MA039–002–00–00
Doc. ver.: 1.24

DSP56xxx v3.6

C CROSS–COMPILER USER'S GUIDE

TASKING

A publication of

Altium BV

Documentation Department

http://www.tasking.com
http://www.altium.com

CONTENTS

TABLE OF CONTENTS

TASKING

CONTENTS

SOFTWARE INSTALLATION 1-1

OVERVIEW 2-1

FLEXIBLE LICENSE MANAGER (FLEXlm) A-1

CONTENTS

MANUAL PURPOSE AND STRUCTURE

PURPOSE

This manual is aimed at users of the TASKING DSP5600x, DSP563xx and DSP566xx Family 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 DSP56xxx family of processors.

2. Overview
   Provides an overview of the TASKING DSP56xxx Family toolchain and gives you some familiarity with the different parts of it and their relationship. A sample session explains how to build a DSP56xxx application from your C file.

3. Language Implementation
   Concentrates on the approach of the DSP56xxx 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 DSP56xxx C application. It deals with items like assembly language interfacing, C startup code and stack/heap size.

8. Support for User–designed Target Boards
   Contains the steps you have to take to support user–designed target boards.

APPENDICES

A. Flexible License Manager (FLEXlm)
   Contains a description of the Flexible License Manager.

B. Motorola Compatibility
   Describes the interoperability between the TASKING and Motorola tool sets. It describes how to create a Motorola CLAS COFF object file and how to link CLAS/COFF object files and libraries.

INDEX

RELATED PUBLICATIONS

C Standards

- The C Programming Language (second edition) by B. Kernighan and D. Ritchie (1988, Prentice Hall)
- ANSI X3.159–1989 standard [ANSI]

TASKING Tools

- DSP56xxx Cross–Assembler, Linker/Locator, Utilities User's Guide [TASKING, MA039–000–00–00]
- DSP56xxx CrossView Pro Debugger User's Guide [TASKING, MA039–049–00–00]

Core Reference Manuals

- DSP56000 Digital Signal Processor Family Manual [Motorola, Inc.]
- DSP560*xx* Digital Signal Processor User's Manual [Motorola, Inc.]

- DSP56300 24–Bit Digital Signal Processor Family Manual [Motorola, Inc.]
- DSP563*xx* 24–Bit Digital Signal Processor User's Manual [Motorola, Inc.]
- DSP56L307 24–Bit Digital Signal Processor User's Manual [Motorola, Inc.]

- DSP56600 Digital Signal Processor Family Manual [Motorola, Inc.]
- DSP5660*x* Digital Signal Processor User's Manual [Motorola, Inc.]

- DSP56652 Baseband Digital Signal Processor User's Manual [Motorola, Inc.]
- DSP56654 Baseband Digital Signal Processor User's Manual [Motorola, Inc.]

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 DSP56xxx Family (DSP563xx/DSP566xx, DSP5600x) on Windows 95/98/XP/NT/2000 and several UNIX hosts.

1.2 INSTALLATION FOR WINDOWS

1. Start Windows 95/98/XP/NT/2000, if you have not already done so.
2. Insert the CD–ROM into the CD–ROM drive.

   If the TASKING Showroom dialog box appears, proceed with Step 5.
3. Click the **Start** button and select **Run...**
4. In the dialog box type **`d:\setup`** (substitute the correct drive letter for your CD–ROM drive) and click on the **OK** button.

   The TASKING Showroom dialog box appears.
5. Select a product and click on the **Install** button.
6. Follow the instructions that appear on your screen.

You can find your serial number on the *Start–up kit envelope*, delivered with the product.

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

1.2.1 SETTING THE ENVIRONMENT

After you have installed the software, you can set some environment variables to make invocation of the tools easier. When you are using EDE all settings are configurable from within EDE. 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, when you invoke the tools from a command prompt. If you installed the software under `c:\c56`, you can include the executable directory `c:\c56\bin` in your search path.

For the DSP563xx/DSP566xx family the default installation path is `\c563`.

In EDE, select the `Project | Directories...` menu item. Add one or more executable directory paths to the `Executable Files Path` field.

The environment variable TMPDIR can be used to specify a directory where programs can place temporary files. The DSP5600x compiler uses the environment variable C56INC to search for include files. Use C563INC for the DSP563xx/DSP566xx family. An example of setting this variable is given below.

See also the section *Include Files* in the chapter *Compiler Use*.

Example Windows 95/98

Add the following line to your `autoexec.bat` file.

```
set C563INC=c:\c563\include
```

You can also type this line in a Command Prompt window but you will loose this setting after you close the window.

Example Windows NT

1. Rright–click on the `My Computer` icon on your desktop and select `Properties`.

   The System Properties dialog appears.

2. Select the `Environment` tab.

3. In the `Variable` edit field enter:

   `C563INC`

4. In the `Value` edit field enter:

   `c:\c563\include`

5. Click on the `Set` button, then click `OK`.

Example Windows XP/2000

1. Rright–click on the `My Computer` icon on your desktop and select `Properties`.

   The System Properties dialog appears.

2. Select the `Advanced` tab.
3. Click on the `Environment Variables` button.

   The Environment Variables dialog appears.
4. In the `System variables` field, click on the `New` button.

   The New System Variable dialog appears.
5. In the `Variable name` field enter:

   `C563INC`
6. In the `Variable value` field enter:

   `c:\c563\include`
7. Click on the `OK` button to accept the changes and close the dialogs.

1.3 INSTALLATION FOR LINUX

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

`SW`*`product-version-RPMrelease`*`.i386.rpm`
`sw`*`product_version-release`*`_i386.deb`
`SW`*`product-version`*`.tar.gz`

These three files contain exactly the same information, so you only have to install one of them. When your Linux distribution supports RPM packages, you can install the `.rpm` file. For a Debian based distribution, you can use the `.deb` file. Otherwise, you can install the product from the `.tar.gz` file.

1.3.1 RPM INSTALLATION

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

3. Go to the directory on which the CD–ROM is mounted:

   ```
   cd /cdrom
   ```

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 verion 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 DEBIAN INSTALLATION

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.

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.

3. Go to the directory on which the CD–ROM is mounted:

   ```
   cd /cdrom
   ```

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

   ```
   dpkg –i sw*.deb
   ```

   This will install or upgrade all products in a subdirectory of the default installation directory `/usr/local`.

1.3.3 TAR.GZ INSTALLATION

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.

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.

3. Go to the directory on which the CD–ROM is mounted:

   ```
   cd /cdrom
   ```

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.

1.3.4 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.

1.4 INSTALLATION FOR UNIX HOSTS

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.

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

2. Insert the CD–ROM into the CD–ROM drive and mount the CD–ROM on a directory, for example `/cdrom`.

   Be sure to use an ISO 9660 file system with Rock Ridge extensions enabled. See the UNIX manual pages about **mount** for details.

3. Go to the directory on which the CD–ROM is mounted:

   ```
   cd /cdrom
   ```

4. Run the installation script:

   ```
   sh install
   ```

   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 ***
   SWxxxxxx 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 SWxxxxxx xxxx.xxxx completed.
```

For the DSP563xx/DSP566xx the directory `c563` will be created. For the DSP5600x this directory will be `c56`.

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

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 Deaemon 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 infomation is entered, the license manager must be started (see section 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 approrpiate 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.

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 approriate sequence number. Refer to you 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 (*lfpath*) 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@elliot
```

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
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
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

TASKING

2 CHAPTER

2.1 INTRODUCTION TO DSP56XXX FAMILY C CROSS-COMPILER

This manual provides a functional description of the TASKING DSP56xxx Family C Cross–Compiler. This manual uses **c563** (the name of the binary) as a shorthand notation for "TASKING DSP563xx/DSP566xx C Compiler", and uses **c56** as a shorthand notation for "TASKING DSP5600x C Compiler".

TASKING offers a complete toolchain for the Motorola DSP56xxx Family of Digital Signal Processors (DSPs) and their derivatives. The DSP563xx (24–bit), the DSP566xx (16–bit), the DSP5600x (24–bit) family are supported. This manual uses 'DSP5600x' to indicate the derivatives that have a '0' in the third position (e.g. DSP56002), 'DSP563xx' for those that have a '3' in the third position (e.g. DSP56366) and 'DSP566xx' along the same lines. In this manual all core versions are treated identical unless implementation differences require otherwise. 'DSP56xxx' is used as a shorthand notation for the Motorola DSP56xxx Family of Digital Signal Processors (DSPs) and their derivatives. The toolchain contains a C++ compiler, a C compiler, an assembler, a linker, a locator, a control program, a make utility, a library maintainer, an object reader utility and a debugger.

The C compilers are dedicated to the DSP architecture of the DSP56xxx. This means that you can access all special features of the DSP56xxx in C. And yet the C compiler conforms to the ANSI standard. It is a single pass, optimizing compiler that generates fast and compact code.

c563 generates assembly source code using the DSP563xx or DSP566xx assembly language specification. You must assemble this code with the TASKING DSP563xx Cross–Assembler. This manual uses **as563** as a shorthand notation for "TASKING DSP563xx/DSP566xx Cross–Assembler".

c56 generates assembly source code using the DSP5600x assembly language specification. You must assemble this code with the TASKING DSP5600x Cross–Assembler. This manual uses **as56** as a shorthand notation for "TASKING DSP5600x Cross–Assembler".

You can link the generated object with other objects and libraries using the TASKING DSP563xx/DSP566xx linker. In this manual we use **lk563** as a shorthand notation for "TASKING DSP563xx/DSP566xx linker". You can also link Motorola COFF objects and libraries with **lk563**. You can locate the linked object to a complete application using the TASKING DSP563xx/DSP566xx locator. In this manual we use **lc563** as a shorthand notation for "TASKING DSP563xx/DSP566xx locator". Use **lk56** and **lc56** for the DSP5600x linker and locator.

The DSP56xxx toolchain also accepts C++ source files. C++ source files or sources using C++ language features need to be preprocessed by **cp563** (DSP563xx/DSP566xx), **cp56** (DSP5600x). The output generated by **cp563** is DSP563xx or DSP566xx C, which can be translated with the C compiler **c563**. The output generated by **cp56** is DSP5600x C, which can be translated with the C compiler **c56**.

Note that the C++ compilers are not part of the C compiler package. They can be ordered separately from TASKING.

The programs **cc56**, and **cc563** are control programs for the DPS5600x, and DSP563xx/DSP566xx respectively. The control program facilitates the invocation of various components of the DSP56xxx toolchain. **cc563** recognizes several filename extensions. C++ source files (`.cc`, `.cxx` or `.cpp`) are passed to the C++ compiler. C source files (`.c`) are passed to the compiler. Assembly sources (`.asm` or `.src`) are 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. A list of supported platforms and emulators is available from TASKING.

In this manual **c563**, **as563**, **lk563**, **lc563**, **cc563**, **mk563**, **ar563** and **pr563** are used to indicate the executables of both DSP56xxx toolchains, unless explicitly stated otherwise.

2.2 GENERAL IMPLEMENTATION

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

2.2.1 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 *frontend* 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 DSP56xxx assembly instructions.

The peephole optimizer / pipeline scheduler phase:

This phase uses pattern matching techniques to perform peephole optimizations on the internal code (e.g. deleting obsolete moves). The pipeline scheduler reorders and combines instructions to minimize the number of instructions. Finally the peephole optimizer translates the internal instruction code into assembly code for **as563**. The generated assembly does not contain any macros. The assembler is also equiped with an optimizer. This optimizer takes care of move parallelization, NOP removal and DO/REP optimization.

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 backend 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.

2.2.2 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, for example, 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.

2.2.3 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.

Loop unrolling

Try to duplicate a loop body 2, 4 or 8 times to reduce the number of branches and to create a longer linear code part. This optimization is only performed when hardware loops are not possible.

Dead store elimination

Expressions from which the result is never used are eliminated.

Hardware loop generation

Where possible replace loops in the program by the zero–overhead hardware loop supported by the DSP.

Register contents tracking

Improving generated assembly code by replacing operands, e.g. replace an immediate value with a register that contains that value already.

2.2.4 SPECIFIC OPTIMIZATIONS

Besides the common optimizations, some special optimizations are performed. These optimizations are specific for each DSP56xxx family. The compiler supports an optimization method that allows instruction re–ordering and parallelization of instructions.

2.2.4.1 REPLACING NOPS

In some cases the contents of an address register are not available in the next instruction due to pipeline effects. In such cases, the compiler generates a NOP instruction. The assembler tries to replace these NOPs with other instructions when possible. The compiler enables the NOP optimization of the assembler by generating an OPT directive in the assembly source.

2.2.4.2 INSTRUCTION PARALLELIZATION (PARALLEL MOVES)

In general, one or two moves can be performed together with an arithmetic instruction, which are called 'parallel moves'. The parallel moves are restricted by the addressing mode used, and not all arithmetic instructions can have all possible parallel moves. In this manual a place where a parallel move can be performed is called a 'move slot'. If there is an arithmetic instruction without a parallel move, you can see this as an 'empty move slot'.

The compiler does not generate parallel moves. So, all move slots are empty. The assembler tries to fill up the move slots as efficiently as possible. The compiler enables the parallel move optimization of the assembler by generating an OPT directive in the assembly source.

2.2.4.3 HARDWARE DO AND REP LOOPS

Typical DSP applications often processes data in loops. The available hardware loop instruction supports fast loops. It is important to have a compiler that generates loops as fast as possible, by making use of the hardware loop instructions and by making use of the parallel move capability of the DSP. The actual hardware DO to REP optimization is performed by the assembler. See also the section *Optimizations* of the chapter *Assembler* in the DSP56xxx Cross–Assembler User's Guide.

