files you want on your EDE desktop.

EDE automatically creates a makefile for the project. EDE updates the makefile every time you modify your project.

2.6.2 USING THE CONTROL PROGRAM

A detailed description of the process using the sample program `sieve` is described below. This procedure is outlined as a guide for you to build your own executables for debugging.

1. Make the subdirectory `dhry` of the `examples` directory the current working directory.
2. Be sure that the directory of the binaries is present in the `PATH` environment variable.
3. Compile, assemble, link and locate the modules using one call to the control program **ccm16**:

```
ccm16 -g -M -o dhry.abs dhry_1.c dhry_2.c
```

The **-g** option specifies to generate symbolic debugging information. This option must always be specified when debugging with CrossView Pro.

The **-M** option specifies to generate map files.

The **-o** option specifies the name of the output file.

The command in step 3 generates the object file `dhry_1.obj`, and `dhry_2.obj` the linker map file `dhry.lnl`, the locator map file `dhry.map` and the absolute output file `dhry.abs`. The file `dhry.abs` is in the IEEE Std. 695 format, and can directly be used by CrossView. No separate formatter is needed.

Now you have created all the files necessary for debugging with CrossView Pro with one call to the control program.

If you want to see how the control program calls the compiler, assembler, linker and locator, you can use the **-v0** option or **-v** option. The **-v0** option only displays the invocations without executing them. The **-v** option also executes them.

```
ccm16 -g -M -o dhry.abs dhry_1.c dhry_2.c -v0
```

The control program shows the following command invocations without executing them (UNIX output):

```
M16C control program va.b rc          SN000000-003 (c)year TASKING, Inc.
dhry_1.c:
+ cml16 -e -g -Ms -o /tmp/cc26910b.src dhry_1.c
+ asml16 /tmp/cc26910b.src -e -g -o dhry_1.obj
dhry_2.c:
+ cml16 -e -g -Ms -o /tmp/cc26910c.src dhry_2.c
+ asml16 /tmp/cc26910c.src -e -g -o dhry_2.obj
+ lkm16 -e -M dhry_1.obj dhry_2.obj -lcs -lms -lfps -lrts -odhry.out -Odhry
+ lcm16 -e -M -odhry.abs -dm16c.dsc dhry.out
```

The **-e** option removes output files after errors occur. The **-Ms** option selects the small memory model. The **-lcs**, **-lfps** and **-lrts** options of the linker specify to link the appropriate C libraries. The **-O** option of the linker specifies the basename of the map file. The **-d** option of the locator specifies the name of the locator description file.

As you can see, the tools use temporary files for intermediate results. Also the file `dhry.out` will be removed afterwards. If you want to keep the intermediate files you can use the **-tmp** option. The following command makes this clear.

```
ccml16 -g -M -o dhry.abs dhry_1.c dhry_2.c -v0 -tmp
```

This command produces the following output:

```
M16C control program va.b rc          SN000000-003 (c)year TASKING, Inc.
dhry_1.c:
+ cml16 -e -g -Ms -o dhry_1.src dhry_1.c
+ asml16 dhry_1.src -e -g -o dhry_1.obj
dhry_2.c:
+ cml16 -e -g -Ms -o dhry_2.src dhry_2.c
+ asml16 dhry_2.src -e -g -o dhry_2.obj
+ lkm16 -e -M dhry_1.obj dhry_2.obj -lcs -lms -lfps -lrts -odhry.out -Odhry
+ lcm16 -e -M -odhry.abs -dm16c.dsc dhry.out
```

As you can see, if you use the **-tmp** option, the assembly source files and linker output file will be created in your current directory also.

Of course, you will get the same result if you invoke the tools separately using the same calling scheme as the control program.

As you can see, the control program automatically calls each tool with the correct options and controls. The control program is described in detail in Chapter *Compiler Use*.

2.6.3 USING THE MAKEFILE

The subdirectories in the `examples` directory each contain a makefile which can be processed by **mk16**. Also each subdirectory contains a `readme.txt` file with a description of how to build the example.

To build the `dhry` demo example follow the steps below. This procedure is outlined as a guide for you to build your own executables for debugging.

1. Make the subdirectory `dhry` of the `examples` directory the current working directory.

This directory contains a makefile for building the `dhry` demo example. It uses the default **mk16** rules.

2. Be sure that the directory of the binaries is present in the `PATH` environment variable.
3. Compile, assemble, link and locate the modules using one call to the program builder **mk16**:

mk16

This command will build the example using the file `makefile`.

To see which commands are invoked by **mk16** without actually executing them, type:

mk16 -n

This command produces the following output:

```
M16C program builder vx.y rz          SN000000-003 (c)year TASKING, Inc.
ccm16 -c -o dhry_1.obj  -g -w91 -w183 -w303 -Wa-gsL    dhry_1.c
ccm16 -c -o dhry_2.obj  -g -w91 -w183 -w303 -Wa-gsL    dhry_2.c
ccm16 -o dhry_1.abs  dhry_1.obj dhry_2.obj
```

The **-g** option in the makefile is used to instruct the C compiler to generate symbolic debug information. This information makes debugging an application written in C much easier to debug.

The **-w** option in the makefile is used to suppress a warning.

The **-o** option specifies the name of the output file.



Depending on the contents of the `makefile` other options can be displayed than the options described above. These are not of interest at this moment.

To remove all generated files type:

```
mkml6 clean
```


OVERVIEW

CHAPTER

3

LANGUAGE IMPLEMENTATION



3

CHAPTER

3.1 INTRODUCTION

The TASKING C cross-compiler (**cm16**) offers a new approach to high-level language programming for the M16C family. It conforms to the ANSI standard, but allows you to control the special functions of the M16C in C.

This chapter describes the C language implementation in relation to the M16C architecture.

The extensions to the C language in **cm16** are:

_bit You can use data type `_bit` for the type definition of scalars in the M16C bit-addressable area, and for the return type of functions.

_sfrbit Data type for the declaration of Special Function Registers. The compiler does not allocate memory for an `_sfrbit`.

_sfrbyte Data type for the declaration of Special Function Registers. The compiler does not allocate memory for an `_sfrbyte`.

_sfrword Data type for the declaration of Special Function Registers. The compiler does not allocate memory for an `_sfrword`.

_sfrlong Data type for the declaration of Special Function Registers. The compiler does not allocate memory for an `_sfrlong`.

_at You can place a variable at an absolute address.

storage types

Apart from a memory category (extern, static, ...) you can specify a storage type in each declaration. This way you obtain a memory model-independent addressing of variables in several address ranges (`_sfr`, `_near`, `_far`, `_rom`, `_farrom`, `_nearrom`).

inline C functions

You can specify to inline a function body instead of calling the function by using the `_inline` keyword.

assembly functions

Assembly functions can be called from C when they are prototyped with the `_asmfunc` keyword.

interrupt functions

You can specify interrupt functions directly through interrupt vectors in the C language (`_interrupt` keyword). You may also specify the register bank to be used.

3.2 ACCESSING MEMORY

The M16C has an address range of 1 Megabyte accessible via 20-bit addresses in one linear memory area.

cm16 offers two ways of dealing with the separate address spaces implemented in the M16C, which can be combined. You can:

- specify a storage type (and perhaps also a target memory of a pointer) with the declaration of a C variable

and you are able to:

- select a memory model, specifying which memory space must be used (as default) for all C variables which do not have an explicit storage specifier. This is very useful for compiling existing C source, which does not need to be adapted for the M16C.

In practice the majority of the C code of a complete application is standard C (without using any language extension). You can compile this part of the application without any modification, using the memory model which fits best to the requirements of the system (code density, amount of external RAM etc.).

Only a small part of the application uses language extensions. These parts often deal with items such as:

- I/O, using the special function registers
- high execution speed needed
- high code density needed
- access to non-default memory required (ROM, internal RAM)
- bit type needed
- C interrupt functions

3.2.1 STORAGE TYPES

An object other than a function or an automatic (stack) variable cannot be referred to solely by its starting address, because this might be valid for several address spaces. You can explicitly assign each variable to one of the logical address spaces (`_near`, `_far`, `_rom`) by using a type specifier. This specifier determines the 'storage type' of static objects.

cm16 recognizes the following storage type specifiers:

Storage Type	Description
<code>_bita</code>	bit-addressable RAM.
<code>_near</code>	Data is located in the first 64 Kb of memory.
<code>_far</code>	Data is located in the SFR space.
<code>_rom</code>	internal/external ROM. The compiler infers the type qualifier <code>const</code> .
<code>_farrom</code>	ROM data located anywhere in memory.
<code>_nearrom</code>	ROM data located in the first 64 Kb of memory.

Table 3-1: Storage type specifiers

Keywords like `_internal` and `_external`, to specify whether code or data should be located in on-chip memory or external memory are NOT provided. However, code or data stored in on-chip memory can be faster accessed, and access consumes less power. On-chip and external memory access is exactly the same from the perspective of the code generator. Locating code and data in on-chip memory or external memory is solely a problem to be solved by the locator, not by the compiler.

The pragma **renamesect** is used to rename a code or data section. The DELFEE locator language is used to specify how to locate the renamed section. Solving locate problems within the locator language instead of in the C source, gives you the ability to tune an existing application for another M16C derivative without updating the source code.

Functions are by default allocated in Program Memory; the storage specifier may be omitted in that case. Also, function return values cannot be assigned to a storage area.

In addition to static storage specifiers, a static object can be assigned to a fixed memory address using the `_at ()` keyword.

cm16 treats the storage specifier `_rom` type in a special way: `_rom` always implies the type qualifier `const`.

Const Qualifier

The ANSI standard states that the type qualifier `const` can be used to specify 'read-only' objects (or: are not 'lvalues'). An ANSI C compiler may allocate static `const` objects in ROM memory. Allocating constants in ROM is therefore possible. Default are constants allocated in the M16C code space. Constants will be placed in sections with the `INIT` attribute by default. So, **cm16** treats `const` just as a type qualifier, which allows the compiler to check on illegal lvalue use. This is exactly the way it is meant to be used in the ANSI definition.

It is also possible to allocate strings and literals in the ROM code area only. this ROM code area. When you use the **-S** option **cm16** places strings in this ROM code area. The **-T** option places all constants in the ROM code area.

Example:

```
func( i )
const int i;
{
    i++; /* results in error message from cm16 */
}
```

3.2.1.1 NON-VOLATILE RAM

The value of variables located in non-volatile ram are retained when the microcontroller system is reset or is turned off. The first time the microcontroller system is started the non-volatile ram should be initialized, afterwards the values stored in the non-volatile memory should not be reinitialized after a reset. However, if the non-volatile memory is 'damaged' due to battery exchange, application program error, etc., the non-volatile memory should be reinitialized after a reset.

Handling non-volatile memory is a problem that should be solved by the locator and the startup code. The pragma **renamesect** is used to associate a symbolic name with the section that contains the objects located in non-volatile RAM. The DELFEE locator language is used to add additional locator attributes to the section (like `noclear`), and to map this section at the right physical memory addresses. For this reason the TASKING M16C C compiler does not support a keyword like `_persistent` to indicate that a data object should be located in non-volatile memory.

3.2.1.2 STORAGE AND SECTION RELATIONS

The following table shows the resulting assembler section types and attributes for each C storage type:

Storage Type	M16C Section Type / Attribute
<code>_bit</code>	BIT CLEAR/INIT
<code>_bita</code>	BITA CLEAR/INIT
<code>_near</code>	DATA CLEAR/INIT
<code>_far</code>	FDATA CLEAR/INIT
<code>_rom</code>	DATA (small memory model) or FDATA (large memory model) ROMDATA
<code>_farrom</code>	FDATA ROMDATA
<code>_nearrom</code>	DATA ROMDATA

Table 3-2: Section types

Examples using explicit storage types:

```

_near char c;
_rom char text[] = "No smoking";
_far int array[100][4];
_near long l;

```

allocating:

1	byte in RAM for c
11	bytes in ROM for the initialized character array text[]
800	bytes in RAM for array



4 bytes in RAM for 1

The storage type specifiers are treated like any other type specifier (e.g. unsigned). This means the examples above can also be declared (exactly the same):

```
char _near c;
char _rom text[] = "No smoking";
int _far array[100][4];
long _near l;
```

An object must be fully contained in a single storage section. See section 3.17, *Structure Tags* for details.

3.2.2 MEMORY MODELS

cm16 supports two reentrant memory models: small and large. You can select one of these models with the **-M** option.

If no memory model is specified on the command line, **cm16** uses the *small* model because this model generates the most efficient code. The table below illustrates the meaning of each data model.

Model	Data	Constants
Small	_near	_near
Large	_far	_far

Table 3-3: **cm16** memory models

Separate versions of the C and runtime libraries are supplied for all supported data models, avoiding the need for the programmer to recompile or rebuild these when using a particular model.

The value of the predefined preprocessor symbol `_MODEL` represents the memory model selected (**-M** option). This can be very helpful in making conditional C code in one source module, used for different applications in different memory models. *Portable C Code*

The value of `_MODEL` is:

```
small model    's'
large model    'l'
```

Code addresses are presumed to be 20-bit (by default). This corresponds to command-line option **-Jf**. To make code addresses 16-bit, use option **-Jn**.

The predefined preprocessor symbol `_CODEMODEL`, which represents the code model selected with the **-J** option, is set to 'n' or 'f', indicating whether code addresses are assumed to be near or far.

Example:

```
#if _MODEL == 'l' /* large model */  
...  
  
#endif
```

3.2.3 THE `_AT()` ATTRIBUTE

In C-M16C it is possible to place certain variables and functions at absolute addresses. Instead of writing a piece of assembly code, a variable or function can be placed on an absolute address using the `_at()` attribute.

Examples:

```
_near unsigned char Display _at( 0x2000 );  
  
int f( int x ) _at(0x0100)  
{  
    return x+1;  
}
```

The example above creates a variable with the name `Display` at address `0x2000`. In the generated assembly code an absolute section appears in the form 'DATA AT 2000H'; at this address, space is reserved for the variable `Display`.

A number of restrictions are in effect when placing variables on an absolute address:

- Only global variables can be placed on absolute addresses. Parameters of functions, or automatics within functions cannot be placed on an absolute address.

- When declared 'extern', the variable is not allocated by the compiler. When the same variable is allocated within another module but on a different address, the compiler, assembler or linker will not notice.
- When the variable is declared 'static', no public symbol will be generated (normal C behavior).
- Absolute variables cannot overlap each other, declaring two absolute variables on the same address will cause an error generated by the assembler or by the linker. The compiler does not check this.
- Declaring the same absolute variable within two modules will also produce conflicts at link time (except when one of the modules declares the variable 'extern').

3.3 DATA TYPES

All ANSI C data types are supported. In addition to these types, the `_sfrbit`, `_sfrbyte`, and `_bit` types are added. Three types of pointers are recognized. Object size and ranges:

Data Type	Size (in bytes)	Range
<code>_bit</code>	1 bit	0 or 1
<code>_sfrbit</code>	1 bit	0 to 1
<code>signedchar</code>	1	-128 to +127
<code>unsigned char</code>	1	0 to 255U
<code>_sfrbyte</code>	1	0 to 255U
<code>signed short</code>	2	-32768 to +32767
<code>unsigned short</code>	2	0 to 65535U
<code>signed int</code>	2	-32768 to +32767
<code>unsigned int</code>	2	0 to 65535U
<code>_sfrword</code>	2	0 to 65535U
<code>_sfrlong</code>	4	0 to 4294967295UL
<code>signed long</code>	4	-2147483648 to +2147483647
<code>unsigned long</code>	4	0 to 4294967295UL
<code>float</code>	4	+/- 1.176E-38 to +/- 3.402E+38
<code>double</code>	8	+/- 2.225E-308 to +/- 1.798E+308
<code>enum</code>	2	0 to 65535U
<code>_near pointer</code>	2	0 to 65535
<code>_far pointer</code>	2 / 4	0 to 1 megabyte

Table 3-4: Data types

- `_bit`, `char`, `_sfrbyte`, `_sfrword`, `_sfrlong`, `short`, `int` and `long` are all integral types, supporting all implicit (automatic) conversions.
- **cm16** generates instructions using (8 bit) character arithmetic, when it is correct to evaluate a character expression this way. This results in a higher code density compared with integer arithmetic. A special section *Character Arithmetic* provides details.

- the M16C convention is used, storing variables with the least significant part at the lower memory address (Little Endian).
- float is implemented in little endian IEEE 32-bit single precision format.
- double is implemented in little endian IEEE 64-bit double precision format.

3.3.1 ANSI C TYPE CONVERSIONS

According to the ANSI C X3.159-1989 standard, a character, a short integer, an integer bit field (either signed or unsigned), or an object of enumeration type, may be used in an expression wherever an integer may be used. If a `signed int` can represent all the values of the original type, then the value is converted to `signed int`; otherwise the value will be converted to `unsigned int`. This process is called *integral promotion*.

Integral promotion is also performed on function pointers and function parameters of integral types using the old-style declaration. To avoid problems with implicit type conversions, you are advised to use function prototypes.

Many operators cause conversions and yield result types in a similar way. The effect is to bring operands into a common type, which is also the type of the result. This pattern is called the *usual arithmetic conversions*.

Integral promotions are performed on both operands; then, if either operand is `unsigned long`, the other is converted to `unsigned long`.

Otherwise, if one operand is `long` and the other is `unsigned int`, the effect depends on whether a `long` can represent all values of an `unsigned int`; if so, the `unsigned int` operand is converted to `long`; if not, both are converted to `unsigned long`.

Otherwise, if one operand is `long`, the other is converted to `long`.

Otherwise, if either operand is `unsigned int`, the other is converted to `unsigned int`.

Otherwise, both operands have type `int`.



See also the section *Character Arithmetic*.



Sometimes surprising results may occur, for example when unsigned char is promoted to int. You can always use explicit casting to obtain the type required. The following example makes this clear:

```
static unsigned char a=0xFF, b, c;

void f()
{
    b=~a;
    if ( b == ~a )
    {
        /* This code is never reached because,
        * 0x0000 is compared to 0xFF00.
        * The compiler converts character 'a' to
        * an int before applying the ~ operator
        */
        ...
    }

    c=a+1;
    while( c != a+1 )
    {
        /* This loop never stops because,
        * 0x0000 is compared to 0x0100.
        * The compiler evaluates 'a+1' as an
        * integer expression. As a side effect,
        * the comparison will also be an integer
        * operation
        */
        ...
    }
}
```

To overcome this 'unwanted' behavior use an explicit cast:

```
static unsigned char a=0xFF, b, c;

void f()
{
    b=~a;
    if ( b == (unsigned char)~a )
    {
        /* This code is always reached */
        ...
    }

    c=a+1;
    while( c != (unsigned char)(a+1) )
    {
        /* This code is never reached */
        ...
    }
}
```

Keep in mind that the arithmetic conversions apply to multiplications also:

```
static int      h, i, j;
static long     k, l, m;

/* In C the following rules apply:
 *      int  * int      result: int
 *      long * long     result: long
 *
 *      NOT int * int    result: long
 */
```

```

void f()
{
    h = i * j;      /* int * int = int */
    k = l * m;      /* long * long = long */

    l = i * j;      /* int * int = int, afterwards
                    * promoted (sign or zero
                    * extended) to long
                    */
    l = (long) i * j; /* long * long = long */
    l = (long)(i * j); /* int * int = int,
                      * afterwards casted to long
                      */
}

```

3.3.2 CHARACTER ARITHMETIC

cm16 generates code using 8 bit (character) arithmetic as long as the result of the expression is exactly the same as if it was evaluated in integer arithmetic. This must be done, because ANSI does not define character arithmetic and character constants. Although the M16C performs 16-bit operation as fast as 8-bit operations, the overhead caused by the integral promotions is suppressed.

So it is recommended to use character variables in expressions, because it saves data space for allocation, and often results in a higher code density. You can always force to use character arithmetic with a character cast.

The following examples clarify when integer arithmetic is used and when character arithmetic:

```

char  a, b, c, d;
int   i;

void
main()
{
    c = a + b;          /* character arithmetic */
    i = a + b;          /* integer arithmetic */
    i = (char)(a + b);  /* character arithmetic */

    c = a / d;          /* character arithmetic */
    c = (a + b) / d;    /* integer arithmetic */
    c = ((char)(a + b)) / d; /* character arithmetic */

    c = a >> d;          /* character arithmetic */
    c = (a + b) >> d;   /* integer arithmetic */
}

```

```

if ( a > b )                /* character arithmetic */
    c = d;
if ( (a + b) > c )          /* integer arithmetic  */
    c = d;
}

```

You can disable character arithmetic with the **-AC** ccommand line option.

3.3.3 THE _bit TYPE

The `_bit` type is used to define scalars in the M16C bit-addressable area and for the return type of functions. A struct containing bit fields cannot be used for this purpose, for example because the struct is aligned at a byte boundary.

The following rules apply to `_bit` type variables:

1. A `_bit` type variable is always unsigned.
2. A `_bit` type variable can be exchanged with all other integral type variables. The compiler generates the correct conversion.

A `_bit` type variable is like a boolean. Therefore, converting an `int` type variable to a `_bit` type variable does **not** mean the `_bit` type variable is the least significant bit of the `int` type variable. It is 1 (true) if the `int` type variable is not equal to 0, and 0 (false) if the `int` type variable is 0. In C:

```
bit_variable = int_variable;
```

can be seen as:

```
bit_variable = int_variable ? 1 : 0;
```

3. A `_bit` type variable is **not** allowed as a function parameter of a function, or as an automatic variable.
4. A function may have return type `_bit`. However, the next rule may not be violated.
5. Evaluation of a complex `_bit` expression (using non `_bit` types or `_bit` return type of a function) is not recursive or reentrant, because the compiler might need temporary static bit space.
6. A `_bit` typed expression is not allowed as switch expression.
7. The `sizeof` of a `_bit` type is 1.

3.3.4 SPECIAL FUNCTION REGISTERS

The `_sfrbyte`, `_sfrword`, `_sfrbit` and `_sfrlong` keywords allow direct access to all special function registers as if they were C variables. These special function registers can be used the same way as any other integral data type, including all automatic conversions.

The `_sfrbyte`, `_sfrword`, `_sfrbit` and `_sfrlong` keywords are handled as `volatile unsigned` variables.

The notation is as follows:

```
_sfrbyte  name _at( address );
```

or

```
_sfrword  name _at( address );
```

or

```
_sfrlong  name _at( address );
```

or

```
_sfrbit   name _at( bitaddr );
```

or

```
_sfrbit   name _atbit( address, bitnumber );
```

Where *name* occurs replace with the name of the SFR you want to specify. *address* is the bit or byte address of the SFR. Because these registers are placed in the sfr-area of the processor, the compiler will not allocate any storage space.

Note, that the words 'sfrbyte', 'sfrword', 'sfrbit' and 'sfrlong' are not reserved words for **cm16**. Therefore, these words can be used as an identifier. **cm16** does not generate symbolic debugging information for special function registers because they are already known by the debugger.

Because the special function registers are dealing with I/O, it is incorrect optimize away the access to them. Therefore, **cm16** deals with the special function registers as if they were declared with the `volatile` qualifier.

3.4 FUNCTION PARAMETERS

cm16 supports (ANSI) prototyping of function parameters. Therefore, **cm16** allows passing parameters of type `char`, **without** converting these parameters to `int` type. This results into higher code density, higher execution speed and less RAM data space needed for parameter passing.

For example, in the following C code:

```
void func( char number, long value );
int  printf( char *format, ... );

void
main(void)
{
    int  i;
    char c;

    func( c, i );
    printf( "c=%d, i=%d\n", c, i );
}
```

the code generator uses the prototype of `func()` and:

- passes `c` as a byte
- promotes `i` to `long` before passing it as a `long`

However, the code generator does not know anything of the `printf()` arguments, because this function is declared with a variable argument list. If there is no prototype (as with the old style K & R functions), the compiler promotes both `char` type parameters to `int` type, the same way an automatic conversion is done in an assignment of a `char` type variable to an `int` type variable. So, with the `printf()` call the code generator:

- promotes `c` to `int` before passing it as `int`
- passes `i` as `int`

3.5 PARAMETER PASSING AND FUNCTION RETURN

A lot of execution time of an application is spent transferring parameters between functions. The fastest parameter transport is via registers. If not enough registers are available, some parameters are passed via registers, the other parameters are passed over the stack. See the table below.

Parameter order	Parameter type			
	char	int, 2-byte pointer, structure <= 2-byte	long, float, 4-byte pointer, structure <= 4-byte	double
1	R0L	R0	R0;A0	
2	R0H	A0		
3	A0			

Table 3-5: Register usage for parameter passing

Example with four register arguments:

```
func1( char a, long b, int c, char d )
```

- a (first parameter) is passed in register R0L.
- b (second parameter) is passed on the stack.
- c (third parameter) is passed in register A0.
- d (fourth parameter) is passed in register R0H.

All parameters of a variable argument list function are always passed over the stack. Parameters are pushed in reverse order, so all ANSI C macros defined in `stdarg.h` can be applied.

Example with variable argument function:

```
_printf( char *format, ... )
```

- all parameters (including `format`) are passed via the stack.

Function return is performed as follows:

Return type	Register(s)
bit	0,R0
char	R0L
short/int	R0



Return type	Register(s)
long	R0;A0
float	R0;A0
double	R3;R2;R1;R0
structure	Stack temporary (ad- dress passed by caller in R0)
2-byte pointer	A0
4-byte pointer	R0;A0

Table 3-6: Register usage for function return types

3.6 AUTOMATIC VARIABLES

In C the `register` type qualifier is a means for the programmer to tell the compiler that the variable will be used very often. So the code generator must try to reserve a register for this variable and use this register instead of the stack location of this automatic variable. The compiler uses an efficient allocation scheme to decide which of the automatic objects and parameter objects that are used the most, are to be allocated within registers. Because of this allocation scheme **cm16** ignores the `register` keyword.

Automatic variables are allocated on the user stack and are addressed using the indexed addressing mode.

3.7 INITIALIZED VARIABLES

Non-automatic initialized variables use the same amount of space in both ROM and RAM (for all possible RAM memory spaces). This is because the initializers are stored in ROM and copied to RAM at start-up. This is completely transparent to the user. The only exception is an initialized variable residing in ROM, by means of the `_rom` storage type specifier or a `const` qualifier used with `_near`, `_far`:

Examples (*small memory model*) :

```
int          i = 100;          /* 2 bytes in rom and
                                2 bytes in DATA */
_rom int     j = 3;            /* 2 bytes in CODE ROMDATA */
_rom char    a[] = "HELP";     /* 5 bytes in CODE ROMDATA */
_near char    c = 'a';         /* 1 byte in DATA ROMDATA and
                                1 byte in DATA */
```

3.8 TYPE QUALIFIER VOLATILE

You can use the `volatile` type qualifier when modifications on the object have undesired side effects when they are performed in the regular way. Memory locations may not be updated because of compiler optimizations, which attempt to save a memory write by keeping the value in a register. When a variable is declared with the `volatile` qualifier, the compiler disables such optimizations. Volatile variables are located in a section of which the `NOCLEAR` attribute is set.

The ANSI report describes that the updates of volatile objects follow the rules of the abstract machine (the target processor) and thus access to a volatile object becomes implementation defined.

Example:

```
const volatile int real_time_clock _at(0x1234);

/*  define the real time clock register;
    it is read-only (const);
    read operations must access the real memory
    location (volatile)
*/
```

3.9 STRINGS

In this section the word 'strings' means the separate occurrence of a string in a C program. So, array variables initialized with strings are just initialized character arrays, which can be allocated in any memory type, and are not considered as 'strings'. See section *Initialized Variables* for more information on this topic.

Strings and literals in a C source program, which are not used to initialize an array, have static storage duration. The ANSI standard does not require that these strings be modifiable. Allocating the strings in ROM is therefore possible. By default strings are allocated in the M16C code space.

It is also possible to allocate strings and literals in the ROM code space only. When you use the **-S** option **cm16** will place strings in this ROM code area.

3.10 POINTERS

Some objects have two types: a 'logical' type and a storage type. For example, a function is residing in ROM (storage type), but the logical type is the return type of this function. The most obvious C type having different storage and logical type is a pointer. For example:

```
_rom char *_near p; /* pointer residing in DATA,
                    pointing to ROM */
```

means `p` has storage type `_near` (allocated in direct addressable RAM), but has logical type 'character in target memory space CODE ROMDATA'. The memory type specifier used left to the `*`, specifies the target memory of the pointer, the memory specifier used right to the `*`, specifies the storage memory of the pointer.

The memory type specifiers are treated like any other type specifier (like unsigned). This means the pointer above can also be declared (exactly the same) using:

```
char _rom *_near p; /* pointer residing in DATA,
                    pointing to ROM */
```

If the target memory and storage memory of a pointer are not explicitly declared, **cm16** uses the default of the memory model selected. For example, in the small model, the declaration:

```
char      *p;
```

is exactly the same as:

```
_near char * _near p;
```

cm16 recognizes two types of pointers `_near` and `_far`. `_near` pointers are 16 bits and can point only to locations in the lowest 64K bytes of memory. `_far` pointers are 32 bits and can point anywhere in memory.

In pointer arithmetic **cm16** checks, besides the type of each pointer, also the target memory of the pointers, which should be the same. You can always convert from 2-byte to 4-byte pointer. When converting to a smaller pointer size, the compiler will warn you for potential loss of information.

3.11 INLINE C FUNCTIONS

The `_inline` keyword is used to signal the compiler to inline the function body instead of calling the function. An inline function must be defined in the same source file before it is 'called'. When an inline function has to be called in several source files, each file must include the definition of the inline function. Usually this is done by defining the inline function in a header file.

Not using a function which is defined as an `_inline` function does not produce any code.

Example (t.c):

```
int  w,x,y,z;

_inline int
add( int a, int b )
{
    return( a + b );
}

void
main( void )
{
    w = add( 1, 2 );
    z = add( x, y );
}
```

No specific debug information is generated for inline functions. The debugger cannot step-into an inline function, it considers the inline function as one HLL source line.

The pragmas `asm` and `endasm` are allowed in inline functions. This makes it possible to define inline assembly functions. See also the section *Inline Assembly* in this chapter.

The generated code is:

```
; t.c          12          w = add( 1, 2 );
                MOV.W    #3,_w
; t.c          13          z = add( x, y );
                MOV.W    _x,R0
                ADD.W    _y,R0
                MOV.W    R0,_z
```

3.12 INLINE ASSEMBLY

cm16 supports inline assembly using the following pragmas:

- #pragma asm** Insert assembly text following this pragma.
- #pragma asm_noflush** As #pragma asm, but the peephole optimizer does not flush the code buffer.
- #pragma endasm** Switch back to the C language.



C modules containing inline assembly are not portable and are very hard to prototype in other environments.

The peephole optimizer in the compiler maintains a code buffer for optimizing sequences of assembly instructions before they are written in the output file. The compiler does not interpret the text of inline assembly. It passes inline assembly lines directly to the output file. To prevent that instructions in the peephole buffer, which belong to C code before the inline assembly lines, will be written in the output file after the inline assembly text, the compiler flushes the instruction buffer in the peephole optimizer. All instructions in the buffer are written to the output file. If this behavior is not desired the pragma **asm_noflush** starts inline assembly without flushing the code buffer.



See also the section *Assembly Language Interfacing* in the chapter *Run Time Environment*.

3.13 CALLING ASSEMBLY FUNCTIONS

For a fixed register-based interface between C and assembly functions the function qualifier `_asmfunc` is available. This function qualifier can be used for a prototype of an assembly function to be called from C or for a function definition of a C function to be called from assembly.

Example:

```

/* prototype of assembly function */
extern _asmfunc int
special_out( int port, long config, int value );

void main( void )
{
    long cfg;
    int y;
    ...
    if( special_out( 1, cfg, y ) ) /* call assembly
                                   function */
    {
        ...
    }
    ...
}
```


The number of arguments that can be passed is limited by the number of available registers. If too many arguments are used, the compiler will issue an error. Passing some parameters over the stack is not an option, because the interface would become complex and the `_asmfunc` qualifier loses its value, i.e., creating a simple interface between C and assembly functions.

3.14 INTRINSIC FUNCTIONS

When you want to use some specific M16C instructions, that have no equivalence in C, you would be forced to write assembly routines to perform these tasks. However, **cm16** offers a way of handling this in C. Therefore, **cm16** has a number of built-in functions, which are implemented as intrinsic functions.

To the programmer intrinsic functions appear as normal C functions, but the difference is that they are interpreted by the code generator, so that more efficient code may be generated. Several pre-declared functions are available to generate inline assembly code for the intrinsic function (call). This avoids the overhead that is normally introduced by parameter passing and context saving before executing the called function.

The names of the intrinsic functions all have a leading underscore, because the ANSI specification states that public C names starting with an underscore are implementation defined.

The advantages of using intrinsic functions, compared with in-line assembly (pragma asm/endasm) are:

- the possibility to use simulation routines or stub functions by a host compiler, to replace the inline assembly code generated by **cm16**
- C level variables can be accessed
- the compiler chooses to generate the most efficient code to access C variables
- intrinsic code is optimized, except for `_nop()`

The following intrinsic functions are implemented:

Function	Comment
signed char _absb (signed char)	
int _absw (int)	
void _brk (void)	
char _dadcb (char, char)	
int _dadcw (int, int)	
char _daddb (char, char)	
int _daddw (int, int)	
int _divb (int, char)	Returns quotient and remainder
char _divb_q (int, char)	Returns quotient

Function	Comment
char _divb_r (int, char)	Returns remainder
long int _divw (long int, int)	Returns quotient and remainder
int _divw_q (long int, int)	Returns quotient
int _divw_r (long int, int)	Returns remainder
int _divub (int, char)	Returns quotient and remainder
char _divub_q (int, char)	Returns quotient
char _divub_r (int, char)	Returns remainder
long int _divuw (long int, int)	Returns quotient and remainder
int _divuw_q (long int, int)	Returns quotient
int _divuw_r (long int, int)	Returns remainder
int _divxb (int, char)	Returns quotient and remainder
char _divxb_q (int, char)	Returns quotient
char _divxb_r (int, char)	Returns remainder
long int _divxw (long int, int)	Returns quotient and remainder
int _divxw_q (long int, int)	Returns quotient
int _divxw_r (long int, int)	Returns remainder
char _dsbbb (char, char)	
int _dsbbw (int, int)	
char _dsubb (char, char)	
int _dsubw (int, int)	
int _enter (int)	
void _exitd (void)	
int _fclr (int)	Use 0 to 7 or _C, _D, _Z, _S, _B, _O, _I, _U to clear a bit in the flag register.
int _fset (int)	Use 0 to 7 or _C, _D, _Z, _S, _B, _O, _I, _U to set a bit in the flag register.
int _int (int)	
void _into (void)	
int _ldc_fb (int)	
int _ldc_sb (int)	
int _ldc_sp (int)	

Function	Comment
int _ldc_isp (int)	
int _ldc_flg (int)	
int _ldintb (int)	
int _ldc_intbh (int)	
int _ldc_intbl (int)	
int _ldipl (int)	
void _nop (void)	
int _popc (int)	The operand is the register as encoded in the opcode.
int _popm (int)	The operand is the register mask as encoded in the opcode.
int _pushc (int)	The operand is the register as encoded in the opcode.
int _pushm (int)	The operand is the register mask as encoded in the opcode.
void _reit (void)	
int _rmpab (_near char *source, _near char *dest, int count)	
long int _rmpaw (_near char *source, _near char *dest, int count)	
char _rotb (signed char, char)	
int _rotw (signed char, int)	
void _rts (void)	
char _shab (signed char, char)	
int _shaw (signed char, int)	
long int _shal (signed char, long int)	
char _shlb (signed char, char)	
int _shlw (signed char, int)	
long int _shll (signed char, long int)	
int _smovbb (_far char *source, _near char *dest, int count)	

Function	Comment
int _smovbw (_far int *source, _near int *dest, int count)	
int _smovfb (_far char *source, _near char *dest, int count)	
int _smovfw (_far int *source, _near int *dest, int count)	
char _sstrb (char, _near char *, int)	
int _sstrw (int, _near int *, int)	
int _stc_fb (void)	
int _stc_sb (void)	
int _stc_sp (void)	
int _stc_isp (void)	
int _stc_flg void()	
int _stc_intbh (void)	
int _stc_intbl (void)	
void _und (void)	
void _wait (void)	

Table 3-7: Intrinsic functions

3.15 INTERRUPTS

The M16C C language introduces three new reserved words: `_interrupt`, `_hw_interrupt` and `_bankswitch`, which can be seen as special type qualifiers, only allowed with function declarations. A function can be declared to serve as an interrupt service routine. Interrupt functions cannot return anything and must have a **void** argument type list. `_interrupt` is used to define functions for the variable vector table. The vector argument must be between 0 and 63. `_hw_interrupt` defines functions for the fixed vector table. Its vector argument must be between 0 and 8. `_hw_interrupt` also initializes the appropriate entry in the fixed vector table. For example, in:

```
_interrupt(vector) _bankswitch void
_isr(void)
{
    ...
};
```

The `_interrupt` function qualifier takes one argument, *vector*, that defines the interrupt vector number, i.e., the vector address.

`_bankswitch` causes register bank 1 to be used for the function.

Because the vector is filled by the compiler (unless disabled by the `-v` option), the interrupt number must be specified. To find out which interrupt number should be used, see the section *Interrupt Functions* in chapter *Run-time Environment*.

Some interrupts are reserved and handled or used by the compiler (run-time library) like:

- Hardware reset.
- Stack overflow and underflow.

`_special` defines functions to be called using the M16C JSRS instruction. The syntax is:

```
_special(number) void
filename (args)
```

number must be between 18 and 255.

3.16 SAFER C

Based upon the 'MISRA guidelines for the application of C language in vehicle based software', the TASKING Safer C technology offers enhanced compiler error checking that will guide the programmer in writing better, more coherent and intrinsically safer applications. Through this configurable system of enhanced C language error checking, the use of error-prone C constructs can be prevented. A predefined configuration for compliance with the 'required rules' described in the MISRA guidelines is selectable through a single click in the EDE|Safer C Options menu. A custom set of applicable Safer C rules can be easily configured using the same menu. It is also possible to have a project team work with a Safer C configuration common to the whole project. In this case the Safer C configuration can be read from an external settings file. This too, is easily selected through the EDE|Safer C Options menu. In order to provide proof that installed company Safer C requirements have in fact been adhered to throughout the entire project, the M16C Linker/Locator can generate a Safer C Quality Assurance report. This report lists the various modules in the project with the respective Safer C settings under which these have been compiled.

Unfortunately it has not been possible to implement support for all 127 rules described in the MISRA guidelines. The reason for this is that a number of rules are beyond the scope of what can be checked in a C compiler environment. These unsupported rules are visible in the EDE|Safer C Options menu dialog boxes, but cannot be selected (grayed out).

MISRA is a registered trademark of MIRA held on behalf of the Motor Industry Software Reliability Association.

Enabling Safer C

From the command line Safer C can be enabled by the following compiler option:

```
-safern,n,...
```

where n specifies the rule(s) which must be checked.

Error Messages

In case a Safer C rule is violated, an error message will be generated e.g.:

E 209: Safer C rule 9 violation: comments shall not be nested.

See Appendix B *Safer C* for the supported and unsupported Safer C rules.

3.17 STRUCTURE TAGS

A tag declaration is intended to specify the lay-out of a structure or union. If a memory type is specified, it is considered to be part of the declarator. A tag name itself, nor its members can be bound to any storage area, although members having type "... pointer to" do require one. A tag may then be used to declare objects of that type, and may allocate them in different memories (if that declaration is in the same scope). The following example illustrates this constraint.

```
struct S {
    _near int i; /* referring to storage: not correct */
    _far char *p; /* used to specify target memory: correct */
};
```

In the example above **cm16** ignores the erroneous `_near` storage specifier (without displaying a warning message).

3.18 TYPEDEF

Typedef declarations follow the same scope rules as any declared object. Typedef names may be (re-)declared in inner blocks but not at the parameter level. However, in typedef declarations, memory specifiers are allowed. A typedef declaration should at least contain one type specifier.

Examples:

```
typedef _near int NEARINT; /* storage type _near: OK */
typedef int _near *PTR; /* logical type _near
                        storage type 'default' */
```


3.19 SWITCH STATEMENT

cm16 supports two ways of code generation for a switch statement: a jump chain (linear switch) or a jump table.

A jump chain is comparable with an if/else-if/else-if/else construction. A jump table is a table filled with case label entry points for each possible switch value. The switch argument is used as an index in the jump table. An indirect jump is performed to the indexed case label entry point.

By default, the compiler will try to use the switch method which uses the least space in ROM.

It is obvious that, especially for large switch statements, the jump table approach executes faster than the jump chain approach. Also, the jump table has a predictable behavior in execution speed. No matter the switch argument, every case is reached in the same execution time.

With a small number of cases, the jump chain method can be faster in execution and shorter in size.

The compiler chosen switch method can be overruled by using one of the following option combinations:

```
-OT -OW /* force jump chain code */
-Ot      /* force jump table code */
-OT -Ow  /* let the compiler decide
           the switch method used */
```

The last one is also the default of the compiler. Using a pragma cannot overrule the restrictions as described earlier.



By default, jump tables contain 2-byte address offsets, and the JMPL.W instruction is void. If the function is very large, you may need to use the -Zw option to use 3-byte absolute addresses with the JMPL.A instruction.

3.20 PORTABLE C CODE

If you are developing C code for the M16C using **cm16**, you might want to test some code on the host you are working on, using a C compiler for that host. Therefore, we deliver the include file `cm16.h`. This header file checks if `_CM16C` is defined (**cm16** only), and redefines the storage type specifiers if it is not defined.

When using this include file, you are able to use the storage type specifiers (when needed) and yet write 'portable C code'.

Furthermore an adapted prototype of each C built-in function is present, because these functions are not known by another ANSI compiler. If you use these functions, you should write them in C, performing the same job as the M16C and link these functions with your application for simulation purposes.

For compatibility with existing C-51 programs, the file `c51.h` is delivered, which just includes `cm16.h`.

3.21 HOW TO PROGRAM SMART

If you want to get the best code out of **cm16**, the following guidelines should be kept in mind:

1. Always use function prototyping. So, `char` variables can be passed as `char` without being promoted to `int`.
2. If you are using the large model (because it is not possible to use a smaller model), try to declare the most frequently used variables (static) with storage type `_near`. If you want your code to remain portable, you can use the `register` keyword.

We recommend to use the smallest model (small before large) that best fits your application and explicitly declare big data items as `_far`. You can make your own C functions to access these far data objects. If you want to use the Standard C library functions on far data objects, you have to use the large model.

3. Try to use the unsigned qualifier as much as possible (e.g. `for (i = 0; i < 500; i++)`), because unsigned comparisons require less code than signed comparisons.

4. Try to use the smallest data type as possible: bit for boolean usage (flags), character for small loops and so on. See also the sections *Character Arithmetic* and *The _bit Type*.
5. If execution speed is important (e.g. interrupt functions and time consuming loops), you should use the **-OS** option or **#pragma optimize S**.

3.22 SOME EXAMPLES OF COMPLEX DECLARATORS

Because **cm16** has some extensions to support the various memory types of the M16C processor family, declarations of objects may need some explanation.

First of all, declaration of simple objects is done exactly the same way as in standard C.

For example:

```
char c;
int i;
long l;
```

When programming portable C code, declaration of pointers is also standard.

For example:

```
char *pc;
int *pi;
long *pl;
```

However, for code density it may be desired to place an object in another memory area, this can be done by preceding the object type by the requested data area specifier.

For example:

```
_near char nc;
_near int ni;
_far long fl;
```

also correct is :

```
char _near nc;
int _near ni;
long _far fl;
```

Now, pointers to another area than the default (specified by the memory model, see the section *Detailed description of the Compiler Options*) are declared as follows:

<code>_near char * pnc;</code>	Pointer resides in default memory, points to a character in near.
<code>_near int * pni;</code>	Pointer resides in default memory, points to an integer in near.
<code>_far long * pfl;</code>	Pointer resides in default memory, points to a long in far.

Even more difficult, these pointers may be placed in some other data area than the default.

For example:

<code>_near char * _near npnc;</code>	Pointer resides in near, points to a character in near.
<code>_near int * _near npni;</code>	Pointer resides in near, points to an integer in near.
<code>_far long * _near fppl;</code>	Pointer resides in near, points to a long in far.



Using objects located in `_near` always produces less code than using objects in `_far`. So the smallest code size (and often the fastest execution speed) can be achieved by placing as many objects as possible in `_near`. When it is not possible to place all objects in internal RAM, select the objects which are most referenced in the code.

Some examples of complex declarators are given below.

```
_near char c;
_near char * _far p = &c;
_near char * _far * pp = &p;
_near char * _far * * _near ppp = &pp;
```

Now `ppp` is a pointer located in `near`, points to a pointer in default memory, this points to a pointer in `far`, which is a pointer to a character in `near`.

```
int _far * func( void );  
int _far (* _near fp)( void ) = func;
```

Now `fp` is a pointer located in `near`, points to a function with no arguments, returning a pointer to an integer in `far`.

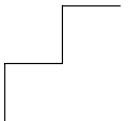
CHAPTER

4

COMPILER USE



TASKING



4

CHAPTER

4.1 CONTROL PROGRAM

The control program **ccm16** facilitates the invocation of the various components of the M16C tool chain, from a single command line. The control program accepts source files and options on the command line in random order.

The invocation syntax of the control program is:

```
ccm16 [ option ] ... [ control ] ... [ file ] ... ] ...
```

Options are preceded by a '-' (minus sign). The input *file* can have one of the extensions explained below.



When you use a **UNIX** shell (Bourne shell, C-shell), arguments containing special characters (such as '(' and '?') must be enclosed with " " or escaped. The -? option (in the C-shell) becomes: "-?" or -\?.

The control program recognizes the following argument types:

- Arguments starting with a '-' character are options. Some options are interpreted by the control program itself; other options are passed to those programs in the tool chain that accept the option.
- Arguments with '(' or arguments without a '.' character are interpreted as assembler controls and are passed to the assembler.
- Arguments with a .cc, .cxx or .cpp suffix are interpreted as C++ source programs and are passed to the C++ compiler.
- Arguments with a .c suffix are interpreted as C source programs and are passed to the compiler.
- Arguments with a .asm suffix are interpreted as assembly source files which have to be preprocessed and passed to the assembler.
- Arguments with a .src suffix are interpreted as compiled assembly source files. They are directly passed to the assembler.
- Arguments with a .out suffix are interpreted as linked object files and are passed to the locator. The locator accepts only one .out file in the invocation.
- Arguments with a .dsc suffix are treated as locator command files. If there is a file with extension .dsc on the command line, the control program assumes a locate phase has to be added. If there is no file with extension .dsc, the control program stops after linking (unless it has been directed to stop in an earlier phase)
- Everything else is considered an object file and is passed to the linker.

Normally, a control program tries to compile and assemble all source files to object files, followed by a link and locate phase which produces an absolute output file. There are however, options to suppress the assembler, linker or locator stage. The control program produces unique filenames for intermediate steps in the compilation process, which are removed afterwards. If the compiler and assembler are called subsequently, the control program prevents preprocessing of the compiler generated assembly file. Normally, assembly input files are preprocessed first.

The following options are interpreted by the control program:

Option	Description
-? or none	Display invocation syntax
-Ccpu	Use special function register definitions for <i>cpu</i>
-Ml	Large memory model
-Ms	Small memory model
-V	Display version header only
-Waaarg	Pass argument directly to the assembler
-Wcarg	Pass argument directly to the C compiler
-Wcparg	Pass argument directly to the C++ compiler
-Wplarg	Pass argument directly to the C++ pre-linker
-Wlkarg	Pass argument directly to the linker
-Wlcarg	Pass argument directly to the locator
-c++	Associate .c files with the c++ compiler
-c	Do not link: stop at .obj
-cc	Do not compile: stop at .c
-cl	Do not locate: stop at .out
-cs	Do not assemble: compile C files to .src and stop
-f file	Read arguments from <i>file</i> ("-" denotes standard input)
-g[f l n]...	Enable symbolic debug information (unless -gn is used)
-ieee	Set locator output file format to IEEE-695 (default)
-ihex	Set locator output file format to Intel Hex
-nolib	Do not link with the standard libraries
-o file	Specify the output file
-srec	Set locator output file format to Motorola S-records
-tiof	Set locator output file format to TIOF-695
-tmp	Keep intermediate files

Option	Description
-v	Show command invocations
-v0	Show command invocations, but do not start them
-wc++	Enable C and assembler warnings for C++ files.

Table 4-1: Control program options

4.1.1 DETAILED DESCRIPTION OF THE CONTROL PROGRAM OPTIONS

- ? Display a short explanation of options at stdout.
 - Ccpu Use special function register definitions for *cpu*.
 - M{s | l} Specify the memory model to be used:
 - small (s)
 - large (l)
 - V The copyright header containing the version number is displayed, after which the control program terminates.
 - Warg
 - Wcarg
 - Wcparg
 - Wplarg
 - Wlkarg
 - Wlcarg
- With these options you can pass a command line argument directly to the assembler (-Wa), C compiler (-Wc), C++ compiler (-Wcp), C++ pre-linker (-Wpl), linker (-Wlk) or locator (-Wlc). These options may be used to pass some options that are not recognized by the control program, to the appropriate program. The argument may be either directly appended to the option, or follow the option as a separate argument of the control program.
- c++ Specify that files with the extension .c are considered to be C++ files instead of C files. So, the C++ compiler is called prior to the C compiler. This option also forces the linker to link C++ libraries.

-c
-cc
-cl
-cs

Normally, the control program invokes all stages to build an absolute file from the given input files. With these options it is possible to skip the C compiler, assembler, linker or locator stage. With the **-cc** option the control program stops after compilation of the C++ files and retains the resulting `.c` files. With the **-cs** option the control program stops after the compilation of the C source files (`.c`) and after preprocessing the assembly source files (`.asm`), and retains the resulting `.src` files. With the **-c** option the control program stops after the assembler, with as output one or more object files (`.obj`). With the **-cl** option the control program stops after the link stage, with as output a linker object file (`.out`).

-f file

Read command line arguments from *file*. The filename `"-"` may be used to denote standard input. To get around the limits on the size of the command line, it is possible to use command files. These command files contain the options that could not be part of the real command line. Command files can also be generated on the fly, for example by the make utility.

Some simple rules apply to the format of the command file:

1. It is possible to have multiple arguments on the same line in the command file.
2. To include whitespace in the argument, surround the argument with either single or double quotes.
3. If single or double quotes are to be used inside a quoted argument, we have to go by the following rules:
 - a. If the embedded quotes are only single or double quotes, use the opposite quote around the argument. Thus, if a argument should contain a double quote, surround the argument with single quotes.
 - b. If both types of quotes are used, we have to split the argument in such a way that each embedded quote is surrounded by the opposite type of quote.

Example:

```
"This has a single quote ' embedded"
```

or

```
'This has a double quote " embedded'
```

or

```
'This has a double quote " and \  
a single quote ''' embedded"
```

4. Some operating systems impose limits on the length of lines within a text file. To circumvent this limitation it is possible to use continuation lines. These lines end with a backslash and newline. In a quoted argument, continuation lines will be appended without stripping any whitespace on the next line. For non-quoted arguments, all whitespace on the next line will be stripped.

Example:

```
"This is a continuation \  
line"  
-> "This is a continuation line"
```

```
control(file1(mode,type),\  
        file2(type))  
->  
control(file1(mode,type),file2(type))
```

5. It is possible to nest command line files up to 25 levels.

-g[f|l|n]... Enable symbolic debug information (unless **-gn** used). With **-gn** you disable all debug, including type checking. With **-gl** you disable lifetime information for all types. If you use **-gf**, high level language type information is also emitted for types which are not referenced by variables. Therefore, this sub-option is not recommended.

- ieee**
-ihex
-srec
-tiof
- With these options you can specify the locator output format of the absolute file. The output file can be an IEEE-695 file (.abs), Intel Hex file (.hex), Motorola S-record file (.sre) or TIOF-695 file (.abs). The default output is IEEE-695 (.abs).
- nolib**
- With this option the control program does not supply the standard libraries to the linker. Normally the control program supplies the default C and run-time libraries to the linker. Which libraries are needed is derived from the compiler options.
- o *file***
- Normally, this option is passed to the locator to specify the output file name. When you use the **-cl** option to suppress the locating phase, the **-o** option is passed to the linker. When you use the **-c** option to suppress the linking phase, the **-o** option is passed to the assembler, provided that only one source file is specified. When you use the **-cs** option to suppress the assembly phase, the **-o** option is passed to the compiler. The argument may be either directly appended to the option, or follow the option as a separate argument of the control program.
- tmp**
- With this option the control program creates intermediate files in the current directory. They are not removed automatically. Normally, the control program generates temporary files for intermediate translation results, such as compiler generated assembly files, object files and the linker output file. If the next phase in the translation process completes successfully, these intermediate files will be removed.
- v**
- When you use the **-v** option, the invocations of the individual programs are displayed on standard output, preceded by a '+' character.
- v0**
- This option has the same effect as the **-v** option, with the exception that only the invocations are displayed, but the programs are not started.

- wc++** Enable C and assembler warnings for C++ files. The assembler and C compiler may generate warnings on C output of the C++ compiler. By default these warnings are suppressed.

4.1.2 ENVIRONMENT VARIABLES

The control program uses the following environment variables:

- TMPDIR** This variable may be used to specify a directory, which the control program should use to create temporary files. When this environment variable is not set, temporary files are created in the directory `"/tmp"` on UNIX systems, and in the current directory on other operating systems.
- CCM16COPT** This environment variable may be used to pass extra options and/or arguments to each invocation of the control program **ccm16**. The control program processes the arguments from this variable before the command line arguments.
- CCM16CBIN** When this variable is set, the control program prepends the directory specified by this variable to the names of the tools invoked.

4.2 COMPILER

The invocation syntax of the C compiler is:

```
cm16 [ option ] ... [file] ... ] ...
```



When you use a **UNIX** shell (Bourne shell, C-shell), arguments containing special characters (such as '(' and '?') must be enclosed with " " or escaped. The -? option (in the C-shell) becomes: "-?" or -\?.

The C compiler accepts C source file names and command line options in random order. Source files are processed in the same order as they appear on the command line (left-to-right). Options are indicated by a leading '-' character. Each C source file is compiled separately and the compiler generates an output file with suffix .src per C source module, containing assembly source code.

The priority of the options is left-to-right: when two options conflict, the first (most left) one takes effect. The -D and -U options are not considered conflicting options, so they are processed left-to-right for each source file. You can overrule the default output file name with the -o option. The compiler uses each -o option only once, so it is possible to specify multiple -o options for multiple source files.

When you invoke **cm16** without any argument, the invocation syntax is displayed (same as -? option).

A summary of the options is given below. The next section describes the options in more detail.

Option	Description
-?	Display invocation syntax
-A[<i>flag</i> ...]	Control language extensions
-C <i>cpu</i>	Use special function registers for <i>cpu</i>
-D <i>macro</i> [= <i>def</i>]	Define preprocessor <i>macro</i>
-E[<i>m</i> <i>l</i>]	Preprocess only or emit dependencies or enable multi-line macros
-H <i>file</i>	Include <i>file</i> before starting compilation
-Idirectory	Look in <i>directory</i> for include files
-J{n f}	Select code memory model (near or far)
-M{s l}	Select memory model: small or large

Option	Description
-Oflag...	Control optimization
-Rmem=name	Change section name
-S	Allocate strings in ROM only
-T	Allocate constants in ROM only
-Umacro	Remove preprocessor <i>macro</i>
-V	Display version header only
-Za	Assume arrays are small (<64KB) if size is unspecified
-Zs	Assume objects don't span 64KB boundary
-Zv	Do not emit fixed vector table initialization
-Zw	Use 3-byte addresses in switch jump tables
-align_data	Align data to an even address
-align_func	Align functions to an even address
-e	Remove output file if compiler errors occur
-err	Send diagnostics to error list file (<i>.err</i>)
-f file	Read options from <i>file</i>
-g[f l n]...	Enable symbolic debug information (unless -gn is used)
-n	Send output to standard output
-o file	Specify name of output <i>file</i>
-s	Merge C-source code with assembly output
-safern,n,...	Enable individual safer C checks
-t	Display lines/min
-u	Treat all 'char' variables as unsigned
-w[num]	Suppress one or all warning messages

Table 4-2: Compiler options (alphabetical)

Description	Options
Include options	
Read options from <i>file</i>	-f file
Include <i>file</i> before starting compilation	-Hfile
Look in <i>directory</i> for include files	-Idirectory

Description	Options
Preprocess options	
Preprocess only or emit dependencies or enable multi-line macros	-E[m l]
Define preprocessor <i>macro</i>	-Dmacro[=def]
Use special function registers for <i>cpu</i>	-Ccpu
Remove preprocessor <i>macro</i>	-Umacro
Allocation control options	
Change section name	-Rmem=name
Allocate strings in ROM only	-S
Allocate constants in ROM only	-T
Code generation options	
Select code model: near or far	-J{n f}
Select memory model: small or large	-M{s l}
Control optimization	-Oflag...
Assume arrays are small (<64KB) if size is unspecified	-Za
Assume objects don't span 64KB boundary	-Zs
Do not emit fixed vector table initialization	-Zv
Use 3-byte addresses in switch jump tables	-Zw
Align data to an even address	-align_data
Align functions to an even address	-align_func
Language control options	
Enable/disable specific language extensions	-A[flag...]
Enable 'float' constants	-Fc
Treat all 'char' variables as unsigned	-u
Output file options	
Remove output file if compiler errors occur	-e
Send output to standard output	-n
Specify name of output <i>file</i>	-o file
Merge C-source code with assembly output	-s
Diagnostic options	
Display invocation syntax	-?
Display version header only	-V

Description	Options
Send diagnostics to error list file (<code>.err</code>)	<code>-err</code>
Enable symbolic debug information (unless <code>-gn</code> is used)	<code>-g[f l n]...</code>
Enable individual safer C checks	<code>-safer_{n,n,...}</code>
Display lines/min	<code>-t</code>
Suppress one or all warning messages	<code>-w[num]</code>

Table 4-3: Compiler options (functional)

4.2.1 DETAILED DESCRIPTION OF THE COMPILER OPTIONS

Option letters are listed below. Each option (except **-o**; see description of the **-o** option) is applied to every source file. If the same option is used more than once, the first (most left) occurrence is used. The placement of command line options is of no importance except for the **-I** and **-o** options. For the **-o** option, the filename may not start immediately after the option. There must be a tab or space in between. All other option arguments must start immediately after the option. Source files are processed in the same order as they appear on the command line (left-to-right).

Some options have an equivalent pragma.

-?

Option:

-?

Description:

Display an explanation of options at stdout.

Example:

cm16 -?

-A

Option:



Select the EDE | C Compiler Options | Project Options... menu item. Set or disable the Language Extensions in the Language tab.



-A[flags]

Arguments:

Optionally one or more language extension flags.

Default:

-A1

Description:

Control language extensions. **-A** without any flags, specifies strict ANSI mode; all language extensions are disabled. This is equivalent to **-ABCDKLPQSTUVX** and **-A0**.

Flags which are controlled by a letter, can be switched on with the lower case letter and switched off with the uppercase letter. Note that the usage of these options might have effect on code density and code execution performance. The following flags are allowed:

- b** Default. Allow 8-bit operations on bitfields.
- B** Do not allow 8-bit operations on bitfields.
- c** Default. Perform character arithmetic. **cm16** generates code using 8-bit character arithmetic as long as the result of the expression is exactly the same as if it was evaluated using integer arithmetic. See also section *Character Arithmetic*.
- C** Conform to ANSI-C by checking for assignments of a constant string to a non constant string pointer. The example above produces warning W130: "operands of '=' are pointers to different types".
- d** Default. Define storage for uninitialized constant rom data, instead of implicit zero initialization. The compiler generates a `'DS 1'` for `'const char i[1];'`.

- D** Uninitialized constant rom data is implicitly zero. The compiler generates a `'DB 1'` for `'const char i[1];'`.
- k** Default. Allow keyword language extensions such as `_far` and `_bit`.
- K** Keyword extensions are not allowed.
- l** Default. 500 significant characters are allowed in an identifier instead of the minimum ANSI-C translation limit of 31 significant characters. Note: more significant characters are truncated without any notice.
- L** Conform to the minimum ANSI-C translation limit of 31 significant characters. This makes it possible to translate your code with any ANSI-C conforming C-compiler. Note: more significant characters are truncated without any notice.
- p** Default. Allow C++ style comments in C source code. For example:

```
// e.g this is a C++ comment line.
```
- P** Do not allow C++ style comments in C source code, to conform to strict ANSI-C.
- q** Default. Allow single quoted strings longer than one character.
- Q** Do not allow single quoted strings longer than one character.
- s** Default. `__STDC__` is defined as `'0'`. The decimal constant `'0'`, intended to indicate a non-conforming implementation. When one of the language extensions are enabled `__STDC__` should be defined as `'0'`.
- S** `__STDC__` is defined as `'1'`. In strict ANSI-C mode (**-A**) `__STDC__` is defined as `'1'`.
- t** Default. Do not promote old-style function parameters when prototype checking.
- T** Perform default argument promotions on old-style function parameters for a strict ANSI-C implementation. `char` type arguments are promoted to `int` type and `float` type arguments are then promoted to `double` type.
- u** Default. Use type `unsigned char` for `0x80-0xff`. The type of an unsuffixed octal or hexadecimal constant is the first of the corresponding list in which its value can be represented:

Character arithmetic enabled **-Ac**:

```
char, unsigned char, int, unsigned int, long,
unsigned long
```

Character arithmetic disabled **-AC** (strict ANSI-C):

```
int, unsigned int, long, unsigned long
```

- U** Do not use type `unsigned char` for 0x80–0xff. The type of an unsuffixed octal or hexadecimal constant is the first of the corresponding list in which its value can be represented:

Character arithmetic enabled **-Ac**:

```
char, int, unsigned int, long, unsigned long
```

Character arithmetic disabled **-AC** (strict ANSI-C):

```
int, unsigned int, long, unsigned long
```

- v** Default. Allow type cast of an lvalue object with incomplete type `void` and lvalue cast which does not change the type and memory of an lvalue object.

Example:

```
void *p; ((int*)p)++;      /* allowed */
int i; (char)i=2;         /* NOT allowed */
```

- V** A cast may not yield an lvalue, to conform strict ANSI-C mode.

- x** Default. Do not check for assignments of a constant string to a non-constant string pointer. With this option the following example produces no warning:

```
char *p;
void main( void ) { p = "hello"; }
```

- X** Conform to ANSI-C by checking for assignments of a constant string to a non-constant string pointer. The example above produces warning W130: "operands of '=' are pointers to different types".

- 0** – same as **-ABCDKLPQSTUVX** (disable all, strict ANSI-C)
- 1** – same as **-Abcdklpqstuvx** (default, enable all)

Example:

To disable character arithmetic and C++ comments enter:

```
cm16 -ACP test.c
```


-align_data

Option:



Select the EDE | C Compiler Options | Project Options... menu item. Enable the Align data to an even address check box in the Code Generation tab.



-align_data

Description:

With this option the compiler generates assembly code to align 16, 32 and 64 bit data variables to even addresses. This optimizes access time but may take extra memory space.

Example:

To specify to the compiler to align data to even addresses, enter:

```
cm16 -align_data test.c
```

-align_func

Option:



Select the EDE | C Compiler Options | Project Options... menu item. Enable the Align functions to an even address check box in the Code Generation tab.



-align_func

Description:

With this option the compiler generates assembly code to align functions to even addresses. This optimizes access time to functions but may take extra memory space.

Example:

To specify to the compiler to align functions to even addresses, enter:

```
cm16 -align_func test.c
```

-C

Option:



Choose a cpu from the EDE | Processor Options... | CPU menu item.



-Ccpu

Arguments:

The cpu name which identifies your M16C derivative.

Description:

Use special function register definitions for *cpu*. The filename looked for is "reg*cpu*.sfr" in the same way include files whose names are enclosed in "" are searched.

Example:

To specify to the compiler to look for a file named `reg61.sfr`, and to use this file as a special function register definition file, enter:

```
cm16 -CM16C61 test.c
```



Section *Special Function Registers* in the previous chapters.

-D

Option:



Select the EDE | C Compiler Options | Project Options... menu item. Define a macro (syntax: *macro*[=*def*]) in the Define user macros field in the Preprocessing tab. You can define more macros by separating them with commas.



-D*macro*[=*def*]

Arguments:

The macro you want to define and optionally its definition.

Description:

Define *macro* to the preprocessor, as in #define. If *def* is not given ('=' is absent), '1' is assumed. Any number of symbols can be defined. The definition can be tested by the preprocessor with #if, #ifdef and #ifndef, for conditional compilations. If the command line is getting longer than the limit of the operating system used, the **-f** option is needed.

Example:

The following command defines the symbol NORAM as 1 and defines the symbol PI as 3.1416.

```
cm16 -DNORAM -DPI=3.1416 test.c
```



-U

-E / -Em / -EI

Option:



Select the **EDE** | **C Compiler Options** | **Project Options...** menu item. Enable the **Preprocess only** and **capture output** check box in the **Preprocessing** tab.



-E[m | I]

Description:

Run the preprocessor of the compiler only and send the output to stdout. When you use the **-E** option, use the **-o** option to separate the output from the header produced by the compiler.

When you use the **-Em** option, the compiler generates dependency rules which can be used by a 'make' utility.

When you use the **-EI** option, you can use multi-line macros. A backslash used to continue a macro on the next source line will be expanded as a new line instead of a concatenation of the lines.

Examples:

The following command preprocesses the file `test.c` and sends the output to the file `preout`.

```
cm16 -E -o preout test.c
```

The following command preprocesses the file `test.c` which may contain multi-line macros, and sends the output to the file `multi`.

```
cm16 -EI test.c -o multi
```

The following command generates dependency rules for the file `test.c` which can be used by **mkcm16** (the M16C 'make' utility).

```
cm16 -Em test.c
```

```
test.src : test.c
```

-e

Option:

EDE always removes the output file on errors.

**-e****Description:**

Remove the output file when an error has occurred. With this option the 'make' utility always does the proper productions.

Example:

```
cm16 -e test.c
```

-err

Option:



In EDE this option is not so useful. If you would use this option you would not see the error messages in the Build tab.



-err

Description:

Write errors to the file *source.err* instead of stderr.

Example:

To write errors to the `test.err` instead of stderr, enter:

```
cm16 -err test.c
```

-f

Option:



Select the EDE | C Compiler Options | Project Options... menu item. Add the option to the Additional options field in the Misc tab.



-f *file*

Arguments:

A filename for command line processing. The filename “-” may be used to denote standard input.

Description:

Use *file* for command line processing. To get around the limits on the size of the command line, it is possible to use command files. These command files contain the options that could not be part of the real command line. Command files can also be generated on the fly, for example by the make utility.

More than one **-f** option is allowed.

Some simple rules apply to the format of the command file:

1. It is possible to have multiple arguments on the same line in the command file.
2. To include whitespace in the argument, surround the argument with either single or double quotes.
3. If single or double quotes are to be used inside a quoted argument, we have to go by the following rules:
 - a. If the embedded quotes are only single or double quotes, use the opposite quote around the argument. Thus, if a argument should contain a double quote, surround the argument with single quotes.
 - b. If both types of quotes are used, we have to split the argument in such a way that each embedded quote is surrounded by the opposite type of quote.

Example:

```
"This has a single quote ' embedded"
```

or

```
'This has a double quote " embedded'
```

or

```
'This has a double quote " and \  
a single quote ''' embedded"
```

4. Some operating systems impose limits on the length of lines within a text file. To circumvent this limitation it is possible to use continuation lines. These lines end with a backslash and newline. In a quoted argument, continuation lines will be appended without stripping any whitespace on the next line. For non-quoted arguments, all whitespace on the next line will be stripped.

Example:

```
"This is a continuation \  
line"  
-> "This is a continuation line"  
  
control(file1(mode,type),\  
        file2(type))  
->  
control(file1(mode,type),file2(type))
```

5. It is possible to nest command line files up to 25 levels.

Example:

Suppose the file `mycmds` contains the following line:

```
-err  
test.c
```

The command line can now be:

```
cm16 -f mycmds
```

-g

Option:



Select the EDE | C Compiler Options | Project Options... menu item. Enable the Generate symbolic debug information check box in the Debug tab. Optionally enable the Include debug information for non referenced types and/or Disable lifetime info for all types check box.



-g[f|l|n]...

Description:

Add directives to the output files, incorporating symbolic information to facilitate high level debugging.

With **-gn** you disable all debug, including type checking.

With **-gl** you disable lifetime information for all types.

If you use **-gf**, high level language type information is also emitted for types which are not referenced by variables. Therefore, this sub-option is not recommended.

When the compiler is set to a high optimization level the debug comfort may decrease.

Examples:

To add symbolic debug information to the output files, enter:

```
cm16 -g test.c
```

To add symbolic debug information to the output files but disable lifetime information for all types, enter:

```
cm16 -gl test.c
```

To disable all symbolic debug information including type checking, enter:

```
cm16 -gn test.c
```

-H

Option:



Select the EDE | C Compiler Options | Project Options... menu item. Enter a filename in the First #include this file field in the Preprocessing tab.



-H*file*

Arguments:

The name of an include file.

Description:

Include *file* before compiling the C-source. This is the same as specifying #include "*file*" at the first line of your C-source.

Example:

```
cm16 -Hstdio.h test.c
```



-I



Option:



Select the EDE | Directories... menu item. Add one or more directory paths to the Include Files Path field.



-I*directory*

Arguments:

The name of the directory to search for include file(s).

Description:

Change the algorithm for searching `#include` files whose names do not have an absolute pathname to look in *directory*. Thus, `#include` files whose names are enclosed in `""` are searched for first in the directory of the file containing the `#include` line, then in directories named in **-I** options in left-to-right order. If the include file is still not found, the compiler searches in a directory specified with the environment variable `CM16CINC`. `CM16CINC` may contain more than one directory. Finally, the directory `../include` relative to the directory where the compiler binary is located is searched. This is the standard include directory supplied with the compiler package.

For `#include` files whose names are in `<>`, the directory of the file containing the `#include` line is not searched. However, the directories named in **-I** options (and the one in `CM16CINC` and the relative path) are still searched.

Example:

```
cm16 -I/proj/include test.c
```



Section *Include Files*.

-J

Option:

-J*model*

Arguments:

The code model to use, where *model* is one of the following:

n near
f far

Default

-Jf

Description:

Select the code model to use.

Example:

```
cm16 -Jn test.c
```

-M

Option:



Select the EDE | C Compiler Options | Project Options... menu item. Choose a Memory Model and a Default Data Memory in the Code Generation tab.



-M*model*

Arguments:

The memory model to be used, where *model* is one of:

s small (default)
l large

Default:

-Ms

Description:

Select memory model to be used.