Take for instance the next C fragment:

```
_fract fir_filter(_fract data[], _fract _Y coef[])
{
     long _fract result;  int i;

     result = 0.0;
     for ( i = 0; i < 100; i++ )
     {
         result += data[i] * (long _fract) coef[i];
     }
     return _round(result);
}
```

The loop is recognized as a simple loop and the array references are interpreted as post increment pointers. The code will be internally rewritten by the compiler as:

```
result = 0.0;
tmp1 = data;
tmp2 = coef;
for ( i = 0; i < 100; i++ )
{
    result += *tmp1++ * *tmp2++;
}
```

Because of the very powerful addressing modes of the DSP families the pointer dereference and the auto–increment are easy to implement. The index `i` is not used anymore and will be optimized away. The compiler generates assembly code similar to the following code (the last column gives the code size in words and the execution time in clock cycles):

```
    clr  a               ; Clear result               1,1
    do   #100,L5         ; Hardware do loop           2,5
    move x:(r0)+,x0      ; Get an element of 'data'   1,1*
    move y:(r4)+,y0      ; Get an element of 'coef'   1,1*
    mac  x0,y0,a         ; Multiply and accumulate    1,1*
L5: void x0, y0, r0, r4  ; End of loop, indicate registers
                         ; unused after loop
    rnd  a               ; Round result               1,1
    rts                  ; Return it                  1,3
```

This results in a loop time of only 3 clock cycles! (The instructions within the loop are marked with an '*' after the size/timing figures.) But, the empty move slot(s) are not yet used. The problem is that `X0` and `Y0` must be known at the start of the mac instruction. It is possible to retrieve these values after the mac instruction for the next loop incarnation if the first values are obtained before the loop:

```
    clr  a               ; Clear result                 1,1
    move x:(r0)+,x0      ; Get first element of 'data'  1,1
    move y:(r4)+,y0      ; Get first element of 'coef'  1,1
    do   #100,L5         ; Hardware do loop             2,5
    mac  x0,y0,a         ; Multiply and accumulate      1,1*
    move x:(r0)+,x0      ; Get next element of 'data'   1,1*
    move y:(r4)+,y0      ; Get next element of 'coef'   1,1*
L5: void x0, y0, r0, r4  ; End of loop, indicate registers
                         ; unused after loop
    rnd   a              ; Round result                 1,1
    rts                  ; Return it                    1,3
```

At first sight, this code is not better. However, after applying the optimization of the assembler to put instructions in parallel, the result becomes:

```
    clr  a        x:(r0)+,x0  y:(r4)+,y0  ; Clear result,      1,1
                                          ; prime loop
    do   #100,L5                          ; Hardware do loop   2,5
    mac  x0,y0,a  x:(r0)+,x0  y:(r4)+,y0  ; Multiply and       1,1*
                                          ; accumulate, and get
                                          ; next elements of 'data'
L5:                                       ; and 'coef'
    rnd  a                                ; Round result       1,1
    rts                                   ; Return it          1,3
```

In this final result, the loop time is one instruction cycle, the absolute minimum. The hardware loop instruction 'do' can be replaced by a 'rep' because there is only one instruction left within the loop.

The optimization of DO to REP loops can be controlled with the **–OR**/**–Or** command line option of the compiler. This option can also be used with the `#pragma optimize` to turn the optimization on or off for some code parts.

Example:

```
#pragma optimize R  /* turn DO to REP optim off */
     for( ... )/* some loop */
#pragma optimize r  /* turn DO to REP optim on */
```

The compiler will not use a hardware loop when:

- the loop conditions cannot be evaluated in DO/REP instructions
- a function call is detected inside the loop, unless hardware stack extension is active
- the allocated hardware stack space is exhausted

When one of these criteria is met the compiler generates loops that consist of branch and jump instructions, which make it possible to do some optimizations not possible on hardware DO loops.

The nesting depth of hardware DO loops can be controlled with the **–L** command line option as each loop takes two hardware stack loads. Counting the nesting is restricted to one function level. Unless hardware stack extension is enabled, loops containing a function call will not be implemented as hardware loops. The compiler cannot determine the hardware stack use of the called function and must avoid exhausting it.. When the stack depth is exceeded, the compiler will not generate hardware DO loops for the highest levels.

Example:

The compiler is called as follows:

```
c563 -L4 example.c
```

The code in `example.c` is:

```
for( ... )        /* depth 1 -> no hardware DO loop */
{
     for( ... )      /* depth 2 -> hardware DO loop */
     {               /* stack level 2 */
          for( ... ) /* depth 3 -> hardware DO loop */
          {          /* stack level 4 */
               ...
          }
     }
}
```

2.2.4.4 BITFIELDS

The DSP56xxx instruction set contains several instructions to do fast operations on a single bit. The **c563** C compiler will use these functions for operations on bitfields.

Example:

```
struct    {
               unsigned int b :1;
          } a;
a.b = 1;
```

Generated code:

```
.
.
bset #0,x:Fa
.
.
```

Where `Fa` is the start address of the structure.

2.2.4.5 MAC INSTRUCTION GENERATION

The compiler will generate the multiply–accumulate instructions for multiplications whenever possible and useful. See section 2.2.4.3, *Hardware DO and REP Loops*, for an example.

2.2.4.6 ABSOLUTE ADDRESSING MODE USAGE

For accessing static and global objects the compiler will use absolute addressing modes whenever possible. When the object is defined as `_near` (see section *Storage Specifiers*) the compiler will use short addressing modes when the used instruction has such an addressing mode.

2.3 COMPILER STRUCTURE

If you want to build a DSP56xxx application you need to invoke the following programs directly, or via the control program:

- One of the C compilers (**c56** or **c563**), will generate 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.
- One of the corresponding cross–assemblers (**as563**, **as56**), which process the generated assembly source file into a relocatable object file with suffix `.obj`.
- The **lk563** linker (**lk56** for DSP5600x), 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 **lc563** locator (**lc56** for DSP5600x), 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 DSP563xx toolchain:

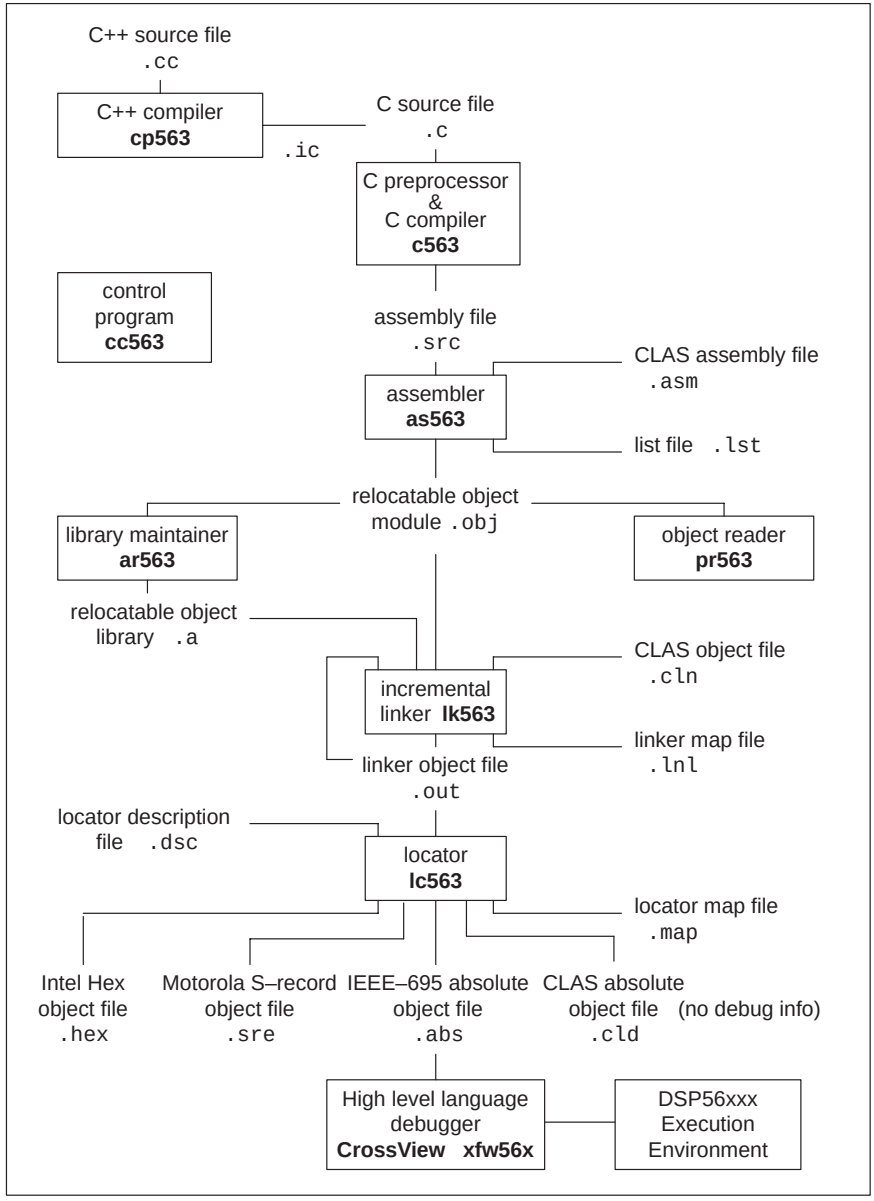


Figure 2–1: DSP563xx development flow

For the DSP5600x toolchain, each executable name ends in '**56**'.

The next figure explains the relationship between the different parts of the TASKING DSP563xx toolchain and the Motorola toolchain. This path is active when the **–S** option of **cc563** is used, or when the tool set with Motorola tools is selected in EDE.

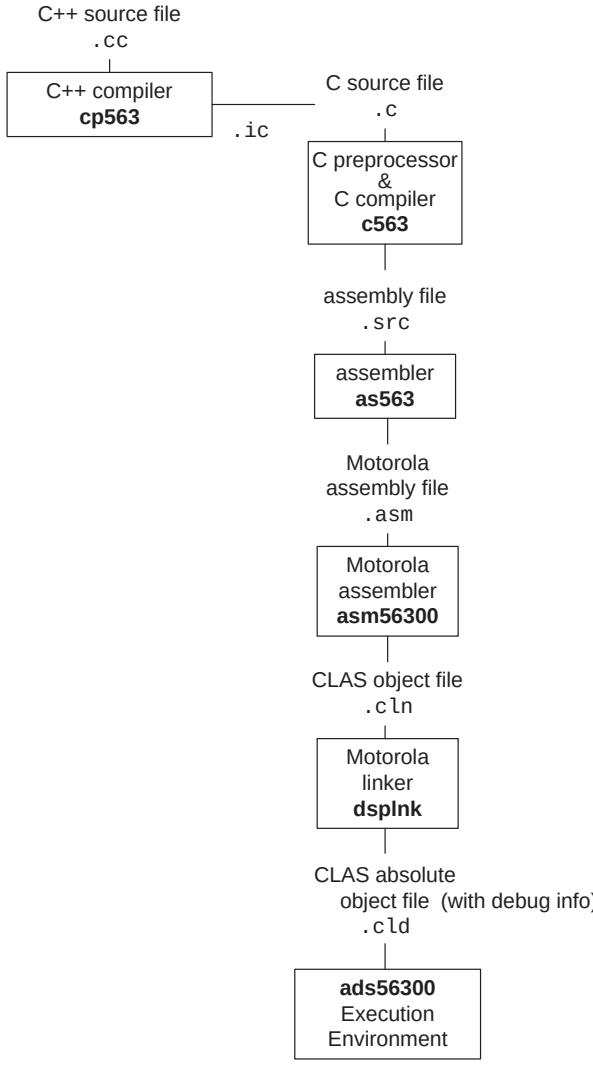


Figure 2–2: Motorola toolchain connection

The program **cc563** is a so–called control program, which facilitates the invocation of various components of the DSP56xxx toolchain. C++ source programs are compiled by the C++ compiler, 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, **lc563** produces output files in Motorola S–record format, Motorola CLAS compatible object 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 DSP56xxx Cross–Assembler User's Guide.

The name of the DSP56xxx CrossView Pro Debugger is **xfw56x**.

2.4 ENVIRONMENT VARIABLES

This section contains an overview of the environment variables used by the DSP56xxx toolchain.

Environment Variable	Description
AS56INC	Specifies an alternative path for include files for the assembler **as56**.
AS563INC	Specifies an alternative path for include files for the assembler **as563**.
C56INC	Specifies an alternative path for #include files for the C compiler **c56**.
C563INC	Specifies an alternative path for #include files for the C compiler **c563**.
C56LIB	Specifies a path to search for library files used by the linker **lk56**.
C563LIB	Specifies a path to search for library files used by the linker **lk563**.
CC56BIN	When this variable is set, the control program, **cc56**, prepends the directory specified by this variable to the names of the tools invoked.
CC563BIN	When this variable is set, the control program, **cc563**, prepends the directory specified by this variable to the names of the tools invoked.
CC56OPT	Specifies extra options and/or arguments to each invocation of **cc56**. The control program processes the arguments from this variable before the command line arguments.
CC563OPT	Specifies extra options and/or arguments to each invocation of **cc563**. The control program processes the arguments from this variable before the command line arguments.
LM_LICENSE_FILE	Identifies the location of the license data file. Only needed for hosts that need the FLEXlm license manager.

Environment Variable	Description
PATH	Specifies the search path for your executables.
TMPDIR	Specifies an alternative directory where programs can create temporary files. Used by **c56**, **c563**, **cc56**, **cc563**, **as56**, **as563**, **lk56**, **lk563**, **lc56**, **lc563**, **ar56**, **ar563**.

Table 2–1: Environment variables

2.5 SAMPLE SESSION

The subdirectory `c` in the `examples` subdirectory contains a demo program for the DSP56xxx toolchain.

In order to debug your programs, you will have to compile, assemble, link and locate them for debugging using the TASKING DSP56xxx 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.

2.5.1 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.

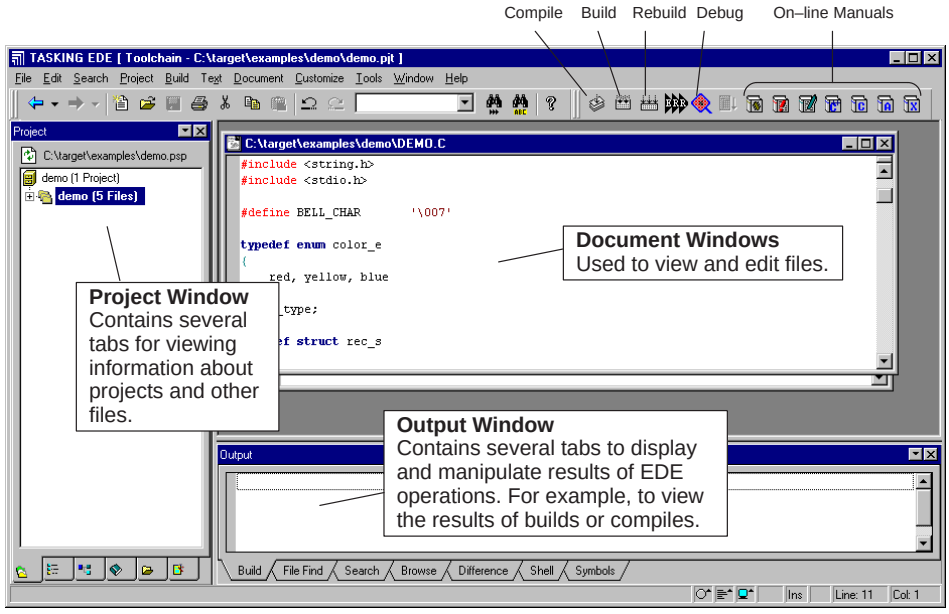
To use EDE on the `calc` demo program in the subdirectory `c` in the `examples` subdirectory of the DSP56xxx product tree follow the steps below. This procedure is outlined as a guide for you to build your own executables for debugging.