Example:

```
cm16 -Ml test.c
```



Section *Memory Models*.

-n

Option:

-n

Description:

Do not create output files; instead, the output is sent to stdout.

Example:

```
cm16 -n test.c
```

-O

Option:



Select the EDE | C Compiler Options | Project Options... menu item. You can control optimizations in the Optimization and Adv. Optim. tabs.



-Oflags

Pragma:

optimize flags

Arguments:

One or more optimization flags.

Default:

-O1

Description:

Control optimization. If you do not use this option, the default optimization of **cm16** is **-O1**, which is an optimization level to let **cm16** generate the smallest code.

Flags which are controlled by a letter, can be switched on with the lower case letter and switched off with the uppercase letter. These options are described together.

All optimization flags can also be given in the source file after a `#pragma optimize`. However, depending on the optimization some `optimize` pragmas affect the entire module (module level), whereas other `optimize` pragmas can be used on a function scope only (function level) or on each source line (flow level). 'On function level' means that if a `pragma optimize` is found within a function, it is interpreted as if it was found just before the function. The optimization level of each `optimize` pragma is described for each **-O** option.

An overview of the flags is given below.

a	- relaxed alias checking (cse required)	(function)
c	- cse optimazition	(function)
d	- optimizations based on data flow analysis	(function)
e	- expression propagation (cse required)	(function)
f	- flow optimization	(function)
i	- invariant code motoin	(function)
l	- duplicate loop condition	(function)
s	- optimize for small code size	(flow)
t	- force jump table for switch statement	(flow)
u	- loop unrolling	(function)
v	- subscript strength reduction on arrays	(function)
w	- switch statement optiminization, table or chain	(flow)
y	- peephole optimization	
0	- same as -OACDEFILSTUVWY	(no optim)
1	- same as -OacdefiLsTUvwy	(default, size)
2	- same as -OACDefiLsTUvwy	(debug, size)
3	- same as -OacdefilSTUvwy	(speed)
4	- same as -OACDEFilSTUvwy	(debug, speed)

Example:

```
cm16 -OACdEfILUv test.c
```



Pragma optimize in section *Pragmas*.

-Onumber

Option:

-Onumber

Arguments:

A number in the range 0 – 2.

Default:

-O1

Description:

Control optimization. You can specify a single number in the range 0 – 2, to enable or disable optimization. The options are a combination of the other optimization flags:

- O0** – same as **-OACDEFILSTUVWY**
Switchable optimizations switched off
- O1** – same as **-OacdefilStUvw**
Default. Set optimization to let **cm16** generate the smallest code.
- O2** – same as **-OacdefilSTUvw**
Set optimization flags to let **cm16** generate the fastest code.



The flags 0 to 2 cannot be concatenated with other flags. For example, **-Oa2c** is not allowed, **-OacF** is allowed.

Example:

To optimize for code size, enter:

```
cm16 -O1 test.c
```

-Oa / -OA

Option:



Select the EDE | C Compiler Options | Project Options... menu item. Select the Advanced optimization level in the Optimization tab. Enable or disable the Relaxed alias checking (requires CSE) check box.



-Oa / -OA

Pragma:

noalias / alias

optimize a / optimize A (on function level)

Default:

-OA

Description:

With **-Oa** you relax alias checking. If you specify this option, **cm16** will not erase remembered register contents of user variables if a write operation is done via an indirect (calculated) address. You must be sure this is not done in your C-code (check pointers!) before turning on this option. Note that the option **-Oc** must be on to use this option.

With **-OA** you specify strict alias checking. If you specify this option, the compiler erases all register contents of user variables when a write operation is done via an indirect (calculated) address.

Example:

An example is given in section *Alias* in this chapter.



-Oc

Pragmas **noalias**, **alias** and **optimize** in section *Pragmas*

-Oc / -OC

Option:



Select the EDE | C Compiler Options | Project Options... menu item. Select the Advanced optimization level in the Optimization tab. Enable or disable the Common subexpression elimination (CSE) check box.



-Oc / -OC

Pragma:

optimize c / optimize C (on function level)

Default:

-Oc

Description:

With **-Oc** you enable CSE (common subexpression elimination). With this option specified, the compiler tries to detect common subexpressions within the C code. The common expressions are evaluated only once, and their result is temporarily held in registers.

Note that the **-Oc** option must be on to enable the relax alias checking (**-Oa**), expression propagation (**-Oe**) and moving invariant code outside a loop (**-Oi**).

With **-OC** you disable CSE (common subexpression elimination). With this option specified, the compiler will not try to search for common expressions. Also relax alias checking, expression propagation and moving invariant code outside a loop will be disabled.

Example:

```
/*
 * Compile with -OC -O0,
 * Compile with -Oc -O0, common subexpressions are found
 *   and temporarily saved.
 */

char x, y, a, b, c, d;

void
main( void )
{
    x = (a * b) - (c * d);

    y = (a * b) + (c * d); /*(a*b) and (c*d) are common */
}
```



Pragma optimize in section *Pragmas*.

-Od / -OD

Option:



Select the EDE | C Compiler Options | Project Options... menu item. Select the Advanced optimization level in the Optimization tab. Enable or disable the Constant and copy propagation (requires CSE) check box.



-Od / -OD

Pragma:

optimize d / optimize D (on function level)

Default:

-Od

Description:

With **-Od** you enable constant and copy propagation. With this option, the compiler tries to find assignments of constant values to a variable, a subsequent assignment of the variable to another variable can be replaced by the constant value.

With **-OD** you disable constant and copy propagation.

Example:

```
/*
 * Compile with -OD -O0, 'i' is actually assigned to 'j'
 * Compile with -Od -O0, 15 is assigned to 'j', 'i' was
 * propagated
 */

int i;
int j;

void
main( void )
{
    i = 10;
    j = i + 5;
}
```



Pragma optimize in section *Pragmas*.

-Oe / -OE

Option:



Select the EDE | C Compiler Options | Project Options... menu item. Select the Advanced optimization level in the Optimization tab. Enable or disable the Expression propagation (requires CSE) check box.



-Oe / -OE

Pragma:

optimize e / optimize E (on function level)

Default:

-Oe

Description:

With **-Oe** you enable expression propagation. With this option, the compiler tries to find assignments of expressions to a variable, a subsequent assignment of the variable to another variable can be replaced by the expression itself. Note that the option **-Oc** must be on to use this option.

With **-OE** you disable expression propagation.

Example:

```
/*
 * Compile with -OE -Oc -O0, normal cse is done
 * Compile with -Oe -Oc -O0, 'i+j' is propagated.
 */

unsigned i, j;

int
main( void )
{
    static int a;
    a = i + j;
    return (a);
}
```

**-Oc**Pragma optimize in section *Pragmas*.

-Of / -OF

Option:



Select the EDE | C Compiler Options | Project Options... menu item. Select the Advanced optimization level in the Optimization tab. Enable or disable the Code flow optimization and order rearranging check box.



-Of / -OF

Pragma:

optimize f / optimize F(on function level)

Default:

-Of

Description:

With **-Of** you enable control flow optimizations and code order rearranging on the intermediate code representation, such as jump chaining and conditional jump reversal.

With **-OF** you disable control flow optimizations.

Examples:

The following example shows a control optimization:

```
/*
 * Compile with -OF -O0
 * Compile with -Of -O0, compiler finds first time 'i'
 * is always < 10, the unconditional jump is removed.
 */
int i;

void
main( void )
{
    for( i=0; i<10; i++ )
    {
        do_something();
    }
}
```

The following example shows a conditional jump reversal:

```
/*
 * Compile with -OF -O0, code as written sequential
 * Compile with -Of -O0, code is rearranged
 *
 * Code rearranging enables other optimizations to
 * optimize better, e.g. CSE
 */

int i;
extern void dummy( void );

void main ()
{
    do
    {
        if ( i )
        {
            i--;
        }
        else
        {
            i++;
            break;
        }
        dummy();
    } while ( i );
}
```



Pragma optimize in section *Pragmas*.

-Oi / -OI

-Oi / -OI



Select the EDE | C Compiler Options | Project Options... menu item. Select the Advanced optimization level in the Optimization tab. Enable or disable the Move invariant code outside loop (requires CSE) check box.



-Oi / -OI

Pragma:

optimize i / optimize I (on function level)

Default:

-Oi

Description:

With **-Oi** you move invariant code outside a loop. Note that the option **-Oc** must be on to use this option.

With **-OI** you disable moving invariant code outside a loop.

Example:

```
/*
 * Compile with -OI -Oc -O0, normal cse is done
 * Compile with -Oi -Oc -O0, invariant code is found in
 *   the loop, code is moved outside the loop.
 */
void
main( void )
{
    char x, y, a, b;
    int i;

    for( i=0; i<20; i++ )
    {
        x = a + b;
        y = a + b;
    }
}
```

**-Oc**Pragma optimize in section *Pragmas*.

-OI / -OL

Option:



Select the EDE | C Compiler Options | Project Options... menu item. Select the Advanced optimization level in the Optimization tab. Enable or disable the Generate fast loops (increases code size) check box.



-OI / -OL

Pragma:

optimize 1 / optimize L(on function level)

Default:

-OL

Description:

With **-OI** you enable fast loops. Duplicate the loop condition. Evaluate the loop condition one time outside the loop, just before entering the loop, and at the bottom of the loop. This saves one unconditional jump and gives less code inside a loop.

With **-OL** you disable fast loops.

Example:

```
/*
 * Compile with -OL -O0
 * Compile with -OI -O0, compiler duplicates the loop
 * condition, the unconditional jump is removed.
 */
int i;

void
main( void )
{
    for( ; i<10; i++ )
    {
        do_something();
    }
}
```



Pragma optimize in section *Pragmas*.

-Os / -OS

Option:



Select the EDE | C Compiler Options | Project Options... menu item. Select the Advanced optimization level in the Optimization tab. Enable or disable the Favor small code size above execution speed check box.



-Os / -OS

Pragma:

optimize s / optimize S (on flow level)

Default:

-Os

Description:

With **-Os** you tell the compiler to generate smaller code. Whenever possible less instructions are used. Note that this may result in more instruction cycles.

With **-OS** you disable the smaller code optimization.



Pragma optimize in section *Pragmas*.

-Ot / -OT

Option:

-Ot / -OT

Pragma:

optimize t / optimize T (on flow level)

Default:

-OT

Description:

With **-Ot** you force the compiler to generate jump tables for switch statements.

With **-OT** it depends on the **-Ow/-OW** option which switch method is used. With **-OT** and **-OW** the compiler generates a jump chain for switch statements. With **-OT** and **-Ow** the compiler chooses the best switch method possible, jump chain or jump table. So, with **-OT** a jump table can still be generated.

Overview:

-Ot -Ow	jump table
-OT -Ow	smart
-Ot -OW	jump table
-OT -OW	jump chain

Example:

```
/*
 * Compile with -OT -OW, generate jump chain.
 * Compile with -Ot -OW, generate jump table.
 */
int i;

void
main( void )
{
    switch (i)
    {
        case 1:      i = 0;
                    case 2:      i = 1;
                    case 3:      i = 2;
                    default:      i = 3;
    }
}
```



Section *Switch Statement*.

Pragma optimize in section *Pragmas*.

-Ou / -OU

Option:



Select the EDE | C Compiler Options | Project Options... menu item. Select the Advanced optimization level in the Optimization tab. Enable or disable the Loop unrolling check box.



-Ou / -OU

Pragma:

optimize u / optimize U (on function level)

Default:

-OU

Description:

With **-Ou** you enable loop unrolling. With this option specified, the compiler tries to eliminate short loops by duplicating a loop body 2, 4 or 8 times. This reduces the number of branches and creates a longer linear code part.

With **-OU** you disable loop unrolling.

Example:

```
/*
 * Compile with -OU, normal loop handling
 * Compile with -Ou, loop is eliminated,
 * body is duplicated
 */
int i, j;

void
main( void )
{
    for( i=0; i<2; i++ )    /* short loop */
    {
        j = 2 * i;
    }
}
```



Pragma optimize in section *Pragmas*.

-Ov / -OV

Option:



Select the EDE | C Compiler Options | Project Options... menu item. Select the Advanced optimization level in the Optimization tab. Enable or disable the Subscript strength reduction check box in the Advanced Optimization tab.



-Ov / -OV

Pragma:

optimize v / optimize V (on function level)

Default:

-Ov

Description:

With **-Ov** you enable subscript strength reduction. With this option specified, the compiler tries to reduce expressions involving an index variable in strength.

With **-OV** you disable subscript strength reduction.

Example:

```
/*
 * Compile with -OV -O0, disable subscript strength
 * reduction
 * Compile with -Ov -O0, begin and end address of 'a'
 * are determined before the loop and temporarily put in
 * registers instead of determining the address each
 * time inside the loop
 */
int i;
int a[4];

void
main( void )
{
    for( i=0; i<4; i++ )
    {
        a[i] = i;
    }
}
```



Pragma optimize in section *Pragmas*.

-Ow / -OW

Option:

-Ow / -OW

Pragma:

optimize w / optimize W (on flow level)

Default:

-Ow

Description:

With **-Ow** the compiler chooses the best switch method possible, jump chain or jump table, unless **-Ot** is used. **-Ot** forces the generation of a jump table.

With **-OW** the compiler generates a jump chain for switch statements, unless **-Ot** is used. **-Ot** forces the generation of a jump table.

Example:

```
/*
 * Compile with -OW -OT, always generate jump chain.
 * Compile with -Ow -OT, choose best switch method, in this
 * case this is also a jump chain.
 */
int i;

void
main( void )
{
    switch (i)
    {
        case 1:    i = 0;
        case 2:    i = 1;
        case 3:    i = 2;
        default:   i = 3;
    }
}
```



Section *Switch Statement*.
Pragma *optimize* in section *Pragmas*.

-Oy / -OY

Option:



Select the EDE | C Compiler Options | Project Options... menu item. Select the Advanced optimization level in the Optimization tab. Enable or disable the Peephole optimization check box in the Advanced Optimization tab.



-Oy / -OY

Pragma:

optimize y / optimize Y (on flow level)

Default:

-Oy

Description:

With **-Oy** you enable peephole optimization. Remove redundant code. The peephole optimizer searches for redundant instructions or for instruction sequences which can be combined to minimize the number of instructions.

With **-OY** you disable peephole optimization.

Example:

```
/*
 * Compile with -OY -O0, unnecessary instructions found
 * Compile with -Oy -O0, peephole optimizer searches
 * for patterns in the generated code which can be
 * removed/combined. E.g.
 */

long a;
long f(void);

void
main( void )
{
    long b;

    b = f();
    a = (a << 1) + b;
}
```



Pragma optimize in section *Pragmas*.

-o

Option:



Select the EDE | C Compiler Options | Project Options... menu item. Add the option to the Additional options field in the Misc tab.



-o *file*

Arguments:

An output filename. The filename may not start immediately after the option. There must be a tab or space in between.

Default:

Module name with `.src` suffix.

Description:

Use *file* as output filename, instead of the module name with `.src` suffix. Special care must be taken when using this option, the first **-o** option found acts on the first file to compile, the second **-o** option acts on the second file to compile, etc.

Example:

When specified:

```
cm16 file1.c file2.c -o file3.src -o file2.src
```

two files will be created, **file3.src** for the compiled file **file1.c** and **file2.src** for the compiled file **file2.c**.

-R

Option:



Select the EDE | C Compiler Options | Project Options... menu item. Add the option to the Additional options field in the Misc tab.



-R*mem=name*

Pragma:

renamesect

Arguments:

A memory space, followed by a section name. *mem* can be one of:

<i>mem</i>	Description
BI	<code>_bit</code>
FD	<code>_far</code>
DA	<code>_near</code>
RO	<code>_rom</code>
CO	near program code
FC	far program code

Table 4-4: Memory spaces

Description:

The compiler defaults to a section naming convention, using the module name and a two letter memory type abbreviation: *name*_RO for executable code. In case a module must be loaded at a fixed address or a data section needs a special place in memory, the **-R** option enables the user to generate a unique section name. In this way the order **lcm16** allocates these sections can be specified in a locator description file.

Example:

To create a new section name (MARK_CLR_BI) for cleared `_bit` sections, enter:

```
cm16 -RBI=MARK test.c
```



Pragma renamesect in section *Pragmas*.

-S

Option:



Select the **EDE | C Compiler Options | Project Options...** menu item. Enable the **Keep strings and constants in ROM** check box in the **Code Generation** tab.



-S

Description:

Default string literals and `const` declared objects are allocated in ROM and copied to RAM, which allows run-time string modification.

With option **-S** string literals and `const` declared objects can be allocated in ROM only.

Example:

```
cm16 -S test.c
```



Section 3.9 *Strings* in Chapter *Language Implementation* and section *const Qualifier* in section 3.2.1 *Storage Types* in Chapter *Language Implementation*.

-S

Option:



Select the EDE | C Compiler Options | Project Options... menu item. Enable the Merge C source code with assembly output check box in the Code Generation/Output tab.



-s

Pragma:

source

Description:

Merge C source code with generated assembly code in output file.

Example:

```
cm16 -s test.c

; test.c      1 long a;
; test.c      2 long f(void);
; test.c      3
; test.c      4 void
; test.c      5 main( void )
; test.c      6 {
                GLOBAL _main
```



Pragmas source and nosource in section *Pragmas*.

-safer

Option:

`-safer n,n,\dots`

Arguments:

The Safer C rules to be checked.

Description:

With this option, the Safer C rules to be checked can be specified. Refer to Appendix B *Safer C* for a list of supported and unsupported Safer C rules.

Example:

```
cm16 -safer9 test.c
```

Will generate an error in case 'test.c' contains nested comments.

-T

Option:

Select the EDE | C Compiler Options | Project Options... menu item. Enable the Keep constants in ROM check box in the Code Generation tab.

**-T****Description:**

Default string literals and `const` declared objects are allocated in ROM and copied to RAM, which allows run-time string modification.

With option **-T** `const` declared objects can be allocated in ROM only.

Example:

```
cm16 -T test.c
```



Section *const Qualifier* in section 3.2.1 *Storage Types* in Chapter *Language Implementation*.

-t

Option:

-t

Description:

Display the number of lines processed and the compilation speed in lines per minute.

Example:

```
cm16 -t test.c
```

```
processed 25 lines at 7075 lines/min
```

-U

Option:

-U*name*

Arguments:

The name macro you want to undefine.

Description:

Remove any initial definition of identifier *name* as in `#undef`, unless it is a predefined ANSI standard macro. ANSI specifies the following predefined symbols to exist, which cannot be removed:

<code>__FILE__</code>	"current source filename"
<code>__LINE__</code>	current source line number (int type)
<code>__TIME__</code>	"hh:mm:ss"
<code>__DATE__</code>	"Mmm dd yyyy"
<code>__STDC__</code>	level of ANSI standard. This macro is set to 1 when the option to disable language extensions (-A) is effective. Whenever language extensions are excepted, <code>__STDC__</code> is set to 0 (zero).

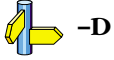
When **cm16** is invoked, also the following predefined symbols exist:

<code>_CM16C</code>	predefined symbol to identify the compiler. This symbol can be used to flag parts of the source which must be recognized by the cm16 compiler only. It expands to the version number of the compiler.
<code>_CODEMODEL</code>	Identifies for which memory model the module was compiled ("n" for near and "f" for far).
<code>_MODEL</code>	identifies for which memory model the module is compiled. It expands to a single character ('s' for small or 'l' for large) that can be tested by the preprocessor. See section <i>Memory Models</i> for details.

These symbols can be turned off with the **-U** option.

Example:

```
cm16 -U_MODEL test.c
```



-u

Option:



Select the EDE | C Compiler Options | Project Options... menu item. Enable the Treat 'char' variables as unsigned check box in the Language2 tab.



-u

Description:

Treat 'character' type variables as 'unsigned character' variables. By default char is the same as specifying signed char. With **-u** char is the same as unsigned char.

Example:

With the following command char is treated as unsigned char:

```
cm16 -u test.c
```

-V

Option:

-V

Description:

Display version information.

Example:

```
cm16 -V
```

```
m16c C compiler vx.y rz          SN000000-015 (c) year TASKING, Inc.
```

-W

Option:



Select the EDE | C Compiler Options | Project Options... menu item. Select the Suppress specific warnings option in the Misc tab and optionally fill in specific message numbers to suppress.



-w*[num]*

Arguments:

Optionally the warning number to suppress.

Description:

-w suppress all warning messages. **-wnum** only suppresses the given warning.

Example:

To suppress warning 135, enter:

```
cm16 file1.c -w135
```

-Z

Option:



Select the **EDE | C Compiler Options | Project Options...** menu item. Select the **Additional options** in the **Misc.** tab and optionally fill in specific code generation options.



-Zargument

Arguments:

Miscellaneous M16C-specific code generation options, where option is one of the following:

- a Assume small (<=64KB) arrays by default
- s Assume that objects don't span 64KB boundaries
- v Do not emit fixed vector table initialization code
- w Use JMPL.A in switch tables

Default:

All options turned off.

Description:

Select certain code generation options.

Example:

```
cm16 -Za -Zw test.c
```

4.3 INCLUDE FILES

You may specify include files in two ways: enclosed in `<>` or enclosed in `""`. When an `#include` directive is seen, **cm16** used the following algorithm trying to open the include file:

1. If the filename is enclosed in `""`, and it is not an absolute pathname (does not begin with a `'\'` the include file is searched for in the directory of the file containing the `#include` line. For example, in:

PC:

```
cm16 ..\..\source\test.c
```

UNIX:

```
cm16 ../../source/test.c
```

cm16 first searches in the directory `..\..\source`.

If you compile a source file in the directory where the file is located (**cm16 test.c**), the compiler searches for include files in the current directory.



This first step is not done for include files enclosed in `< >`.

2. Use the directories specified with the `-I` options, in a left-to-right order. For example:

PC:

```
cm16 -I..\..\include demo.c
```

UNIX:

```
cm16 -I../../include demo.c
```

3. Check if the environment variable `CM16CINC` exists. If it does exist, use the contents as a directory specifier for include files. You can specify more than one directory in the environment variable `CM16CINC` by using a separator character. Instead of using `-I` as in the example above, you can specify the same directory using `CM16CINC`:

PC:

```
set CM16CINC=..\..\include
cm16 demo.c
```

UNIX:

if using the Bourne shell (sh)

```
CM16CINC=../../include
export CM16CINC
cm16 demo.c
```

or if using the C-shell (csh)

```
setenv CM16CINC ../../include
cm16 demo.c
```

4. When an include file is not found with the rules mentioned above, the compiler tries the subdirectory `include`, one directory higher than the directory containing the **cm16** binary. For example:

PC:

cm16.exe is installed in the directory `C:\CM16C\BIN`
 The directory searched for the include file is `C:\CM16C\INCLUDE`

UNIX:

cm16 is installed in the directory `/usr/local/cm16c/bin`
 The directory searched for the include file is
`/usr/local/cm16c/include`

The compiler determines run-time which directory the binary is executed from to find this include directory.

A directory name specified with the **-I** option or in `CM16CINC` may or may not be terminated with a directory separator, because **cm16** inserts this separator, if omitted.

When you specify more than one directory to the environment variable `CM16CINC`, you have to use one of the following separator characters:

PC:

`; , space`

e.g. `set CM16CINC=..\..\include;\proj\include`

UNIX:

`: ; , space`

e.g. `setenv CM16CINC ../../include:/proj/include`

4.4 PRAGMAS

According to ANSI (3.8.6) a preprocessing directive of the form:

```
#pragma pragma-token-list new-line
```

causes the compiler to behave in an implementation-defined manner. The compiler ignores pragmas which are not mentioned in the list below. Pragmas give directions to the code generator of the compiler. Besides the pragmas there are two other possibilities to steer the code generation process: command line options and keywords (e.g., `_bit` type variables) in the C application itself. The compiler acknowledges these three groups using the following rules:

Command line options can be overruled by keywords and pragmas. Keywords can be overruled by pragmas. Hence, pragmas have the highest priority.

This approach makes it possible to set a default optimization level for a source module, which can be overridden temporarily within the source by a pragma.

The C compiler **cm16** supports the following pragmas:

alias

Default. Same as **-OA**. Perform strict alias checking. When a pointer is dereferenced, all register contents are assumed to be invalid afterwards. See also the section *Alias*.

noalias

Same as **-Oa**. Relax alias checking.

asm

Insert the following (non preprocessor lines) as assembly language source code into the output file. The inserted lines are not checked for their syntax. The code buffer of the peephole optimizer is flushed. Thus the compiler will stop optimizations like peephole pattern replacement and resumes these optimizations after the **endasm** pragma as if it starts at the beginning of a function.

For advanced assembly in-lining, intrinsic functions can be used. The defined set of intrinsic functions cover most of the specific M16C features which could otherwise not be accessed by the C language. For more information on intrinsic functions see section 3.14 *Intrinsic Functions*.

asm_noflush

Same as pragma **asm**, except that the peephole optimizer does not flush the code buffer and assumes register contents remain valid.

endasm

Switch back to the C language.

The section *Inline Assembly* in the chapter *Language Implementation* contains more information.

listinc

Expand include files in generated list file.

nolistinc

Default. Do not expand include files in list file.

optimize flags

Controls the amount of optimization. The remainder of the source line is scanned for option characters, which are processed like the flags of the **-O** command line option. Please refer to the **-O** option for the list of available flags.

Depending on the optimization some **optimize** pragmas can be used on a function scope only (function level), whereas other **optimize** pragmas can be used on each source line (flow level). 'On function level' means that if a pragma **optimize** is found within a function, it is interpreted as if it was found just before the function. The optimization level of each **optimize** pragma is described for each *flag* of the **-O** option.

endoptimize

End a region that was optimized with a **#pragma optimize**. The pragma **endoptimize** restores the situation as it was before the corresponding pragma **optimize**. **#pragma optimize/endoptimize** pairs can be nested.

Example:

```
#pragma optimize 0
/* disable all optimizations */
. . .
#pragma optimize t
/* force generation of table for switch */
switch(...)
{
    . . .
}
#pragma endoptimize
/* back to all optimizations disabled */
#pragma endoptimize
/* back to default optimizations */
```

renamesect mem=new

Rename a section. This pragma is used to create unique section names. Section renaming is used in combination with locator description files to assign specific locator attributes to pieces of C code. Same as **-R** option.

source

Same as the **-s** option. Enable mixing C source with assembly code.

nosource

Default. Disable generation of C source within assembly code.

4.5 ALIAS

By default the compiler assumes that each pointer may point to any object created in the program, so when any pointer is dereferenced, all register contents are assumed to be invalid afterwards.

When it is known that aliasing problems do not occur in the written C source, alias checking may be relaxed (use the **-Oa** option). Note that the option **-Oc** must be on to use this option. Relaxing alias checking may reduce code size.

Example 1:

```
int i;

void
func( )
{
    char    * p;
    char    c;
    char    d;

    if( i )
        p = &c;
    else
        p = &d;

    c = 2;
    d = 3;

    *p = 4; /* may write to 'c' or 'd'          */
           /* --> aliasing object 'c' or 'd'    */

    i = c; /* '*p' might have changed the value of 'c', */
           /* so 'c' may not be used from register */
           /* contents, but MUST be read from memory */
           /* --> alias checking MUST be ON in this case */
}
```

Example 2:

```
int i;

void
func( char *p )
{
    char    c;
    char    d;

    c = 2;
    d = 3;

    *p = 4; /* cannot write to 'c' or 'd', but to some other
            object */

    i = c; /* '*p' cannot have changed the value of 'c', */
          /* so 'c' may be used from register contents */
          /* --> alias checking may be OFF in this case */
}

```

Example 3:

```
int array[2];

main()
{
    array[0] = 1;
    array[1] = -1;

    array[0] = array[0] + array[1];
    /* an interrupt might have changed the value */
    /* of 'array', so 'array' may not be used */
    /* from register contents, but MUST be read */
    /* from memory */
    /* --> alias checking MUST be ON in this case */
}

```

4.6 COMPILER LIMITS

The ANSI C standard [1-2.2.4] defines a number of translation limits, which a C compiler must support to conform to the standard. The standard states that a compiler implementation should be able to translate and execute a program that contains at least one instance of every one of the limits listed below. **cm16**'s actual limits are given within parentheses.

Most of the actual compiler limits are determined by the amount of free memory in the host system. In this case a 'D' (Dynamic) is given between parentheses. Some limits are determined by the size of the internal compiler parser stack. These limits are marked with a 'P'. Although the size of this stack is 200, the actual limit can be lower and depends on the structure of the translated program.

- 15 nesting levels of compound statements, iteration control structures and selection control structures (P > 15)
- 8 nesting levels of conditional inclusion (50)
- 12 pointer, array, and function declarators (in any combinations) modifying an arithmetic, a structure, a union, or an incomplete type in a declaration (15)
- 31 nesting levels of parenthesized declarators within a full declarator (P > 31)
- 32 nesting levels of parenthesized expressions within a full expression (P > 32)
- 31 significant characters in an external identifier (full ANSI-C mode),
500 significant characters in an external identifier (non ANSI-C mode)
- 511 external identifiers in one translation unit (D)
- 127 identifiers with block scope declared in one block (D)
- 1024 macro identifiers simultaneously defined in one translation unit (D)
- 31 parameters in one function declaration (D)
- 31 arguments in one function call (D)
- 31 parameters in one macro definition (D)
- 31 arguments in one macro call (D)
- 509 characters in a logical source line (1500)
- 509 characters in a character string literal or wide string literal (after concatenation) (1500)

- 8 nesting levels for **#included** files (50)
- 257 case labels for a switch statement, excluding those for any nested switch statements (D)
- 127 members in a single structure or union (D)
- 127 enumeration constants in a single enumeration (D)
- 15 levels of nested structure or union definitions in a single struct-declaration-list (D)



CHAPTER

5

COMPILER DIAGNOSTICS



5

CHAPTER

5.1 INTRODUCTION

cm16 has three classes of messages: user errors, warnings and internal compiler errors.

Some user error messages carry extra information, which is displayed by the compiler after the normal message. The messages with extra information are marked with 'I' in the list below. They never appear without a previous error message and error number. The number of the information message is not important, and therefore, this number is not displayed. A user error can also be fatal (marked as 'F' in the list below), which means that the compiler aborts compilation immediately after displaying the error message and may generate a 'not complete' output file.

The error numbers and warning numbers are divided in two groups. The frontend part of the compiler uses numbers in the range 0 to 499, whereas the backend (code generator) part of the compiler uses numbers in the range 500 and higher. Note that most error messages and warning messages are produced by the frontend.

If you program a non fatal error, **cm16** displays the C source line that contains the error, the error number and the error message on the screen. If the error is generated by the code generator, the C source line displayed always is the last line of the current C function, because code generation is started when the end of the function is reached by the frontend. However, in this case, **cm16** displays the line number causing the error before the error message. **cm16** always generates the error number in the assembly output file, exactly matching the place where the error occurred.

So, when a compilation is not successful, the generated output file is not accepted by the assembler, thus preventing a corrupt application to be made (see also the **-e** option).

Warning messages do not result into an erroneous assembly output file. They are meant to draw your attention to assumptions of the compiler, for a situation which may not be correct. Warning messages can be controlled with the **-w[num]** option.

The last class of messages are the internal compiler errors. The following format is used:

S number: internal error - please report

These errors are caused by failed internal consistency checks and should never occur. However, if such a 'SYSTEM' error appears, please report the occurrence to TASKING, using a Problem Report form. Please include a diskette or tape, containing a small C program causing the error.

5.2 RETURN VALUES

cm16 returns an exit status to the operating system environment for testing.

For example,

in a MS-DOS BATCH-file you can examine the exit status of the program executed with ERRORLEVEL:

```
cm16 -s %1.c
IF ERRORLEVEL 1 GOTO STOP_BATCH
```

In a bourne shell script, the exit status can be found in the `$?` variable, for example:

```
cm16 $*
case $? in
0)      echo ok ;;
1|2|3)  echo error ;;
esac
```

The exit status of **cm16** is one of the numbers of the following list:

- 0 Compilation successful, no errors
- 1 There were user errors, but terminated normally
- 2 A fatal error, or System error occurred, premature ending
- 3 Stopped due to user abort

5.3 ERRORS AND WARNINGS

Errors start with an error type, followed by a number and a message. The error type is indicated by a letter:

- I information
- E error
- F fatal error
- S internal compiler error
- W warning

Frontend

- F 1 evaluation expired

Your product evaluation period has expired. Contact your local TASKING office for the official product.

- W 2 unrecognized option: '*option*'

The option you specified does not exist. Check the invocation syntax for the correct option.

- E 4 expected *number* more '#endif'

The preprocessor part of the compiler found the '#if', '#ifdef' or '#ifndef' directive but did not find a corresponding '#endif' in the same source file. Check your source file that each '#if', '#ifdef' or '#ifndef' has a corresponding '#endif'.

- E 5 no source modules

You must specify at least one source file to compile.

- F 6 cannot create "*file*"

The output file or temporary file could not be created. Check if you have sufficient disk space and if you have write permissions in the specified directory.

- F 7 cannot open "*file*"

Check if the file you specified really exists. Maybe you misspelled the name, or the file is in another directory.

- F 8 attempt to overwrite input file "*file*"

The output file must have a different name than the input file.

E 9 unterminated constant character or string

This error can occur when you specify a string without a closing double-quote (") or when you specify a character constant without a closing single-quote ('). This error message is often preceded by one or more E 19 error messages.

F 11 file stack overflow

This error occurs if the maximum nesting depth (50) of file inclusion is reached. Check for #include files that contain other #include files. Try to split the nested files into simpler files.

F 12 memory allocation error

All free space has been used. Free up some memory by removing any resident programs, divide the file into several smaller source files, break expressions into smaller subexpressions or put in more memory.

W 13 prototype after forward call or old style declaration – ignored

Check that a prototype for each function is present before the actual call.

E 14 ';' inserted

An expression statement needs a semicolon. For example, after ++i in { int i; ++i }.

E 15 missing filename after -o option

The **-o** option must be followed by an output filename.

E 16 bad numerical constant

A constant must conform to its syntax. For example, 08 violates the octal digit syntax. Also, a constant may not be too large to be represented in the type to which it was assigned. For example, `int i = 0x1234567890;` is too large to fit in an integer.

E 17 string too long

This error occurs if the maximum string size (1500) is reached. Reduce the size of the string.

E 18 illegal character (*0xbexnumber*)

The character with the hexadecimal ASCII value *0xbexnumber* is not allowed here. For example, the '#' character, with hexadecimal value 0x23, to be used as a preprocessor command, may not be preceded by non-white space characters. The following is an example of this error:

```
char *s = #S ; // error
```

E 19 newline character in constant

The newline character can appear in a character constant or string constant only when it is preceded by a backslash (\). To break a string that is on two lines in the source file, do one of the following:

- End the first line with the line-continuation character, a backslash (\).
- Close the string on the first line with a double quotation mark, and open the string on the next line with another quotation mark.

E 20 empty character constant

A character constant must contain exactly one character. Empty character constants (' ') are not allowed.

E 21 character constant overflow

A character constant must contain exactly one character. Note that an escape sequence (for example, \t for tab) is converted to a single character.

E 22 '#define' without valid identifier

You have to supply an identifier after a '#define'.

E 23 '#else' without '#if'

'#else' can only be used within a corresponding '#if', '#ifdef' or '#ifndef' construct. Make sure that there is a '#if', '#ifdef' or '#ifndef' statement in effect before this statement.