How to Start EDE

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

This will launch the EDE.

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.



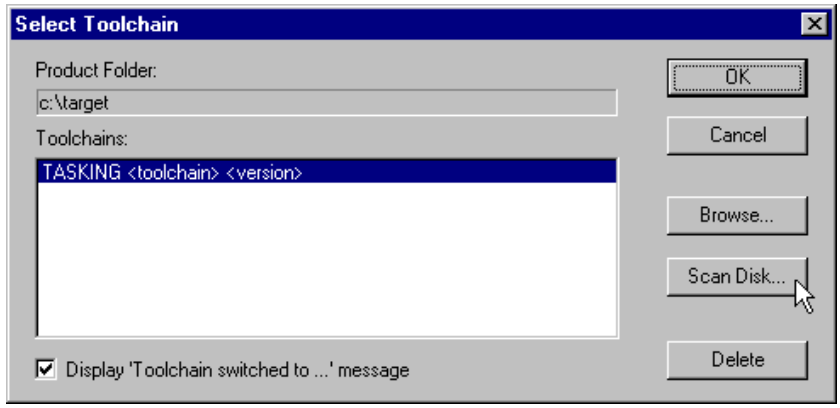
How to Select a Toolchain

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

If you have more than one TASKING product installed and you want to change toolchains, do the following::

1. From the `Project` menu, select `Select Toolchain...`

   The `Select Toolchain` dialog appears.



2. Select the toolchain you want. You can do this by clicking on a toolchain in the `Toolchains` list box and click `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:

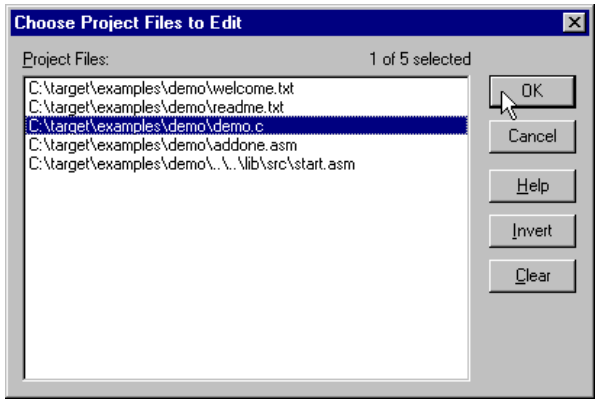
1. From the `Project` menu, select `Set Current ->`.
2. Select the project file to open. For the `calc` demo program select the file `calc.pjt` in the subdirectory `c` in the `examples` subdirectory of the DSP56xxx product tree. If you have used the defaults, the file `calc.pjt` is in the directory `c:\c563\examples\c` for the DSP563xx/DSP566xx (`c56` for the DSP5600x).

How to Load/Open Files

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

1. From the `Project` menu, select `Load Files...`

   The `Choose Project Files to Edit` dialog appears.



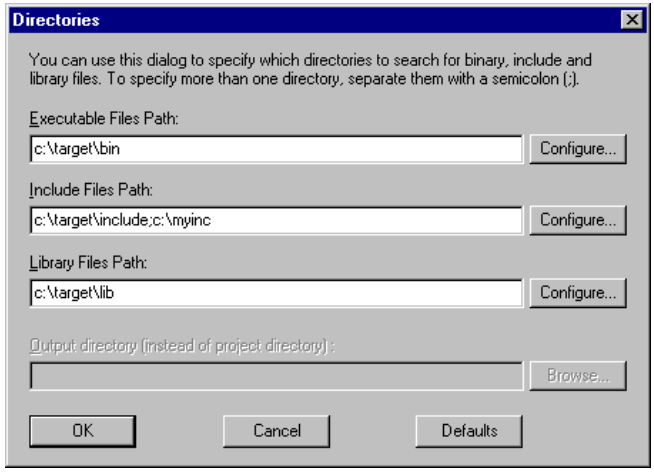
2. Choose the file(s) you want to open by clicking on it. You can select multiple files by pressing the `<Ctrl>` or `<Shift>` key while you click on a file. With the `<Ctrl>` key you can make single selections and with the `<Shift>` key you can select everything from the first selected file to the file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

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

Check the directory paths

1. From the `Project` menu, select `Directories...`.

   The `Directories` dialog appears.



2. Check the directory paths for programs, include files and libraries. You can add your own directories here, separated by semicolons.

3. Click `OK`.

How to Select a CPU Type

The next step is to compile the file(s) together with its dependent files so you can debug the application. But first you need to specify for which CPU type you want to build your application:

1. From the `Project` menu, select `Project Options...`

   The Project Options dialog box appears.

2. Select `CPU Selection`.

3. In the `CPU family/type` box, select the CPU or CPU family for which you want to build your application and click `OK`.

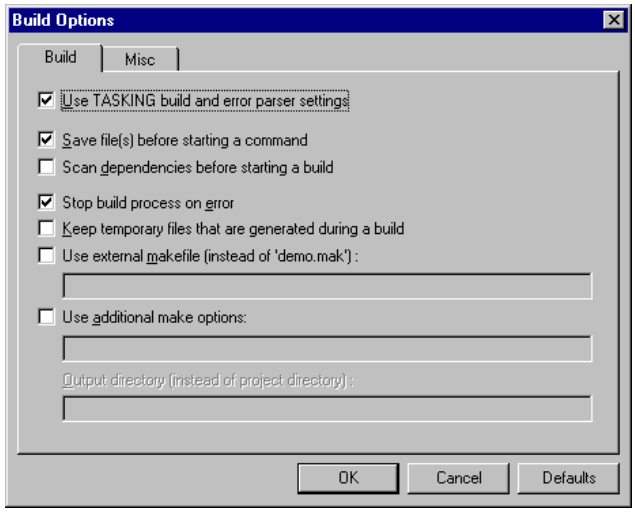
How to Build the Demo Application

Now you can build your 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 keep temporary files that are generated during a build.

1. From the `Build` menu, select `Options...`

   The `Build Options` dialog appears.



2. Make your changes and press the `OK` button.
3. From the `Build` menu, select `Scan All Dependencies`.
4. Click on the `Execute 'Make' command` button. The following button is the execute Make button which is located in the ribbon bar.

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 `Output` window.

You can see which commands (and corresponding output captured) which have been executed by the build process in the `Build` tab:

```
TASKING program builder vx.yrz   Build nnn SN 00000000
Compiling "calc.c"
Assembling "calc.src"
Linking to "calc.out"
Creating IEEE-695 absolute file "calc.abs"
```

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 start 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. From the `File` menu, select `Load Symbolic Debug Info...`

   The `Load Symbolic Debug Info` dialog box appears.

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. From the `View` menu, select `Source | Source lines`.

   The source window opens.

   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. From the `Run` menu, select `Reset Target System`.

   To run your application step–by–step:

3. From the `Run` menu, select `Animate`.

The program `calc.abs` is now stepping through the high level language statements. Using the toolbar 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. From the `File` menu, select `New Project Space...`

   The `Create a New Project Space` dialog appears.

2. Give your project space a name and then click `OK`.

   The `Project Properties` dialog box appears.

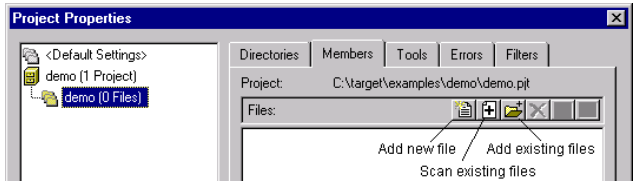
3. Click on the `Add new project to project space` button.

   The `Add New Project to Project Space` dialog appears.

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. From the `Project` menu, select `Load Files...` to open the 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.

2.5.2 USING THE CONTROL PROGRAM

A detailed description of the process using the sample program `calc.c` is described below for the DSP563xx/DSP566xx. For the DSP5600x use **cc56** where **cc563** is used. This procedure is outlined as a guide for you to build your own executables for debugging.

1. Make the subdirectory `c` 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 **cc563**:

   ```
   cc563 –g –M calc.c –o calc.abs
   ```

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

   Some optimizations may affect the ability to debug the code in a high level language debugger. Therefore, the **–O2** option is automatically selected with **–g** to switch off these optimizations. When the **–g** option is specified to the compiler with a higher optimization level, the compiler will issue warning message W555.

   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 `calc.obj`, the locator map file `calc.map` and the absolute output file `calc.abs`. The file `calc.abs` is in the IEEE Std. 695 format, and can directly be used by CrossView Pro. 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.

```
cc563 –g –M calc.c –o calc.abs –v0
```

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

```
+ c563 –o /tmp/cc7208b.src –g –M24x calc.c
+ as563 –o calc.obj –gs –M24x /tmp/cc7208b.src
+ lk563 –o/tmp/cc7208c.out –ddef_targ.dsc –uR_def_targ calc.obj
–lc24 –lfp24 –lrt24
+ lc563 –ocalc.abs –ddef_targ.dsc –f1 –M /tmp/cc7208c.out
```

The **–M24x** option of the compiler selects the 24–bit memory model with default data space in X memory. The **–lc24**, **–lfp24** and **–lrt24** options of the linker specify to link the appropriate C library, floating point library and run–time library.

As you can see, the tools use temporary files for intermediate results. If you want to keep the intermediate files you can use the **–tmp** option. The following command makes this clear.

```
cc563 –g –M calc.c –o calc.abs –v0 –tmp
```

This command produces the following output:

```
+ c563 –o calc.src –g –M24x calc.c
+ as563 –o calc.obj –gs –M24x calc.src
+ lk563 –ocalc.out –ddef_targ.dsc –uR_def_targ calc.obj
–lc24 –lfp24 –lrt24
+ lc563 –ocalc.abs –ddef_targ.dsc –f1 –M calc.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.

2.5.3 USING THE MAKEFILE

The subdirectories in the `examples` directory each contain a `makefile` which can be processed by **mk563**. Also each subdirectory contains a `readme.txt` file with a description of how to build the example.

To build the `calc` 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 `c` of the `examples` directory the current working directory.

   This directory contains a makefile for building the `calc` demo example. It uses the default **mk563** 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 **mk563**:

   ```
   mk563
   ```

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

   To see which commands are invoked by **mk563** without actually executing them, type:

   ```
   mk563 -n -a
   ```

The option **-a** causes all files to be rebuild, regardless wether they are out of date or not.

This command produces the following output:

```
TASKING DSP563xx/6xx program builder   vx.yrz Build nnn
Copyright 1996-year Altium BV          Serial# 00000000
cc563 -g  calc.c -o calc.abs
cc563 -cs -gn -s  intrnsic.c
cc563 -cs -gn -s  intrpt.c
```

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 **–o** option specifies the name of the output file.

To remove all generated files type:

```
mk563 clean
```

CHAPTER 3

LANGUAGE IMPLEMENTATION

TASKING

CHAPTER 3

3.1 INTRODUCTION

The TASKING DSP563xxx/DSP566xx Family C cross–compiler (**c563**) offers a new approach to high–level language programming for the DSP563xx/DSP566xx family. **c56** is the DSP5600x Family C cross–compiler. They conform to the ANSI standard, but allow you to control the special functions of the DSP5600x, DSP563xx and DSP566xx in C.

The extensions to the C language in the C compiler are:

additional data types

In addition to the standard data types, **c563** supports the fractional type (`_fract`), long fractional type (`long _fract`) and complex data type (`_complex`).

_at

You can specify a variable to be at an absolute address.

_nosat

You can specify a fractional variable to wrap around instead of going into saturation during calculations.

storage specifiers

Apart from a memory category (extern, static, ...) you can specify a storage specifier in each declaration. This way you obtain a memory model–independent addressing of variables in several address ranges (`_X`, `_Y`, `_L`, `_P`, `_near`, `_internal`, `_external`). The `_near`, `_internal` and `_external` modifiers can also be used on functions to force them in a specific memory region.

reentrant functions

In the mixed model (DSP5600x only) you can selectively define functions as reentrant (`_reentrant` keyword). Reentrant functions can be invoked recursively. Interrupt programs can also call reentrant functions.

interrupt functions

You can specify interrupt functions directly through interrupt vectors in the C language (`_fast_interrupt`, `_long_interrupt` keyword).

inline C functions

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

special calling conventions

With the `_compatible` keyword you can specify that a function must have the same calling convention as the Motorola C compiler. The `_callee_save` keyword can be used to indicate that a function must save all registers, instead of leaving this to the caller.

intrinsic functions

A number of pre–declared functions can be used to generate inline assembly code at the location of the intrinsic (built–in) function call. This avoids the overhead which is normally used to do parameter passing and context saving before executing the called function.

circular buffers

c563 supports the type modifier `_circ` for circular data structures and pointers.

bank switching

You can specify a function to be located in a particular bank with the keyword `_bank()`. A bank is a combination of an address range and a page number**.**

3.2 ACCESSING MEMORY

Members of the DSP56xxx family have separate program memory and data memory, and the data memory of the DSP5600x, DSP563xx and DSP566xx is, in turn, divided into two separate memory spaces, X and Y. The combination of X and Y memory space is called L memory space. Each address range is accessible through 16–bit or 24–bit addresses. Also, the entire address range is bit–addressable.

c563 offers language extensions to deal with the separate memory spaces. You can specify a storage type (and in case of a pointer the storage type of the object it points to) with the declaration of a C variable.

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 may in fact require language extensions. These parts often have some of the following properties. They

- access I/O, using the special function registers
- need high execution speed
- need high code density
- access non–default memory
- are used to service interrupts

3.2.1 STORAGE SPECIFIERS

Static storage specifiers can be used to allocate **static** objects in a particular memory area of the addressing space of the processor. All objects taking **static** storage may be declared with an explicit storage specifier. By default static variables will be allocated in X memory.

c56 and **c563** recognize the following storage type specifiers:

Storage Specifier	Description
_X	X memory specifier (default)
_Y	Y memory specifier
_L	L memory specifier
_P	program memory
_near	lowest 64 addresses of data memory. For functions: short addressable memory
_internal	internal memory
_external	external memory

Table 3–1: Storage type specifiers

***Example** (mixed* DSP56xxx *lines):*