E 24 '#endif' without matching '#if'

'#endif' appeared without a matching '#if', '#ifdef' or '#ifndef' preprocessor directive. Make sure that there is a matching '#endif' for each '#if', '#ifdef' and '#ifndef' statement.

E 25 missing or zero line number

'#line' requires a non-zero line number specification.

E 26 undefined control

A control line (line with a '*#identifier*') must contain one of the known preprocessor directives.

W 27 unexpected text after control

'*#ifdef*' and '*#ifndef*' require only one identifier. Also, '*#else*' and '*#endif*' only have a newline. '*#undef*' requires exactly one identifier.

W 28 empty program

The source file must contain at least one external definition. A source file with nothing but comments is considered an empty program.

E 29 bad '*#include*' syntax

A '*#include*' must be followed by a valid header name syntax. For example, *#include <stdio.h* misses the closing '>'.

E 30 include file "*file*" not found

Be sure you have specified an existing include file after a '*#include*' directive. Make sure you have specified the correct path for the file.

E 31 end-of-file encountered inside comment

The compiler found the end of a file while scanning a comment. Probably a comment was not terminated. Do not forget a closing comment **/* when using ANSI-C style comments.

E 32 argument mismatch for macro "*name*"

The number of arguments in invocation of a function-like macro must agree with the number of parameters in the definition. Also, invocation of a function-like macro requires a terminating *)* token. The following are examples of this error:

```
#define A(a) 1
int i = A(1,2); /* error */

#define B(b) 1
int j = B(1;    /* error */
```

E 33 *"name"* redefined

The given identifier was defined more than once, or a subsequent declaration differed from a previous one. The following examples generate this error:

```
int i;
char i;          /* error */
main()
{
}

main()
{
    int j;
    int j;        /* error */
}
```

W 34 illegal redefinition of macro *"name"*

A macro can be redefined only if the body of the redefined macro is exactly the same as the body of the originally defined macro.

This warning can be caused by defining a macro on the command line and in the source with a `#define` directive. It also can be caused by macros imported from include files. To eliminate the warning, either remove one of the definitions or use an `#undef` directive before the second definition.

E 35 bad filename in `#line`

The string literal of a `#line` (if present) may not be a "wide-char" string. So, `#line 9999 L"t45.c"` is not allowed.

W 36 'debug' facility not installed

`#pragma debug` is only allowed in the debug version of the compiler.

W 37 attempt to divide by zero

A divide or modulo by zero was found. Adjust the expression or test if the second operand of a divide or modulo is zero.

E 38 +non integral switch expression

A `switch` condition expression must evaluate to an integral value. So, `char *p = 0; switch (p)` is not allowed.

F 39 unknown error number: *number*

This error may not occur. If it does, contact your local TASKING office and provide them with the exact error message.

W 40 non-standard escape sequence

Check the spelling of your escape sequence (a backslash, \, followed by a number or letter), it contains an illegal escape character. For example, \c causes this warning.

E 41 `#elif` without `#if`

The `#elif` directive did not appear within an `#if`, `#ifdef` or `#ifndef` construct. Make sure that there is a corresponding `#if`, `#ifdef` or `#ifndef` statement in effect before this statement.

E 42 syntax error, expecting parameter type/declaration/statement

A syntax error occurred in a parameter list a declaration or a statement. This can have many causes, such as, errors in syntax of numbers, usage of reserved words, operator errors, missing parameter types, missing tokens.

E 43 unrecoverable syntax error, skipping to end of file

The compiler found an error from which it could not recover. This error is in most cases preceded by another error. Usually, error E 42.

I 44 in initializer "*name*"

Informational message when checking for a proper constant initializer.

E 46 cannot hold that many operands

The value stack may not exceed 20 operands.

E 47 missing operator

An operator was expected in the expression.

E 48 missing right parenthesis

)' was expected.

W 49 attempt to divide by zero – potential run-time error

An expression with a divide or modulo by zero was found. Adjust the expression or test if the second operand of a divide or modulo is zero.

E 50 missing left parenthesis

(' was expected.

- E 51 cannot hold that many operators
The state stack may not exceed 20 operators.
- E 52 missing operand
An operand was expected.
- E 53 missing identifier after 'defined' operator
An identifier is required in a `#if defined(identifier)`.
- E 54 +non scalar controlling expression
Iteration conditions and 'if' conditions must have a scalar type (not a struct, union or a pointer). For example, after `static struct {int i;} si = {0};` it is not allowed to specify `while (si) ++si.i;`.
- E 55 operand has not integer type
The operand of a '#if' directive must evaluate to an integral constant. So, `#if 1.` is not allowed.
- W 56 '<*debugoption*><*level*>' no associated action
This warning can only appear in the debug version of the compiler. There is no associated debug action with the specified debug option and level.
- W 58 invalid warning number: *number*
The warning number you supplied to the `-w` option does not exist. Replace it with the correct number.
- F 59 sorry, more than *number* errors
Compilation stops if there are more than 40 errors.
- E 60 label "*label*" multiple defined
A label can be defined only once in the same function. The following is an example of this error:

```
f()  
{  
  lab1:  
  lab1:      /* error */  
}
```

E 61 type clash

The compiler found conflicting types. For example, a `long` is only allowed on `int` or `double`, no specifiers are allowed with `struct`, `union` or `enum`. The following is an example of this error:

```
unsigned signed int i;    /* error */
```

E 62 bad storage class for "*name*"

The storage class specifiers `auto` and `register` may not appear in declaration specifiers of external definitions. Also, the only storage class specifier allowed in a parameter declaration is `register`.

E 63 "*name*" redeclared

The specified identifier was already declared. The compiler uses the second declaration. The following is an example of this error:

```
struct T { int i; };
struct T { long j; };    /* error */
```

E 64 incompatible redeclaration of "*name*"

The specified identifier was already declared. All declarations in the same function or module that refer to the same object or function must specify compatible types. The following is an example of this error:

```
f()
{
    int i;
    char i;    /* error */
}
```

W 66 function "*name*": variable "*name*" not used

A variable is declared which is never used. You can remove this unused variable or you can use the **-w66** option to suppress this warning.

W 67 illegal suboption: *option*

The suboption is not valid for this option. Check the invocation syntax for a list of all available suboptions.

W 68 function "*name*": parameter "*name*" not used

A function parameter is declared which is never used. You can remove this unused parameter or you can use the **-w68** option to suppress this warning.

- E 69 declaration contains more than one basic type specifier

Type specifiers may not be repeated. The following is an example of this error:

```
int char i;      /* error */
```

- E 70 +‘break’ outside loop or switch

A break statement may only appear in a switch or a loop (do, for or while). So, `if (0) break;` is not allowed.

- E 71 illegal type specified

The type you specified is not allowed in this context. For example, you cannot use the type void to declare a variable. The following is an example of this error:

```
void i;          /* error */
```

- W 72 duplicate type modifier

Type qualifiers may not be repeated in a specifier list or qualifier list. The following is an example of this warning:

```
{ long long i; }    /* error */
```

- E 73 object cannot be bound to multiple memories

Use only one memory attribute per object. For example, specifying both rom and ram to the same object is not allowed.

- E 74 declaration contains more than one class specifier

A declaration may contain at most one storage class specifier. So, `register auto i;` is not allowed.

- E 75 +‘continue’ outside a loop

continue may only appear in a loop body (do, for or while). So, `switch (i) {default: continue;}` is not allowed.

- E 76 duplicate macro parameter “name”

The given identifier was used more than one in the format parameter list of a macro definition. Each macro parameter must be uniquely declared.

- E 77 parameter list should be empty

An identifier list, not part of a function definition, must be empty. For example, `int f (i, j, k);` is not allowed on declaration level.

- E 78 'void' should be the only parameter

Within a function prototype of a function that does not except any arguments, `void` may be the only parameter. So, `int f(void, int);` is not allowed.

- E 79 +constant expression expected

A constant expression may not contain a comma. Also, the bit field width, an expression that defines an enum, array-bound constants and `switch` case expressions must all be integral constant expressions.

- E 80 '#' operator shall be followed by macro parameter

The '#' operator must be followed by a macro argument.

- E 81 '##' operator shall not occur at beginning or end of a macro

The '##' (token concatenation) operator is used to paste together adjacent preprocessor tokens, so it cannot be used at the beginning or end of a macro body.

- W 86 escape character truncated to 8 bit value

The value of a hexadecimal escape sequence (a backslash, \, followed by a 'x' and a number) must fit in 8 bits storage. The number of bits per character may not be greater than 8. The following is an example of this warning:

```
char c = '\xabc';      /* error */
```

- E 87 concatenated string too long

The resulting string was longer than the limit of 1500 characters.

- W 88 "name" redeclared with different linkage

The specified identifier was already declared. This warning is issued when you try to redeclare an object with a different basic storage class, and both objects are not declared `extern` or `static`. The following is an example of this warning:

```
int i;
int i();    /* error E 64 and warning */
```

- E 89 illegal bitfield declarator

A bit field may only be declared as an integer, not as a pointer or a function for example. So, `struct {int *a:1;} s;` is not allowed.

E 90 *#error message*

The *message* is the descriptive text supplied in a *'#error'* preprocessor directive.

W 91 no prototype for function "*name*"

Each function should have a valid function prototype.

W 92 no prototype for indirect function call

Each function should have a valid function prototype.

I 94 hiding earlier one

Additional message which is preceded by error E 63. The second declaration will be used.

F 95 protection error: *message*

Something went wrong with the protection key initialization. The message could be: "Key is not present or printer is not correct.", "Can't read key.", "Can't initialize key.", or "Can't set key-model".

E 96 syntax error in *#define*

#define id(requires a right-parenthesis *)*'.

E 97 "... " incompatible with old-style prototype

If one function has a parameter type list and another function, with the same name, is an old-style declaration, the parameter list may not have ellipsis. The following is an example of this error:

```
int f(int, ...);  
int f();          /* error, old-style */
```

E 98 function type cannot be inherited from a typedef

A typedef cannot be used for a function definition. The following is an example of this error:

```
typedef int INTFN();  
INTFN f {return (0);} /* error */
```

F 99 conditional directives nested too deep

'#if', *'#ifdef'* or *'#ifndef'* directives may not be nested deeper than 50 levels.

- E 100 +case or default label not inside switch

The `case:` or `default:` label may only appear inside a `switch`.

- E 101 vacuous declaration

Something is missing in the declaration. The declaration could be empty or an incomplete statement was found. You must declare array declarators and `struct`, `union`, or `enum` members. The following are examples of this error:

```
int ;                                /* error */

static int a[2] = { };              /* error */
```

- E 102 +duplicate case or default label

Switch case values must be distinct after evaluation and there may be at most one `default:` label inside a `switch`.

- E 103 may not subtract pointer from scalar

The only operands allowed on subtraction of pointers is pointer – pointer, or pointer – scalar. So, scalar – pointer is not allowed. The following is an example of this error:

```
int i;
int *pi = &i;
ff(1 - pi);                          /* error */
```

- E 104 left operand of *operator* has not struct/union type

The first operand of a `'.'` or `'->'` must have a `struct` or `union` type.

- E 105 zero or negative array size – ignored

Array bound constants must be greater than zero. So, `char a[0];` is not allowed.

- E 106 different constructors

Compatible function types with parameter type lists must agree in number of parameters and in use of ellipsis. Also, the corresponding parameters must have compatible types. This error is usually followed by informational message I 111. The following is an example of this error:

```
int f(int);
int f(int, int);                      /* error different
                                     parameter list */
```

E 107 different array sizes

Corresponding array parameters of compatible function types must have the same size. This error is usually followed by informational message I 111. The following is an example of this error:

```
int f(int[][2]);  
int f(int[][3]);          /* error */
```

E 108 different types

Corresponding parameters must have compatible types and the type of each prototype parameter must be compatible with the widened definition parameter. This error is usually followed by informational message I 111. The following is an example of this error:

```
int f(int);  
int f(long);              /* error different type  
                           in parameter list */
```

E 109 floating point constant out of valid range

A floating point constant must have a value that fits in the type to which it was assigned. See section *Data Types* for the valid range of a floating point constant. The following is an example of this error:

```
float d = 10E9999;        /* error, too big */
```

E 110 function cannot return arrays or functions

A function may not have a return type that is of type array or function. A pointer to a function is allowed. The following are examples of this error:

```
typedef int F(); F f();   /* error */  
  
typedef int A[2]; A g(); /* error */
```

I 111 parameter list does not match earlier prototype

Check the parameter list or adjust the prototype. The number and type of parameters must match. This message is preceded by error E 106, E 107 or E 108.

E 112 parameter declaration must include identifier

If the declarator is a prototype, the declaration of each parameter must include an identifier. Also, an identifier declared as a `typedef` name cannot be a parameter name. The following are examples of this error:

```
int f(int g, int) {return (g);} /* error */

typedef int int_type;
int h(int_type) {return (0);} /* error */
```

E 114 incomplete struct/union type

The `struct` or `union` type must be known before you can use it. The following is an example of this error:

```
extern struct unknown sa, sb;
sa = sb; /* 'unknown' does not have a
          defined type */
```

The left side of an assignment (the lvalue) must be modifiable.

E 115 label "*name*" undefined

A `goto` statement was found, but the specified label did not exist in the same function or module. The following is an example of this error:

```
f1() { a: ; } /* W 116 */
f2() { goto a; } /* error, label 'a:' is
                  not defined in f2() */
```

W 116 label "*name*" not referenced

The given label was defined but never referenced. The reference of the label must be within the same function or module. The following is an example of this warning:

```
f() { a: ; } /* 'a' is not referenced */
```

E 117 "*name*" undefined

The specified identifier was not defined. A variable's type must be specified in a declaration before it can be used. This error can also be the result of a previous error. The following is an example of this error:

```
unknown i; /* error, 'unknown' undefined */
i = 1; /* as a result, 'i' is also
        undefined */
```

W 118 constant expression out of valid range

A constant expression used in a case label may not be too large. Also when converting a floating point value to an integer, the floating point constant may not be too large. This warning is usually preceded by error E 16 or E 109. The following is an example of this warning:

```
int i = 10E88;    /* error and warning */
```

E 119 cannot take 'sizeof' bitfield or void type

The size of a bit field or void type is not known. So, the size of it cannot be taken.

E 120 cannot take 'sizeof' function

The size of a function is not known. So, the size of it cannot be taken.

E 121 not a function declarator

This is not a valid function. This may be due to a previous error. The following is an example of this error:

```
int f() return 0;    /* missing '{ }' */
int g() { }         /* error, 'g' is not a
                     formal parameter and
                     therefore, this is not a
                     valid function declaration */
```

E 122 unnamed formal parameter

The parameter must have a valid name.

W 123 function should return something

A return in a non-void function must have an expression.

E 124 array cannot hold functions

An array of functions is not allowed.

E 125 +function cannot return anything

A return with an expression may not appear in a void function.

W 126 missing return (function "name")

A non-void function with a non-empty function body must have a return statement.

E 129 cannot initialize "*name*"

Declarators in the declarator list may not contain initializations. Also, an `extern` declaration may have no initializer. The following are examples of this error:

```
{ extern int i = 0; }      /* error */
int f( i ) int i=0;        /* error */
```

W 130 operands of *operator* are pointers to different types

Pointer operands of an operator or assignment (`=`), must have the same type. For example, the following code generates this warning:

```
long *pl;
int *pi = 0;
pl = pi;      /* warning */
```

E 131 bad operand type(s) of *operator*

The operator needs an operand of another type. The following is an example of this error:

```
int *pi;
pi += 1.;      /* error, pointer on left; needs
                integral value on right */
```

W 132 value of variable "*name*" is undefined

This warning occurs if a variable is used before it is defined. For example, the following code generates this warning:

```
int a,b;
a = b;        /* warning, value of b unknown */
```

E 133 illegal struct/union member type

A function cannot be a member of a `struct` or `union`. Also, bit fields may only have type `int` or `unsigned`.

E 134 bitfield size out of range – set to 1

The bit field width may not be greater than the number of bits in the type and may not be negative. The following example generates this error:

```
struct i { unsigned i : 999; }; /* error */
```

W 135 statement not reached

The specified statement will never be executed. This is for example the case when statements are present after a `return`.

E 138 illegal function call

You cannot perform a function call on an object that is not a function. The following example generates this error:

```
int i, j;
j = i();      /* error, i is not a function */
```

E 139 *operator* cannot have aggregate type

The type name in a (cast) must be a scalar (not a `struct`, union or a pointer) and also the operand of a (cast) must be a scalar. The following are examples of this error:

```
static union ui {int a;} ui ;
ui = (union ui)9;      /* cannot cast to union */
ff( (int)ui );          /* cannot cast a union
                        to something else */
```

E 140 *type* cannot be applied to a register/bit/bitfield object or builtin/inline function

For example, the `'&'` operator (address) cannot be used on registers and bit fields. So, `func(&r6);` and `func(&bitf.a);` are invalid.

E 141 *operator* requires modifiable lvalue

The operand of the `'++'`, or `'--'` operator and the left operand of an assignment or compound assignment (lvalue) must be modifiable. The following is an example of this error:

```
const int i = 1;
i = 3;          /* error, const cannot be
                modified */
```

E 143 too many initializers

There may be no more initializers than there are objects. The following is an example of this error:

```
static int a[1] = {1, 2}; /* error,
                        only one object can be initialized */
```

- W 144 enumerator "*name*" value out of range

An enum constant exceeded the limit for an int. The following is an example of this warning:

```
enum { A = INT_MAX, B }; /* warning,
                          B does not fit in an int anymore */
```

- E 145 requires enclosing curly braces

A complex initializer needs enclosing curly braces. For example, `int a[] = 2;` is not valid, but `int a[] = {2};` is.

- E 146 argument *#number*: memory spaces do not match

With prototypes, the memory spaces of arguments must match.

- W 147 argument *#number*: different levels of indirection

With prototypes, the types of arguments must be assignment compatible. The following code generates this warning:

```
int i; void func(int,int);
func( 1, &i ); /* warning, argument 2 */
```

- W 148 argument *#number*: struct/union type does not match

With prototypes, both the prototyped function argument and the actual argument was a struct or union., but they have different tags. The tag types should match. The following is an example of this warning:

```
f(struct s); /* prototype */
main()
{
    struct { int i; } t;
    f( t ); /* t has other type than s */
}
```

- E 149 object "*name*" has zero size

A struct or union may not have a member with an incomplete type. The following is an example of this error:

```
struct { struct unknown m; } s; /* error */
```

- W 150 argument *#number*: pointers to different types

With prototypes, the pointer types of arguments must be compatible. The following example generates this warning:

```
int f(int*);
long *l;
f(l);           /* warning */
```

- W 151 ignoring memory specifier

Memory specifiers for a `struct`, `union` or `enum` are ignored.

- E 152 operands of *operator* are not pointing to the same memory space

Be sure the operands point to the same memory space. This error occurs, for example, when you try to assign a pointer to a pointer from a different memory space.

- E 153 'sizeof' zero sized object

An implicit or explicit `sizeof` operation references an object with an unknown size. This error is usually preceded by error E 119 or E 120, cannot take 'sizeof'.

- E 154 argument *#number*: struct/union mismatch

With prototypes, only one of the prototyped function argument or the actual argument was a `struct` or `union`. The types should match. The following is an example of this error:

```
f(struct s);           /* prototype */

main()
{
    int i;
    f(i);               /* i is not a struct */
}
```

- E 155 casting lvalue 'type' to 'type' is not allowed

The operand of the '++', or '--' operator or the left operand of an assignment or compound assignment (lvalue) may not be cast to another type. The following is an example of this error:

```
int i = 3;
++(unsigned)i;         /* error, cast expression
                        is not an lvalue */
```

- E 157 *"name"* is not a formal parameter

If a declarator has an identifier list, only its identifiers may appear in the declarator list. The following is an example of this error:

```
int f( i ) int a; /* error */
```

- E 158 right side of *operator* is not a member of the designated struct/union

The second operand of '.' or '->' must be a member of the designated struct or union.

- E 160 pointer mismatch at *operator*

Both operands of *operator* must be a valid pointer. The following example generates this error:

```
int *pi = 44; /* right side not a pointer */
```

- E 161 aggregates around *operator* do not match

The contents of the structs, unions or arrays on both sides of the *operator* must be the same. The following example causes this error:

```
struct {int a; int b;} s;
struct {int c; int d; int e;} t;
s = t; /* error */
```

- E 162 *operator* requires an lvalue or function designator

The '&' (address) operator requires an lvalue or function designator. The following is an example of this error:

```
int i;
i = &( i = 0 );
```

- W 163 operands of *operator* have different level of indirection

The types of pointers or addresses of the operator must be assignment compatible. The following is an example of this warning:

```
char **a;
char *b;
a = b; /* warning */
```

- E 164 operands of *operator* may not have type 'pointer to void'

The operands of *operator* may not have operand (void *).

- W 165 operands of *operator* are incompatible: pointer vs. pointer to array

The types of pointers or addresses of the operator must be assignment compatible. A pointer cannot be assigned to a pointer to array. The following is an example of this warning:

```
main()
{
    typedef int array[10];
    array a;
    array *ap = a;      /* warning */
}
```

- E 166 *operator* cannot make something out of nothing

Casting type void to something else is not allowed. The following example generates this error:

```
void f(void);
main()
{
    int i;

    i = (int)f();      /* error */
}
```

- E 170 recursive expansion of inline function "*name*"

An `_inline` function may not be recursive. The following example generates this error:

```
_inline int a (int i)
{
    a(i);              /* recursive call */
    return i;
}
main()
{
    a(1);              /* error */
}
```


- E 171 +too much tail-recursion in inline function "*name*"

If the function level is greater than or equal to 40 this error is given. The following example generates this error:

```
_inline void a ()
{
    a();
}
main()
{
    a();
}
```

- W 172 adjacent strings have different types

When concatenating two strings, they must have the same type. The following example generates this warning:

```
char b[] = L"abc""def";    /* strings have
                             different types */
```

- E 173 'void' function argument

A function may not have an argument with type void.

- E 174 not an address constant

A constant address was expected. Unlike a static variable, an automatic variable does not have a fixed memory location and therefore, the address of an automatic is not a constant. The following is an example of this error:

```
int *a;
static int *b = a; /* error */
```

- E 175 not an arithmetic constant

In a constant expression no assignment operators, no '++' operator, no '--' operator and no functions are allowed. The following is an example of this error:

```
int a;
static int b = a++; /* error */
```

- E 176 address of automatic is not a constant

Unlike a static variable, an automatic variable does not have a fixed memory location and therefore, the address of an automatic is not a constant. The following is an example of this error:

```
int a;                /* automatic */
static int *b = &a; /* error */
```

W 177 static variable "*name*" not used

A static variable is declared which is never used. To eliminate this warning remove the unused variable.

W 178 static function "*name*" not used

A static function is declared which is never called. To eliminate this warning remove the unused function.

E 179 +inline function "*name*" is not defined

Possibly only the prototype of the inline function was present, but the actual inline function was not. The following is an example of this error:

```
_inline int a(void); /* prototype */

main()
{
    int b;
    b = a();          /* error */
};
```

E 180 illegal target memory (*memory*) for pointer

The pointer may not point to *memory*. For example, a pointer to bitaddressable memory is not allowed.

E 181 invalid cast to function

A cast to type function is not allowed. A cast to a function pointer type is allowed.

W 182 argument *#number*: different types

With prototypes, the types of arguments must be compatible.

- W 183 variable '*name*' possibly uninitialized

Possibly an initialization statement is not reached, while you tried to use the variable. The following is an example of this warning:

```
int a;

int f(void)
{
    int i;

    if ( a )
    {
        i = 0; /* statement not reached */
    }
    return i; /* warning */
}
```

- I 185 (prototype synthesized at line *number* in "*name*")

This is an informational message containing the source file position where an old-style prototype was synthesized. This message is preceded by error E 146, W 147, W 148, W 150, E 154, W 182 or E 203.

- E 186 array of type bit is not allowed

An array cannot contain bit type variables.

- E 187 illegal structure definition

A structure can only be defined (initialized) if its members are known. So, `struct unknown s = { 0 };` is not allowed.

- E 188 structure containing bit-type fields is forced into bitaddressable area

This error occurs when you use a bitaddressable storage type for a structure containing bit-type members.

- E 189 pointer is forced to bitaddressable, pointer to bitaddressable is illegal

A pointer to bitaddressable memory is not allowed.

- W 190 "long float" changed to "float"

In ANSI C floating point constants are treated having type double, unless the constant has the suffix 'f'. If you have specified an option to use float constants, a long floating point constant such as `123.12f1` is changed to a float.

E 191 recursive struct/union definition

A `struct` or `union` cannot contain itself. The following example generates this error:

```
struct s { struct s a; } b;      /* error */
```

E 192 missing filename after `-f` option

The `-f` option requires a filename argument.

E 193 only one `-f` option allowed

You can use the `-f` option only once.

E 194 cannot initialize typedef

You cannot assign a value to a typedef variable. So, `typedef i=2;` is not allowed.

W 195 constant expression out of range — truncated

The resulting constant expression is too large to fit in the specified data type. The value is truncated. The following example generates this warning:

```
int i = 140000L;      /* warning, value is too large
                      to fit in an int */
```

W 196 constant expression out of range due to signed/unsigned type mismatch

The resulting constant expression is too large to fit in the specified data type. The following example generates this warning:

```
int i = 40000U; /* the unsigned value is too large
                to fit in a signed int */
/* unsigned int i = 40000U; is OK */
```

W 197 unrecognized `-w` argument: *argument*

The `-w` option only accepts a warning number or the text 'strict' as an argument. See the description of the `-w` option for details.

W 198 trigraph sequence replaced

Trigraphs are used in the C language to create special characters on obsolete terminals with a limited character set. When they are replaced in your source, e.g. in a string, they may give rise to very obscure errors.

F 199 demonstration package limits exceeded

The demonstration package has certain limits which are not present in the full version. Contact TASKING for a full version.

W 200 unknown pragma – ignored

The compiler ignores pragmas that are not known. For example, `#pragma unknown`.

W 201 *name* cannot have storage type – ignored

A register variable or an automatic/parameter cannot have a storage type. To eliminate this warning, remove the storage type or place the variable outside a function.

E 202 '*name*' is declared with 'void' parameter list

You cannot call a function with an argument when the function does not accept any (void parameter list). The following is an example of this error:

```
int f(void);          /* void parameter list */

main()
{
    int i;
    i = f(i);         /* error */
    i = f();          /* OK */
}
```

E 203 too many/few actual parameters

With prototyping, the number of arguments of a function must agree with the prototype of the function. The following is an example of this error:

```
int f(int);          /* one parameter */

main()
{
    int i;
    i = f(i,i);       /* error, one too many */
    i = f(i);         /* OK */
}
```

W 204 U suffix not allowed on floating constant – ignored

A floating point constant cannot have a 'U' or 'u' suffix.

- W 205 F suffix not allowed on integer constant – ignored
An integer constant cannot have a 'F' or 'f' suffix.
- E 206 '*name*' named bit-field cannot have 0 width
A bit field must be an integral constant expression with a value greater than zero.
- E 207 list of rule numbers expected after "-safer" option
Add the numbers of the Safer C rules to the -safer option to specify the rules that must be checked. See Appendix B *Safer C*
- W 208 unsupported Safer C rule number *number*.
Specified Safer C rule number is not supported.
- E 209 +Safer C rule *number* violation: *rule_description*
A specified Safer C rule is violated.
- E 212 "*name*": missing static function definition
A function with a `static` prototype misses its definition.
- W 213 invalid string/character constant in non-active part of source
This part of the source is skipped.
- E 214 second occurrence of `#pragma asm` or `asm_noflush`
`#pragma asm/#pragma endasm` blocks cannot be nested. Use `#pragma endasm` before starting a new `#pragma asm/#pragma endasm` block.
- E 215 "`#pragma endasm`" without a "`#pragma asm`"
A `#pragma endasm` must always have a corresponding `#pragma asm` or `#pragma asm_noflush`.
- W 216 suggest parentheses around assignment used as truth value
Generated when the argument of an `if` statement is actually an assignment (might indicate a typing error).

W 303 variable '*name*' possibly uninitialized

Possibly an initialization statement is not reached, while you tried to use the variable. The following is an example of this warning:

```
int a;

int f(void)
{
    int i;

    if ( a )
    {
        i = 0; /* statement not reached */
    }
    return i; /* warning */
}
```

E 327 too many arguments to pass in registers for `_asmfunc` '*name*'

An `_asmfunc` function uses a fixed register-based interface between C and assembly, but the number of arguments that can be passed is limited by the number of available registers. With function *name* this limit was reached.

Backend

W 507 duplicate qualifier

Only one function qualifier is allowed. Duplicate function qualifiers are ignored.

W 510 function qualifier used on non-function

A function qualifier can only be used on functions.

E 511 interrupt function must have void result and void parameter list

A function declared with `_interrupt(n)` may not accept any arguments and may not return anything.

W 512 (*number*) not within valid range (*num1,num2*)

An interrupt vector number must be in the range *num1* to *num2*. Any other number is illegal.

E 514 conflicting '*name*' attribute from previous definition

The attributes of the current function qualifier declaration and the previous function qualifier declaration are not the same.

- E 515 difference in vector number (*old_num* != *new_num*) from previous declaration
The function prototype of an interrupt service routine must have the same vector number as in the function definition.
- E 516 '*memory_type*' is illegal memory for function
The storage type is not valid for this function.
- W 517 address would be truncated
This warning is issued when pointer conversion is needed, for example, when you assign a `_far` pointer to a `_near` pointer.
- E 526 function qualifier '`_asmfunc`' not allowed in function definition
`_asmfunc` is only allowed in the function prototype.
- E 528 `_at()` requires a numerical address
You can only use an expression that evaluates to a numerical address.
- E 529 `_at()` address out of range for this type of object
The absolute address is not present in the specified memory space.
- E 530 `_at()` only valid for global variables
Only global variables can be placed on absolute addresses.
- E 531 `_at()` only allowed for uninitialized variables
Absolute variables cannot be initialized.
- E 532 `_at()` has no effect on external declaration
When declared `extern` the variable is not allocated by the compiler.
- W 533 language extension keyword used as identifier
A language extension keyword is a reserved word, and reserved words cannot be used as an identifier.
- W 536 truncating the value to: *hexnumber*
The value does not fit in the specified storage class and is truncated.
- W 542 optimization stack underflow, no optimization options are saved with `#pragma optimize`
This warning occurs if you use a `#pragma endoptimize` while there were no options saved by a previous `#pragma optimize`.

- E 562 Bit type parameter not allowed
Parameters cannot be of type `_bit`.
- E 563 Bit-type struct member not allowed
Struct members cannot be of type `_bit`.
- E 566 special function registers may not be initialized
For example, the construction `_sfrbyte a = 2;` is not allowed.
- E 567 operand of '*memory_type*' must be a constant
The operand from an intrinsic function should specify an immediate constant value.

Example: `int i; -enter (i);` The `-enter` mnemonic requires a constant number.
- E 568 `_at()` only valid for global variables
Only global variables can be placed on absolute addresses. Only global variables can be placed on absolute addresses.
- E 569 Bit offset out of range
With `_atbit` you can specify a bit offset within `_sfrbyte` from 0 to 7, within `_sfrword` from 0 to 15 and within `_sfrlong` from 0 to 31.

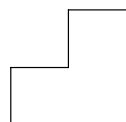
CHAPTER

6

LIBRARIES



TASKING



6

CHAPTER

6.1 INTRODUCTION

This chapter describes the library functions delivered with the compiler. Some functions (e.g. `printf()`, `scanf()`) can be edited to match your needs. **cm16** come with libraries in object format per memory model and with header files containing the appropriate prototype of the library functions. The library functions are also shipped in source code (C or assembly).

A number of standard operations within C are too complex to generate inline code for. These operations are implemented as run-time library functions. The run-time library routines are added to the C library.

6.2 HEADER FILES

The following header files are delivered with the C compiler:

- <assert.h>** `assert`
- <ctype.h>** `isalnum`, `isalpha`, `isascii`, `isctrl`, `isdigit`, `isgraph`, `islower`, `isprint`, `ispunct`, `isspace`, `isupper`, `isxdigit`, `toascii`, `_tolower`, `tolower`, `_toupper`, `toupper`
- <errno.h>** Error numbers. No C functions.
- <fcntl.h>** Definition of flags used by `open()`.
- <float.h>** `copysign`, `copysignf`, `isfinite`, `isfinitef`, `isinf`, `isinff`, `isnan`, `isnan`, `scalbf`. Constants related to floating point arithmetic.
- <limits.h>** Limits and sizes of integral types. No C functions.
- <locale.h>** `localeconv`, `setlocale`. Delivered as skeletons.
- <malloc.h>** Non-ANSI C header file with prototypes of `malloc` and `free`.
- <math.h>** `acos`, `asin`, `atan`, `atan2`, `ceil`, `cos`, `cosh`, `exp`, `fabs`, `floor`, `fmod`, `frexp`, `ldexp`, `log`, `log10`, `modf`, `pow`, `sin`, `sinh`, `sqrt`, `tan`, `tanh`
- <setjmp.h>** `longjmp`, `setjmp`
- <signal.h>** `raise`, `signal`. Functions are delivered as skeletons.
- <simio.h>** `_simi`, `_simo`
- <stdarg.h>** `va_arg`, `va_end`, `va_start`

- <stddef.h> offsetof, definition of special types.
- <stdio.h> clearerr, _close, fclose, feof, ferror, fflush, fgetc, fgetpos, fgets, fopen, fprintf, fputc, fputs, fread, freopen, fscanf, fseek, fsetpos, ftell, fwrite, getc, getchar, gets, _lseek, _open, perror, printf, putc, putchar, puts, _read, remove, rename, rewind, scanf, setbuf, setvbuf, sprintf, sscanf, tmpfile, tmpnam, ungetc, vfprintf, vprintf, vsprintf, _unlink, _write
- <stdlib.h> abort, abs, atexit, atof, atoi, atol, bsearch, calloc, div, exit, free, getenv, labs, ldiv, malloc, mblen, mbstowcs, mbtowc, qsort, rand, realloc, srand, strtod, strtol, strtoul, system, wcstombs, wctomb
- <string.h> memchr, memcmp, memcpy, memmove, memset, strcat, strchr, strcmp, strcol, strcpy, strcspn, sterror, strlen, strncat, strncmp, strncpy, strpbrk, strchr, strspn, strstr, strtok, strxfrm
- <time.h> asctime, clock, ctime, difftime, gmtime, localtime, mktime, strftime, time. All functions are delivered as skeletons.

6.3 C LIBRARIES

The C library contains C library functions. All C library functions are described in this chapter. These functions are only called by explicit function calls in your application program.

The lib directory contains the following libraries:

Compiler Model	Library to link
Small	libcs.a
Large	libcl.a

Table 6-1: C libraries

Compiler Model	Library to link
Small	libms.a
Large	libml.a

Table 6-2: Math libraries

Compiler Model	Library to link
Small	printfss.a printfsm.a scanfss.a
Large	printfs.a printfm.a scanf.s.a

Table 6-3: Printf and scanf libraries

Compiler Model	Library to link
Small	librts.a
Large	librtl.a

Table 6-4: Run-time libraries

Compiler Model	Library to Link
Small	libfps.a
Large	libfpl.a

Table 6-5: Floating point libraries



The **lkm16** linker is using this naming convention when specifying the **-l** option. For example, with **-lcs** the linker is looking for `libcs.a` in the system `lib` directory. Specifying the libraries is a job taken care of by the control program.

When you use floating point, the floating point library must be the last library linked. It should be placed after the C library. Arithmetic routines like `sin()`, `cos()`, etc. are not present in these libraries. Only basic floating point operations can be done.

6.3.1 SINGLE PRECISION FLOATING POINT

In ANSI C all mathematical functions (`<math.h>`), are based on `double` arguments and `double` return type. So, even if you are using only `float` variables in your code, the language definition dictates promotion to `double`, when using the math functions or floating point formatters (`printf()` and `scanf()`). The result is more code and less execution speed. In fact the ANSI approach introduces a performance penalty.

To improve the code size and execution speed, the compiler supports the option **-F** to force single precision floating point usage. If you use **-F**, a `float` variable passed as an argument is no longer promoted to `double` when calling a variable argument function or an old style K&R function, and the type `double` is treated as `float`. It is obvious that this affects the whole application (including libraries). Therefore, special single precision versions of the floating point libraries are delivered with the package. When using **-F**, these libraries must be used. It is not possible to mix C modules created with the **-F** option and C modules which are using the regular ANSI approach.

The **-Fc** option only treats floating point constants (having no suffix) as `float` instead of `double`.

The single precision floating point C libraries have an additional 's' in the filename.

6.3.2 C LIBRARY IMPLEMENTATION DETAILS

A detailed description of the delivered C library is shown in the following list.

Explanation :

- Y – Fully implemented
- I – Implemented, but needs some user written low level routine
- L – Delivered as a skeleton

File	Imple- mented	Routine name	Description / Reason
assert.h	Y	'assert()' macro	Macro definition
ctype.h	Y		Most of the routines are delivered as macro AND as function (as prescribed by ANSI).
	Y	isalnum	
	Y	isalpha	
	Y	iscntrl	
	Y	isdigit	
	Y	isgraph	
	Y	islower	
	Y	isprint	
	Y	ispunct	
	Y	isspace	
	Y	isupper	
	Y	isxdigit	
	Y	tolower	
	Y	toupper	
	Y	_tolower	Not defined by ANSI
	Y	_toupper	Not defined by ANSI
	Y	isascii	Not defined by ANSI
	Y	toascii	Not defined by ANSI
errno.h	Y		Only Macros
fcntl.h	Y		Definitions of flags used by _open
	I	open	
float.h	Y		
limits.h	Y		Only Macros
locale.h	Y		
	L	localeconv	No OS present
	L	setlocale	No OS present



File	Imple- mented	Routine name	Description / Reason
math.h	Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y	acos asin atan atan2 ceil cos cosh exp fabs floor fmod frexp ldexp log log10 modf pow sin sinh sqrt tan tanh	
setjmp.h	Y Y Y	longjmp setjmp	
signal.h	Y Y Y	raise signal	
stdarg.h	Y Y Y Y	va_arg va_end va_start	
stddef.h	Y		Only Macros

File	Imple- mented	Routine name	Description / Reason
stdio.h	Y		
	Y	clearerr	
	I	fclose	Needs _close
	Y	feof	
	Y	ferror	
	I	fflush	Needs _write/_lseek
	I	fgetc	Needs _read
	I	fgetpos	Needs _lseek
	I	fgets	Needs _read
	I	fopen	Needs _open
	I	fprintf	Needs _write
	I	fputc	Needs _write
	I	fputs	Needs _write
	I	fread	Needs _read
	I	freopen	Needs _close/_open
	I	fscanf	Needs _read
	I	fseek	Needs _lseek
	I	fsetpos	Needs _lseek
	I	ftell	Needs _lseek
	I	fwrite	Needs _write
	I	getc	Needs _read
	I	getchar	Needs _read
	I	gets	Needs _read
	Y	perror	
	I	printf	Needs _write
	I	putc	Needs _write
	I	putchar	Needs _write
	I	puts	Needs _write
	I	remove	Needs _unlink
	L	rename	
	I	rewind	Needs _lseek
	I	scanf	Needs _read
	Y	setbuf	
	Y	setvbuf	
	Y	sprintf	
	Y	sscanf	
	L	tmpfile	
	L	tmpnam	Delivered as a random name generator, but should use some process ID.
	Y	ungetc	
	I	vfprintf	Needs _write
	I	vprintf	Needs _write
	Y	vsprintf	

File	Imple- mented	Routine name	Description / Reason
	I I I I I I	_close _open _lseek _read _unlink _write	Low level file close routine Low level file open routine Low level file positioning routine Low level block input routine, when not customized, will use _simi Low level file remove routine Low level block write routine, when not customized, will use _simo
stdlib.h	Y Y Y Y Y Y Y Y Y Y Y L Y Y Y Y Y Y Y L L L L L L	abort abs atexit atof atoi atol bsearch calloc div exit free getenv labs ldiv malloc qsort strtod strtol strtoul rand realloc srand system mblen mbstowcs mbtowc wcstombs wctomb	Calls _exit() in cstart Calls _exit() in cstart No OS present No OS present wide chars not supported wide chars not supported wide chars not supported wide chars not supported wide chars not supported

[illegible]

6.3.3 C LIBRARY INTERFACE DESCRIPTION

_close

```
#include <stdio.h>
int _close( int fd );
```

Low level file close function. `_close` is used by the functions `close` and `fclose`. The given file descriptor should be properly closed, any buffer is already flushed.

_lseek

```
#include <stdio.h>
off_t _lseek( int fd, off_t offset, int whence );
```

Low level file positioning function. `_lseek` is used by all file positioning functions (`fgetpos`, `fseek`, `fsetpos`, `ftell`, `rewind`).

_open

```
#include <stdio.h>
int _open( int fd, int flags );
```

Low level file open function. `_open` is used by the functions `fopen` and `freopen`. The given file descriptor should be properly opened.

_read

```
#include <stdio.h>
size_t
_read( FILE *fin, char *base, size_t size );
```

Low level block input function. It reads a block of characters from the given stream. This function interfaces to CrossView Pro's simulated I/O feature.

Returns the number of characters read.

_simi

```
#include <simio.h>
int _simi( int stream, char *port, int len );
```

CrossView Simulated input interface function.



See also "_read()".

_simo

```
#include <simio.h>
int _simo( int stream, char *port, int len );
```

CrossView Simulated output interface function.



See also "_write()".

_tolower

```
#include <ctype.h>
int _tolower( int c );
```

Converts *c* to a lowercase character, does not check if *c* really is an uppercase character. This is a non-ANSI function.

Returns the converted character.

_toupper

```
#include <ctype.h>
int _toupper( int c );
```

Converts *c* to an uppercase character, does not check if *c* really is a lowercase character. This is a non-ANSI function.

Returns the converted character.

_unlink

```
#include <stdio.h>
int _unlink( const char *name );
```

Low level file remove function. `_unlink` is used by the function `remove`.

_write

```
#include <stdio.h>
size_t
_write( FILE *iop, char *base, size_t size );
```

Low level block output function. It writes a block of characters to the given stream. This function interfaces to CrossView Pro's simulated I/O feature.

Returns the number of characters correctly written.

abort

```
#include <stdlib.h>
void abort( void );
```

Terminates the program abnormally. It calls the function `_exit`, which is defined in the start-up module.

Returns nothing.

abs

```
#include <stdlib.h>
int abs( int n );
```

Returns the absolute value of the signed int argument.

acos

```
#include <math.h>
double acos( double x );
```

Returns the arccosine $\cos^{-1}(x)$ of x in the range $[0, \pi]$,
 $x \in [-1, 1]$.

asctime

```
#include <time.h>
char *asctime( const struct tm *tp );
```

Converts the time in the structure **tp* into a string of the form:

```
Mon Jan 21 16:15:14 1989\n\0
```

Returns the time in string form.

asin

```
#include <math.h>
double asin( double x );
```

Returns the arcsine $\sin^{-1}(x)$ of *x* in the range $[-\pi/2, \pi/2]$, $x \in [-1, 1]$.

assert

```
#include <assert.h>
void assert( int expr );
```

When compiled with NDEBUG, this is an empty macro. When compiled without NDEBUG defined, it checks if *expr* is true. If it is true, then a line like:

```
"Assertion failed: expression, file filename, line
num"
```

is printed.

Returns nothing.

atan

```
#include <math.h>
double atan( double x );
```

Returns the arctangent $\tan^{-1}(x)$ of *x* in the range $[-\pi/2, \pi/2]$. $x \in [-1, 1]$.

atan2

```
#include <math.h>
double atan2( double y, double x );
```

Returns the result of: $\tan^{-1}(y/x)$ in the range $[-\pi, \pi]$.

atexit

```
#include <stdlib.h>
int atexit( void (*fcn)( void ) );
```

Registers the function *fcn* to be called when the program terminates normally.

Returns zero, if program terminates normally.
non-zero, if the registration cannot be made.

atof

```
#include <stdlib.h>
double atof( const char *s );
```

Converts the given string to a double value. White space is skipped, conversion is terminated at the first unrecognized character.

Returns the double value.

atoi

```
#include <stdlib.h>
int atoi( const char *s );
```

Converts the given string to an integer value. White space is skipped, conversion is terminated at the first unrecognized character.

Returns the integer value.

atol

```
#include <stdlib.h>
long atol( const char *s );
```

Converts the given string to a long value. White space is skipped, conversion is terminated at the first unrecognized character.

Returns the long value.

bsearch

```
#include <stdlib.h>
_reentrant void *bsearch( const void *key,
    const void *base, size_t n, size_t size, int (* cmp)
    (const void *, const void *) );
```

This function searches in an array of *n* members, for the object pointed to by *ptr*. The initial base of the array is given by *base*. The size of each member is specified by *size*. The given array must be sorted in ascending order, according to the results of the function pointed to by *cmp*.

Returns a pointer to the matching member in the array, or NULL when not found.

calloc

```
#include <stdlib.h>
void *calloc( size_t nobj,
    size_t size );
```

The allocated space is filled with zeros. The maximum space that can be allocated can be changed by customizing the heap size (see the section *Heap*). By default no heap is allocated. When "calloc()" is used while no heap is defined, the locator gives an error.

Returns a pointer to space in external memory for *nobj* items of *size* bytes length.
NULL if there is not enough space left.

ceil

```
#include <math.h>
double ceil( double x );
```

Returns the smallest integer not less than *x*, as a double.

clearerr

```
#include <stdio.h>
void clearerr( FILE *stream );
```

Clears the end of file and error indicators for stream.

Returns nothing.

clock

```
#include <time.h>
clock_t clock( void );
```

Determines the processor time used.

Returns -1.

copysign

```
#include <float.h>
double copysign( double d, double sign );
```

IEEE-754-1985 Recommended function. Copy the sign of the second argument to the value of the first argument and return that as result.

Returns the first argument with the sign of the second argument.

copysignf

```
#include <float.h>
float copysignf( float f, float sign );
```

IEEE-754-1985 Recommended function. Copy the sign of the second argument to the value of the first argument and return that as result.

Returns the first argument with the sign of the second argument.

cos

```
#include <math.h>
double cos( double x );
```

Returns the cosine of *x*.

cosh

```
#include <math.h>
double cosh( double x );
```

Returns the hyperbolic cosine of *x*.

ctime

```
#include <time.h>
char *ctime( const time_t *tp );
```

Converts the calendar time **tp* into local time, in string form. This function is the same as:

```
asctime( localtime( tp ) );
```

Returns the local time in string form.

difftime

```
#include <time.h>
double
difftime( time_t time2, time_t time1 );
```

Returns the result of `time2 - time1` in seconds.

div

```
#include <stdlib.h>
div_t div( int num, int denom );
```

Both arguments are integers. The returned quotient and remainder are also integers.

Returns a structure containing the quotient and remainder of `num` divided by `denom`.

exit

```
#include <stdlib.h>
void exit( int status );
```

Terminates the program normally. Acts as if `'main()'` returns with `status` as the return value.

Returns zero, on successful termination.

exp

```
#include <math.h>
double exp( double x );
```

Returns the result of the exponential function e^x .

fabs

```
#include <math.h>
double fabs( double x );
```

Returns the absolute double value of `x`. $|x|$

fclose

```
#include <stdio.h>
int fclose( FILE *stream )
```

Flushes any unwritten data for stream, discards any unread buffered input, frees any automatically allocated buffer, then closes the stream.

Returns zero if the stream is successfully closed, or EOF on error.

feof

```
#include <stdio.h>
int feof( FILE *stream );
```

Returns a non-zero value if the end-of-file indicator for stream is set.

ferror

```
#include <stdio.h>
int ferror( FILE *stream );
```

Returns a non-zero value if the error indicator for stream is set.

fflush

```
#include <stdio.h>
int fflush( FILE *stream );
```

Writes any buffered but unwritten data, if stream is an output stream. If stream is an input stream, the effect is undefined.

Returns zero if successful, or EOF on a write error.

fgetc

```
#include <stdio.h>
int fgetc( FILE *stream );
```

Reads one character from the given `stream`.

Returns the read character, or EOF on error.

fgetpos

```
#include <stdio.h>
int fgetpos( FILE *stream, fpos_t *ptr );
```

Stores the current value of the file position indicator for the stream pointed to by `stream` in the object pointed to by `ptr`. The type `fpos_t` is suitable for recording such values.

Returns zero if successful,
a non-zero value on error.

fgets

```
#include <stdio.h>
char *fgets( char *s, int n, FILE *stream );
```

Reads at most the next `n-1` characters from the given `stream` into the array `s` until a newline is found.

Returns `s`, or NULL on EOF or error.

floor

```
#include <math.h>
double floor( double x );
```

Returns the largest integer not greater than `x`, as a double.

fmod

```
#include <math.h>
double fmod( double x, double y );
```

Returns the floating-point remainder of x/y , with the same sign as x . If y is zero, the result is implementation-defined.

fopen

```
#include <stdio.h>
FILE *fopen( const char *filename,
             const char *mode );
```

Opens a file for a given mode.

Returns a stream. If the file cannot not be opened, NULL is returned.

You can specify the following values for mode:

"r"	read; open text file for reading
"w"	write; create text file for writing; if the file already exists its contents is discarded
"a"	append; open existing text file or create new text file for writing at end of file
"r+"	open text file for update; reading and writing
"w+"	create text file for update; previous contents if any is discarded
"a+"	append; open or create text file for update, writes at end of file

The update mode (with a '+') allows reading and writing of the same file. In this mode the function `fflush` must be called between a read and a write or vice versa. By including the letter `b` after the initial letter, you can indicate that the file is a binary file. E.g. `"rb"` means read binary, `"w+b"` means create binary file for update. The filename is limited to `FILENAME_MAX` characters. At most `FOPEN_MAX` files may be open at once.

fprintf

```
#include <stdio.h>
int fprintf( FILE *stream,
             const char *format, ... );
```

Performs a formatted write to the given stream.



See also "printf()", "_write()" and section *Printf and Scanf Formatting Routines*.

fputc

```
#include <stdio.h>
int fputc( int c, FILE *stream );
```

Puts one character onto the given stream.



See also "_write()".

Returns EOF on error.

fputs

```
#include <stdio.h>
int fputs( const char *s, FILE *stream );
```

Writes the string to a stream. The terminating NULL character is not written.



See also "_write()".

Returns NULL if successful, or EOF on error.

fread

```
#include <stdio.h>
size_t fread( void *ptr,
              size_t size, size_t nobj, FILE *stream );
```

Reads `nobj` members of `size` bytes from the given `stream` into the array pointed to by `ptr`.



See also `"_read()"`.

Returns the number of successfully read objects.

free

```
#include <stdlib.h>
void free( void *p );
```

Deallocates the space pointed to by `p`. `p` Must point to space earlier allocated by a call to `"calloc()", "malloc()"` or `"realloc()"`. Otherwise the behavior is undefined.



See also `"calloc()", "malloc()"` and `"realloc()"`.

Returns nothing

freopen

```
#include <stdio.h>
FILE *
freopen( const char *filename,
         const char *mode, FILE *stream );
```

Opens a file for a given mode associates the `stream` with it. This function is normally used to change the files associated with `stdin`, `stdout`, or `stderr`.



See also `"fopen()"`.

Returns `stream`, or `NULL` on error.

frexp

```
#include <math.h>
double frexp( double x, int *exp );
```

Splits *x* into a normalized fraction in the interval $[1/2, 1)$, which is returned, and a power of 2, which is stored in **exp*. If *x* is zero, both parts of the result are zero. For example: `frexp(4.0, &var)` results in $0.5 \cdot 2^3$. The function returns 0.5, and 3 is stored in *var*.

Returns the normalized fraction.

fscanf

```
#include <stdio.h>
int fscanf( FILE *stream,
           const char *format, ... );
```

Performs a formatted read from the given stream.



See also "scanf()", "_read()" and section *Printf and Scanf Formatting Routines*.

Returns the number of items converted successfully.

fseek

```
#include <stdio.h>
int
fseek( FILE *stream, long offset, int origin );
```

Sets the file position indicator for *stream*. A subsequent read or write will access data beginning at the new position. For a binary file, the position is set to *offset* characters from *origin*, which may be `SEEK_SET` for the beginning of the file, `SEEK_CUR` for the current position in the file, or `SEEK_END` for the end-of-file. For a text stream, *offset* must be zero, or a value returned by `ftell`. In this case *origin* must be `SEEK_SET`.

Returns zero if successful,
a non-zero value on error.

fsetpos

```
#include <stdio.h>
int fsetpos( FILE *stream,
            const fpos_t *ptr );
```

Positions *stream* at the position recorded by *fgetpos* in **ptr*.

Returns zero if successful,
 a non-zero value on error.

ftell

```
#include <stdio.h>
long ftell( FILE *stream );
```

Returns the current file position for *stream*, or
 -1L on error.

fwrite

```
#include <stdio.h>
size_t fwrite( const void *ptr,
              size_t size, size_t nobj,
              FILE *stream );
```

Writes *nobj* members of *size* bytes to the given *stream* from the array pointed to by *ptr*.

Returns the number of successfully written objects.

getc

```
#include <stdio.h>
int getc( FILE *stream );
```

Reads one character out of the given *stream*. Currently #defined as *getchar()*, because FILE I/O is not supported.



See also "*_read()*".

Returns the character read or EOF on error.

getchar

```
#include <stdio.h>
int getchar( void );
```

Reads one character from standard input.



See also ”_read()”.

Returns the character read or EOF on error.

getenv

```
#include <stdlib.h>
char *getenv( const char *name );
```

Returns the environment string associated with name, or NULL if no string exists.

gets

```
#include <stdio.h>
char *gets( char *s );
```

Reads all characters from standard input until a newline is found. The newline is replaced by a NULL-character.



See also ”_read()”.

Returns a pointer to the read string or NULL on error.

gmtime

```
#include <time.h>
struct tm *gmtime( const time_t *tp );
```

Converts the calendar time *tp into Coordinated Universal Time (UTC).

Returns a structure representing the UTC, or NULL if UTC is not available.

isalnum

```
#include <ctype.h>
int isalnum( int c );
```

Returns a non-zero value when *c* is an alphabetic character or a number ([A-Z][a-z][0-9]).

isalpha

```
#include <ctype.h>
int isalpha( int c );
```

Returns a non-zero value when *c* is an alphabetic character ([A-Z][a-z]).

isascii

```
#include <ctype.h>
int isascii( int c );
```

Returns a non-zero value when *c* is in the range of 0 and 127. This is a non-ANSI function.

isctrl

```
#include <ctype.h>
int isctrl( int c );
```

Returns a non-zero value when *c* is a control character.

isdigit

```
#include <ctype.h>
int isdigit( int c );
```

Returns a non-zero value when *c* is a numeric character ([0-9]).

isfinite

```
#include <float.h>
int isfinite( double d );
```

IEEE-754-1985 recommended function. Test the given variable on being a finite (IEEE-754) value.

Returns zero if the variable is not finite, else non-zero.

isfinitef

```
#include <float.h>
int isfinitef( float f );
```

IEEE-754-1985 recommended function. Test the given variable on being a finite (IEEE-754) value.

Returns zero if the variable is not finite, else non-zero.

isgraph

```
#include <ctype.h>
int isgraph( int c );
```

Returns a non-zero value when *c* is printable, but not a space.

isinf

```
#include <float.h>
int isinf( double d );
```

IEEE-754-1985 Recommended function. Test the given variable on being an infinite (IEEE-754) value.

Returns zero if the variable is not \pm -infinite, else non-zero.

isinf

```
#include <float.h>
int isinf( float f );
```

IEEE-754-1985 Recommended function. Test the given variable on being an infinite (IEEE-754) value.

Returns zero if the variable is not \pm -infinite, else non-zero.

islower

```
#include <ctype.h>
int islower( int c );
```

Returns a non-zero value when *c* is a lowercase character ([a-z]).

isnan

```
#include <float.h>
int isnan( double d );
```

IEEE-754-1985 Recommended function. Test the given variable on being a NaN (Not a Number, IEEE-754) value.

Returns zero if the variable is not NaN, else non-zero.

isnanf

```
#include <float.h>
int isnanf( float f );
```

IEEE-754-1985 Recommended function. Test the given variable on being a NaN (Not a Number, IEEE-754) value.

Returns zero if the variable is not NaN, else non-zero.

isprint

```
#include <ctype.h>
int isprint( int c );
```

Returns a non-zero value when *c* is printable, including spaces.

ispunct

```
#include <ctype.h>
int ispunct( int c );
```

Returns a non-zero value when *c* is a punctuation character (such as `'`, `,`, `!`, etc.).

isspace

```
#include <ctype.h>
int isspace( int c );
```

Returns a non-zero value when *c* is a space type character (space, tab, vertical tab, formfeed, linefeed, carriage return).

isupper

```
#include <ctype.h>
int isupper( int c );
```

Returns a non-zero value when *c* is an uppercase character (`[A-Z]`).

isxdigit

```
#include <ctype.h>
int isxdigit( int c );
```

Returns a non-zero value when *c* is a hexadecimal digit (`[0-9][A-F][a-f]`).

labs

```
#include <stdlib.h>
long labs( long n );
```

Returns the absolute value of the signed long argument.

ldexp

```
#include <math.h>
double ldexp( double x, int n );
```

Returns the result of: $x \cdot 2^n$.

ldiv

```
#include <stdlib.h>
ldiv_t ldiv( long num, long denom );
```

Both arguments are long integers. The returned quotient and remainder are also long integers.

Returns a structure containing the quotient and remainder of num divided by denom.

localeconv

```
#include <locale.h>
struct lconv *localeconv( void );
```

Sets the components of an object with type `struct lconv` with values appropriate for the formatting of numeric quantities according to the rules of the current locale.

Returns a pointer to the filled-in object.

localtime

```
#include <time.h>
struct tm *localtime( const time_t *tp );
```

Converts the calendar time **tp* into local time.

Returns a structure representing the local time.

log

```
#include <math.h>
double log( double x );
```

Returns the natural logarithm $\ln(x)$, $x > 0$.

log10

```
#include <math.h>
double log10( double x );
```

Returns the base 10 logarithm $\log_{10}(x)$, $x > 0$.

longjmp

```
#include <setjmp.h>
void longjmp( jmp_buf env, int val);
```

Restores the environment previously saved with a call to `setjmp()`. The function calling the corresponding call to `setjmp()` may not be terminated yet. The value of *val* may not be zero.

Returns nothing.

malloc

```
#include <stdlib.h>
void *malloc( size_t size );
```

The allocated space is not initialized. The maximum space that can be allocated can be changed by customizing the heap size (see the section *Heap*). By default no heap is allocated. When "malloc()" is used while no heap is defined, the locator gives an error.

Returns a pointer to space in external memory of size bytes length. NULL if there is not enough space left.

mblen

```
#include <stdlib.h>
int mblen( const char *s, size_t n );
```

Determines the number of bytes comprising the multi-byte character pointed to by *s*, if *s* is not a null pointer. Except that the shift state is not affected. At most *n* characters will be examined, starting at the character pointed to by *s*.

Returns the number of bytes, or 0 if *s* points to the null character, or -1 if the bytes do not form a valid multi-byte character.

mbstowcs

```
#include <stdlib.h>
size_t mbstowcs( wchar_t *pwcs,
                 const char *s, size_t n );
```

Converts a sequence of multi-byte characters that begins in the initial shift state from the array pointed to by *s*, into a sequence of corresponding codes and stores these codes into the array pointed to by *pwcs*, stopping after *n* codes are stored or a code with value zero is stored.

Returns the number of array elements modified (not including a terminating zero code, if any), or (size_t)-1 if an invalid multi-byte character is encountered.

mbtowc

```
#include <stdlib.h>
int mbtowc( wchar_t *pwc,
            const char *s, size_t n );
```

Determines the number of bytes that comprise the multi-byte character pointed to by *s*. It then determines the code for value of type `wchar_t` that corresponds to that multi-byte character. If the multi-byte character is valid and *pwc* is not a null pointer, the `mbtowc` function stores the code in the object pointed to by *pwc*. At most *n* characters will be examined, starting at the character pointed to by *s*.

Returns the number of bytes, or 0 if *s* points to the null character, or -1 if the bytes do not form a valid multi-byte character.

memchr

```
#include <string.h>
void *memchr( const void *cs, int c,
              size_t n );
```

Checks the first *n* bytes of *cs* on the occurrence of character *c*.

Returns NULL when not found, otherwise a pointer to the found character is returned.

memcmp

```
#include <string.h>
int memcmp( const void *cs,
            const void *ct, size_t n );
```

Compares the first *n* bytes of *cs* with the contents of *ct*.

Returns a value < 0 if *cs* < *ct*,
0 if *cs* == *ct*,
or a value > 0 if *cs* > *ct*.

memcpy

```
#include <string.h>
void *memcpy( void *s,
              const void *ct, size_t n );
```

Copies *n* characters from *ct* to *s*. No care is taken if the two objects overlap.

Returns *s*

memmove

```
#include <string.h>
void *memmove( void *s,
               const void *ct, size_t n );
```

Copies *n* characters from *ct* to *s*. Overlapping objects will be handled correctly.

Returns *s*

memset

```
#include <string.h>
void *memset( void *s, int c,
              size_t n );
```

Fills the first *n* bytes of *s* with character *c*.

Returns *s*

mktime

```
#include <time.h>
time_t mktime( struct tm *tp );
```

Converts the local time in the structure **tp* into calendar time.

Returns the calendar time, or -1 if it cannot be represented.

modf

```
#include <math.h>
double modf( double x, double *ip );
```

Splits *x* into integral and fractional parts, each with the same sign as *x*. It stores the integral part in **ip*.

Returns the fractional part.

offsetof

```
#include <stddef.h>
int offsetof( type, member );
```

Returns the offset for the given member in an object of type.

perror

```
#include <stdio.h>
void perror( const char *s );
```

Prints *s* and an implementation-defined error message corresponding to the integer *errno*, as if by:

```
fprintf( stderr, "%s: %s\n", s, "error message" );
```

The contents of the error message are the same as those returned by the *strerror* function with the argument *errno*.



See also the "strerror()" function.

Returns nothing.

pow

```
#include <math.h>
double pow( double x, double y );
```

A domain error occurs if *x*=0 and *y*<=0, or if *x*<0 and *y* is not an integer.

Returns the result of *x* raised to the power of *y*: *x^y*.

printf

```
#include <stdio.h>
int printf( const char *format,...);
```

Performs a formatted write to the standard output stream.



See also “_write()” and section *Printf and Scanf Formatting Routines*.

Returns the number of characters written to the output stream.

The `format` string may contain plain text mixed with conversion specifiers. Each conversion specifier should be preceded by a ‘%’ character. The conversion specifier should be build in order:

- Flags (in any order):
 - specifies left adjustment of the converted argument.
 - + a number is always preceded with a sign character.
+ has higher precedence as space.
 - space a negative number is preceded with a sign, positive numbers with a space.
 - 0 specifies padding to the field width with zeros (only for numbers).
 - # specifies an alternate output form. For o, the first digit will be zero. For x or X, “0x” and “0X” will be prefixed to the number. For e, E, f, g, G, the output always contains a decimal point, trailing zeros are not removed.
- A number specifying a minimum field width. The converted argument is printed in a field with at least the length specified here. If the converted argument has fewer characters than specified, it will be padded at the left side (or at the right when the flag ‘-’ was specified) with spaces. Padding to numeric fields will be done with zeros when the flag ‘0’ is also specified (only when padding left). Instead of a numeric value, also ‘*’ may be specified, the value is then taken from the next argument, which is assumed to be of type `int`.
- A period. This separates the minimum field width from the precision.

- A number specifying the maximum length of a string to be printed. Or the number of digits printed after the decimal point (only for floating point conversions). Or the minimum number of digits to be printed for an integer conversion. Instead of a numeric value, also '*' may be specified, the value is then taken from the next argument, which is assumed to be of type int.
- A length modifier 'h', 'l' or 'L'. 'h' indicates that the argument is to be treated as a short or unsigned short number. 'l' should be used if the argument is a long integer. 'L' indicates that the argument is a long double.

Flags, length specifier, period, precision and length modifier are optional, the conversion character is not. The conversion character must be one of the following, if a character following '%' is not in the list, the behavior is undefined:

Character	Printed as
d, i	int, signed decimal
o	int, unsigned octal
x, X	int, unsigned hexadecimal in lowercase or uppercase respectively
u	int, unsigned decimal
c	int, single character (converted to unsigned char)
s	char *, the characters from the string are printed until a NULL character is found. When the given precision is met before, printing will also stop
f	double
e, E	double
g, G	double
n	int *, the number of characters written so far is written into the argument. This should be a pointer to an integer in default memory. No value is printed.
p	pointer (hexadecimal 32-bit value)
%	No argument is converted, a '%' is printed.

Table 6-6: Printf conversion characters

putc

```
#include <stdio.h>
int putc( int c, FILE *stream );
```

Puts one character onto the given stream.



See also ”_write()”.

Returns EOF on error.

putchar

```
#include <stdio.h>
int putchar( int c );
```

Puts one character onto standard output.



See also ”_write()”.

Returns the character written or EOF on error.

puts

```
#include <stdio.h>
int puts( const char *s );
```

Writes the string to stdout, the string is terminated by a newline.



See also ”_write()”.

Returns NULL if successful, or EOF on error.

qsort

```
#include <stdlib.h>
_reentrant void qsort(
    const void *base, size_t n, size_t size,
    int (* cmp)(const void *, const void *) );
```

This function sorts an array of *n* members. The initial base of the array is given by *base*. The size of each member is specified by *size*. The given array is sorted in ascending order, according to the results of the function pointed to by *cmp*.

Returns nothing.

raise

```
#include <signal.h>
int raise( int sig );
```

Sends the signal *sig* to the program.



See also "signal()".

Returns zero if successful, or a non-zero value if unsuccessful.

rand

```
#include <stdlib.h>
int rand( void );
```

Returns a sequence of pseudo-random integers, in the range 0 to RAND_MAX.

realloc

```
#include <stdlib.h>
void *realloc( void *p, size_t size );
```

Reallocates the space for the object pointed to by `p`. The contents of the object will be the same as before calling `realloc()`. The maximum space that can be allocated can be changed by customizing the heap size (see the section *Heap*). By default no heap is allocated. When "realloc()" is used while no heap is defined, the linker gives an error.



See also "malloc".

Returns NULL and `*p` is not changed, if there is not enough space for the new allocation. Otherwise a pointer to the newly allocated space for the object is returned.

remove

```
#include <stdio.h>
int remove( const char *filename );
```

Removes the named file, so that a subsequent attempt to open it fails.

Returns zero if file is successfully removed, or a non-zero value, if the attempt fails.

rename

```
#include <stdio.h>
int rename( const char *oldname,
            const char *newname );
```

Changes the name of the file.

Returns zero if file is successfully renamed, or a non-zero value, if the attempt fails.

rewind

```
#include <stdio.h>
void rewind( FILE *stream );
```

Sets the file position indicator for the stream pointed to by `stream` to the beginning of the file. This function is equivalent to:

```
(void) fseek( stream, 0L, SEEK_SET );
clearerr( stream );
```

Returns nothing.

scalb

```
#include <float.h>
double scalb( double d, int power );
```

IEEE-754-1985 Recommended function.

Returns $d * 2^{\text{power}}$ for integral values power without computing 2^N .

scalbf

```
#include <float.h>
double scalbf( float d, int power );
```

IEEE-754-1985 Recommended function.

Returns $d * 2^{\text{power}}$ for integral values power without computing 2^N .

scanf

```
#include <stdio.h>
int scanf( const char *format, ... );
```

Performs a formatted read from the standard input stream.



See also `”_read()”` and section *Printf and Scanf Formatting Routines*.

Returns the number of items converted successfully.

All arguments to this function should be pointers to variables (in default memory) of the type which is specified in the format string.

The format string may contain :

- Blanks or tabs, which are skipped.
- Normal characters (not '%'), which should be matched exactly in the input stream.
- Conversion specifications, starting with a '%' character.

Conversion specifications should be built as follows (in order) :

- A '*', meaning that no assignment is done for this field.
- A number specifying the maximum field width.
- The conversion characters `d`, `i`, `n`, `o`, `u` and `x` may be preceede by 'h' if the argument is a pointer to short rather than `int`, or by 'l' (letter ell) if the argument is a pointer to long. The conversion characters `e`, `f`, and `g` may be preceede by 'l' if a pointer double rather than `float` is in the argument list, and by 'L' if a pointer to a long double.
- A conversion specifier: '*', maximum field width and length modifier are optional, the conversion character is not. The conversion character must be one of the following, if a character following '%' is not in the list, the behavior is undefined.

Length specifier and length modifier are optional, the conversion character is not. The conversion character must be one of the following, if a character following '%' is not in the list, the behavior is undefined.

Character	Scanned as
d	int, signed decimal.
i	int, the integer may be given octal (i.e. a leading 0 is entered) or hexadecimal (leading "0x" or "0X"), or just decimal.
o	int, unsigned octal.
u	int, unsigned decimal.
x	int, unsigned hexadecimal in lowercase or uppercase.
c	single character (converted to unsigned char).

Character	Scanned as
s	char *, a string of non white space characters. The argument should point to an array of characters, large enough to hold the string and a terminating NULL character.
f	float
e, E	float
g, G	float
n	int *, the number of characters written so far is written into the argument. No scanning is done.
p	pointer; hexadecimal 32-bit value which must be entered without 0x- prefix.
[...]	Matches a string of input characters from the set between the brackets. A NULL character is added to terminate the string. Specifying [...] includes the ']' character in the set of scanning characters.
[^...]	Matches a string of input characters not in the set between the brackets. A NULL character is added to terminate the string. Specifying [^...] includes the ']' character in the set.
%	Literal '%', no assignment is done.

Table 6-7: *Scanf conversion characters*

setbuf

```
#include <stdio.h>
void
setbuf( FILE *stream, char *buf );
```

Buffering is turned off for the `stream`, if `buf` is `NULL`. Otherwise, `setbuf` is equivalent to:

(void) `setvbuf(stream, buf, _IOFBF, BUFSIZ)`

Returns nothing.



See also `"setvbuf()"`.

setjmp

```
#include <setjmp.h>
int setjmp( jmp_buf env );
```

Saves the current environment for a subsequent call to `longjmp`.

Returns the value 0 after a direct call to `setjmp()`. Calling the function `longjmp()` using the saved `env` will restore the current environment and jump to this place with a non-zero return value.



See also `longjmp()`.

setlocale

```
#include <locale.h>
char *setlocale( int category,
                 const char *locale );
```

Selects the appropriate portion of the program's locale as specified by the `category` and `locale` arguments.

Returns the string associated with the specified `category` for the new locale if the selection can be honored.
null pointer if the selection cannot be honored.

setvbuf

```
#include <stdio.h>
int
setvbuf( FILE *stream, char *buf,
         int mode, size_t size );
```

Controls buffering for the *stream*; this function must be called before reading or writing. *mode* can have the following values:

_IOFBF causes full buffering
_IOLBF causes line buffering of text files
_IONBF causes no buffering

If *buf* is not NULL, it will be used as a buffer; otherwise a buffer will be allocated. *size* determines the buffer size.