```
int _X            Var_in_X;       /* allocate integer variable
                                     in X memory */
int _near _X      Var_in_low_X;   /* fast accessible integer
                                     in low 64 addresses
                                     of X memory */
int _X * _Y       Ptr_in_Y_to_X;  /* allocate
                                     pointer in Y memory,
                                     used to point to integers
                                     in X */
char _P   string[] = "DSP56xxx";  /* string in program memory*/
long _L   Long_in_L;              /* allocate long variable in
                                     L memory */
_internal int _X  Intern_in_X;    /* allocate integer variable
                                     in internal X memory */
```

The **c56** and **c563** compilers use the advantage of the fact that two objects allocated in X and Y memory, can be accessed simultaneously in the same instruction.

Using the `_near` storage qualifier, allows the compiler to generate faster access code for frequently used variables and functions. The 'near' range depends on the core for which the program is compiled.

Using the `_L` storage qualifier only makes sense for the types of a double word size, which are `unsigned long`, `signed long`, `float` and `long _fract`. The most significant word is allocated in X memory and the least significant word is allocated in Y memory at the same address (X:Y). When `_L` is used the compiler can generate faster code for accessing these types. The compiler does not apply `_L` to any other data type.

For the `_internal` and `_external` storage qualifier there is no difference in code generation. The information is passed to the locator which knows which part of the memory is internal and which part is external from the description file. By default, internal memory is filled first, then external memory. With the `_internal` storage qualifier (high priority) you force an object in internal memory. If there is not enough internal memory the locator will give an error. With the `_external` storage qualifier (low priority) you give a preference for external memory, but sections declared with `_external` will still be located in internal memory if it is available. Using internal memory means a higher performance and lower power consumption than using external memory. Sections with none of the two modifiers (medium priority) will be located before sections created with the `_external` memory modifier. Except for `_internal` declared sections, sections may cross the border between internal and external memory if there is no gap between them.

Functions are allocated in program memory by default; the storage specifier may be omitted in that case. Also, function return values cannot be assigned to a storage area. The `_near, _internal` and `_external` storage qualifiers can be used to express storage preferences.

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

```
int _X myvar _at(0x100);
```

This is useful to interface to object programs using fixed memory schemes. It allows two objects, when declared in two different memory areas, to have the same starting address, for which the compiler can attempt to generate more efficient code when both objects are used as operands of one operation.

Examples using storage specifiers:

Some examples of using storage specifiers:

```
int _X *p;   // pointer to int in X memory
int _Y *g;   // pointer to int in Y memory

g = p;       // the compiler issues a warning
```

If a library function declares:

```
extern int _X foo;  //extern int in X memory
```

and a data object is declared as:

```
int _Y foo;         //int in Y memory
```

the linker will flag this as an error. The usage of the variables is always without a storage specifier:

```
int _Y example;     /* define an int in Y memory */
example = 2;        /* assign example            */
```

The generated assembly would be:

```
move #2,X0
move X0,Y:example
```

All allocations with the same storage specifiers are collected in units called 'sections'. The section with the `_near` attribute will be located from address 0. An error will be generated if this section exceeds 64 words. Next, the sections with the `_internal` attribute will be located. The size of the internal memory is known by the locator (it reads the DELFEE description file). If the total size of `_near` and `_internal` memory exceeds the internal memory size, an error will be generated. It is always possible to control the location of sections manually.

3.2.2 MEMORY MODELS

c563 supports the 24–bit and 16–bit modes of the DSP563xx by three models: the 24–bit model, the 16/24–bit model and the 16–bit model. Furthermore it can generate code for DSP566xx targets (16–bit). By default, the **c563** compiles for the 24–bit model.

c56 supports three memory models for the DSP5600x: static, mixed and reentrant. By default, the **c56** compiles for the mixed model.

You can select one of these models with the **–M** option. Programs for the DSP563xx and DSP566xx are always compiled using a reentrant model, since a static model would not improve code performance.

The compiler also enables you to select a different default memory space. The default memory space is where all data without memory space modifiers is placed. This can be important for backward compatibility and special hardware layouts. On the DSP5600x and DSP563xx X, Y, L and P can be selected.

Moreover, if you select X or Y memory as default memory the stack will be placed in L memory by default. This improves the execution speed because on L memory double–word moves are possible.

Separate versions of the C and run–time libraries are supplied for all supported models, avoiding the need for you to re–compile or re–build these when using a particular model.

3.2.2.1 16 AND 24-BIT MODELS FOR DSP563XX

The DSP563xx supports 24 and 16–bit arithmetic modes and 24 and 16–bit addressing modes. The **c563** C compiler supports the following models for these modes:

24–bit arithmetic and 24–bit addresses	24–bit model
16–bit arithmetic and 24–bit addresses	16/24–bit model
16–bit arithmetic and 16–bit addresses	16–bit model

The 24–bit model is the default for **c563**. **c563** can also generate code for the DSP566xx, which is always 16–bit arithmetic with 16–bit addresses.

All DSP563xx models are reentrant.

3.2.2.2 DSP566XX MEMORY MODEL

The DSP566xx is supported with a special compiler model for **c563**. This model creates 16–bit code for the 16–bit address space of the DSP566xx, and takes into account the specific properties of the DSP566xx. The DSP566xx model is reentrant.

3.2.2.3 STATIC MODEL FOR DSP5600X

In the static model, C function parameters and automatics are passed via a static, overlayable area in memory. The linker uses a function call graph of the entire application for this purpose. Data areas of functions which do not call each other can be overlayed, since these functions will never be active simultaneously. However, this cannot be accomplished for functions called through pointers.

The static model approach for the DSP5600x version implies that function parameters and local variables are allocated statically instead of using a stack. The reason for this is that there is no efficient way of addressing objects on a stack. When it is assumed that register R7 is used as the stack pointer, three instructions would be needed to access an arbitrary object on the stack:

```
move #offset,n7
NOP
move x:(r7+n7),a
```

Absolute addressing only requires a single instruction:

```
move x:Fvar,a
```

When the NOP instruction cannot be replaced by something useful, the first sequence requires 3 or 4 instruction words and 7 or 8 cycles, depending on the size of the offset. Using absolute addressing, accessing an object requires only 1 or 2 instruction words and 2 or 7 cycles, depending on the size of the address.

For each function, the compiler creates a separate section for parameters, locals and temporary storage. To minimize the memory requirements of an application, the linker overlays the sections of functions that are not simultaneously active. The compiler generates information about function calls, so that the linker is able to construct a function call graph for the application.

As an example, consider a function `main` which calls the two functions `a` and `b`. The functions `a` and `b` are not simultaneously active, so they may use the same memory for local storage. This situation is similar to a stack based approach, where two functions also use the same memory for local storage, but where the memory is allocated on the stack.

```
void a( int x )
{
    printf( "a has %d\n", x );
}

void b( int x )
{
    printf( "b has %d\n", x );
}

void main(void)
{
    a( 1 );   //call function a()
    b( 2 );   //a() and b() are not
              //simultaneously active
}
```

Using a static memory model implies that the functions are not reentrant, and recursion is therefore not possible. Because of the limited depth, the hardware stack is only used for function return addresses and for hardware loops. To further limit the hardware stack requirements, a loop is not implemented by a hardware loop instruction when the loop body contains function calls.

To limit the hardware stack usage the return address of non–leaf functions is saved in the separate section for parameters, locals and temporary storage.

Whenever possible, variables and common subexpressions (CSE) are allocated in a register. Using registers increases performance and reduces power consumption.

Local variables that are allocated in a register during the entire function do not need to be allocated in memory.

3.2.2.4 MIXED MODEL FOR DSP5600X

Using a mixed memory model implies that the functions are not reentrant, and recursion is therefore not possible, except for functions explicitly declared `_reentrant`. This is the default memory model for the DSP5600x. The difference with the static model is that in the static model no register is reserved as stack pointer, while in the mixed model register R7 is always reserved as stack pointer. This implies that the compiler cannot use this register for storing automatics or temporary results in the mixed model.

3.2.2.5 DSP5600X STATIC AND MIXED MODEL LIMITATIONS

Function Pointers

When using the static memory model (DSP5600x only), the parameter space and local variable space is allocated in a fixed area in memory. When programming with function pointers, the compiler is not able to find out where to place arguments. Normally the compiler places arguments specified in the function call directly in the variable/argument space of the called function. Doing so, the caller passes the arguments implicitly to the callee. Thus, functions requiring stacked pointers may not be called via function pointers in the static memory model, since the run–time contents of the function pointer is unknown. Consequently, it is unknown which static memory area must be used.

Individual functions may be declared reentrant when compiling for the mixed model:

```
int _reentrant
multiply( int first, int second )
{
     return( first * second );
}
```

Reentrant functions use the stack for passing parameters and automatic variable allocation. If the function does not use any static data, does not call any direct or extended function, then the function is both recursive and reentrant. When using the reentrant memory model, all functions are in fact implicitly reentrant.

So, function pointers are only allowed to point to functions compiled using the reentrant model. Parameters are passed to these functions via the stack. In the reentrant memory model a function pointer may point to any function in the application.

Variable Argument Lists

When compiling functions using a static model (DSP5600x only), the compiler allocates a fixed amount of static memory space for all necessary arguments and local variables. When a function has a variable argument list, the required amount of space is defined by the calling function. The actual space needed may vary from call to call. As a result, the compiler does not know how much space should be allocated for passing parameters to the function with a variable argument list.

For non–reentrant functions with variable argument lists the compiler generates eleven words. For long argument lists this may not be enough.

To overcome this problem, functions with variable argument lists are implicitly `_reentrant` in the mixed memory model.

3.2.2.6 REENTRANT MODEL

If recursion or reentrancy is required, the application must be compiled using the reentrant scheme. It implies that all functions are declared `_reentrant`. The code generator then uses a software stack to pass parameters and for allocation of automatics. Register R7 is used as stack pointer by default.

3.2.2.7 _MODEL, _DSP, _DEFMEM AND _STKMEM

c563 introduces the predefined preprocessor symbols _MODEL, _DSP and _DEFMEM. The value of _MODEL represents the memory model selected (**–M***model* option). The value of _DSP represents the processor family (0, 1, 3 or 6 for the different families). The value of _DEFMEM represents the default memory space selected (**–M***mem* option). This can be very helpful in making conditional C code in one source module, used for different applications in different memory models. See also the section *Portable C Code*, explaining the include file `c56.h`.

The _STKMEM symbol represents the memory space used for the stack, in the same way as the _DEFMEM symbol. The stack may be in a different memory if default memory is X or Y. In this case it is placed in L memory by default, which can be changed with the option **–ML**.

For **c563** the value of _MODEL is:

16–bit model	16	(16–bit arithmetic/addresses)
16/24–bit model	1624	(16–bit arithmetic, 24–bit addresses)
24–bit model	24	(24–bit arithmetic/addresses)
DSP566xx model	16	(16–bit arithmetic/addresses)

For **c56** the value of _MODEL is:

static model	's'
mixed model	'm'
reentrant model	'r'

The value of _DEFMEM and _STKMEM can be **x**, **y**, **l** or **p**.

Example:

```
#if _MODEL == 's'   /* static model */
...

#endif
```

3.2.3 THE _AT() ATTRIBUTE

In DSP56xxx C it is possible to place certain variables at absolute addresses. Instead of writing a piece of assembly code, a variable can be placed on an absolute address using the `_at()` attribute.

Example:

```
_external unsigned char Display _at( 0x2000 );
```

The example above creates a variable with the name `Display` at address 0x2000 in external memory. In the generated assembly code an absolute section will appear. On this position 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 be initialized, except for absolute variables declared in rom.
- Functions cannot be declared absolute.
- 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 during link time (except when one of the modules declares the variable 'extern').

3.3 DATA TYPES

All ANSI C data types are supported, except `double` and `long double`, which both are evaluated as floats. In addition to these types, the `_fract`, `long _fract`, `enum` and `_complex` types are added. Object size and ranges for all DSP56xxx families are given in tables 3–2 and 3–3:

	DSP563xx/6xx 16–bit		DSP563xx 24–bit		DSP563xx 16/24–bit	
Data Type	**Size (bit)**	**Range**	**Size (bit)**	**Range**	**Size (bit)**	**Range**
signed char	8	–128 to +127	8	–128 to +127	8	–128 to +127
unsigned char	8	0 to 255U	8	0 to 255U	8	0 to 255U
signed short	16	–32768 to +32767	16	–32768 to +32767	16	–32768 to +32767
unsigned short	16	0 to 65535U	16	0 to 65535U	16	0 to 65535U
signed int	16	–32768 to +32767	24	–8388608 to +8388607	16	–32768 to +32767
unsigned int	16	0 to 65535U	24	0 to 16777215U	16	0 to 65535U
signed long	32	–2147483648 to +2147483647	48	–140737488355328 to +140737488355327	32	–2147483648 to +2147483647
unsigned long	32	0 to 4294967295UL	48	0 to 281474976710655	32	0 to 4294967295UL
_fract	16	[–1, 1]	24	[–1, 1]	16	[–1, 1]
long _fract	32	[–1, 1]	48	[–1, 1]	32	[–1, 1]
pointer	16	0 to 65535U	24	0 to 16777215U	24	0 to 16777215U
_circ pointer	16+16	0 to 65535U	24+24	0 to 16777215U	24+24	0 to 16777215U
float/double	16+8	+/– 1.1750E–38 +/– 3.4028E+38	24+8	+/– 1.1754940E–38 +/– 3.4028235E+38	16+8	+/– 1.1750E–38 +/– 3.4028E+38
enum	16	–32768 to +32767	24	–8388608 to +8388607	16	–32768 to +32767
_complex	2*16	[–1, 1] for both fields	2*24	[–1, 1] for both fields	2*16	[–1, 1] for both fields

Table 3–2: Data types DSP563xx/6xx

	DSP5600x	
Data Type	**Size (bit)**	**Range**
signed char	8	–128 to +127
unsigned char	8	0 to 255U
signed short	16	–32768 to +32767
unsigned short	16	0 to 65535U
signed int	24	–8388608 to +8388607
unsigned int	24	0 to 16777215U
signed long	48	–140737488355328 to +140737488355327
unsigned long	48	0 to 281474976710655
_fract	24	[–1, 1]
long _fract	48	[–1, 1]
pointer	24	0 to 16777215U
_circ pointer	24+24	0 to 16777215U
float/double	24+8	+/– 1.1754940E–38 +/– 3.4028235E+38
enum	24	–8388608 to +8388607
_complex	2*24	[–1, 1] for both fields

Table 3–3: Data types DSP5600x

- `char`, `short`, `int` and `long` are all integral types, supporting all implicit (automatic) conversions.
- although the `char` type is 8 bit, each `char` occupies one memory word, because the DSP56xxx has no instructions to access one byte efficiently.
- `char` and `short` are treated as 8–bit and 16–bit `int` respectively.
- the DSP56xxx convention is used, storing variables with the most significant part at the higher memory address (Little Endian).
- `Double` is equal to `float`.

3.3.1 THE FRACTIONAL DATA TYPE

The compiler supports the additional data type `_fract` to do fixed point arithmetic without the use of (expensive) special libraries. The implementation of this data type depends on the selected family member. Fractions can have values between −1 and +1 and can be used like integers and floating points, and combine little code overhead with a high dynamic range.

```
/* constant with value 0.3333... */

const _fract  one_third = 1.0/3;
```

Fixed point arithmetic follows the rules for floating point calculations.

Floating point constants in `[-1,1]` are interpreted as a fractional type which allows fixed point arithmetic with fractional constants without suffixes or casting. Floating point value `+1.0` will be interpreted as the closest fractional value to 1 possible.

Long Fractional Type

The `long` keyword can be used in combination with the `_fract` data type. The result is a fractional type with double precision, 32–bit or 48–bit depending on the selected family member.

Example:

```
long _fract  Lfval;
```

Rounding

Rounding can be controlled by the intrinsic function `_round()` (see section *Intrinsic Functions*).

Operations

You can use all operations on fractional data which are also allowed on floating point numbers. As an example, you can add two fractional numbers, but you cannot exclusive–or them. In addition, you can use the shift operators on a fractional number; the right side of the shift operator must be an integral type.

Example:

```
_fract f;
void main()
{
    f <<= 2;   // multiply f by 4.
}
```

Wrapping and Saturation

The default behavior of C fractional variables for an overflow is going into saturation. (Integer calculations will always truncate on overflows). For some calculations (such as those related to the maintenance of phase angle), it is useful to let an *overflow* wrap around (e.g., .75 + .5 => 1.25 , wraps to –.75). In the Motorola DSP, it can be done efficiently by storing A1 directly, which bypasses the saturation.

The `_nosat` keyword allows to define variables which will wrap around instead of going into saturation during calculations.

Example:

```
_fract _nosat Angle;
```

The variable `Angle` in this example will wrap around instead of going into saturation during calculations.

3.3.2 THE COMPLEX DATA TYPE

The compiler supports a pre–declared type definition of type complex, which is equivalent to:

```
typedef struct
{
    _fract   re;
    _fract   im;
} _complex;
```

This type is supported by means of intrinsic functions. Four standard operations are provided by intrinsic services:

```
_complex _cadd(_complex,_complex);           /* complex addition */
_complex _csub(_complex,_complex);        /* complex subtraction */
_complex _cmul(_complex,_complex);     /* complex multiplication */
_complex _cdiv(_complex,_complex);           /* complex division */
_complex _cmac(_complex,_complex,_complex);       /* complex mac */
```

This offers a simple but effective way to use complex arithmetic in C applications, without sacrificing portability or efficiency.

3.3.3 UNSIGNED CHARACTERS

The character type is treated as `unsigned char` by default. Arithmetic on unsigned characters can be done more efficiently than on signed characters. You can overrule this default with the **–u** command line option, which sets the default to signed char.

Examples:

The following declarations are identical when **–u** is not used.

```
char c;
unsigned char c;
```

The following declarations are identical when **–u** is used.

```
char c;
signed char c;
```

3.3.4 ANSI C TYPE CONVERSIONS

According to the ANSI C X3.159–1989 standard, a character, a short integer, an integer bitfield (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`.

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.

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
 *
 * and 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
                          */
}
```

3.3.5 MEMORY MAPPED REGISTERS

Each derivative of the DSP56xxx core can have its own features like timers, ADCs and I/O ports, etc. These features are implemented using memory mapped hardware registers. Most of these registers consist of bitfields. A register and its bitfields can be accessed with a structure defined at the fixed address of the register with all fields defined. The compiler will generate efficient code for accessing structures at absolute addresses and bitfields in these structures. This makes special keywords for defining registers, which is used in other compilers, superfluous.

Example:

```
typedef union            /* PLL Control Register      */
{
   struct
   {
      unsigned MF  : 12;  /*  0: Multiplication Factor */
      unsigned DF  : 4;   /* 12: Division Factor       */
      int XTLD     : 1;   /* 16: XTAL Disable          */
      int PSTP     : 1;   /* 17: STOP processing State */
      int PEN      : 1;   /* 18: PLL Enable            */
      unsigned COD : 2;   /* 19: Clock Output Disable  */
      int CSRC     : 1;   /* 21: Chip Clcok Source     */
      int CKOS     : 1;   /* 22: CKOUT Clock Source    */
      int rsvd     : 1;   /* 23: Reserved              */
   } B;

   int I;

} pctl_type;

#define PCTL  (*(pctl_type*)  0xFFFD)  /* PLL Control Register */

void
main( void )
{
   PCTL.B.PEN = 1;
   PCTL.I = 0x0010;
}
```

Generated code:

```
   bset   #18,x:<<$FFFD
   movep  #>$000010,x:<<$FFFD
```

To simplify programming for the various different DSP56xxx derivatives, special files are supplied containing the declarations of all memory mapped register names for a specific derivative. For example, `reg56002.h` for the DSP56002 derivative.

You can also add your own memory mapped register definitions within the C–source. All bits without a special meaning to the chip can be given a unique name with a unique purpose.

3.4 AUTOMATIC VARIABLES

The implementation of automatic variables in the static and mixed models of the DSP5600x has some limitations. In non–reentrant functions recursion is not possible. In these functions automatic variables are not allocated on a stack, but in a static area. In a reentrant function automatic variables are treated the conventional way: coming and going with a function on the stack. In static functions automatics are still overlayable with automatics of other functions. Allocation of automatics is subject to the memory model selected. In a static model this means static allocation in one of the RAM memory spaces. In the reentrant model this means dynamic allocation on the stack.

Although automatic variables are allocated in a static area with non–reentrant functions, they are **not** the same as local variables (within a function) which are declared to be static by means of the `static` keyword. The difference is:

- it is not guaranteed that an automatic variable still has the same value as the last time the function returned, because it may have been overlaid with another automatic variable of another module.
- it is guaranteed that the value of a static variable is the same as the last time the function returned. Static variables are never overlaid.

3.5 REGISTER VARIABLES

In C the `register` type qualifier tells 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. Whenever possible, the compiler allocates automatic objects and parameter objects within registers. **c563** therefore ignores the `register` keyword.

For every object not placed in registers, the next rules apply.

static: (DSP5600x only) In this model automatic variables are allocated on fixed positions in X or Y memory, which is directly addressable RAM. The code using this memory has a very high execution speed, so in this model there is no need to treat a register variable in a special way, because all automatic variables are accessed with a speed comparable to a real register.

reentrant: In this model automatic variables are allocated on the user stack and are addressed using the indexed addressing mode.

The code generator of **c563** uses a 'save by caller' strategy. This means that a function which needs the contents of one or more registers over a function call, must save the contents of these 'registers' and restore them after the call. The major advantage of this approach is, that only registers which are really used after the call are saved.

Conclusion: the usage of the `register` keyword is not necessary for improving code density or speed.

3.6 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 are initialized variables declared with the `const` specifier.

Examples *(static memory model)* **:**

```
int     i = 100;        /* 1 word  in P memory and
                           1 word  in X memory     */
_P int  j = 3;          /* 2 words in P memory     */
char   *t = "TEXT";     /* 6 words in P memory and
                           6 words in X memory:
                           1 word  for p,
                           5 words for "TEXT"      */
_P char t[] = "HELP";   /* 10 words in P memory    */
_X char c = 'a';        /* 1 word in P memory and
                           1 word in X memory      */
const int k = 400;      /* 1 word in X memory      */
```

3.7 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. It may be undesired that the compiler attempts to optimize a memory update by keeping the value in a register (e.g., a hardware register). When a variable is declared with the `volatile` qualifier, the compiler disables such optimizations. The ANSI standard 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 _X int real_time_clock _at(0x1200);

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

3.8 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 have static storage. The ANSI X3.159–1989 standard permits string literals to be put in ROM. **c563** offers the possibility to allocate a static initialized variable in ROM only, when you declare it with the `const` qualifier. This enables the initialization of a (const) character array in ROM (if not enough ROM is available, the string may be located in RAM as well):

```
const char romhelp[] = "help";
/* allocation of 5 words in ROM only */
```

Or a pointer array in ROM only, initialized with the addresses of strings, also ROM only:

```
char * const message[] = {"hello","alarm","exit"};
```

ANSI string concatenation is supported: adjacent strings are concatenated – only when they appear as primary expressions – to a single new one. The result may not be longer than the maximum string length (ANSI limit 509 characters, actual compiler limit 1500 characters).

The ANSI Standard states that identical string literals need not be distinct, i.e. may share the same memory. Because memory can be very scarce with DSP applications, **c563** overlays identical strings within the same module.

In section 3.1.4 the Standard states that behavior is undefined if a program attempts to modify a string literal. Because it is a common extension to ANSI (A.6.5.5) that string literals are modifiable, there may be existing C source modifying strings at run–time. This can be done either with pointers, or even worse:

```
"st ing"[2] = 'r';
```

c563 accepts this statement when strings are in both ROM and RAM.

3.9 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:

```
_P char *_X p;  /* pointer residing in X,
                   pointing to P */
```

means `p` has storage type `_X` (allocated in RAM), but has logical type 'character in target memory space P'. 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 _P *_X p;  /* pointer residing in X,
                   pointing to ROM (P) */
```

If the target memory and storage memory of a pointer are not explicitly declared, **c563** uses the default `_X` storage specifier. You can overrule the default memory with the **–M** command line option.

In pointer arithmetic **c563** checks, besides the type of each pointer, also the target memory of the pointers, which must be the same. For example, it is invalid (and has no use) to assign a pointer to `_X` to a pointer to `_L`. Of course, an appropriate cast corrects the error.

3.10 INTEGER DIVISION AND MODULO

If you divide a negative number by a positive one, there are two possible outputs, one smaller than the actual (`float`) value, one larger than it, and they have different modulo values too. For example, `–5 / +3` equals `–1.666`, so converted to integers we can get:

```
–5 / +3 = –2,  –5 % +3 = +1   ==>  (–5 / +3) * +3 + (–5 % +3) = –5
```

OR:

```
–5 / +3 = –1,  –5 % +3 = –2   ==>  (–5 / +3) * +3 + (–5 % +3) = –5
```

The ANSI C standard (section 3.3.5) allows both sets of outcomes, it only requires that the calculation sums up correctly as shown after the arrow. In the TASKING C compiler we have chosen to implement the first form, as this gives the most compact code: we can do divisions with shifts, and modulo operations with bitwise–and, if the second operand is a power of two.

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 );
        move    #3,r5
        move    r5,x:Fw
; t.c        13          z = add( x, y );
        move    x:Fy,y1
        move    x:Fx,b
        add     y1,b
        move    b1,x:Fz
```

3.12 INLINE ASSEMBLY

c563 supports using assembly in the following ways:

1. by using the `_asm()` intrinsic function;
2. by using the `__asm( )` intrinsic function, which allows passing and returning values to the assembly code;
3. by using pragmas;
4. by linking with separate assembly modules.

C modules that contain inline assembly are not portable and harder to prototype in other environments.

3.12.1 USING THE _ASM INTRINSIC FUNCTION

c563 supports the intrinsic function `_asm()` for inline assembly. The argument of this function is a string that contains the assembly code to be inlined.

For example:

```
_asm("ori  #$03,MR");    // disable interrupts
```

This is equal to:

```
#pragma asm
     ori  #$03,MR
#pragma endasm
```

The advantage is that the `_asm()` intrinsic can be used in pre–processor defines:

```
#define _disable_interrupts()  _asm("ori #$03,MR")
```

3.12.2 USING THE __ASM INTRINSIC FUNCTION

To allow access to local variables from inline assembly, the compiler provides the `__asm()` (two leading underscores) intrinsic function. This function is highly compatibly with the Motorola C compiler function, although differences may exist in compiler–structure dependent areas. Note that other uses of the `__asm` keyword in the Motorola C compiler (e.g. forcing variables in specific registers) are not supported. With the `__asm` keyword C expressions can be assigned to a register from a group (e.g. an address register), and result values can be written to local or global C variables.

The general syntax of the `__asm` statement is:

```
__asm [volatile] (  instruction_template
                    [ : output_param_list
                    [ : input_param_list
                    [ : regsave]]] );
```

where,

instruction_template	is a string that may contain **%**[*mod_char*]*parm_nr* parameters from the in– or output list.
mod_char	is an operand modifier character (see table 3–5)
parm_nr	is a parameter number in the range 0..31.
output_param_list	[[**"=***constraint_char***"(***c_expression***)**],...]
input_param_list	[[**"***constraint_char***"(***c_expression***)**],...]
constraint _char	is an operand constraint character (see table 3–4)
regsave	[[**"***register_name***"**],...]
register _name	is one of the work register names (see below)

The constraint characters specify a group of registers for the parameter. The modifier characters select a specific part of the register selected for the parameter. For output parameters, the *c_expression* must be something that can be assigned to (an lvalue). Registers that are used in the assembly instructions must be reserved, either in the parameter lists or in the reserved register list (*regsave*, above). The compiler takes account of these lists (but does not interpret the instruction template!), and no unnecessary register saves and restores are placed around the inline assembly instructions.

Except for the instruction template, all parts within the braces are optional. The TASKING C compiler accepts but ignores the `volatile` keyword after the `__asm` keyword. Also the superfluous '=' equals signs in the output parameter list are accepted but ignored.

Available input/output operand constraints

Char	Type	Operand	Remark
A	address register	r0..r7	The compiler excludes the user stack pointer and reserved address registers
N	offset register	n0..n7	The compiler excludes the user stack pointer offset and reserved offset registers
C	address + modulo register	r0/m0..r7/m7	For circular pointers
D	accumulator	a, b	The complete 56–bit accumulator registers
R	input register	x0, y0, x, y	Selection depends on operand size
r	GP register	a, b, x0, y0, x1, y1, r0..r7, n0..n7	All general–purpose int–sized registers except user stack pointer and offset
S	source register	x0, x1, y0, y1, x, y	Selection depends on operand size
i	immediate value	#value	

Char	Type	Operand	Remark
m	memory	x:Fvar, x:(r7–i)	Stack or memory operand
number	other operand	same as *number*	Used when in– and output operands must be the same

Table 3–4: Available input/output operand constraints

Available operand modifiers

Char	Constraint	Operand	Remark
j	A, C	n0..n7	Offset register matching with address register. You must save and restore these registers.
v	A, C	m0..m7	Modulo register matching with address register (this modifier is not supported in Motorola C).
e	D	a1, b1	High part of accumulator.
h	D	a0, b0	Low part of accumulator.
k	D	a2, b2	Extension word of accumulator.
g	R, S	x1, x0, y1, y0	Other part of input register
i	R, S	x, y	Long input register matching with integer–sized input register.
f	m	x:,y:,p:,l:	Insert memory space for the operand.
p	i	$0000..$FFFF	Force 16–bit sized constant
q	i	$000000..$FFFFFF	Force 24–bit sized constant

Table 3–5: Available operand modifiers

Possible work registers

```
a, a0, a1, a2, b, b0, b1, b2, x, x0, x1, y, y0, y1,
r0..r7, n0..n7 (except user stack pointer r and n
register)
```

`m0..m7` cannot be reserved, they must be restored in the template to –1 after use. Of course the associated r–register must be set free, either by making it an output parameter, or by placing it in the reserved register list.