Returns zero if successful
 a non-zero value for an error.



See also "setbuf()".

signal

```
#include <signal.h>
void (*signal( int sig,
               void (*handler)(int)))(int);
```

Determines how subsequent signals will be handled. If *handler* is **SIG_DFL**, the default behavior is used; if *handler* is **SIG_IGN**, the signal is ignored; otherwise, the function pointed to by *handler* will be called, with the argument of the type of signal. Valid signals are:

SIGABRT abnormal termination, e.g. from **abort**
SIGFPE arithmetic error, e.g. zero divide or overflow
SIGILL illegal function image, e.g. illegal instruction
SIGINT interactive attention, e.g. interrupt
SIGSEGV illegal storage access, e.g. access outside memory limits
SIGTERM termination request sent to this program

When a signal `sig` subsequently occurs, the signal is restored to its default behavior; then the signal-handler function is called, as if by `(*handler)(sig)`. If the handler returns, the execution will resume where it was when the signal occurred.

Returns the previous value of `handler` for the specific signal, or `SIG_ERR` if an error occurs.

sin

```
#include <math.h>
double sin( double x );
```

Returns the sine of `x`.

sinh

```
#include <math.h>
double sinh( double x );
```

Returns the hyperbolic sine of `x`.

sprintf

```
#include <stdio.h>
int sprintf( char *s, const char *format, ... );
```

Performs a formatted write to a string.



See also “`printf()`” and section *Printf and Scanf Formatting Routines*.

sqrt

```
#include <math.h>
double sqrt( double x );
```

Returns the square root of `x`. \sqrt{x} , where $x \geq 0$.

srand

```
#include <stdlib.h>
void srand( unsigned int seed );
```

This function uses `seed` as the start of a new sequence of pseudo-random numbers to be returned by subsequent calls to `srand()`. When `srand` is called with the same seed value, the sequence of pseudo-random numbers generated by `rand()` will be repeated.

Returns pseudo random numbers.

sscanf

```
#include <stdio.h>
int sscanf( char *s, const char *format, ... );
```

Performs a formatted read from a string.



See also "scanf()" and section *Printf and Scanf Formatting Routines*.

strcat

```
#include <string.h>
char *strcat( char *s, const char *ct );
```

Concatenates string `ct` to string `s`, including the trailing NULL character.

Returns `s`

strchr

```
#include <string.h>
char *strchr( const char *cs, int c );
```

Returns a pointer to the first occurrence of character `c` in the string `cs`. If not found, NULL is returned.

strcmp

```
#include <string.h>
int strcmp( const char *cs, const char *ct );
```

Compares string *cs* to string *ct*.

Returns <0 if *cs* < *ct*,
 0 if *cs* == *ct*,
 >0 if *cs* > *ct*.

strcoll

```
#include <string.h>
int strcoll( const char *cs, const char *ct );
```

Compares string *cs* to string *ct*. The comparison is based on strings interpreted as appropriate to the program's locale.

Returns <0 if *cs* < *ct*,
 0 if *cs* = *ct*,
 >0 if *cs* > *ct*.

strcpy

```
#include <string.h>
char *strcpy( char *s, const char *ct );
```

Copies string *ct* into the string *s*, including the trailing NULL character.

Returns *s*

strcspn

```
#include <string.h>
size_t strcspn( const char *cs, const char *ct );
```

Returns the length of the prefix in string *cs*, consisting of characters not in the string *ct*.

strerror

```
#include <string.h>
char *strerror( size_t n );
```

Returns pointer to implementation-defined string corresponding to error n.

strftime

```
#include <time.h>
size_t
strftime( char *s, size_t smax,
          const char *fmt,
          const struct tm *tp );
```

Formats date and time information from the structure *tp into s according to the specified format fmt. fmt is analogous to a printf format. Each %c is replaced as described below:

%a	abbreviated weekday name
%A	full weekday name
%b	abbreviated month name
%B	full month name
%c	local date and time representation
%d	day of the month (01-31)
%H	hour, 24-hour clock (00-23)
%I	hour, 12-hour clock (01-12)
%j	day of the year (001-366)
%m	month (01-12)
%M	minute (00-59)
%p	local equivalent of AM or PM
%S	second (00-59)
%U	week number of the year, Sunday as first day of the week (00-53)
%w	weekday (0-6, Sunday is 0)
%W	week number of the year, Monday as first day of the week (00-53)
%x	local date representation
%X	local time representation
%y	year without century (00-99)
%Y	year with century

%Z time zone name, if any
%% %

Ordinary characters (including the terminating ‘\0’) are copied into *s*. No more than *smax* characters are placed into *s*.

Returns the number of characters (‘\0’ not included), or zero if more than *smax* characters were produced.

strlen

```
#include <string.h>
size_t strlen( const char *cs );
```

Returns the length of the string in *cs*, not counting the NULL character.

strncat

```
#include <string.h>
char *strncat( char *s,
               const char *ct, size_t n );
```

Concatenates string *ct* to string *s*, at most *n* characters are copied. Add a trailing NULL character.

Returns *s*

strncmp

```
#include <string.h>
int strncmp( const char *cs,
             const char *ct, size_t n );
```

Compares at most *n* bytes of string *cs* to string *ct*.

Returns <0 if *cs* < *ct*,
 0 if *cs* == *ct*,
 >0 if *cs* > *ct*.

strncpy

```
#include <string.h>
char *strncpy( char *s,
               const char *ct, size_t n );
```

Copies string *ct* onto the string *s*, at most *n* characters are copied. Add a trailing NULL character if the string is smaller than *n* characters.

Returns *s*

strpbrk

```
#include <string.h>
char *strpbrk( const char *cs,
               const char *ct );
```

Returns a pointer to the first occurrence in *cs* of any character out of string *ct*. If none are found, NULL is returned.

strrchr

```
#include <string.h>
char *strrchr( const char *cs,
               int c );
```

Returns a pointer to the last occurrence of *c* in the string *cs*. If not found, NULL is returned.

strspn

```
#include <string.h>
size_t strspn( const char *cs,
               const char *ct );
```

Returns the length of the prefix in string *cs*, consisting of characters in the string *ct*.

strstr

```
#include <string.h>
char *strstr( const char *cs,
              const char *ct );
```

Returns a pointer to the first occurrence of string *ct* in the string *cs*. Returns NULL if not found.

strtod

```
#include <stdlib.h>
double strtod( const char *s, char **endp );
```

Converts the initial portion of the string pointed to by *s* to a double value. Initial white spaces are skipped. When *endp* is not a NULL pointer, after this function is called, **endp* will point to the first character not used by the conversion.

Returns the read value.

strtok

```
#include <string.h>
char *strtok( char *s, const char *ct );
```

Search the string *s* for tokens delimited by characters from string *ct*. It terminates the token with a NULL character.

Returns a pointer to the token. A subsequent call with *s* == NULL will return the next token in the string.

strtol

```
#include <stdlib.h>
long strtol( const char *s,
             char **endp, int base );
```

Converts the initial portion of the string pointed to by *s* to a long integer. Initial white spaces are skipped. Then a value is read using the given base. When base is zero, the base is taken as defined for integer constants. I.e. numbers starting with an '0' are taken octal, numbers starting with '0x' or '0X' are taken hexadecimal. Other numbers are taken decimal. When *endp* is not a NULL pointer, after this function is called, **endp* will point to the first character not used by the conversion.

Returns the read value.

strtoul

```
#include <stdlib.h>
unsigned long strtoul(
    const char *s, char **endp, int base );
```

Converts the initial portion of the string pointed to by *s* to an unsigned long integer. Initial white spaces are skipped. Then a value is read using the given base. When base is zero, the base is taken as defined for integer constants. I.e. numbers starting with an '0' are taken octal, numbers starting with '0x' or '0X' are taken hexadecimal. Other numbers are taken decimal. When *endp* is not a NULL pointer, after this function is called, **endp* will point to the first character not used by the conversion.

Returns the read value.

strxfrm

```
#include <string.h>
size_t
strncmp( char *ct, const char *cs, size_t n );
```

Transforms the string pointed to by *cs* and places the resulting string into the array pointed to by *ct*. No more than *n* characters are placed into the resulting string pointed to by *ct*, including the terminating null character.

Returns the length of the transformed string.

system

```
#include <stdlib.h>
int system( const char *s );
```

Passes the string *s* to the environment for execution.

Returns a non-zero value if there is a command processor, if *s* is NULL; or an implementation-dependent value, if *s* is not NULL.

tan

```
#include <math.h>
double tan( double x );
```

Returns the tangent of *x*.

tanh

```
#include <math.h>
double tanh( double x );
```

Returns the hyperbolic tangent of *x*.

time

```
#include <time.h>
time_t time( time_t *tp );
```

The return value is also assigned to **tp*, if *tp* is not NULL.

Returns the current calendar time, or -1 if the time is not available.

tmpfile

```
#include <stdio.h>
FILE *tmpfile( void );
```

Creates a temporary file of the mode "wb+" that will be automatically removed when closed or when the program terminates normally.

Returns a stream if successful, or NULL if the file could not be created.

tmpnam

```
#include <stdio.h>
char *tmpnam( char s[L_tmpnam] );
```

Creates a temporary name (not a file). Each time `tmpnam` is called a different name is created.

`tmpnam(NULL)` creates a string that is not the name of an existing file, and returns a pointer to an internal static array. `tmpnam(s)` creates a string and stores it in `s` and also returns it as the function value. `s` must have room for at least `L_tmpnam` characters. At most `TMP_MAX` different names are guaranteed during execution of the program.

Returns a pointer to the temporary name, as described above.

toascii

```
#include <ctype.h>
int toascii( int c );
```

Converts `c` to an ascii value (strip highest bit). This is a non-ANSI function.

Returns the converted value.

tolower

```
#include <ctype.h>
int tolower( int c );
```

Returns *c* converted to a lowercase character if it is an uppercase character, otherwise *c* is returned.

toupper

```
#include <ctype.h>
int toupper( int c );
```

Returns *c* converted to an uppercase character if it is a lowercase character, otherwise *c* is returned.

ungetc

```
#include <stdio.h>
int ungetc( int c, FILE *fin );
```

Pushes at the most one character back onto the input buffer.

Returns EOF on error.

va_arg

```
#include <stdarg.h>
va_arg( va_list ap, type );
```

Returns the value of the next argument in the variable argument list. It's return type has the type of the given argument *type*. A next call to this macro will return the value of the next argument.

va_end

```
#include <stdarg.h>
va_end( va_list ap );
```

This macro must be called after the arguments have been processed. It should be called before the function using the macro 'va_start' is terminated (ANSI specification).

va_start

```
#include <stdarg.h>
va_start( va_list ap, lastarg );
```

This macro initializes ap. After this call, each call to va_arg() will return the value of the next argument. In our implementation, va_list cannot contain any bit type variables. Also the given argument lastarg must be the last non-bit type argument in the list.

vfprintf

```
#include <stdio.h>
int vfprintf( FILE *stream,
              const char *format, va_list arg );
```

Is equivalent to vprintf, but writes to the given stream.



See also "vprintf()", "_write()" and section *Printf and Scanf Formatting Routines*.

vprintf

```
#include <stdio.h>
int vprintf( const char *format,
             va_list arg );
```

Does a formatted write to standard output. Instead of a variable argument list as for printf(), this function expects a pointer to the list.



See also "printf()", "_write()" and section *Printf and Scanf Formatting Routines*.

vsprintf

```
#include <stdio.h>
int vsprintf( char *s,
              const char *format, va_list arg );
```

Does a formatted write a string. Instead of a variable argument list as for `printf()`, this function expects a pointer to the list.



See also "`printf()`", "`_write()`" and section *Printf and Scanf Formatting Routines*.

wcstombs

```
#include <stdlib.h>
size_t wcstombs( char *s,
                 const wchar_t *pwcs, size_t n );
```

Converts a sequence of codes that correspond to multi-byte characters from the array pointed to by `pwcs`, into a sequence of multi-byte characters that begins in the initial shift state and stores these multi-byte characters into the array pointed to by `s`, stopping if a multi-byte character would exceed the limit of `n` total bytes or if a null character is stored.

Returns the number of bytes modified (not including a terminating null character, if any), or `(size_t)-1` if a code is encountered that does not correspond to a valid multi-byte character.

wctomb

```
#include <stdlib.h>
int wctomb( char *s, wchar_t wchar );
```

Determines the number of bytes needed to represent the multi-byte corresponding to the code whose value is `wchar` (including any change in the shift state). It stores the multi-byte character representation in the array pointed to by `s` (if `s` is not a null pointer). At most `MB_CUR_MAX` characters are stored. If the value of `wchar` is zero, the `wctomb` function is left in the initial shift state.

Returns the number of bytes, or `-1` if the value of `wchar` does not correspond to a valid multi-byte character.

6.3.4 PRINTF AND SCANF FORMATTING ROUTINES

The functions `printf()`, `fprintf()`, `vfprintf()`, `vsprintf()`, ... call one single function that deals with the format string and arguments. This function is `_doprint()`. This is a rather big function because the number of possibilities of the format specifiers in a format string are large. If you do not use all the possibilities of the format specifiers a smaller `_doprint()` function can be used. Three different versions exist:

LARGE	the full formatter, no restrictions
MEDIUM	floating point printing is not supported
SMALL	as MEDIUM, but also the length specifier cannot be used.

The same applies to all `scanf` type functions, which all call the function `_doscan()`.

The formatters included in the standard C libraries are LARGE. You can select different formatters by linking other libraries with your application. The following extra libraries are included:

<code>lib/printfss</code>	small model, SMALL formatter
<code>lib/printfsm</code>	small model, MEDIUM formatter
<code>lib/scanfss</code>	small model, SMALL formatter
<code>lib/printfsl</code>	large model, SMALL formatter
<code>lib/printflm</code>	large model, MEDIUM formatter
<code>lib/scanfsl</code>	large model, SMALL formatter

Example:

To use the MEDIUM printf formatter for the small model:



Select the EDE | C Compiler Options | Project Options... menu item. Enable the Allow width specifier, precision and flags check box in the Printf/Scanf tab. Disable the Print floating point values check box.



`ccm16 -Ms hello.obj -lprintfsm`

6.4 RUN-TIME LIBRARY

Some compiler generated code contains calls to run-time library functions that would use too much code if generated as inline code. The name of a run-time library function always contains two leading underscores.

Because **cm16** generates assembly code (and not object code) it prepends an underscore '_' for the names of (public) C variables to distinguish these symbols from M16C registers. So if you use a function with a leading underscore, the assembly label for this function contains two leading underscores. This function name could cause a name conflict (double defined) with one of the run-time library functions. However, ANSI states that it is not portable to use names starting with an underscore for public C variables and functions, because results are implementation defined.



LIBRARIES

CHAPTER 7

RUN-TIME ENVIRONMENT



TASKING



7 | CHAPTER

7.1 **STARTUP CODE**

When linking your C modules with the library, you automatically link the object module, containing the C startup code. This module is called `cstart.obj` and is present in every C library (once for every compiler model).

EDE generates the file `cstart.src` for you, when you make selections in the EDE | Processor Options dialog. You must manually add the file `cstart.src` to your project properties.

7.2 **REGISTER USAGE**

cm16 uses the following registers for C function return types:

Return type	Register(s)
bit	0,R0
char	R0L
short/int	R0
long	R0;A0
float	R0;A0
double	R3;R2;R1;R0
structure	Stack temporary (address passed by caller in R0)
2-byte pointer	A0
4-byte pointer	R0;A0

Table 7-1: Register usage

7.3 SECTION USAGE

cm16 uses a large number of section. This section contains a list of all possible section names of a complete C application:

BIT

M16C_INI_BI initialized user C `_bit` variables

BIT ADDRESSABLE

M16C_INI_BA initialize user C `_bita` variables

*modulename*_CLR_BA cleared C `_bita` type variables

NEAR DATA

M16C_INI_DA initialized C near variables

*modulename*_CLR_DA cleared C near variables

FAR DATA

M16C_INI_FD initialized C far variables

*modulename*_CLR_FD cleared C far variables

NEAR CODE

*modulename*_CO near C code

FAR CODE

*modulename*_FC far C code

HARDWARE INTERRUPTS

*modulename*_FV_0xFFx generated for `_hw_interrupt` vector table entries `xx` is a hex offset

*modulename*_FV the interrupt code

SPECIAL SUBROUTINES

*modulename*_CO_0xFFx generated for `_hw_interrupt` vector table entries `xx` is a hex offset.

*modulename*_FC_HIG the special subroutine code

If you use the **-R** option (or **renamesect** pragma), to specify the name **cm16** must use for a certain section, this name is added to this list. Note that **lcm16** produces a locator map (suffix **.map**) which shows the addresses of all sections used in the application.

7.4 STACK

The stack is defined in the locator description file (`m16c.dsc` in directory `etc`) with the keyword `stack`, which results in a section called `stack`. The description file tells **lcm16** to allocate the stack after all other data sections and the heap.

The stack size can be controlled with the keyword `length=size` in the description file. If you do not specify the stack size, the locator will allocate the rest of the available memory for the stack, as done in the startup code. You can use the locator defined labels `__lc_bs` and `__lc_es` in your application to retrieve the beginning and end address of the stack. Please note that the locator will only allocate a stack section if the application refers to one of the locator defined symbols `__lc_bs` or `__lc_es`. Remember that there must be enough space allocated for the stack, which grows downwards.

The following diagram shows the stack frame when using reentrant functions. The processor mode, user-mode or system-mode, does not influence the stack frame.

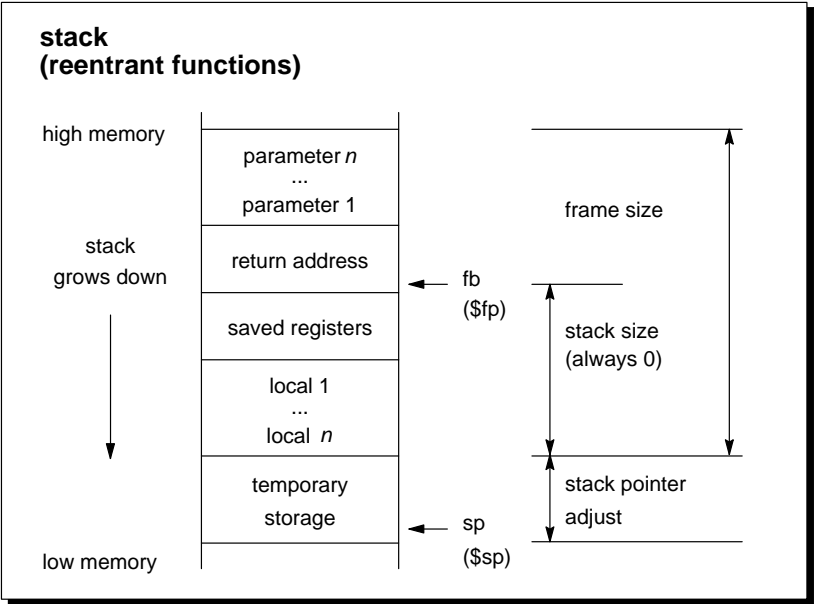


Figure 7-1: Stack diagram

The **stack** saves the return addresses of functions, non-register automatic and parameter variables of reentrant functions.

Automatics and parameters are all accessed using the stack pointer register. The stack pointer `SP` points to the last item pushed on the stack.

The stack frame also contains a so-called virtual frame pointer (`fp`). The virtual frame pointer points to the lower byte of the function's return address. In case of an `_interrupt` function `fp` points to the save contents of the M16C FLG register. All stack offsets in the debug info are relative to this virtual frame pointer. To be able to access automatic variables, the debugger needs to know two offsets, the stack size and the stack pointer adjust.

The stack size is passed as a function constant by the compiler. The stack size is always 0 (zero), because stack pointer adjust information is also generated in the function prologue. The stack pointer adjust reflects the number of pushes/pops done since the functions prologue.

Be aware that an interrupt function pushes both the PC and the current value of the FLG register on the stack.

7.5 HEAP

The heap is only needed when dynamic memory management library functions are used: `malloc()`, `calloc()`, `free()` and `realloc()`. The heap is a reserved area in the data space. So, only if you use one of the memory allocation functions listed above, the locator automatically allocates a heap, as specified in the locator description file with the keyword `heap`.

A special section called `heap` is used for the allocation of the heap area. You can place the heap section anywhere in memory, using the locator description file. You can specify the size of the heap using the keyword `length=size` in the locator description file. If you do not specify the heap size and yet refer to it (e.g. call `malloc()`), the locator will allocate the rest of the available IDATA for the heap. The locator defined labels `__lc_bh` and `__lc_eh` (beginning and end of heap) are used by the library function `sbrk()`, which is called by `malloc()` when memory is needed from the heap.

Please note that, when using the heap, you should not forget to clear all IDATA memory in the startup code.

After editing, you must process the C startup file with **asm16** to make the correct object file. For a detailed description, see the section *Startup Code*.

Example part of the locator description file defining the heap size and location:

```
amode idata
{
    section selection=w;
    heap length=1000;
}
```



The special `heap` section is only allocated when its locator labels are used in the program.

7.6 FLOATING POINT ARITHMETIC

Floating point arithmetic support for the **cm16** is included in software as a separate set of libraries. When linking, the desired floating point library must be specified after the C library. The libraries are reentrant, and only use temporary program stack memory.

To ensure portability of floating point arithmetic, floating point arithmetic for the **cm16** has been implemented adhering to the IEEE-754 standard for floating point arithmetic. See the *IEEE Standard for Binary Floating-Point Arithmetic* document, as published in 1985 by the IEEE Computer Society, for more details on these floating point arithmetic definitions. This document is referred to as IEEE-754 in this manual.

cm16 supports both single and double precision floating point operations, usable via the ANSI C types `float` and `double` respectively. For the sole purpose of speed, also a non-trapping library is included for each memory model. For the library name syntax, see section *C Libraries*.



M16C does not support floating point exceptions.

7.6.1 SPECIAL FLOATING POINT VALUES

Below is a list of special, IEEE-754 defined, floating point values as they can occur during run-time.

Special value	Sign	Exponent	Mantissa
+0.0 (Positive Zero)	0	all zeros	all zeros
-0.0 (Negative Zero)	1	all zeros	all zeros
+INF (Positive Infinite)	0	all ones	all zeros
-INF (Negative Infinite)	1	all ones	all zeros
NaN (Not a number)	0	all ones	all ones

Table 7-2: Special floating point values

7.7 INTERRUPT FUNCTIONS

Interrupt functions may be implemented directly in C, by using the `_interrupt(n)` function qualifier. A function declared with this qualifier differs from a normal function definition in a number of ways:

1. The appropriate interrupt vector, consisting of an interrupt function entry label. For more details, see section *Interrupts*. The vector may be suppressed with the `-v` option.
2. If the `_bank_switch` is not used, all registers that might possibly be corrupted during the execution of the interrupt function are saved on function entry and restored on function exit. Normally, only the registers directly used by the interrupt function will be saved.
3. `_bank_switch` can be used to cause register bank 1 to be used. This saves the overhead of saving registers.
4. The function is terminated with an REIT instruction.

Example:

```
; M16C C compiler v1.0 r9      SN00000000-??? (c) 1998 TASKING, Inc.
; options: -s -gn -Ms
$CASE ON
$OPTJ ON

NAME "hand"

; hand.c 1 int count;
; hand.c 2
; hand.c 3 _interrupt(1) _bank_switch void handler( void )
; hand.c 4 {
GLOBAL _handler
DEFSECT "hand_FC", CODE
SECT "hand_FC"
__int__1:
_handler:
    fset b
; hand.c 5 count++;
    add.w #1,_count
```

```
    ; hand.c      6      }
        fclr  b
        reit

DEFSECT "interpt_co", CODE
SECT "interpt_co"
    POP.W R0
    POP.W R1
    REIT
    DW      0008FH,_handler

    DEFSECT "hand_CLR_DA", DATA, CLEAR
    SECT "hand_CLR_DA"
    GLOBAL      _count
_count:      DS      2
    END
```

7.8 ASSEMBLY LANGUAGE INTERFACING

Assembly language functions can be called from C and vice versa. The names used by **cm16** are case sensitive, so you must tell **asm16** to act case sensitive too, using the `$CASE` control. **cm16** prepends an underscore for the name of the C variable, to distinguish these names from the M16C registers. So, any names used or defined in M16C C must have a leading underscore in assembly code. Internal compiler symbols (run-time library) use two underscores.

When using strict ANSI, and when you call an assembly routine that has a name of e.g. 50 characters, you get a link error "UNRESOLVED EXTERNAL". The reason for it is that the C compiler truncates names to 32 characters, but the assembler and linker do not. The solution is, when calling assembly routines, use names of 31 characters or less (if you do not count the leading '_' for a moment). The same rule applies when you call a C function from your assembly code.

The quickest (and most reliable) way to make an assembly language function, which must conform to M16C C, is to make the body of this function in C, and compile this module with the memory model used by all other C modules. If the assembly function must return something, specify the return type in the 'assembler function' using C syntax, and let it return something. If parameters are used, force code generation for accessing these parameters with a dummy statement (e.g. an assignment) or declare the parameter as volatile and just access it:

```
int assem( char volatile a, char c, int i )
{
    a;
    return( c + i );
}
```

Now compile this module, using the correct memory model. The compiler makes the correct frame, and you can edit the generated assembly module, to make the real assembly function inside this frame.

A second method to create an interface to assembly is to make use of the feature of the compiler to have inline assembly.

Assembly lines in the C source must be introduced by a `'#pragma asm'`, the end is indicated by a `'#pragma endasm'`. For example:

```
int assem( char c, int i )
{
    int j;
    j = i;
    #pragma asm
        MOV.B #01,R0L
    #pragma endasm
    j = c;
}
```

When the assembly does not change any registers, like in the example above, also `'#pragma asm_noflush'` may be used instead of `'#pragma asm'`. For an explanation of the used pragmas see the section *Pragmas*.



APPENDIX

A

FLEXIBLE LICENSE MANAGER (FLEXlm)



A

APPENDIX

1 INTRODUCTION

This appendix discusses Globetrotter Software's Flexible License Manager and how it is integrated into the TASKING toolchain. It also contains descriptions of the Flexible License Manager license administration tools that are included with the package, the daemon log file and its contents, and the use of daemon options files to customize your use of the TASKING toolchain.

2 LICENSE ADMINISTRATION

2.1 OVERVIEW

The Flexible License Manager (FLEXlm) is a set of utilities that, when incorporated into software such as the TASKING toolchain, provides for managing access to the software.

The following terms are used to describe FLEXlm concepts and software components:

feature	A feature could be any of the following: <ul style="list-style-type: none">• A TASKING software product.• A software product from another vendor.
license	The right to use a feature. FLEXlm restricts licenses for features by counting the number of licenses for features in use when new requests are made by the application software.
client	A TASKING application program.
daemon	A process that "serves" clients. Sometimes referred to as a <i>server</i> .
vendor daemon	The daemon that dispenses licenses for the requested features. This daemon is built by an application's vendor, and contains the vendor's personal encryption code. Tasking is the vendor daemon for the TASKING software.

license daemon

The daemon process that sends client processes to the correct vendor daemon on the correct machine. The same license daemon is used by all applications from all vendors, as this daemon neither performs encryption nor dispenses licenses. The license daemon processes no user requests on its own, but forwards these requests to other daemons (the vendor daemons).

server node A computer system that is running both the license and vendor daemon software. The server node will contain all the dynamic information regarding the usage of all the features.

license file An end-user specific file that contains descriptions of the server nodes that can run the license daemons, the various vendor daemons, and the restrictions for all the licensed features.

The TASKING software is granted permission to run by FLEXlm daemons; the daemons are started when the TASKING toolchain is installed and run continuously thereafter. Information needed by the FLEXlm daemons to perform access management is contained in a license data file that is created during the toolchain installation process. As part of their normal operation, the daemons log their actions in a daemon log file, which can be used to monitor usage of the TASKING toolchain.

The following sections discuss:

- Installation of the FLEXlm daemons to provide for access to the TASKING toolchain.
- Customizing your use of the toolchain through the use of a daemon options file.
- Utilities that are provided to assist you in performing license administration functions.
- The daemon log file and its contents.

For additional information regarding the use of FLEXlm, refer to the chapter *Software Installation*.

2.2 PROVIDING FOR UNINTERRUPTED FLEXLM OPERATION

TASKING products licensed through FLEXlm contain a number of utilities for managing licenses. These utilities are bundled in the form of an extra product under the name SW000098. TASKING products themselves contain two additional files for FLEXlm in a *flexlm* subdirectory:

Tasking	The Tasking daemon (vendor daemon).
license.dat	A template license file.

If you have already installed FLEXlm (e.g. as part of another product) then it is not needed to install the bundled SW000098. After installing SW000098 on UNIX, the directory `/usr/local/flexlm` will contain two subdirectories, `bin` and `licenses`. After installing SW000098 on Windows the directory `c:\flexlm` will contain the subdirectory `bin`. The exact location may differ if FLEXlm has already been installed as part of a non-TASKING product but in general there will be a directory for executables such as `bin`. That directory must contain a copy of the **Tasking** daemon shipped with every TASKING product. It also contains the files:

lmgrd	The FLEXlm daemon (license daemon).
lm*	A group of FLEXlm license administration utilities.

Next to it, a license file must be present containing the information of all licenses. This file is usually called `license.dat`. The default location of the license file is in directory `c:\flexlm` for Windows and in `/usr/local/flexlm/licenses` for UNIX. If you did install SW000098 then the `licenses` directory on UNIX will be empty, and on Windows the file `license.dat` will be empty. In that case you can copy the `license.dat` file from the product to the `licenses` directory after filling in the data from your "License Information Form".



Be very careful not to overwrite an existing `license.dat` file because it contains valuable data.

Example `license.dat`:

```
SERVER HOSTNAME HOSTID PORT
DAEMON Tasking /usr/local/flexlm/bin/Tasking
FEATURE SW008002-32 Tasking 3.000 EXPDATE NUSERS PASSWORD SERIAL
```

After modifications from a license data sheet (example):

```
SERVER elliot 5100520c 7594
DAEMON Tasking /usr/local/flexlm/bin/Tasking
FEATURE SW008002-32 Tasking 3.000 1-jan-00 4 0B1810310210A6894 "123456"
```

If the `license.dat` file already exists then you should make sure that it contains the DAEMON and FEATURE lines from your license data sheet. An appropriate SERVER line should already be present in that case. You should only add a new SERVER line if no SERVER line is present. The third field of the DAEMON line is the pathname to the **Tasking** daemon and you may change it if necessary.

The default location for the license file on Windows is:

```
c:\flexlm\license.dat
```

On UNIX this is:

```
/usr/local/flexlm/licenses/license.dat
```

If the pathname of the resulting license file differs from this default location then you must set the environment variable **LM_LICENSE_FILE** to the correct pathname. If you have more than one product using the FLEXlm license manager you can specify multiple license files by separating each pathname (*lfp_{path}*) with a ';' (on UNIX also ':') :

Windows:

```
set LM_LICENSE_FILE=lfppath;lfppath...
```

UNIX:

```
setenv LM_LICENSE_FILE lfppath:lfppath...
```

If you are running the TASKING software on multiple nodes, you have three options for making your license file available on all the machines:

1. Place the license file in a partition which is available (via NFS on Unix systems) to all nodes in the network that need the license file.
2. Copy the license file to all of the nodes where it is needed.
3. Set LM_LICENSE_FILE to "*port@host*", where *host* and *port* come from the SERVER line in the license file.

When the main license daemon **lmgrd** already runs it is sufficient to type the command:

```
lmreread
```

for notifying the daemon that the `license.dat` file has been changed. Otherwise, you must type the command:

```
lmgrd >/usr/tmp/license.log &
```

Both commands reside in the flexlm bin directory mentioned before.

2.3 DAEMON OPTIONS FILE

It is possible to customize the use of TASKING software using a daemon options file. This options file allows you to reserve licenses for specified users or groups of users, to restrict access to the TASKING toolchain, and to set software timeouts. The following table lists the keywords that are recognized at the start of a line of a daemon options file.

Keywords	Function
RESERVE	Ensure that TASKING software will always be available to one or more users or on one or more host computer systems.
INCLUDE	Specify a list of users who are allowed exclusive access to the TASKING software.
EXCLUDE	Specify a list of users who are not allowed to use the TASKING software.
GROUP	Specify a group of users for use in the other commands.
TIMEOUT	Allow licenses that are idle for a specified time to be returned to the free pool, for use by someone else.
NOLOG	Causes messages of the specified type to be filtered out of the daemon's log output.

Table A-1: Daemon options file keywords

In order to use the daemon options capability, you must create a daemon options file and list its pathname as the fourth field on the **DAEMON** line for the **Tasking** daemon in the license file. For example, if the daemon options were in file `/usr/local/flexlm/Tasking.opt` (UNIX), then you would modify the license file **DAEMON** line as follows:

```
DAEMON Tasking /usr/local/Tasking /usr/local/flexlm/Tasking.opt
```

A daemon options file consists of lines in the following format:

```
RESERVE      number feature {USER | HOST | DISPLAY | GROUP} name
INCLUDE      feature {USER | HOST | DISPLAY | GROUP} name
EXCLUDE      feature {USER | HOST | DISPLAY | GROUP} name
GROUP        name <list_of_users>
TIMEOUT      feature timeout_in_seconds
NOLOG        {IN | OUT | DENIED | QUEUED}
REPORTLOG    file
```

Lines beginning with the sharp character (#) are ignored, and can be used as comments. For example, the following options file would reserve one copy of feature SWxxxxxx-xx for user “pat”, three copies for user “lee”, and one copy for anyone on a computer with the hostname of “terry”; and would cause QUEUED messages to be omitted from the log file. In addition, user “joe” and group “pinheads” would not be allowed to use the feature SWxxxxxx-xx:

```
GROUP        pinheads moe larry curley
RESERVE 1    SWxxxxxx-xx USER pat
RESERVE 3    SWxxxxxx-xx USER lee
RESERVE 1    SWxxxxxx-xx HOST terry
EXCLUDE      SWxxxxxx-xx USER joe
EXCLUDE      SWxxxxxx-xx GROUP pinheads
NOLOG        QUEUED
```

3 LICENSE ADMINISTRATION TOOLS

The following utilities are provided to facilitate license management by your system administrator. In certain cases, execution access to a utility is restricted to users with root privileges. Complete descriptions of these utilities are provided at the end of this section.

lmcksum

Prints license checksums.

lmdiag (Windows only)

Diagnoses license checkout problems.

lmdown

Gracefully shuts down all license daemons (both **lmgrd** all vendor daemons, such as **Tasking**) on the license server.

lmgrd

The main daemon program for FLEXlm.

lmbostid

Reports the hostid of a system.

lmremove

Removes a single user's license for a specified feature.

lmreread

Causes the license daemon to reread the license file and start any new vendor daemons.

lmstat

Helps you monitor the status of all network licensing activities.

lmswitchr

Switches the report log file.

lmver

Reports the FLEXlm version of a library or binary file.

lmtools (*Windows only*)

This is a graphical Windows version of the license administration tools.

3.1 LMCKSUM

Name

lmcksum – print license checksums

Synopsis

lmcksum [**-c** *license_file*] [**-k**]

Description

The **lmcksum** program will perform a checksum of a license file. This is useful to verify data entry errors at your location. **lmcksum** will print a line-by-line checksum for the file as well as an overall file checksum.