Most of the examples below are very trivial, and could be coded in C just as well with similar results. They are provided just to demonstrate the use of the `__asm` keyword. Situations where the `__asm` keyword is used should give a definite improvement of the generated code, to justify the complexity and lack of portability of the `__asm` statement. Assembly functions larger than ten statements should preferably be written in a separate assembly module, not inline, to improve readability and portability.

It is not allowed to create loops with multiple `__asm` statements, or generate (conditional) jumps across `__asm` statements. The compiler cannot detect these and will generate incorrect code for the registers involved. If you want to create a loop with `__asm`, the whole loop must be contained in a single `__asm` statement, independent of it being a hardware loop or a backward jump. The same restriction applies to (conditional) jumps. As a rule of thumb, all references to a label in an `__asm` statement must be contained in the same statement.

Example 1

A simple example without input or output parameters. The `mr` register is not a register used by the compiler and therefore can be used without reserving it.

```
__asm( "andi #$FC,mr");
```

Generated code:

```
andi $FC,mr
```

Example 2

The simplest example, it only generates a label. Because the compiler inserts a tab character before emitting the first character of the template, we must provide a newline first (or follow the label with a semicolon).

```
__asm( "\nmylabel");
```

Generated code (note that the first line is empty):

```

mylabel
```

Example 3

Assign the result of inline assembly to a variable. An accumulator–type register is chosen for the parameter due to the constraint `D`; the compiler decides which accumulator is best to use. The string `%0` in the template is replaced with the name of this accumulator. Finally, the compiler generates code to assign the result to the output variable.

```
int var1;

void main(void)
{
    __asm( "move #<$FF,%0" : "=D"(var1));
}
```

Generated code:

```
move #<$FF,a
move a1,x:Fvar1
```

Example 4

Multiply two expressions and assign the result to a variable. Two expressions are multiplied and the result is placed in a variable. An accumulator–type register is necessary for the output parameter (constraint `D`, `%0` in the template); alu registers (X0, Y0, X1 or Y1) must be used for the input registers (constraint `S`, `%1` and `%2` in the template). The compiler generates code to move the input expressions into the input registers and to assign the result to the output variable.

```
int var1, var2;  long result;

void main(void)
{
    __asm( "mpy %1,%2,%0" : "=D"(result) : "S"(var1),
           "S"(var2) );
}
```

Generated code:

```
move x:Fvar1,x0
move x:Fvar2,x1
mpy  x0,x1,a
move a0,x:Fresult
move a1,x:Fresult+1
```

Example 5

Execute a block move in Y memory. Two pointer expressions are given and the contents of a memory block in Y memory is copied. This example has no output, and uses a scratch register for the copying action; it also demonstrates the use of C expressions in the parameters. Notice that the address registers have been declared as output registers because they are destroyed in the routine. To get the same registers in the input parameter list, the number of the output parameter is used instead of the constraint character. The instruction template has been split up over several shorter source lines for better readability. The lines can be written on a single line as well (with '\n' sequences!).

```
int _Y array[100];

void main(void)
{
    __asm( ”do   #20,_copy    \n”
    ” move y:(%0)+,y0         \n”
    ” move y0,y:(%1)+         \n”
    ”_copy:                   \n”
    : ”=A”, ”=A”
    : ”0”(&array[20]), ”1”(&array[40]) : ”y0” );
}
```

Generated code:

```
        move  #Farray+20,r0
        move  #Farray+40,r1
        do    #20,_copy
        move  y:(r0)+,y0
        move  y0,y:(r1)+
_copy:
```

Example 6

The previous example can be improved in two ways: the loop count can be made a parameter as well, and we can leave it to the compiler to decide which scratch integer register to use. It is better to reserve a work register through an unassigned output parameter than through the reserved list, because specifying a group gives the compiler more optimization opportunities than handpicking a register.

```
#define BLOCKSIZE       10
int _Y array[10][BLOCKSIZE];

void main(void)
{
    __asm( ”do   #%5,_copy    \n”
    ” move y:(%0)+,%2         \n”
    ” move %2,y:(%1)+         \n”
    ”_copy:                   \n”
    : ”=A”, ”=A”, ”=r”
    : ”0”(&array[2]), ”1”(&array[4]), ”i”(2*BLOCKSIZE)
);
}
```

Generated code:

```
      move  #Farray+20,r0
      move  #Farray+40,r1
      do    #20,_copy
      move  y:(r0)+,n6
      move  n6,y:(r1)+
_copy:
```

Example 7

The register used in the assembly template can be changed with a modifier, which is placed between the '%' and the number of the parameter. This way you can use partial registers and tell the compiler to reserve the complete register. The following example swaps the two halves of a long. An accumulator is used as the input to the function, because the function parameter will be passed in A. A work register is chosen from the alu register set.

```
void swap_halves(long li)
{
    __asm ("move %e0,%1       \n"
    " move %h0,%0             \n"
    " move %1,%h0             \n"
    : "=D"(li), "S" : "0"(li) );
    return li;
}
```

Generated code:

```
Fswap_halves:
      move  a1,x0
      move  a0,a
      move  x0,a0
      rts
```

Example 8

When a modulo register is used in the template, it must be restored to the default value –1 (linear addressing mode) after use. The modifier 'v' can be used to select the modifier register associated with an address register.

```
void circ_get(int *p)
{
    int result;

    __asm ("move #24-1,%v1    \n"
    " move x:(%1),%0          \n"
    " move #-1,%v1            \n"
    : "=D"(result) : "A"(p) );
    return result;
}
```

Generated code:

```
Fcirc_get:
      move  #24-1,m0
      move  x:(r0),a
      move  #-1,m0
      rts
```

3.12.3 USING INLINE ASSEMBLY PRAGMAS

c563 supports inline assembly using the following pragmas:

#pragma asm	Insert assembly text following this pragma.
#pragma asm_noflush	Same as #pragma asm but the peephole optimizer will not flush the code buffer.
#pragma endasm	Switch back to the C language.

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.

Example:

```
#pragma asm
     ori  #$03,MR
#pragma endasm
```

The compiler will not save any registers prior to the pragma **asm** and also does not restore the original contents of the registers after the pragma **endasm**. This implies that when registers must be used within the pragma **asm**/**endasm** you have to save the contents of these registers first.

For using registers within the inline assembly you cannot rely on the registers the compiler allocated for C variables. Changing the C code may change the register allocation. Also the register allocation may change in future versions of the C compiler. You should use the `__asm()` intrinsic function if your code requires parameter passing.

These pragmas can also be used outside the scope of a function, for instance to create temporary storage for an assembly routine or to specify a vector. Using NOP instructions to place code at specific positions must be avoided, as the assembler will remove these when in optimizing mode. This is a common method in the vector table in combination with short jumps, but it can easily be avoided with multiple ORG statements.

3.12.4 LINKING WITH SEPARATE ASSEMBLY ROUTINES

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 */
     {
          ...
     }
     ...
}
```

With the `_asmfunc` function qualifier the parameter interface is identical to the standard calling convention (see section 7.3 Calling Conventions), but the stack is not used for passing arguments to functions. When there are more arguments to be passed than fit in the registers the compiler will issue an error message.

All registers can be used in the assembly function. On return, the M–registers must be reset to –1 (except M0 if the function returns a circular pointer).

3.13 INTRINSIC FUNCTIONS

When you want to use some specific DSP56xxx instructions, that have no equivalence in C, you would be forced to write assembly routines to perform these tasks. However, **c563** offers a way of handling this in C with a number of built–in functions, which are implemented as intrinsic functions.

Intrinsic functions appear to the programmer as normal C functions, but the difference is that they are interpreted by the code generator, and as a result more efficient code may be generated. 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 inline 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 **c563**
- 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	Description
_abs()	Absolute value of int argument
_asm()	Generate inline assembly
__asm()	Generate inline assembly with parameter passing
_cache_get_start()	Get first address of cache region
_cache_get_end()	Get last address of cache region
_cadd()	complex addition
_cdiv()	complex division
_cmac()	complex multiply–accumulate
_cmul()	complex multiplication
_csub()	complex division
_ext()	Extend high order part of accumulator for (unsigned) chars and shorts

Function	Description
_fabs()	Absolute value of _fract argument
_fract2int()	Convert _fract to int
_fsqrt()	Generate a JSR to the run–time library function Rfsqrt()
_int2fract()	Convert int to _fract
_labs()	Absolute value of long argument
_lfabs()	Absolute value of long _fract argument
_lfract2long()	Convert long _fract to long
_long2lfract()	Convert long to long _fract
_memcpy()	Copy block of memory. Compact memcpy replacement
_memset()	Fill block of memory. Compact memset replacement
_nop()	NOP instruction, not optimized away
_pdiv()	Calculates a/b inline, when it is known that a and b are both positive
_pflush()	Flush cache
_pflushun()	Flush unlocked sectors
_pfree()	Global unlock
_plock()	Lock 1 sector in cache
_punlock()	Unock 1 sector in cache
_rol()	Rotate left
_ror()	Rotate right
_round()	Specifies rounding (fractional arithmetic operations)
_sema_clr()	Clear semaphore
_sema_set()	Set semaphore
_sema_tst()	Test semaphore
_stop()	STOP, stop mode saves power consumption
_strcmp()	String compare. Compact strcmp replacement
_strcpy()	String copy. Compact strcpy replacement
_strlen()	String length. Compact strlen replacement
_swi()	SWI (**c56**) or TRAP (**c563**), software interrupt

Function	Description
_vsl()	Viterbi shift left (**c563** only)
_wait()	WAIT, wait mode saves power consumption

Table 3–6: Intrinsic functions

Prototypes for the intrinsic functions are present in `c56.hc`. Below are the details of the implemented intrinsic functions in alphabetical order (sample C source with generated assembly are given below):

_abs

```
int _abs( int operand );
```

Absolute integer value. Generate ABS instruction.

Returns the result.

Example:

```
volatile int ia, ib;
ia = _abs( ib );
```

... Code ...

```
     move x:ss_main+23,b     ; ib
     abs  b                  ; _abs(ib)
     move b1,x:ss_main+24    ; ia
```

_asm

```
void _asm( const char * asm );
```

This function can be used to generate inline assembly. The *asm* argument must be a string constant with valid assembly code. The compiler will prefix the string with a tab. For more information, see section 3.12.1, *Using the _asm Intrinsic Function*.

Returns nothing.

Example:

```
#define SAVEINTSTAT _asm( "move SR,X:(r2)+" ); \
     _asm( "ori  #$03,MR" )

     SAVEINTSTAT;
```

... Code ...

```
     move SR,X:(r2)+
     ori  #$03,MR
```

__asm

```
void __asm( const char * instruction_template
            [ : output_param_list
            [ : input_param_list
            [ : regsave ]]] );
```

This function can be used to generate inline assembly with parameter passing. The *instruction_template* argument must be a string constant with valid assembly code and optional register placeholders. The compiler will prefix the string with a tab. For more information, see section 3.12.2, *Using the __asm Intrinsic Function*.

Returns nothing.

_cache_get_start

```
int _P * _cache_get_start
                 ( void _cache_region (* fptr )() );
```

This function gets the start address of the cache region belonging to *fptr*.

Returns the start address of the cache region belonging to *fptr*.

Example:

See example with **_plock** function.

This function is only available for the DSP563xx. For more information about instruction cache support, see section 3.16 *DSP563xx Cache Support*.

_cache_get_end

```
int _P * _cache_get_end
                    ( void _cache_region (* fptr )() );
```

This function gets the end address of the cache region belonging to *fptr*.

Returns the end address of the cache region belonging to *fptr*.

Example:

See example in section 3.16.4 *Examples*.

This function is only available for the DSP563xx. For more information about instruction cache support, see section 3.16 *DSP563xx Cache Support*.

_cadd

```
_complex _cadd( _complex op1, _complex op2 );
```

Complex addition.

Returns the result.

Example:

```
_complex ca, cb, cc;
cc = _cadd( ca, cb );
```

... Code ...

```
      move x:Fca,b
      move x:Fca+1,a
      move x:Fcb,x0
      move x:Fcb+1,x1
      add  x0,b
      add  x1,a
      move a,x:Fcc+1 ; cc
      move b,x:Fcc   ; cc
```

_cdiv

```
_complex _cdiv( _complex op1, _complex op2 );
```

Complex division. Calculate *op1* / *op2*.

Returns the result.

Example:

```
_complex ca, cb, cc;
cc = _cdiv( ca, cb );
```

... Code ...

```
     move x:Fca,b
     move x:Fca+1,a
     move x:Fcb,x0
     move x:Fcb+1,x1
     jsr  F_cdiv     ; _cdiv(ca, cb)
     move a,x:Fcc+1  ; cc
     move b,x:Fcc    ; cc
```

_cmac

```
_complex _cmac( _complex op1, _complex op2,
                _complex op3 );
```

Complex multiply and accumulate. Calculate *op1* + *op2* * *op3* generating MAC instructions.

Returns the result.

Example:

```
_complex ca, cb, cc, cd;
cd = _cmac( ca, cb, cc );
```

... Code ...

```
move x:Fca,b
move x:Fca+1,a
move x:Fcb,x0
move x:Fcb+1,x1
move x:Fcc,y0
move x:Fcc+1,y1
mac  x0,y0,b
macr -x1,y1,b
mac  x0,y1,a
macr x1,y0,a
move a,x:Fcc+1     ; cc
move b,x:Fcc       ; cc
```

_cmul

```
_complex _cmul( _complex op1, _complex op2 );
```

Complex multiplication.

Returns the result.

Example:

```
_complex ca, cb, cc;
cc = _cmul( ca, cb );
```

... Code ...

```
move x:Fca,x0
move x:Fca+1,x1
move x:Fcb,y0
move x:Fcb+1,y1
mpy  x0,y0,b
macr -x1,y1,b
mpy  x0,y1,a
macr x1,y0,a
move a,x:Fcc+1     ; cc
move b,x:Fcc       ; cc
```

_csub

```
_complex _csub( _complex op1, _complex op2 );
```

Complex subtraction. Subtract operand *op2* from *op1*.

Returns the result.

Example:

```
_complex ca, cb, cc;
cc = _csub( ca, cb );
```

... Code ...

```
     move x:Fca,b
     move x:Fca+1,a
     move x:Fcb,x0
     move x:Fcb+1,x1
     sub  x0,b
     sub  x1,a
     move a,x:Fcc+1     ; cc
     move b,x:Fcc       ; cc
```

_ext

```
unsigned long _ext( unsigned long operand );
```

Generate instructions to extend high order part of accumulator for (unsigned) chars and shorts. Generate MOVE instructions.

Returns the result.

Example:

```
volatile unsigned long ula, ulb;
ula = _ext( ulb );
```

... Code ...

```
     move x:ss_main+14,b
     move x:ss_main+13,b0
     move #0,b2
     move b1,x:ss_main+16     ; ula
     move b0,x:ss_main+15     ; ula
```

_fabs

```
_fract _fabs( _fract operand );
```

Absolute `_fract` value. Generate ABS instruction.

Returns the result.

Example:

```
volatile _fract fa, fb;
fb = _fabs( fa );
```

... Code ...

```
     move x:ss_main+11,b     ; fb
     abs  b                  ; _fabs(fb)
     move b,x:ss_main+12     ; fa
```

_fract2int

```
int _fract2int( _fract operand );
```

This intrinsic function changes a fractional value into an integer value without generating any conversion code.

Returns the result.

Example:

```
int i;
_fract f = 0.5;
i = _fract2int(f);
```

... Code ...

```
     move #<$40,y0
     move y0,x:Ff
     move y0,x:Fi
```

_fsqrt

```
_fract _fsqrt( _fract operand );
```

Calculate the square root of the fractional operand. Calls the run–time library function `Rfsqrt`.

Returns the result.

Example:

```
volatile _fract fa, fb;
fa = _fsqrt( fb );

... Code ...
     move x:ss_main+11,b     ; fb
     jsr  Rfsqrt             ; _fsqrt(fb)
     move b,x:ss_main+12     ; fa
```

_int2fract

```
_fract _int2fract( int operand );
```

This intrinsic function changes an integer value into a fractional value without generating any conversion code.

Returns the result.

Example:

```
int i = 0x400000;
_fract f;
f = _int2fract(i);

... Code ...
     move #<$40,y0
     move y0,x:Fi
     move y0,x:Ff
```

_labs

```
long _labs( long operand );
```

Absolute long value. Generate ABS instruction.

Returns the result.

Example:

```
volatile long la, lb;
la = _labs( lb );
```

... Code ...

```
     move x:ss_main+18,b
     move x:ss_main+17,b0
     abs  b               ; _labs(lb)
     move b1,x:ss_main+20 ; la
     move b0,x:ss_main+19 ; la
```

_lfabs

```
long _fract _lfabs( long _fract operand );
```

Absolute `long _fract` value. Generate ABS instruction.

Returns the result.

Example:

```
long _fract lfa, lfb;
lfb = _lfabs( lfa );
```

... Code ...

```
     move x:ss_main+8,b
     move x:ss_main+7,b0
     abs  b               ; _lfabs(lfb)
     move b,x:ss_main+10  ; lfa
     move b0,x:ss_main+9  ; lfa
```

_lfract2long

```
long _lfract2long( _lfract operand );
```

This intrinsic function changes a long fractional value into a long value without generating any conversion code.

Returns the result.

Example:

```
long l;
long _fract lf = 0.5;
l = _lfract2long(lf);
```

... Code ...

```
     move #<$40,x1
     move #0,x0
     move x1,x:Flf+1
     move x0,x:Flf
     move x1,x:Fl+1
     move x0,x:Fl
```

_long2lfract

```
_lfract _long2lfract( long operand );
```

This intrinsic function changes a long value into a long fractional value without generating any conversion code.

Returns the result.

Example:

```
long l = 0x4000000000L;
long _fract lf;
lf = _long2lfract(l);
```

... Code ...

```
     move #$4000,b
     move b1,x:Fl+1
     move b0,x:Fl
     move b,x:Flf+1
     move b0,x:Flf
```

_memcpy

```
void * _memcpy( void * dest, const void * src,
                  unsigned int size );
```

Copy a block of memory. An optimized inlined version of the library function `memcpy`, that works on all memory spaces.

The *size* parameter type is `unsigned int`, not `size_t`. This is a small difference on the 16/24–bit model of the DSP563xx between `_memcpy` and `memcpy`.

Returns the copy destination

Example:

```
/* _memcpy for all memory spaces */

_X int a[10];
_Y int b[10];

void main(void)
{
  _memset(a,1,sizeof(a));
  _memcpy(b,a,sizeof(a));
}
```

_memset

```
void * _memset( void * dest, int src,
                  unsigned int size );
```

Fill a block of memory. An optimized inlined version of the library function `memset`, that works on all memory spaces.

The *size* parameter type is `unsigned int`, not `size_t`. This is a small difference on the 16/24–bit model of the DSP563xx between `_memset` and `memset`.

Returns the destination

Example:

```
/* _memset for all memory spaces */

_X int a[10];
_Y int b[10];

_X void * const p = a+5;
_Y void * const q = b+5;

void main(void)
{                          /*   space derived from   */
                           /* ---------------------- */
  _memset(a,1,sizeof(a)); /*  space 'a' resides in  */
  _memset(b,2,sizeof(b)); /*  space 'b' resides in  */
  _memset(p,3,5);         /*  space 'p' points to   */
  _memset(q,4,5);         /*  space 'q' points to   */
}
```

_nop

```
void _nop( void );
```

Generate NOP instructions.

Returns nothing.

Example:

```
_nop();
... Code ...
      opt  noopnop
      nop
      opt  opnop
```

_pdiv

```
_fract _pdiv( _fract op1, _fract op2 );
```

Calculate the division of *op1*/*op2* for positive fractional operands. Generate DIV instruction.

Returns the result.

Example:

```
volatile _fract fa, fb;
fa = _pdiv( fa, fb );
```

... Code ...

```
     move x:ss_main+12,b     ; fa
     move x:ss_main+11,x0    ; fb
     andi #$FE,ccr
     rep  #24
     div  x0,b               ; _pdiv(fa, fb)
     move b,x:ss_main+12     ; fa
```

_pflush

```
void _pflush( void );
```

Use the PFLUSH instruction to flush the instruction cache.

Returns nothing.

Example:

```
_pflush();
```

... Code ...

```
     pflush
```

This function only generates code for the DSP563xx. For more information about instruction cache support, see section 3.16 *DSP563xx Cache Support*.

_pflushun

```
void _pflushun( void );
```

Use the PFLUSHUN instruction to flush unlocked instruction cache sectors.

Returns nothing.

Example:

```
_pflushun();
```

... Code ...

```
     pflushun
```

This function only generates code for the DSP563xx. For more information about instruction cache support, see section 3.16 *DSP563xx Cache Support*.

_pfree

```
void _pfree( void );
```

Use the PFREE instruction to unlock all the locked cache sectors in the instruction cache.

Returns nothing.

Example:

```
_pfree();

... Code ...
      pfree
```

This function only generates code for the DSP563xx. For more information about instruction cache support, see section 3.16 *DSP563xx Cache Support*.

_plock

```
void _plock( int _P *addr );
```

This function uses the PLOCK instruction to lock one sector in the instruction cache. The argument of this function is the sector address to be locked.

Returns nothing.

Example:

```
void foo( void )
{
     void _cache_region creg(void);

     _plock( _cache_get_start( creg ) );
          /* loop fits in one sector */
#pragma cache_align_now
#pragma cache_region_start creg
     for ( ... )
     {
          ...
     }
#pragma cache_region_end creg
     _punlock( _cache_get_start( creg ) )
```

... Code ...

```
     move    #Fcreg,r6
     nop
     plock   (r6)
     align   cache
     ...
     move    #Fcreg,r6
     nop
     punlock (r6)
```

This function only generates code for the DSP563xx. For more information about instruction cache support, see section 3.16 *DSP563xx Cache Support*.

_punlock

```
void _punlock( int _P * addr );
```

This function uses the PUNLOCK instruction to unlock one sector in cache. The argument of this function is the sector address to be unlocked.

Returns nothing.

Example:

See example with **_plock** function.

This function only generates code for the DSP563xx. For more information about instruction cache support, see section 3.16 *DSP563xx Cache Support*.

_rol

```
unsigned int _rol( unsigned int operand,
                   unsigned int count );
```

Use the ROL instruction to rotate (left) *operand count* times. The carry bit is reset before rotation.

Returns the result.

Example:

```
volatile unsigned int uia, uib;
/* rotate left, using int variable */
uia = _rol( uia, uib );
```

... Code ...

```
     move x:ss_main+22,b     ; uia
     move x:ss_main+21,a     ; uib
     tst  a                  ; uib
     jeq  L3
     rep  a1
     rol  b                  ; _rol(uia, uib)
L3:  move b1,x:ss_main+22    ; uia
```

_ror

```
unsigned int _ror( unsigned int operand,
                   unsigned int count );
```

Use the ROR instruction to rotate (right) *operand count* times. The carry bit is reset before rotation.

Returns the result.

Example:

```
volatile unsigned int uia, uib;
/* rotate right, using constant */
uia = _ror( uib, 2 )
uia = _ror( uia, 3 );
```

... Code ...

```
     move x:ss_main+21,b     ; uib
     ror  b                  ; _ror(uib, 2)
     ror  b
     move b1,x:ss_main+22    ; uia

     move x:ss_main+22,b     ; uia
     rep  #3
     ror  b                  ; _ror(uia, 3)
     move b1,x:ss_main+22    ; uia
```

_round

```
_fract _round( long _fract operand );
```

Round the fractional operand. Generate RND instruction.

Returns the result.

Example:

```
volatile _fract fa, fb;
volatile long _fract lfa, lfb;

void main( void )
{
     fa = _round(fa);
     fb = _round(lfa);
     lfb = _round(lfa);
}
```

... Code ...

```
     move x:ss_main+12,b     ; fa
     rnd  b                  ; _round(fa)
     move b,x:ss_main+12     ; fa
;
     move x:ss_main+10,b
     move x:ss_main+9,b0
     rnd  b                  ; _round(lfa)
     move b,x:ss_main+11     ; fb
;
     move x:ss_main+10,b
     move x:ss_main+9,b0
     rnd  b                  ; _round(lfa)
     move b,x:ss_main+8      ; lfb
```

Semaphore intrinsics

Semaphores are program flags that are used to synchronize processes and to force exclusive access to a resource. Examples are an interrupt function that sets a semaphore when data is read, and a processing loop that clears it when done; or a serial port that is used by several parallel processes and must be line–blocked. To guarantee correctness, semaphore actions must be atomic (non–interruptable). The semaphore intrinsic functions guarantee this without the overhead of blocking and re– enabling interrupts. To do this, the generated code contains a bit clear, bit set or bit test assembly instruction to access the semaphore.

_sema_clr

```
int _sema_clr(volatile int *p, int bitnumber);
```

This function clears a semaphore (a bit in a `volatile int` field pointed to by `p`). '*bitnumber*' must be an integral constant expression and should not exceed the width of an `int` in the DSP program. It can either be used to clear a semaphore (ignoring the result), or to test and clear a semaphore (result contains previous state).

Returns the previous state of the semaphore.

Example:

```
typedef volatile int semaphore_t;

void f(void)
{
        static semaphore_t flag = 1;

        while( !_sema_clr( &flag, 0 ) )
            ; /* wait until we reset flag ourselves */
        /* code to handle device */
        _sema_set( &flag, 0 );
        /* free device for other processes */
}
```

```
... Code ...
L3:     dc      $000001
        org     p,".ptext":
L4:     bclr    #0,x:L3         ; _sema_clr
        jcc     L4
        bset    #0,x:L3         ; _sema_set
        rts
```

_sema_set

```
int _sema_set(volatile int *p, int bitnumber);
```

This function sets a semaphore (a bit in a `volatile int` field pointed to by `p`). '*bitnumber*' must be an integral constant expression and should not exceed the width of an `int` in the DSP program. It can either be used to set a semaphore (ignoring the result), or to test and set a semaphore (result contains previous state).

Returns the previous state of the semaphore.

Example:

```
typedef volatile int semaphore_t;

void f(void)
{
        static semaphore_t flag = 0;

        while( _sema_set( &flag, 0 ) )
             ; /* wait until we set flag ourselves */
        /* code to handle device */
        _sema_clr( &flag, 0 );
        /* free device for other processes */
}

... Code ...
L3:     dc      $000001
        org     p,".ptext":
L4:     bset    #0,x:L3         ; _sema_set
        jcs     L4
        bclr    #0,x:L3         ; _sema_clr
        rts
```

_sema_tst

```
int _sema_tst(volatile int *p, int bitnumber);
```

This function tests a semaphore (a bit in a `volatile int` field pointed to by `p`). It does not change the state of the semaphore. '*bitnumber*' must be an integral constant expression and should not exceed the width of an `int` in the DSP program.

Returns the current state of the semaphore.

Example:

```
typedef volatile int semaphore_t;
semaphore_t dev_started;    /* set by other routines */

void f(void)
{
        while( ! _sema_tst( &dev_started, 15 ) )
             ; /* wait until device is operational */
        /* code using device */
}
```

... Code ...

```
        org     p,".ptext":
L3:     btst    #15,x:Fdev_started     ; _sema_tst
        jcc     L3
        rts
```

_stop

```
void _stop( void );
```

Generate STOP instruction.

Returns nothing.

Example:

```
_stop();
```

... Code ...

```
     stop
```

_strcmp

```
int _strcmp( const char * op1, const char * op2 );
```

Perform a string compare. An optimized inlined version of the library function `strcmp`, that works on all memory spaces.

Returns <0 if *op1* < *op2*
0 if *op1* == *op2*
>0 if *op1* > *op2*

Example:

```
char * sa="strna";
char * sb="strnb";

main()
{
     int result;
     result = _strcmp( sa, sb );
}
```

... Code ...

```
        move    x:Fsa,r6
        move    x:Fsb,r5
        move    x:(r5)+,y0
L5:     move    x:(r6)+,a
        sub     y0,a
        jne     L6
        add     y0,a    x:(r5)+,y0
        jne     L5
L6:
```

_strcpy

```
char * _strcpy( char * op1, const char * op2 );
```

Perform a string copy. An optimized inlined version of the library function `strcpy`, that works on all memory spaces.

Returns *op1*

Example:

```
char * sa=”strna”;
char * sb=”strnb”;

main()
{
     _strcpy( sa, sb );
}
```

... Code ...