The following fields participate in the checksum:

- hostid on the SERVER lines
- daemon name on the DAEMON lines
- feature name, version, daemon name, expiration date, # of licenses, encryption code, vendor string and hostid on the FEATURE lines
- daemon name and encryption code on FEATURESET lines

Options

-c *license_file*

Use the specified *license_file*. If no **-c** option is specified, **lmcksum** looks for the environment variable LM_LICENSE_FILE in order to find the license file to use. If that environment variable is not set, **lmcksum** looks for the file c:\flexlm\license.dat (Windows), or /usr/local/flexlm/licenses/license.dat (UNIX).

-k

Case-sensitive checksum. If this option is specified, **lmcksum** will compute the checksum using the exact case of the FEATURE's and FEATURESET's encryption code.

3.2 LMDIAG (Windows only)

Name

lmdiag – diagnose license checkout problems

Synopsis

lmdiag [**-c** *license_file*] [**-n**] [*feature*]

Description

lmdiag (Windows only) allows you to diagnose problems when you cannot check out a license.

If no *feature* is specified, **lmdiag** will operate on all features in the license file(s) in your path. **lmdiag** will first print information about the license, then attempt to check out each license. If the checkout succeeds, **lmdiag** will indicate this. If the checkout fails, **lmdiag** will give you the reason for the failure. If the checkout fails because **lmdiag** cannot connect to the license server, then you have the option of running "extended connection diagnostics".

These extended diagnostics attempt to connect to each port on the license server node, and can detect if the port number in the license file is incorrect. **lmdiag** will indicate each port number that is listening, and if it is an **lmgrd** process, **lmdiag** will indicate this as well. If **lmdiag** finds the vendor daemon for the feature being tested, then it will indicate the correct port number for the license file to correct the problem.

Parameters

feature Diagnose this feature only.

Options

-c *license_file*

Diagnose the specified *license_file*. If no **-c** option is specified, **lmdiag** looks for the environment variable `LM_LICENSE_FILE` in order to find the license file to use. If that environment variable is not set, **lmdiag** looks for the file `c:\flexlm\license.dat` (Windows), or `/usr/local/flexlm/licenses/license.dat` (UNIX).

-n

Run in non-interactive mode; **lmdiag** will not prompt for any input in this mode. In this mode, extended connection diagnostics are not available.

3.3 LMDOWN

Name

lmdown – graceful shutdown of all license daemons

Synopsis

lmdown [**-c** *license_file*] [**-q**]

Description

The **lmdown** utility allows for the graceful shutdown of all license daemons (both **lmgrd** and all vendor daemons, such as **Tasking**) on all nodes. You may want to protect the execution of **lmdown**, since shutting down the servers causes users to lose their licenses. See the **-p** option in Section 3.4, **lmgrd**.

lmdown sends a message to every license daemon asking it to shut down. The license daemons write out their last messages to the log file, close the file, and exit. All licenses which have been given out by those daemons will be revoked, so that the next time a client program goes to verify his license, it will not be valid.

Options

-c *license_file*

Use the specified *license_file*. If no **-c** option is specified, **lmdown** looks for the environment variable **LM_LICENSE_FILE** in order to find the license file to use. If that environment variable is not set, **lmdown** looks for the file `c:\flexlm\license.dat` (Windows), or `/usr/local/flexlm/licenses/license.dat` (UNIX).

-q

Quiet mode. If this switch is not specified, **lmdown** asks for confirmation before asking the license daemons to shut down. If this switch is specified, **lmdown** will not ask for confirmation.



lmgrd, **lmstat**, **lmreread**

3.4 LMGRD

Name

lmgrd – flexible license manager daemon

Synopsis

lmgrd [**-c** *license_file*] [**-l** *logfile*] [**-2 -p**] [**-t** *timeout*] [**-s** *interval*]

Description

lmgrd is the main daemon program for the FLEXlm distributed license management system. When invoked, it looks for a license file containing all required information about vendors and features. On UNIX systems, it is strongly recommended that **lmgrd** be run as a non-privileged user (not root).

Options

-c *license_file*

Use the specified *license_file*. If no **-c** option is specified, **lmgrd** looks for the environment variable `LM_LICENSE_FILE` in order to find the license file to use. If that environment variable is not set, **lmgrd** looks for the file `c:\flexlm\license.dat` (Windows), or `/usr/local/flexlm/licenses/license.dat` (UNIX).

-l *logfile*

Specifies the output log file to use. Instead of using the **-l** option you can use output redirection (**>** or **>>**) to specify the name of the output log file.

-2 -p

Restricts usage of **lmdown**, **lmreread**, and **lmremove** to a FLEXlm administrator who is by default root. If there is a UNIX group called "lmadmin" then use is restricted to only members of that group. If root is not a member of this group, then root does not have permission to use any of the above utilities.

-t *timeout*

Specifies the *timeout* interval, in seconds, during which the license daemon must complete its connection to other daemons if operating in multi-server mode. The default value is 10 seconds. A larger value may be desirable if the daemons are being run on busy systems or a very heavily loaded network.

-s *interval* Specifies the log file timestamp *interval*, in minutes. The default is 360 minutes. This means that every six hours **lmgrd** logs the time in the log file.



lmdown, lmstat

3.5 LMHOSTID

Name

lmhostid – report the hostid of a system

Synopsis

lmhostid

Description

lmhostid calls the FLEXlm version of `gethostid` and displays the results.

The output of **lmhostid** looks like this:

```
lmhostid - Copyright (C) 1989, 1999 Globetrotter Software, Inc.  
The FLEXlm host ID of this machine is "1200abcd"
```

Options

lmhostid has no command line options.

3.6 LMREMOVE

Name

lmremove – remove specific licenses and return them to license pool

Synopsis

lmremove [**-c** *license_file*] *feature user host* [*display*]

Description

The **lmremove** utility allows the system administrator to remove a single user's license for a specified feature. This could be required in the case where the licensed user was running the software on a node that subsequently crashed. This situation will sometimes cause the license to remain unusable. **lmremove** will allow the license to return to the pool of available licenses.

lmremove will remove all instances of “user” on node “host” on display “display” from usage of “feature”. If the optional **-c file** is specified, the indicated file will be used as the license file. Since removing a user's license can be disruptive, execution of **lmremove** is restricted to users with root privileges.

Options

-c *license_file*

Use the specified *license_file*. If no **-c** option is specified, **lmremove** looks for the environment variable `LM_LICENSE_FILE` in order to find the license file to use. If that environment variable is not set, **lmremove** looks for the file `c:\flexlm\license.dat` (Windows), or `/usr/local/flexlm/licenses/license.dat` (UNIX).



lmstat

3.7 LMREREAD

Name

lmreread – tells the license daemon to reread the license file

Synopsis

lmreread [**-c** *license_file*]

Description

lmreread allows the system administrator to tell the license daemon to reread the license file. This can be useful if the data in the license file has changed; the new data can be loaded into the license daemon without shutting down and restarting it.

The license administrator may want to protect the execution of **lmreread**. See the **-p** option in Section 3.4, *lmgrd* for details about securing access to **lmreread**.

lmreread uses the license file from the command line (or the default file, if none specified) only to find the license daemon to send it the command to reread the license file. The license daemon will always reread the file that it loaded from the original path. If you need to change the path to the license file read by the license daemon, then you must shut down the daemon and restart it with that new license file path.

You cannot use **lmreread** if the *SERVER* node names or port numbers have been changed in the license file. In this case, you must shut down the daemon and restart it in order for those changes to take effect.

lmreread does not change any option information specified in an options file. If the new license file specifies a different options file, that information is ignored. If you need to reread the options file, you must shut down (**lmdown**) the daemon and restart it.

Options

-c *license_file*

Use the specified *license_file*. If no **-c** option is specified, **lmreread** looks for the environment variable *LM_LICENSE_FILE* in order to find the license file to use. If that environment variable is not set, **lmreread** looks for the file *license.dat* in the default location.



lmdown

3.8 LMSTAT

Name

lmstat – report status on license manager daemons and feature usage

Synopsis

```
lmstat [ -a ] [ -A ] [ -c license_file ] [ -f feature ]
      [ -l regular_expression ] [ -s server ] [ -S daemon ] [ -t timeout ]
```

Description

License administration is simplified by the **lmstat** utility. **lmstat** allows you to instantly monitor the status of all network licensing activities.

lmstat allows a system administrator to monitor license management operations including:

- Which daemons are running
- Users of individual features
- Users of features served by a specific DAEMON

Options

-a Display all information.

-A List all active licenses.

-c *license_file*

Use the specified *license_file*. If no **-c** option is specified, **lmstat** looks for the environment variable LM_LICENSE_FILE in order to find the license file to use. If that environment variable is not set, **lmstat** looks for the file c:\flexlm\license.dat (Windows), or /usr/local/flexlm/licenses/license.dat (UNIX).

-f *feature* List all users of the specified *feature*(s).

-l *regular_expression*

List all users of the features matching the given *regular_expression*.

-s *server* Display the status of the specified *server* node(s).

-S *daemon* List all users of the specified *daemon*'s features.

- t *timeout*** Specifies the amount of time, in seconds, **lmstat** waits to establish contact with the servers. The default value is 10 seconds. A larger value may be desirable if the daemons are being run on busy systems or a very heavily loaded network.



lmgrd

3.9 LMSWITCHR (Windows only)

Name

lmswitchr – switch the report log file

Synopsis

lmswitchr [**-c** *license_file*] *feature new-file*

or:

lmswitchr [**-c** *license_file*] *vendor new-file*

Description

lmswitchr (Windows only) switches the report writer (REPORTLOG) log file. It will also start a new REPORTLOG file if one does not already exist.

Parameters

<i>feature</i>	Any feature this daemon supports.
<i>vendor</i>	The name of the vendor daemon (such as Tasking).
<i>new-file</i>	New file path.

Options

-c *license_file* Use the specified *license_file*. If no **-c** option is specified, **lmswitchr** looks for the environment variable LM_LICENSE_FILE in order to find the license file to use. If that environment variable is not set, **lmswitchr** looks for the file `c:\flexlm\license.dat` (Windows), or `/usr/local/flexlm/licenses/license.dat` (UNIX).

3.10 LMVER

Name

lmver – report the FLEXlm version of a library or binary file

Synopsis

lmver *filename*

Description

The **lmver** utility reports the FLEXlm version of a library or binary file.

Alternatively, on UNIX systems, you can use the following commands to get the FLEXlm version of a binary:

strings *file* | grep Copy

Parameters

filename Name of the executable of the product.

3.11 LICENSE ADMINISTRATION TOOLS FOR WINDOWS

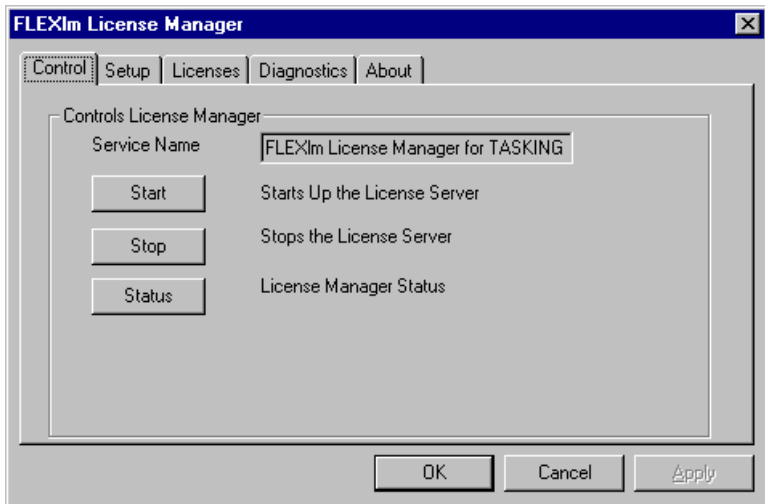
3.11.1 LMTOOLS FOR WINDOWS

For the 32 Bit Windows Platforms, an **lmtools.exe** Windows program is provided. It has the same functionality as listed in the previous sections but is graphically-oriented. Simply run the program (Start | Programs | TASKING FLEXlm | FLEXlm Tools) and choose a button for the functionality required. Refer to the previous sections for information about the options of each feature. The command line interface is replaced by pop-up dialogs that can be filled out. The central EDIT field is where the license file path is placed. This will be used for all other functions and replaces the "**-c** *license_file*" argument in the other utilities.

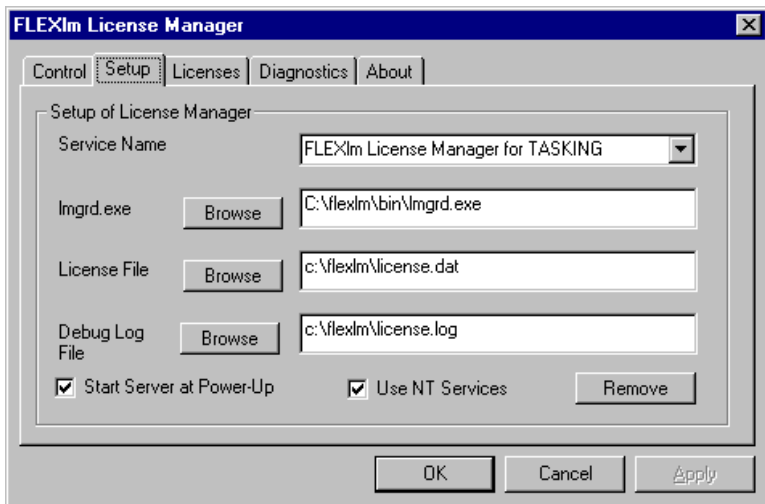
The HOSTID button displays the hostid's for the computer on which the program is running. The TIME button prints out the system's internal time settings, intended to diagnose any time zone problems. The TCP Settings button is intended to fix a bug in the Microsoft TCP protocol stack which has a symptom of very slow connections to computers. After pressing this button, the system will need to be rebooted for the settings to become effective.

3.11.2 FLEXLM LICENSE MANAGER FOR WINDOWS

lmgrd.exe can be run manually or using the graphical Windows tool. You can start this tool from the FLEXlm program folder. Click on Start | Programs | TASKING FLEXlm | FLEXlm Tools



From the Control tab you can start, stop, and check the status of your license server. Select the Setup tab to enter information about your license server.



Select the **Control** tab and click the **Start** button to start your license server. **lmgrd.exe** will be launched as a background application with the license file and debug log file locations passed as parameters.

If you want **lmgrd.exe** to start automatically on NT, select the **Use NT Services** check box and **lmgrd.exe** will be installed as an NT service. Next, select the **Start Server at Power-UP** check box.

The **Licenses** tab provides information about the license file and the **Advanced** tab allows you to perform diagnostics and check versions.

4 THE DAEMON LOG FILE

The FLEXlm daemons all generate log files containing messages in the following format:

mm/dd hh:mm (DAEMON name) message

Where:

mm/dd hh:mm Is the month/day hour:minute that the message was logged.

DAEMON name Either “license daemon” or the string from the DAEMON line that describes your daemon.

In the case where a single copy of the daemon cannot handle all of the requested licenses, an optional “_” followed by a number indicates that this message comes from a forked daemon.

message The text of the message.

The log files can be used to:

- Inform you when it may be necessary to update your application software licensing arrangement.
- Diagnose configuration problems.
- Diagnose daemon software errors.

The messages are grouped below into the above three categories, with each message followed by a brief description of its meaning.

4.1 INFORMATIONAL MESSAGES

Connected to node

This daemon is connected to its peer on node *node*.

CONNECTED, master is name

The license daemons log this message when a quorum is up and everyone has selected a master.

DEMO mode supports only one SERVER host!

An attempt was made to configure a demo version of the software for more than one server host.

DENIED: N feature to user (mm/dd/yy hh:mm)

user was denied access to *N* licenses of *feature*. This message may indicate a need to purchase more licenses.

EXITING DUE TO SIGNAL mm

EXITING with code mm

All daemons list the reason that the daemon has exited.

EXPIRED: feature

feature has passed its expiration date.

IN: feature by user (N licenses) (used: d:hh:mm:ss) (mm/dd/yy hh:mm)

user has checked back in *N* licenses of *feature* at *mm/dd/yy hh:mm*.

IN server died: feature by user (number licenses) (used: d:hh:mm:ss) (mm/dd/yy hh:mm)

user has checked in *N* licenses by virtue of the fact that his server died.

License Manager server started

The license daemon was started.

Lost connection to host

A daemon can no longer communicate with its peer on node *host*, which can cause the clients to have to reconnect, or cause the number of daemons to go below the minimum number, in which case clients may start exiting. If the license daemons lose the connection to the master, they will kill all the vendor daemons; vendor daemons will shut themselves down.

Lost quorum

The daemon lost quorum, so will process only connection requests from other daemons.

MASTER SERVER died due to signal mm

The license daemon received fatal signal *mm*.

MULTIPLE xxx servers running. Please kill, and restart license daemon

The license daemon has detected that multiple copies of vendor daemon *xxx* are running. The user should kill all *xxx* daemon processes and re-start the license daemon.

OUT: feature by user (N licenses) (mm/dd/yy hh:mm)

user has checked out *N* licenses of *feature* at *mm/dd/yy hh:mm*

Removing clients of children

The top-level daemon logs this message when one of the child daemons dies.

RESERVE feature for HOST name***RESERVE feature for USER name***

A license of *feature* is reserved for either user *name* or host *name*.

REStarted xxx (internet port mm)

Vendor daemon *xxx* was restarted at internet port *mm*.

Retrying socket bind (address in use)

The license servers try to bind their sockets for approximately 6 minutes if they detect *address in use* errors.

Selected (EXISTING) master node

This license daemon has selected an existing master (node) as the master.

SERVER shutdown requested

A daemon was requested to shut down via a user-generated kill command.

[NEW] Server started for: feature-list

A (possibly new) server was started for the features listed.

Shutting down xxx

The license daemon is shutting down the vendor daemon *xxx*.

SIGCHLD received. Killing child servers

A vendor daemon logs this message when a shutdown was requested by the license daemon.

Started name

The license daemon logs this message whenever it starts a new vendor daemon.

Trying connection to node

The daemon is attempting a connection to *node*.

4.2 CONFIGURATION PROBLEM MESSAGES

hostname: Not a valid server host, exiting

This daemon was run on an invalid hostname.

hostname: Wrong hostid, exiting

The hostid is wrong for *hostname*.

BAD CODE for feature-name

The specified feature name has a bad encryption code.

CANNOT OPEN options file “file”

The options file specified in the license file could not be opened.

Couldn't find a master

The daemons could not agree on a master.

license daemon: lost all connections

This message is logged when all the connections to a server are lost, which often indicates a network problem.

lost lock, exiting

Error closing lock file

Unable to re-open lock file

The vendor daemon has a problem with its lock file, usually because of an attempt to run more than one copy of the daemon on a single node. Locate the other daemon that is running via a **ps** command, and kill it with **kill -9**.

NO DAEMON line for daemon

The license file does not contain a DAEMON line for *daemon*.

No “license” service found

The TCP *license* service did not exist in `/etc/services`.

No license data for “feat”, feature unsupported

There is no feature line for *feat* in the license file.

No features to serve!

A vendor daemon found no features to serve. This could be caused by bad data in the license file.

UNSUPPORTED FEATURE request: feature by user

The *user* has requested a feature that this vendor daemon does not support. This can happen for a number of reasons: the license file is bad, the feature has expired, or the daemon is accessing the wrong license file.

Unknown host: hostname

The hostname specified on a `SERVER` line in the license file does not exist in the network database (probably `/etc/hosts`).

lm_server: lost all connections

This message is logged when all the connections to a server are lost. This probably indicates a network problem.

NO DAEMON lines, exiting

The license daemon logs this message if there are no `DAEMON` lines in the license file. Since there are no vendor daemons to start, there is nothing to do.

NO DAEMON line for name

A vendor daemon logs this error if it cannot find its own `DAEMON` name in the license file.

4.3 DAEMON SOFTWARE ERROR MESSAGES

accept: message

An error was detected in the accept system call.

ATTEMPT TO START VENDOR DAEMON xxx with NO MASTER

A vendor daemon was started with no master selected. This is an internal consistency error in the daemons.

BAD PID message from mm: pid: xxx (msg)

A top-level vendor daemon received an invalid PID message from one of its children (daemon number xxx).

BAD SCONNECT message: (message)

An invalid “server connect” message was received.

Cannot create pipes for server communication

The pipe call failed.

Can't allocate server table space

A malloc error. Check swap space.

Connection to node TIMED OUT

The daemon could not connect to *node*.

Error sending PID to master server

The vendor server could not send its PID to the top-level server in the hierarchy.

Illegal connection request to DAEMON

A connection request was made to DAEMON, but this vendor daemon is not DAEMON.

Illegal server connection request

A connection request came in from another server without a DAEMON name.

KILL of child failed, errno = mm

A daemon could not kill its child.

No internet port number specified

A vendor daemon was started without an internet port.

Not enough descriptors to re-create pipes

The “top-level” daemon detected one of its sub-daemon’s death. In trying to restart the chain of sub-daemons, it was unable to get the file descriptors to set up the pipes to communicate. This is a fatal error, and the daemons must be re-started.

read: error message

An error in a read system call was detected.

recycle_control BUT WE DIDN'T HAVE CONTROL

The hierarchy of vendor daemons has become confused over who holds the control token. This is an internal error.

return_reserved: can't find feature listhead

When a daemon is returning a reservation to the “free reservation” list, it could not find the listhead of features.

select: message

An error in a select system call was detected.

Server exiting

The server is exiting. This is normally due to an error.

SHELLO for wrong DAEMON

This vendor daemon was sent a “server hello” message that was destined for a different DAEMON.

Unsolicited msg from parent!

Normally, the top-level vendor daemon sends no unsolicited messages. If one arrives, this message is logged. This is a bug.

WARNING: CORRUPTED options list (o->next == 0)***Options list TERMINATED at bad entry***

An internal inconsistency was detected in the daemon’s option list.

5 FLEXLM LICENSE ERRORS

FLEXlm license error, encryption code in license file is inconsistent

Check the contents of the license file using the license data sheet for the product. Correct the license file and run the **lmreread** command.

However, do not change the last (fourth) field of a SERVER line in the license file. This cannot have any effect on the error message but changing it will cause other problems.

license file does not support this version

If this is a first time install then follow the procedure for the error message:

```
FLEXlm license error, encryption code in license file is
inconsistent
```

because there may be a typo in the fourth field of a FEATURE line of your license file. In all other cases you need a new license because the current license is for an older version of the product.

Replace the FEATURE line for the old version of the product with a FEATURE line for the new version (it can be found on the new license data sheet). Run the **lmreread** command afterwards. You can have only one version of a feature (previous versions of the product will continue to work).

FLEXlm license error, cannot find license file

Make sure the license file exists. If the pathname printed on the line after the error message is incorrect, correct this by setting the `LM_LICENSE_FILE` environment variable to the full pathname of the license file.

FLEXlm license error, cannot read license file

Every user needs to have read access on the license file and at least execute access on every directory component in the pathname of the license file. Write access is never needed. Read access on directories is recommended.

FLEXlm license error, no such feature exists

Check the license file. There should be a line starting with:

```
FEATURE SWiiiiiii-jj
```


where "iiiiii" is a six digit software code and "jj" is a two digit host code for identifying a compatible host architecture. During product installations the product code is shown, e.g. SW008002, SW019002. The number in the software code is the same as the number in the product code except that the first number may contain an extra leading zero (it must be six digits long).

The line after the license error message describes the expected feature format and includes the host code.

Correct the license file using the license data sheet for the product and run the **lmreread** command. There is one catch: do not add extra SERVER lines or change existing SERVER lines in the license file.

FLEXlm license error, license server does not support this feature

If the LM_LICENSE_FILE variable has been set to the format *number@host* then see first the solution for the message:

```
FLEXlm license error, no such feature exists
```

Run the **lmreread** program to inform the license server about a changed license data file. If **lmreread** succeeds informing the license server but the error message persists, there are basically three possibilities:

1. The license key is incorrect. If this is the case then there must be an error message in the log file of **lmgrd**. Correct the key using the license data sheet for the product. Finally rerun **lmreread**. The log file of **lmgrd** is usually specified to **lmgrd** at startup with the **-l** option or with **>**.
2. Your network has more than one FLEXlm license server daemon and the default license file location for **lmreread** differs from the default assumed by the program. Also, there must be more than one license file. Try one of the following solutions on the same host which produced the error message:

- type:

```
lmreread -c /usr/local/flexlm/licenses/license.dat
```

- set LM_LICENSE_FILE to the license file location and retry the **lmreread** command.
- use the **lmreread** program supplied with the product SW000098, Flexible License Manager. SW000098 is bundled with all TASKING products.

3. There is a protocol version mismatch between **lmgrd** and the daemon with the name "Tasking" (the vendor daemon according to FLEXlm terminology) or there is some other internal error. These errors are always written to the log file of **lmgrd**. The solution is to upgrade the **lmgrd** daemon to the one supplied in SW000098, the bundled Flexible License Manager product.

On the other hand, if **lmreread** complains about not being able to connect to the license server then follow the procedure described in the next section for the error message "Cannot read license file data from server". The only difference with the current situation is that not the product but a license management utility shows a connect problem.

FLEXlm license error, Cannot read license file data from server

This indicates that the program could not connect to the license server daemon. This can have a number of causes. If the program did not immediately print the error message but waited for about 30 seconds (this can vary) then probably the license server host is down or unreachable. If the program responded immediately with the error message then check the following if the LM_LICENSE_FILE variable has been set to the format *number@host*:

- is the number correct? It should match the fourth field of a SERVER line in the license file on the license server host. Also, the host name on that SERVER line should be the same as the host name set in the LM_LICENSE_FILE variable. Correct LM_LICENSE_FILE if necessary.

In any case one should verify if the license server daemon is running. Type the following command on the host where the license server daemon (**lmgrd**) is supposed to run.

On SunOS 4.x:

```
ps wwax | grep lmgrd | grep -v grep
```

On HP-UX or SunOS 5.x (Solaris 2.x):

```
ps -ef | grep lmgrd | grep -v grep
```

If the command does not produce any output then the license server daemon is not running. See below for an example how to start **lmgrd**.

Make sure that both license server daemon (**lmgrd**) and the program are using the same license data. All TASKING products use the license file `/usr/local/flexlm/licenses/license.dat` unless overruled by the environment variable `LM_LICENSE_FILE`. However, not all existing **lmgrd** daemons may use the same default. In case of doubt, specify the license file pathname with the `-c` option when starting the license server daemon. For example:

```
lmgrd -c /usr/local/flexlm/licenses/license.dat \  
-l /usr/local/flexlm/licenses/license.log &
```

and set the `LM_LICENSE_FILE` environment variable to the `license.dat` pathname mentioned with the `-c` option of **lmgrd** before running any license based program (including **lmreread**, **lmstat**, **lmdown**). If **lmgrd** and the program run on different hosts, transparent access to the license file is assumed in the situation described above (e.g. NFS). If this is not the case, make a local copy of the license file (not recommended) or set `LM_LICENSE_FILE` to the form *number@host*, as described earlier.

If none of the above seems to apply (i.e. **lmgrd** was already running and `LM_LICENSE_FILE` has been set correctly) then it is very likely that there is a TCP port mismatch. The fourth field of a `SERVER` line in the license file specifies a TCP port number. That number can be changed without affecting any license. However, it must never be changed while the license server daemon is running. If it has been changed, change it back to the original value. If you do not know the original number anymore, restart the license server daemon after typing the following command on the license server host:

```
kill PID
```

where `PID` is the process id of **lmgrd**.

6 FREQUENTLY ASKED QUESTIONS (FAQS)

6.1 LICENSE FILE QUESTIONS

I've received FLEXlm license files from 2 different companies. Do I have to combine them?

You don't have to combine license files. Each license file that has any 'counted' lines (the 'number of licenses' field is >0) requires a server. It's perfectly OK to have any number of separate license files, with different **lmgrd** server processes supporting each file. Moreover, since **lmgrd** is a lightweight process, for sites without system administrators, this is often the simplest (and therefore recommended) way to proceed. With v6+ **lmgrd/lmdown/lmreread**, you can stop/reread/restart a single vendor daemon (of any FLEXlm version). This makes combining licenses more attractive than previously. Also, if the application is v6+, using 'dir/*.lic' for license file management behaves like combining licenses without physically combining them.

When is it recommended to combine license files?

Many system administrators, especially for larger sites, prefer to combine license files to ease administration of FLEXlm licenses. It's purely a matter of preference.

Does FLEXlm handle dates in the year 2000 and beyond?

Yes. The FLEXlm date format uses a 4-digit year. Dates in the 20th century (19xx) can be abbreviated to the last 2 digits of the year (xx), and use of this feature is quite widespread. Dates in the year 2000 and beyond must specify all 4 year digits.

6.2 FLEXLM VERSION

Which FLEXlm versions does TASKING deliver?

For Windows we deliver FLEXlm v6.1 and for UNIX we deliver v2.4.

I have products from several companies at various FLEXlm version levels. Do I have to worry about how these versions work together?

If you're not combining license files from different vendors, the simplest thing to do is make sure you use the tools (especially **lmgrd**) that are shipped by each vendor.

lmgrd will always correctly support older versions of vendor daemons and applications, so it's **always** safe to use the latest version of **lmgrd** and the other FLEXlm utilities. If you've combined license files from 2 vendors, you **must** use the latest version of **lmgrd**.

If you've received 2 versions of a product from the same vendor, you must use the latest vendor daemon they sent you. An older vendor daemon with a newer client will cause communication errors.

Please ignore letters appended to FLEXlm versions, i.e., v2.4d. The appended letter indicates a patch, and does NOT indicate any compatibility differences. In particular, some elements of FLEXlm didn't require certain patches, so a 2.4 **lmgrd** will work successfully with a 2.4b vendor daemon.

I've received a new copy of a product from a vendor, and it uses a new version of FLEXlm. Is my old license file still valid?

Yes. Older FLEXlm license files are always valid with newer versions of FLEXlm.

6.3 WINDOWS QUESTIONS

What Windows Host Platforms can be used as a server for Floating Licenses?

The system being used as the server (where the FLEXlm License Manager is running) for Floating licenses, must be Windows NT. The FLEXlm License Manager does not run properly with Windows 95/98.

Why do I need to include NWlink IPX/SPX on NT?

This is necessary for either obtaining the Ethernet card address, or to provide connectivity with a Netware License server.

6.4 TASKING QUESTIONS

How will the TASKING licensing/pricing model change with License Management (FLEXlm)?

TASKING will now offer the following types of licenses so you can purchase licenses based upon usage:

License	Description	Pricing
Node Locked	This license can only be used on a specific system. It cannot be moved to another system.	The pricing for this license will be the current product pricing.
Floating	This license requires a network (license server and a TCP/IP (or IPX/SPX) connection between clients and server) and can be used on any host system (using the same operating system) in the network.	The pricing for this license will be 50% higher than the node locked license.

How does FLEXlm affect future product ordering?

For all licenses, node locked or floating, you must provide information that is used to create a license key. For node locked licenses we must have the HOST ID. Floating licenses require the HOST ID and HOST NAME. The HOST ID is a unique identification of the machine, which is based upon different hardware depending upon host platform. The HOST NAME is the network name of the machine.



TASKING Logistics CANNOT ship ANY orders that do not include the HOST ID and/or HOST NAME information.

What if I do not know the information needed for the license key?

We have a software utility (**tkhostid.exe**) which will obtain and display the HOST ID so a customer can easily obtain this information. This utility is available from our web site, placed on all product CDs (which support FLEXlm), and from technical support. If you have already installed FLEXlm, you can also use **lmhostid**.

- In the case of a *Node locked license*, it is important that the customer runs this utility on the exact machine he intends to run the TASKING tools on.

- In the case of a *Floating License*, the **tkhostid.exe** (or **lmhostid**) utility should be run on the machine on which the FLEXlm license manager will be installed, e.g. the server. The HOST NAME information can be obtained from within the Windows Control Panel. Select "Network", click on "Identification", look for "Computer name".

How will the "locking" mechanism work?

- For node locked licenses, FLEXlm will first search for an ethernet card. If one exists, it will lock onto the number of the ethernet card. If an ethernet card does not exist, FLEXlm will lock onto the hard disk serial number.
- For floating licenses, the ethernet card number will be used.

What happens if I try to move my node locked license to another system?

The software will not run.

What does linger-time for floating licenses mean?

When the TASKING product starts to run, it will try to obtain a license from the license server. The license server keeps track of the number of licenses already issued, and grants or denies the request. When the software has finished running, the license is kept by the license server for a period of time known as the "linger-time". If the same user requests the TASKING product again within the linger-time, he is granted the license again. If another user requests a license during the linger-time, his request is denied until the linger-time has finished.

What is the length of the linger-time for floating licenses?

The length of the linger-time for both the PC and UNIX floating licenses is 5 minutes.

Can the linger-time be changed?

Yes. A customer can change the linger-time to be larger (but not shorter) than the time specified by TASKING.

What happens if my system crashes or I upgrade to a new system?

You will need to contact Technical Support for temporary license keys due to a system crash or to move from one system to another system. You will then need to work with your local sales representative to obtain a permanent new license key.

6.5 USING FLEXLM FOR FLOATING LICENSES

Does FLEXlm work across the internet?

Yes. A server on the internet will serve licenses to anyone else on the internet. This can be limited with the 'INTERNET=' attribute on the FEATURE line, which limits access to a range of internet addresses. You can also use the INCLUDE and EXCLUDE options in the daemon option file to allow (or deny) access to clients running on a range of internet addresses.

Does FLEXlm work with Internet firewalls?

Many firewalls require that port numbers be specified to the firewall. FLEXlm v5 **lmgrd** supports this.

If my client dies, does the server free the license?

Yes, unless the client's whole system crashes. Assuming communications is TCP, the license is automatically freed immediately. If communications are UDP, then the license is freed after the UDP timeout, which is set by each vendor, but defaults to 45 minutes. UDP communications is normally only set by the end-user, so TCP should be assumed. If the whole system crashes, then the license is not freed, and you should use '**lmremove**' to free the license.

What happens when the license server dies?

FLEXlm applications send periodic heartbeats to the server to discover if it has died. What happens when the server dies is then up to the application. Some will simply continue periodically attempting to re-checkout the license when the server comes back up. Some will attempt to re-checkout a license a few times, and then, presumably with some warning, exit. Some GUI applications will present pop-ups to the user periodically letting them know the server is down and needs to be re-started.

How do you tell if a port is already in use?

99.44% of the time, if it's in use, it's because **lmgrd** is already running on the port – or was recently killed, and the port isn't freed yet. Assuming this is not the case, then use '**telnet host port**' – if it says "*can't connect*", it's a free port.

Does FLEXlm require root permissions?

No. There is no part of FLEXlm, **lmgrd**, vendor daemon or application, that requires root permissions. In fact, it is strongly recommended that you do not run the license server (**lmgrd**) as root, since root processes can introduce security risks.

If **lmgrd** must be started from the root user (for example, in a system boot script), we recommend that you use the '**su**' command to run **lmgrd** as a non-privileged user:

```
su username -c"/path/lmgrd -c /path/license.dat \  
-l /path/log"
```

where *username* is a non-privileged user, and *path* is the correct paths to **lmgrd**, *license.dat* and debug log file. You will have to ensure that the vendor daemons listed in */path-to-license/license.dat* have execute permissions for *username*. The paths to all the vendor daemons in the license file are listed on each DAEMON line.

Is it ok to run lmgrd as 'root' (UNIX only)?

It is not prudent to run any command, particularly a daemon, as root on UNIX, as it may pose a security risk to the Operating System. Therefore, we recommend that **lmgrd** be run as a non-privileged user (not 'root'). If you are starting **lmgrd** from a boot script, we recommend that you use

```
su username -c"umask 022; /path/lmgrd \  
-c /path/license.dat -l /path/log"
```

to run **lmgrd** as a non-privileged user.

Does FLEXlm licensing impose a heavy load on the network?

No, but partly this depends on the application, and end-user's use. A typical checkout request requires 5 messages and responses between client and server, and each message is < 150 bytes.

When a server is not receiving requests, it requires virtually no CPU time. When an application, or **lmstat**, requests the list of current users, this can significantly increase the amount of networking FLEXlm uses, depending on the number of current users. Also, prior to FLEXlm v5, use of 'port@host' can increase network load, since the license file is down-loaded from the server to the client. 'port@host' should be, if possible, limited to small license files (say < 50 features). In v5, 'port@host' actually improves performance.

Does FLEXlm work with NFS?

Yes. FLEXlm has no direct interaction with NFS. FLEXlm uses an NFS-mounted file like any other application.

Does FLEXlm work with ATM, ISDN, Token-Ring, etc.?

In general, these have no impact on FLEXlm. FLEXlm requires TCP/IP or SPX (Novell Netware). So long as TCP/IP works, FLEXlm will work.

Does FLEXlm work with subnets, fully-qualified names, multiple domains, etc.?

Yes, although this behavior was improved in v3.0, and v6.0. When a license server and a client are located in different domains, fully-qualified host names have to be used. A fully-qualified hostname is of the form:

node.domain

where *node* is the local hostname (usually returned by the '**hostname**' command or '**uname -n**') *domain* is the internet domain name, e.g. 'globes.com'.

To ensure success with FLEXlm across domains, do the following:

1. Make the sure the fully-qualified hostname is the name on the SERVER line of the license file.
2. Make sure ALL client nodes, as well as the server node, are able to 'telnet' to that fully-qualified hostname. For example, if the host is locally called 'speedy', and the domain name is 'corp.com', local systems will be able to logon to speedy via 'telnet speedy'. But very often, 'telnet speedy.corp.com' will fail, locally.
Note that this telnet command will always succeed on hosts in other domains (assuming everything is configured correctly), since the network will resolve speedy.corp.com automatically.
3. Finally, there must be an 'alias' for speedy so it's also known locally as speedy.corp.com. This alias is added to the `/etc/hosts` file, or if NIS/Yellow Pages are being used, then it will have to be added to the NIS database. This requirement goes away in version 3.0 of FLEXlm.

If all components (application, **lmgrd** and vendor daemon) are v6.0 or higher, no aliases are required; the only requirement is that the fully-qualified domain name, or IP-address, is used as a hostname on the SERVER, or as a hostname in `LM_LICENSE_FILE port@host, or @host`.

Does FLEXlm work with NIS and DNS?

Yes. However, some sites have broken NIS or DNS, which will cause FLEXlm to fail. In v5 of FLEXlm, NIS and DNS can be avoided to solve this problem. In particular, sometimes DNS is configured for a server that's not current available (e.g., a dial-up connection from a PC). Again, if DNS is configured, but the server is not available, FLEXlm will fail.

In addition, some systems, particularly Sun, SGI, HP, require that applications be linked dynamically to support NIS or DNS. If a vendor links statically, this can cause the application to fail at a site that uses NIS or DNS. In these situations, the vendor will have to relink, or recompile with v5 FLEXlm. Vendors are strongly encouraged to use dynamic libraries for libc and networking libraries, since this tends to improve quality in general, as well as making NIS/DNS work.

On PCs, if a checkout seems to take 3 minutes and then fails, this is usually because the system is configured for a dial-up DNS server which is not currently available. The solution here is to turn off DNS.

Finally, hostnames must NOT have periods in the name. These are not legal hostnames, although PCs will allow you to enter them, and they will not work with DNS.

We're using FLEXlm over a wide-area network. What can we do to improve performance?

FLEXlm network traffic should be minimized. With the most common uses of FLEXlm, traffic is negligible. In particular, checkout, checkin and heartbeats use very little networking traffic. There are two items, however, which can send considerably more data and should be avoided or used sparingly:

- **'lmstat -a'** should be used sparingly. **'lmstat -a'** should not be used more than, say, once every 15 minutes, and should be particularly avoided when there's a lot of features, or concurrent users, and therefore a lot of data to transmit; say, more than 20 concurrent users or features.
- Prior to FLEXlm v5, the 'port@host' mode of the LM_LICENSE_FILE environment variable should be avoided, especially when the license file has many features, or there are a lot of license files included in LM_LICENSE_FILE. The license file information is sent via the network, and can place a heavy load. Failures due to 'port@host' will generate the error LM_SERVNOREADLIC (-61).

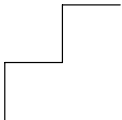
APPENDIX

B

SAFER C



TASKING



B

APPENDIX

Supported and unsupported Safer C rules

1. no language extensions shall be used
- * 2. other languages should only be used with an interface standard
3. inline assembly is only allowed in dedicated C functions
- * 4. provision should be made for appropriate run-time checking
5. only use characters defined by the C standard
- * 6. character values shall be restricted to a subset of ISO 106460-1
7. trigraphs shall not be used
8. multibyte characters and wide string literals shall not be used
9. comments shall not be nested
- * 10. sections of code should not be "commented out"
11. identifiers shall not rely on significance of more than 31 characters
12. the same identifier shall not be used in multiple name spaces
13. specific-length typedefs should be used instead of the basic types
14. use 'unsigned char' or 'signed char' instead of plain 'char'
- * 15. floating point implementations should comply with a standard
- * 16. the bit representation of floating point numbers shall not be used
17. typedef names should not be reused
- * 18. numeric constants should be suffixed to indicate type
19. octal constants (other than zero) shall not be used
20. all object and function identifiers shall be declared before use
21. identifiers shall not hide identifiers in an outer scope
22. declarations should be at function scope where possible ("%s")
- * 23. all declarations at file scope should be static where possible
24. identifiers shall not have both internal and external linkage

- * 25. identifiers with external linkage shall have exactly one definition
- 26. multiple declarations for objects or functions shall be compatible
- * 27. external objects should not be declared in more than one file
- 28. the 'register' storage class specifier should not be used
- 29. the use of a tag shall agree with its declaration
- 30. all automatics shall be initialized before being used
- 31. braces shall be used in the initialization of arrays and structures
- 32. only the first, or all enumeration constants may be initialized
- 33. the right hand side of && or || shall not contain side effects
- 34. the operands of a logical && or || shall be primary expressions
- 35. assignment operators shall not be used in Boolean expressions
- * 36. logical operators should not be confused with bitwise operators
- 37. bitwise operations shall not be performed on signed integers
- 38. a shift count shall be between 0 and the operand width minus 1
- 39. the unary minus shall not be applied to an unsigned expression
- 40. 'sizeof' should not be used on expressions with side effects
- * 41. the implementation of integer division should be documented
- 42. the comma operator shall only be used in a 'for' condition
- 43. don't use implicit conversions which may result in information loss
- 44. redundant explicit casts should not be used
- 45. type casting from any type to/from pointers shall not be used
- 46. the value of an expression shall be evaluation order independent
- * 47. no dependence should be placed on operator precedence rules
- * 48. mixed arithmetic should use explicit casting
- * 49. tests of a (non-Boolean) value against 0 should be made explicit

- 50. F.P. variables shall not be tested for exact equality or inequality
- * 51. constant unsigned integer expressions should not wrap-around
- 52. there shall be no unreachable code
- 53. all non-null statements shall have a side-effect
- 54. a null statement shall only occur on a line by itself
- 55. labels should not be used
- 56. the 'goto' statement shall not be used
- 57. the 'continue' statement shall not be used
- 58. the 'break' statement shall not be used (except in a 'switch')
- 59. an 'if' or loop body shall always be enclosed in braces
- 60. all 'if', 'else if' constructs should contain a final 'else'
- 61. every non-empty 'case' clause shall be terminated with a 'break'
- 62. all 'switch' statements should contain a final 'default' case
- 63. a 'switch' expression should not represent a Boolean case
- 64. every 'switch' shall have at least one 'case'
- 65. floating point variables shall not be used as loop counters
- * 66. a "for" should only contain expressions concerning loop control
- * 67. iterator variables should not be modified in a "for" loop
- 68. functions shall always be declared at file scope
- 69. functions with variable number of arguments shall not be used
- 70. functions shall not call themselves
- 71. function prototypes shall be visible at the definition and call
- 72. the function prototype of the declaration shall match the definition
- 73. identifiers shall be given for all prototype parameters or for none
- 74. parameter identifiers shall be identical for declaration/definition

- 75. every function shall have an explicit return type
- 76. functions with no parameters shall have a 'void' parameter list
- * 77. an actual parameter type shall be compatible with the prototype
- 78. the number of actual parameters shall match the prototype
- 79. the values returned by 'void' functions shall not be used
- 80. void expressions shall not be passed as function parameters
- * 81. "const" should be used for reference parameters not modified
- 82. a function should have a single point of exit
- 83. every exit point shall have a 'return' of the declared return type
- 84. for 'void' functions, 'return' shall not have an expression
- 85. function calls with no parameters should have empty parentheses
- * 86. if a function returns error information, it should be tested
- 87. #include shall only be preceded by another directives or comments
- 88. non-standard characters shall not occur in #include directives
- 89. #include shall be followed by either <filename> or "filename"
- 90. plain macros shall only be used for constants/qualifiers/specifiers
- 91. macros shall not be defined/undefined within a block
- 92. #undef should not be used
- * 93. a function should be used in preference to a function-like macro
- 94. a function-like macro shall not be used without all arguments
- * 95. macro arguments shall not contain pre-preprocessing directives
- 96. macro definitions/parameters should be enclosed in parentheses
- 97. don't use undefined identifiers in pre-processing directives
- 98. a macro definition shall contain at most one # or ## operator

- * 99. all uses of the `#pragma` directive shall be documented
- 100. `'defined'` shall only be used in one of the two standard forms
- 101. pointer arithmetic should not be used
- 102. no more than 2 levels of pointer indirection should be used
- * 103. no relational operators between pointers to different objects
- 104. non-constant pointers to functions shall not be used
- 105. functions assigned to the same pointer shall be of identical type
- 106. an automatic address may not be assigned to a longer lived object
- * 107. the null pointer shall not be de-referenced
- * 108. all struct/union members shall be fully specified
- * 109. overlapping variable storage shall not be used
- * 110. unions shall not be used to access the sub-parts of larger types
- 111. bit fields shall have type `'unsigned int'` or `'signed int'`
- 112. bit fields of type `'signed int'` shall be at least 2 bits long
- 113. all struct/union members shall be named
- 114. reserved and standard library names shall not be redefined
- 115. standard library function names shall not be reused
- * 116. production libraries shall comply with the Safer C restrictions
- * 117. the validity of library function parameters shall be checked
- 118. dynamic heap memory allocation shall not be used
- 119. `'errno'` should not be used
- 120. the macro `'offsetof()'` shall not be used
- 121. `<locale.h>` and the `'setlocale'` function shall not be used
- 122. the `'setjmp'` and `'longjmp'` functions shall not be used
- 123. the signal handling facilities of `<signal.h>` shall not be used

- 124. the <stdio.h> library shall not be used in production code
- 125. the functions atof/atoi/atol shall not be used
- 126. the functions abort/exit/getenv/system shall not be used
- 127. the time handling functions of library <time.h> shall not be used



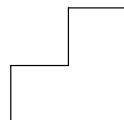
* = Not supported by the TASKING M16C C compiler

INDEX

INDEX



TASKING



INDEX

Symbols

#define, 4-23
 #include, 4-31, 4-73
 #pragma, 4-75
 alias, 4-75
 asm, 3-24, 4-75
 asm_noflush, 3-24, 4-76
 endasm, 3-24, 4-76
 endoptimize, 4-76
 listinc, 4-76
 noalias, 4-75
 nolistinc, 4-76
 nosource, 4-77
 optimize, 4-76
 renamesec, 4-77
 source, 4-77
 #pragma optimize, 4-35
 #undef, 4-67
 -M option, 3-8
 -O option, 3-36
 -v option, 3-31
 __DATE__, 4-67
 __FILE__, 4-67
 __LINE__, 4-67
 __STDC__, 4-67
 __TIME__, 4-67
 __asmfunc, 3-25
 __at attribute, 3-9
 __bdat, 3-5
 __bit, 3-11, 3-16
 __close, 6-12
 __CM16C, 3-35, 4-67
 __CODEMODEL, 4-67
 __far, storage type, 3-5
 __farrom, 3-5
 __inline, 3-23
 __interrupt, 3-31
 __lseek, 6-12
 __MODEL, 4-67
 __near, storage type, 3-5
 __nearrom, 3-5
 __open, 6-12

 __read, 6-12
 __rom, 3-5
 __sfrbit, 3-11, 3-17
 __sfrbyte, 3-11, 3-17
 __sfrlong, 3-11, 3-17
 __sfrword, 3-11, 3-17
 __simi, 6-13
 __simo, 6-13
 __tolower, 6-13
 __toupper, 6-13
 __unlink, 6-14
 __write, 6-14

A

abort, 6-14
 abs, 6-14
 acos, 6-14
 adding files to a project, 2-22
 address range, 3-4
 address spaces, 3-5
 alias, 4-38, 4-75, 4-78
 allocation graph, 2-10
 ansi standard, 2-3, 3-3, 4-67
 asctime, 6-15
 asin, 6-15
 asm, 4-75
 asm_noflush, 4-76
 asm16, 2-11
 assembly functions, 3-25
 assembly language interfacing, 7-12
 assembly routine, 7-12
 assembly source file, 2-11
 assert, 6-15
 assert.h, 6-3
 atan, 6-15
 atan2, 6-16
 atexit, 6-16
 atof, 6-16
 atoi, 6-16
 atol, 6-17
 automatic variables, 3-21

B

backend

compiler phase, 2-6

optimization, 2-6, 2-10

bit, 3-16, 7-4

bit addressable, 7-4

branch tail merging, 2-9

bsearch, 6-17

built-in functions, 3-27

C

C

inline functions, 3-23

language extensions, 3-3

C library, 6-4

implementation details, 6-6

interface description, 6-12

C startup code, 7-3

calloc, 6-17

CCM16CBIN, 4-9

CCM16COPT, 4-9

ceil, 6-18

character arithmetic, 3-15, 4-16

clearerr, 6-18

clock, 6-18

CM16CINC, 4-31, 4-73

code density, 4-50

code generator, 2-7, 3-18, 3-21

command file, 4-6, 4-27

command line processing, 4-27

comments, C++ style, 4-17

common subexpression elimination,
2-9

compiler, invocation, 4-10

compiler diagnostics, 5-1

compiler limits, 4-80

compiler options

-?, 4-15

-A, 4-16

-align_data, 4-20

-align_func, 4-21

-C, 4-22

-D, 4-23

-E, 4-24

-e, 4-25

-El, 4-24

-Em, 4-24

-err, 4-26

-F, 6-6

-f, 4-27

-Fc, 6-6

-g, 4-29

-gf, 4-29

-gl, 4-29

-gn, 4-29

-H, 4-30

-I, 4-31

-J, 4-32

-M, 4-33

-n, 4-34

-O, 4-35, 4-37

-o, 4-59

-Oa / -OA, 4-38

-Oc / -OC, 4-39

-Od / -OD, 4-41

-Oe / -OE, 4-42

-Of / -OF, 4-44

-Oi / -OI, 4-46

-Ol / -OL, 4-48

-Os / -OS, 4-50

-Ot / -OT, 4-51

-Ou / -OU, 4-53

-Ov / -OV, 4-54

-Ow / -OW, 4-56

-Oy / -OY, 4-57

-R, 4-60

-S, 4-62

-s, 4-63

-safer, 4-64

-T, 4-65

-t, 4-66

- U, 4-67
- u, 4-69
- V, 4-70
- w, 4-71
- Z, 4-72
- detailed description*, 4-14
- overview*, 4-10
- overview in functional order*, 4-11
- priority*, 4-10
- compiler phases, 2-6
 - backend*, 2-6
 - code generator phase*, 2-7
 - optimization phase*, 2-6
 - peephole optimizer phase*, 2-7
 - frontend*, 2-6
 - optimization phase*, 2-6
 - parser phase*, 2-6
 - preprocessor phase*, 2-6
 - scanner phase*, 2-6
- compiler structure, 2-11
- compiler use, 4-1
- compound assignment, 4-42
- conditional jump reversal, 2-8, 4-44
- const, 3-6
- constant folding, 2-7
- constant propagation, 2-9, 4-41
- control flow optimization, 2-8, 4-44
- control program, 4-3
 - options overview*, 4-4
- control program options
 - , 4-5
 - C, 4-5
 - c, 4-6
 - c++, 4-5
 - cc, 4-6
 - cl, 4-6
 - cs, 4-6
 - f, 4-6
 - g, 4-7
 - ieee, 4-8
 - ibex, 4-8
 - ML, 4-5
 - Ms, 4-5
 - nolib, 4-8
 - o, 4-8
 - srec, 4-8
 - tiof, 4-8
 - tmp, 4-8
 - V, 4-5
 - v, 4-8
 - v0, 4-8
 - Wa, 4-5
 - Wc, 4-5
 - wc++, 4-9
 - Wcp, 4-5
 - Wlc, 4-5
 - Wlk, 4-5
 - Wpl, 4-5
 - detailed description*, 4-5
- conversions, ANSI C, 3-12
- copy propagation, 2-9, 4-41
- copysign, 6-18
- copysignf, 6-19
- cos, 6-19
- cosh, 6-19
- creating a makefile, 2-23
- cross jumping, 2-9
- cross-assembler, 2-11
- CSE, 2-9, 4-39
- ctime, 6-19
- ctype.h, 6-3
 - _tolower*, 6-13
 - _toupper*, 6-13
- isalnum*, 6-29
- isalpha*, 6-29
- isascii*, 6-29
- iscntrl*, 6-29
- isdigit*, 6-29
- isgraph*, 6-30
- islower*, 6-31
- isprint*, 6-32
- ispunct*, 6-32
- isspace*, 6-32
- isupper*, 6-32
- isxdigit*, 6-32
- toascii*, 6-58

tolower, 6-59

toupper, 6-59

D

data types, 3-11-3-17

_bit, 3-11

_far pointer, 3-11

_near pointer, 3-11

_sfrbyte, 3-11

_sfrlong, 3-11

_sfrword, 3-11

double, 3-11

enum, 3-11

float, 3-11

signed char, 3-11

signed int, 3-11

signed long, 3-11

signed short, 3-11

unsigned char, 3-11

unsigned int, 3-11

unsigned long, 3-11

unsigned short, 3-11

dead code elimination, 2-9

dead store elimination, 2-10

debug information, 4-29

debugger, starting, 2-21

derivatives, 2-5, 4-22

detailed option description

compiler, 4-14-4-72

control program, 4-5-4-9

development flow, 2-12

difftime, 6-20

directory separator, 4-74

div, 6-20

double, 3-11

E

EDE, 2-15

build an application, 2-17

load files, 2-17

open a project, 2-16

select a toolchain, 2-16

start a new project, 2-22

starting, 2-15

embedded development environment.

See EDE

enabling safer c, 3-32

endasm, 4-76

endoptimize, 4-76

enum, 3-11

environment variable

CCM16CBIN, 4-9

CCM16COPT, 4-9

CM16CINC, 4-31, 4-73

LM_LICENSE_FILE, 1-16, A-6

overview of, 2-14

PATH, 1-6, 1-9

TMPDIR, 1-6, 1-9, 4-9

used by control program, 4-9

used by tool chain, 2-14

errno.h, 6-3

error level, 5-4

Error Messages, 3-33

errors, 5-5

backend, 5-32

FLEXlm license, A-33

frontend, 5-5

example

starting EDE, 2-15

using EDE, 2-15

using the control program, 2-23

using the makefile, 2-25

execution speed, 4–50
 execution time, 3–19
 exit, 6–20
 exit status, 5–4
 exp, 6–20
 expression propagation, 4–42
 expression rearrangement, 2–7
 expression simplification, 2–8
 extensions to C, 3–3

F

fabs, 6–20
 FAQ, FLEXlm, A–37
 far code, 7–4
 far data, 7–4
 fast loops, 4–48
 fclose, 6–21
 fcntl.h, 6–3
 feof, 6–21
 ferror, 6–21
 fflush, 6–21
 fgetc, 6–22
 fgetpos, 6–22
 fgets, 6–22
 Flexible License Manager, A–1
 FLEXlm, A–1
 daemon log file, A–25
 daemon options file, A–7
 FAQ, A–37
 frequently asked questions, A–37
 license administration tools, A–8
 for Windows, A–22
 license errors, A–33
 float, 3–11
 float.h, 6–3
 copysign, 6–18
 copysignf, 6–19
 isfinite, 6–30
 isfinitef, 6–30
 isinf, 6–30
 isinf, 6–31
 isnan, 6–31
 isnanf, 6–31
 scalb, 6–44
 scalbf, 6–44
 floating license, 1–9
 floating point, 7–9
 single precision, 6–6
 special values, 7–9
 floor, 6–22
 fmod, 6–23
 fopen, 6–23
 formatters
 printf, 6–62
 scanf, 6–62
 fprintf, 6–24
 fputc, 6–24
 fputs, 6–24
 fread, 6–25
 free, 6–25
 freopen, 6–25
 frexp, 6–26
 frontend
 compiler phase, 2–6
 optimization, 2–6, 2–7
 fscanf, 6–26
 fseek, 6–26
 fsetpos, 6–27
 ftell, 6–27
 function parameters, 3–18
 function qualifier, *_asmfunc*, 3–25
 function return, 3–19
 function return types, 7–3
 functions
 built-in, 3–27
 intrinsic, 3–27
 fwrite, 6–27

G

getc, 6-27
 getchar, 6-28
 getenv, 6-28
 gets, 6-28
 gmtime, 6-28

H

header files, 6-3
 heap, 7-8
 beginning of, 7-8
 end of, 7-8
 heap size, 7-8
 hostid, determining, 1-17
 hostname, determining, 1-18

I

identifier, 4-17
 IEEE 32-bit single precision format,
 3-12
 IEEE 64-bit double precision format,
 3-12
 IEEE-695, 2-13
 IEEE-754, 7-9
 include files, 4-73
 default directory, 4-74
 initialized variables, 3-21
 inline assembly, 3-24
 installation
 licensing, 1-9
 Linux, 1-4
 RPM, 1-4
 tar.gz, 1-5
 UNIX, 1-7
 Windows, 1-3
 Windows 95/98, 1-3
 Windows NT, 1-3

integral promotion, 3-12
 Intel hex format, 2-13
 function, inline C, 3-23
 interrupt frame optimization, 2-10
 interrupt functions, 7-10
 intrinsic functions, 3-27
 introduction, 2-3
 invariant code, 4-46
 invocation
 compiler, 4-10
 control program, 4-3
 isalnum, 6-29
 isalpha, 6-29
 isascii, 6-29
 iscntrl, 6-29
 isdigit, 6-29
 isfinite, 6-30
 isfinitf, 6-30
 isgraph, 6-30
 isinf, 6-30
 isinff, 6-31
 islower, 6-31
 isnan, 6-31
 isnanf, 6-31
 isprint, 6-32
 ispunct, 6-32
 isspace, 6-32
 isupper, 6-32
 isxdigit, 6-32

J

jump chain, 3-34, 4-56
 jump chaining, 2-8, 4-44
 jump table, 2-10, 3-34, 4-51, 4-56

K

keyword, `_inline`, 3-23

L

- labs, 6-33
- language extensions, 4-16
- language implementation, 3-1
- lcm16, 2-11
- ldexp, 6-33
- ldiv, 6-33
- leaf function handling, 2-10
- libraries, 6-1
 - C*, 6-4
 - C (single precision floating point)*, 6-6
 - floating point*, 6-5
 - run-time*, 6-63
- license
 - floating*, 1-9
 - node-locked*, 1-9
 - obtaining*, 1-10
- license file
 - default location*, A-6
 - location*, 1-16
- licensing, 1-9
- limits, compiler, 4-80
- limits.h, 6-3
- linker, 2-11
- listinc, 4-76
- lkm16, 2-11
- LM_LICENSE_FILE, 1-16, A-6
- lmcksum, A-10
- lmdia, A-11
- lmdown, A-12
- lmgrd, A-13
- lmhostid, A-15
- lmremove, A-16
- lmreread, A-17
- lmstat, A-18
- lmswitchr, A-20
- lmver, A-21
- locale.h, 6-3
 - localeconv*, 6-33
 - setlocale*, 6-47
- localeconv, 6-33

- localtime, 6-34
- locator, 2-11
- log, 6-34
- log10, 6-34
- logical expression optimization, 2-8
- longjmp, 6-34
- loop optimization, 2-9
- loop rotation, 2-8, 4-48
- loop unrolling, 2-9, 4-53
- loop variable detection, 4-39

M

- makefile
 - automatic creation of*, 2-23
 - updating*, 2-23
- malloc, 6-35
- malloc.h, 6-3
- math.h, 6-3
 - acos*, 6-14
 - asin*, 6-15
 - atan*, 6-15
 - atan2*, 6-16
 - ceil*, 6-18
 - cos*, 6-19
 - cosh*, 6-19
 - exp*, 6-20
 - fabs*, 6-20
 - floor*, 6-22
 - fmod*, 6-23
 - frexp*, 6-26
 - ldexp*, 6-33
 - log*, 6-34
 - log10*, 6-34
 - modf*, 6-38
 - pow*, 6-38
 - sin*, 6-49
 - sinh*, 6-49
 - sqrt*, 6-49
 - tan*, 6-57
 - tanh*, 6-57
- mblen, 6-35

mbstowcs, 6-35
 mbtowc, 6-36
 memchr, 6-36
 memcmp, 6-36
 memcpy, 6-37
 memmove, 6-37
 memory access, 3-4
 memory model, 3-8
 large, 3-8
 small, 3-8
 memory type, 3-23
 memset, 6-37
 mktime, 6-37
 modf, 6-38
 Motorola S-record, 2-13
 multi-line macros, 4-24

N

names, 7-12
 near code, 7-4
 near data, 7-4
 noalias, 4-75
 node-locked license, 1-9
 nolistinc, 4-76
 non-volatile ram, 3-6
 nosource, 4-77

O

offsetof, 6-38
 optimization, 4-35, 4-37
 backend, 2-6, 2-10
 frontend, 2-6, 2-7
 optimization (backend)
 allocation graph, 2-10
 dead store elimination, 2-10
 interrupt frame, 2-10
 jump table, 2-10
 leaf function handling, 2-10
 peephole optimizations, 2-10

tail recursion elimination, 2-10
 optimization (frontend)
 common subexpression elimination,
 2-9
 conditional jump reversal, 2-8
 constant folding, 2-7
 constant/copy propagation, 2-9
 control flow optimization, 2-8
 cross jumping and branch tail
 merging, 2-9
 dead code elimination, 2-9
 expression rearrangement, 2-7
 expression simplification, 2-8
 jump chaining, 2-8
 logical expression optimization, 2-8
 loop optimization, 2-9
 loop rotation, 2-8
 loop unrolling, 2-9
 remove useless jumps, 2-8
 sharing of string literals and floating
 point constants, 2-9
 switch optimization, 2-8
 optimize, 4-76
 options, control program, 4-4
 output file, 4-59
 overview, 2-1

P

parameter passing, 3-19
 parameters, 3-18
 parser, 2-6
 PATH, 1-6, 1-9
 peephole optimization, 2-10, 4-57
 peephole optimizer, 2-7
 perror, 6-38
 pointer
 _far, 3-11
 _near, 3-11
 pointers, 3-22
 portable C code, 3-35
 pow, 6-38

pragma
 alias, 4-75
 asm, 4-75
 asm_noflush, 4-76
 endasm, 4-76
 endoptimize, 4-76
 listinc, 4-76
 noalias, 4-75
 nolistinc, 4-76
 nosource, 4-77
 optimize, 4-76
 renamesec, 4-77
 source, 4-77
 pragma optimize
 flow level, 4-35
 function level, 4-35
 pragmas, 4-75
 predefined symbols, 4-67
 _CM16C, 4-67
 _CODEMODEL, 4-67
 _MODEL, 4-67
 printf, 6-39
 printf formatter, 6-62
 product definition, 2-5
 project files, adding files, 2-22
 prototyping, 3-18
 putc, 6-41
 putchar, 6-41
 puts, 6-41

Q

qsort, 6-42

R

raise, 6-42
 RAM, 3-4, 3-21
 rand, 6-42
 realloc, 6-43
 reentrant, 3-16

register usage, 3-19, 7-3
 remove, 6-43
 remove useless jumps, 2-8
 rename, 6-43
 renamesec, 4-77
 return address, 7-7
 return values, 5-4
 rewind, 6-44
 ROM, 3-22
 rom, 3-5
 ROM memory, 3-6
 run-time library, 6-63

S

Safer C, B-1
 safer c, 3-32
 sample session, 2-15
 scalb, 6-44
 scalbf, 6-44
 scanf, 6-44
 scanf formatter, 6-62
 scanner, 2-6
 section
 attribute, 3-7
 type, 3-7
 section name, 4-60
 section usage, 7-4
 setbuf, 6-46
 setjmp, 6-47
 setjmp.h, 6-3
 longjmp, 6-34
 setjmp, 6-47
 setlocale, 6-47
 setting the environment, 1-6, 1-9
 setvbuf, 6-48
 sfr, 3-17
 sharing of string literals and floating
 point constants, 2-9
 SIGABRT, 6-48
 SIGFPE, 6-48
 SIGILL, 6-48

SIGINT, 6-48
 signal, 6-49
 signal.h, 6-3
 raise, 6-42
 signal, 6-48
 signals, 6-48
 signed
 char, 3-11
 int, 3-11
 long, 3-11
 short, 3-11
 SIGSEGV, 6-48
 SIGTERM, 6-48
 simio.h, 6-3
 _simi, 6-13
 _simo, 6-13
 sin, 6-49
 sinh, 6-49
 smart programming, 3-35
 source, 4-77
 special function registers, 3-17, 4-22
 sprintf, 6-49
 sqrt, 6-49
 srand, 6-50
 sscanf, 6-50
 stack, 3-5, 7-6, 7-7
 beginning of, 7-6
 end of, 7-6
 organization of, 7-6
 stack size, 7-6
 start.obj, 7-3
 startup code, 7-3
 stdarg.h, 6-3
 va_arg, 6-59
 va_end, 6-60
 va_start, 6-60
 stddef.h, 6-4
 offsetof, 6-38
 stdio.h, 6-4
 _close, 6-12
 _lseek, 6-12
 _open, 6-12
 _read, 6-12
 _unlink, 6-14
 _write, 6-14
 clearerr, 6-18
 fclose, 6-21
 feof, 6-21
 ferror, 6-21
 fflush, 6-21
 fgetc, 6-22
 fgetpos, 6-22
 fgets, 6-22
 fopen, 6-23
 fprintf, 6-24
 fputc, 6-24
 fputs, 6-24
 fread, 6-25
 freopen, 6-25
 fscanf, 6-26
 fseek, 6-26
 fsetpos, 6-27
 ftell, 6-27
 fwrite, 6-27
 getc, 6-27
 getchar, 6-28
 gets, 6-28
 perror, 6-38
 printf, 6-39
 putc, 6-41
 putchar, 6-41
 puts, 6-41
 remove, 6-43
 rename, 6-43
 rewind, 6-44
 scanf, 6-44
 setbuf, 6-46
 setvbuf, 6-48
 sprintf, 6-49
 sscanf, 6-50
 tmpfile, 6-58
 tmpnam, 6-58
 ungetc, 6-59
 vsprintf, 6-60
 vprintf, 6-60
 vsprintf, 6-61

- stdlib.h, 6-4
 - abort*, 6-14
 - abs*, 6-14
 - atexit*, 6-16
 - atof*, 6-16
 - atoi*, 6-16
 - atol*, 6-17
 - bsearch*, 6-17
 - calloc*, 6-17
 - div*, 6-20
 - exit*, 6-20
 - free*, 6-25
 - getenv*, 6-28
 - labs*, 6-33
 - ldiv*, 6-33
 - malloc*, 6-35
 - mblen*, 6-35
 - mbstowcs*, 6-35
 - mbtowc*, 6-36
 - qsort*, 6-42
 - rand*, 6-42
 - realloc*, 6-43
 - srand*, 6-50
 - strtod*, 6-55
 - strtol*, 6-56
 - strtoul*, 6-56
 - system*, 6-57
 - wcstombs*, 6-61
 - wctomb*, 6-61
- storage type, 3-5
 - _bdat*, 3-5
 - _far*, 3-5
 - _farrom*, 3-5
 - _near*, 3-5
 - _nearrom*, 3-5
 - _rom*, 3-5, 3-21
 - relation to section*, 3-7
- strcat*, 6-50
- strchr*, 6-50
- strcmp*, 6-51
- strcoll*, 6-51
- strcpy*, 6-51
- strcspn*, 6-51
- strerror*, 6-52
- strftime*, 6-52
- string*, 3-22
- string.h*, 6-4
 - memchr*, 6-36
 - memcmp*, 6-36
 - memcpy*, 6-37
 - memmove*, 6-37
 - memset*, 6-37
 - strcat*, 6-50
 - strchr*, 6-50
 - strcmp*, 6-51
 - strcoll*, 6-51
 - strcpy*, 6-51
 - strcspn*, 6-51
 - strerror*, 6-52
 - strlen*, 6-53
 - strncat*, 6-53
 - strncmp*, 6-53
 - strncpy*, 6-54
 - strpbrk*, 6-54
 - strrchr*, 6-54
 - strspn*, 6-54
 - strstr*, 6-55
 - strtok*, 6-55
 - strxfrm*, 6-56
- strlen*, 6-53
- strncat*, 6-53
- strncmp*, 6-53
- strncpy*, 6-54
- strpbrk*, 6-54
- strchr*, 6-54
- strspn*, 6-54
- strstr*, 6-55
- strtod*, 6-55
- strtok*, 6-55
- strtol*, 6-56
- strtoul*, 6-56
- structure tag, 3-33
- strxfrm*, 6-56
- subscript strength reduction, 4-54
- switch optimization, 2-8, 4-51, 4-56
- switch statement, 3-34

symbols, predefined, 4-67
system, 6-57

T

tail recursion elimination, 2-10
tan, 6-57
tanh, 6-57
target memory, 3-23
target processors, 2-5
temporary files, 4-9
time, 6-57
time.h, 6-4
 asctime, 6-15
 clock, 6-18
 ctime, 6-19
 difftime, 6-20
 gmtime, 6-28
 localtime, 6-34
 mktime, 6-37
 strftime, 6-52
 time, 6-57
TMPDIR, 1-6, 1-9, 4-9
tmpfile, 6-58
tmpnam, 6-58
toascii, 6-58
tolower, 6-59
toupper, 6-59
transferring parameters between
 functions, 3-19
type qualifier
 const, 3-6, 3-21
 volatile, 3-21
typedef, 3-33

U

ungetc, 6-59
unresolved external, 7-12
unsigned
 char, 3-11
 int, 3-11
 long, 3-11
 short, 3-11
updating makefile, 2-23

V

va_arg, 6-59
va_end, 6-60
va_start, 6-60
variable, automatic, 3-21
variables, initialized, 3-21
version information, 4-70
vfprintf, 6-60
volatile, 3-21
vprintf, 6-60
vsprintf, 6-61

W

warnings, 5-5
warnings (suppress), 4-71
wcstombs, 6-61
wctomb, 6-61