```
        move    x:Fsa,r6
        move    x:Fsb,r5
L5:     move    x:(r5)+,a
        tst     a       a,x:(r6)+
        jne     L5
```

_strlen

```
unsigned int _strlen( const char * op1 );
```

Calculate the length of a string. An optimized inlined version of the library function `strlen`, that works on all memory spaces.

Returns the length of the string.

The result type is `unsigned int`, not `size_t`. This is a small difference on the 16/24–bit model of the DSP563xx between `_strlen` and `strlen`.

Example:

```
char * sa=”strna”;

main()
{
     int length;
     length = _strlen( sa );
}
```

... Code ...

```
        move    x:Fsa,r5
        move    x:(r5)+,b
        lua     (r5)+,n5
L4:     tst     b       x:(r5)+,b
        jne     L4
        lua     (r5)-n5,b
```

_swi

```
void _swi( void );
```

Software interrupt. Generate SWI (**c56**) or TRAP (**c563**) instruction.

Returns nothing.

Example:

```
_swi();

... Code ...
     swi
```

_vsl

```
void _vsl( long op1, int op2,
           long _L * op3 );
```

Perform a Viterbi shift left operation. Generates a VSL assembler instruction.

Returns Nothing. The high part of op1 is stored in _X:op3, the low part of op1 is shifted left one bit and the value of op2 (0 or 1) is added to it. and stored in _Y:op3.

Example:

```
long _L result;
long input = 0x12345678;

main()
{
     _vsl(input, 0, &result);
     _vsl(input, 1, &result);
}

... Code ...
     vsl  b,#0,l:Fresult
     vsl  b,#1,l:Fresult
```

This function only generates code for the DSP563xx and DSP566xx. On the DSP563xx a warning is generated because the opcode is not supported on all silicon revisions.

_wait

```
void _wait( void );
```

Generate WAIT instruction.

Returns nothing.

Example:

```
_wait();

... Code ...
      wait
```

3.14 INTERRUPTS

DSP56xxx C introduces two new reserved words: `_long_interrupt` and `_fast_interrupt`, 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. For example, in:

```
void _long_interrupt( vector )
l_isr(void)
{ ... }

void _fast_interrupt( vector )
f_isr( void )
{ ... }
```

The compiler will generate an interrupt service frame for long interrupts and no frame in case of fast interrupts. The vector specifies the interrupt number of a two word interrupt vector area. Some interrupts are reserved and handled or used by the compiler (run–time) library, like:

- Hardware RESET (used for C main())
- Stack Error a default handling

The interrupt vector –1 is reserved for a so–called symbolic interrupt. This means that **c563** does not assign an interrupt number to this C function. Symbolic interrupts can only be used with `_long_interrupt`.

When the `_fast_interrupt` modifier is used:

- The compiler will not use instructions that modify the status register
- The compiler will not use any register
- The compiler may generate at most 2 words coding; this is checked after optimization in the assembler.
- If the compiler detects that it is impossible to generate a fast interrupt function, it generates a long interrupt function. In this case the compiler gives a warning message. All other errors will only be detected in the assembly phase.

Example of _fast_interrupt:

Suppose, you want an interrupt function for a peripheral, and the vector number is 17:

```
#include <reg56302.h>

int c;

void
_fast_interrupt( 17 )
host_transmit(void)
{
     HTX = c;
}
```

This will result in assembly:

```
     org     p,".irq17":$22+R_VBAVALUE
irq17:
Fhost_transmit:
     movep   x:Fc,x:<<$FFFFC7
     align   2
```

3.15 CIRCULAR BUFFERS

The DSP56xxx family supports circular buffers, for which no representation in C exists. A circular buffer is a linear array that can be accessed using modulo address arithmetic, i.e., a pointer that wraps–around automatically, thus creating a virtual circular buffer. To allow you to use circular buffers in C, **c563** supports the data type `_circ` as an extended data type.

Example:

```
int _circ Circ_Buf[5];
int _circ *Ptr_to_Circ_Buf = Circ_Buf;
```

Here, `Circ_Buf` is declared as a circular buffer. The compiler will emit alignment directives to ensure circular buffers will start at addresses that are a multiple of the smallest power of two that is equal to or larger than the buffer size. The buffer size is kept by the compiler and will be used to control pointer arithmetic of pointers that are assigned to the buffer later.

In the above example, the circular pointer `Ptr_to_circ_Buf` will be stored in an R–type register and the proper modulo value will be stored in its corresponding M–type register. Operations on the circular pointers can be done using the usual C pointer arithmetic with the difference that the pointer will wrap.

When the circular buffer is accessed using a circular pointer, it will wrap at the buffer limits.

Example:

```
while( *Ptr_to_Circ_Buf++ );
```

Indexing in the circular buffer, using an integer index, is treated equally to indexing in a non–circular array.

Example:

```
int i = Circ_Buf[3];
```

The index is not calculated modulo; indexing outside the array boundaries yields undefined results.

Example:

```
_fract SomeVariable;
_fract _circ input_buf[100];

void func(void)
{
_fract _circ *inPtr = input_buf;

        .
        .
        .
  *inPtr++ = SomeVariable;
        .
        .
        .
}
```

the following code will be used:

```
Ffunc:
     .
     .
     .
     move #Finput_buf,r3      ; load buffer address
                              ; in pointer
     move #100–1,m3           ; load modulo value
     move x:FSomeVariable,x0  ; get value of
                              ; SomeVariable
     move x0,x:(r3)+          ; store value at
                              ; buffer location  and
                              ; increment pointer
      .                       ; more actions with
      .                       ; inPtr
     move m7,m3               ; disable modulo of m3
                              ; (m7 always contains
                              ;  –1 in the mixed or
                              ;  reentrant model)
     .
     .
```

The moves to and from `input_buf` will be optimized away mostly. Also setting the modulo register will be done only once, unless register R3 is used for another purpose that requires other or no modulo arithmetic.

3.16 DSP563XX CACHE SUPPORT

The DSP563xx is equipped with an instruction cache. The compiler supports this cache with the features described in the following sections.

The **c56** compiler and the **c563** compiler in DSP566xx memory model ignore the cache features, which makes it possible to write portable code.

3.16.1 CACHE ALIGNMENT

Instructions are cached in sectors of 128 or 256 words, depending on the cache size. The size of the cache varies by DSP563xx derivative.

The following pragma is supported for forcing alignment on a 128 or 256–word boundary:

#pragma cache_align_now

Aligns current address at a cache boundary. This can be done only once per function. The alignment may introduce an unused memory alignment gap at the beginning of the section.

Whether the cache alignment is on a 128 or 256–word boundary depends on the compiler command line option **–c***size* or the `#pragma cache_sector_size` *size*. The *size* is either 128 or 256 words. The compiler has a predefined macro for the cache sector size `_CACHE_SECTOR_SIZE` that expands to *size* value.

3.16.2 CACHE REGIONS

Cache regions are a start and end address of memory to be locked in the cache. Such a region can be bound to a function or function pointer. The user can lock and unlock the cache sectors in a region using intrinsic functions. The function or function pointer must have the function qualifier `_cache_region`. For a function with this function qualifier the compiler will generate a start and an end label. Functions having this qualifier are always aligned on the cache sector size. A cache function definition will look as follows:

```
void _cache_region sample( void )
{
     ...
}
```

When a function pointer with the `_cache_region` qualifier is defined, the begin and end labels can be set manually. For this purpose two pragmas are available:

#pragma cache_region_start *fptr*
Mark start position of a cache region.

#pragma cache_region_end *fptr*
Mark end position of a cache region.

The *fptr* can be defined as follows:

```
void _cache_region fptr( void );
```

See section 3.16.4 for examples.

3.16.3 CACHE INTRINSIC FUNCTIONS

The following intrinsic functions are available for cache locking:

void _pflush(void); flush cache

void _pflushun(void); flush unlocked sectors

void _pfree(void); global unlock

void _plock(int _P **addr*);
lock 1 sector in cache with address *addr*

void _punlock(int _P **addr*);
unlock 1 cache sector with the address *addr*

int _P * _cache_get_start(void _cache_region * *fptr*) ());
get first address of cache region belonging to *fptr*

int _P * _cache_get_end(void _cache_region * *fptr*) ());
get last address of cache region belonging to *fptr*

3.16.4 EXAMPLES

Example 1:

Locking a loop in the cache:

```
void foo( void )
{
     void _cache_region creg( void );

     _plock( _cache_get_start( creg ) );
     /* loop fits in one sector */
#pragma cache_align_now
#pragma cache_region_start creg
     for ( ... )
     {
               ...
     }
#pragma cache_region_end creg
     _punlock( _cache_get_start( creg ) );
}
```

This example assumes that the loop fits in one cache sector. Therefore, only the start address is locked in the cache.

Example 2:

This example shows how to lock a function from another module. This example does not assume that the whole function fits in one cache sector.

Module A:

```
void _cache_region sample( void )
{
     ...
}
```

Module B:

```
extern void _cache_region sample( void );

void main( void )
{
     void _P *p;

     for ( p = _cache_get_start( sample );
           p < _cache_get_end( sample );
           p += _CACHE_SECTOR_SIZE )
     {
          _plock( p );
     }
     . . .
     for ( p = _cache_get_start( sample );
           p < _cache_get_end( sample );
           p += _CACHE_SECTOR_SIZE )
     {
          _punlock( p );
     }
}
```

3.17 PATRIOT BANK SWITCHING SUPPORT

The banked program memory feature of the Patriot chips (DSP56622 / DSP56671 / DSP56679 / DSP56690 / DSP56691 / DSP56694) is supported with the `_bank()` keyword. A bank is combination of a particular address range and a page number.

_bank(*range***,** *page***)**

range a number from 0 to 3, corresponding to the following available address ranges:

range#	address range
0	0X8000–0X9FFF
1	0XA000–0XBFFF
2	0XC000–0XDFFF
3	0XE000–0XEFFF

Table 3–7: Address ranges

page a number of 0 or 7, corresponding to the page a function should be located in.

Use the command line option **–p***page* to specify the total number of pages (*page* is 1..8). Default is two pages.

A function cannot call functions in the same range but within a different page number. For function calls to functions in the same range and page or in main memory (address 0x000 – 0x7FFF) no extra instructions for switching pages are generated. When the called function is in a different range, switch instructions will be generated. When a function switches pages it saves the value of the DDM Page Configuration Register (DPCR) at the start of the calling function and restores it at the function end.

When you define a function with the `_bank` keyword, the compiler generates an special section name. The section name begins with ".ptext_", followed by a letter indicating the range (where A is range 0, B is range 1, etc.) and followed by a number indicating the page.

For the S–record format the banks are coded in the section information. For the other formats the locator places the sections at the appropriate addresses. For page 0 this is as indicated in the table above. Ranges within page 1 will be located at the address plus 0x10000. The CrossView Pro debugger/loader will interpret these addresses and switch to the correct bank when memory in those ranges is accessed. The user can use the same addresses to look at the program memory. Furthermore the PC register the user sees will also reflect the current state of the DPCR register for and if it points to the paged memory.

Restrictions:

- A function can not call functions in the same range but with a different page number.
- Function pointers pointing to banked functions need to be declared within the same bank as the function they point to.

Example

The following code calls function foo() from function bar().
Function foo() is located in range 1, page 0 and function bar() is located in range 3, page 1.

```
void _bank(1,0) foo( void );
void _bank(3,1) bar( void );

void _bank(3,1) bar( void )
{
     foo();
}
```

3.18 PACKED STRINGS

The **c56** and **c563** compilers support packed characters and packed strings. The `_packed` type modifier will be used to define a packed string. The `_packed` modifier can only be applied to characters, character arrays or pointers to character. Incrementing a packed character pointer means incrementing to the next word. Thus, two (**c563** in 16–bit model) or three (**c56**, **c563** in 24–bit model) characters are skipped.

3.18.1 LIBRARY FUNCTIONS

To be able to access packed strings, the C library has been extended with functions to deal with packed strings:

char *_unpackstr(char * *unp*, const _packed char * *p*);
Unpack string pointed to by *p* in the buffer pointed to by *unp*. Return a pointer to the unpacked string.

_packed char *_packstr(_packed char * *p*, const char * *unp*);
Pack string pointed to by *unp* in the buffer pointed to by *p*. Return a pointer to the packed string.

size_t _unpstrlen(const _packed char **p*);
Return the length in number of characters of the packed string pointed to by *p*. This is the number of characters when the string would be unpacked.

size_t _packsize(const char * p);
Calculate the size of a string when it is packed.

char _pstr_get(const _packed char *p, size_t *idx*);
Return character at index *idx* in packed string *p*.

void _pstr_put(_packed char *p, size_t *idx*, char *c*);
Put character *c* on index *idx* in packed string *p*.

Printf and scanf formatters support the packed string conversion specification %S, which otherwise behaves like the normal string conversion specification %s.

3.18.2 PRAGMAS

Two pragmas for packed string support are available:

#pragma pack_strings After this pragma all string constants will be packed string constants. Using such a string as a not packed string (e.g., passing to a function with a not packed string argument type) will yield a type conflict error.

#pragma nopack_strings
After this pragma string constants will no longer be packed string constants.

3.18.3 EXAMPLES

Example 1:

```
_packed char somestring[] = ”Some String”;

void example( void )
{
      // allocate space for unpacking
      char *p = (char *)malloc( _unpstrlen( somestring) + 1) );
      _unpackstr( p, somestring );
      printf( ”unpacked %s, packed %S\n”, p, somestring );
      free( p );
}
```

Example 2:

```
int printpstring( _packed char *p )
{
     int idx = 0;
     char c;
     while( c = _pstr_get( p, idx++ ) )
          putchar( c );
     return( idx );
}

void main( void )
{
     int parm;
#pragma pack_strings      // make packed strings
                          // of all string constants
     printpstring( "Continue? (Y/N): " );
     parm = getchar();
     printpstring( "Action: " );
     printpstring( parm == 'Y' ? "Continuing" :
"Stopping" );
#pragma nopack_strings
}
```

The `sizeof( _packed char )` is one, just like `sizeof( char )`. This implies that pointer arithmetic on pointers to packed strings goes per three characters.

Example 3:

```
_packed char *p = "123456789";
char         *n = "123456789";
  . . .
char c;
c = _pstr_get( p, 1 );    // c will be '1'
p++;
c = _pstr_get( p, 1 );    // c will be '4'
c = *n;                   // c will be '1'
n++;
c = *n;                   // c will be '2'
```

3.19 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 {
      _X int i;     /* referring to storage: not correct */
      _P char *p;   /* used to specify target memory: correct */
};
```

In the example above **c56** ignores the erroneous _X storage specifier (without displaying a warning message).

3.20 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 _P int PINT;          /* storage type _P: OK */
typedef int _X *DATAPTR;      /* logical type _X
                                 storage type 'default' */
```

3.21 SWITCH STATEMENT

c563 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 JMP instructions for each possible switch value. The switch argument is used as an index to jump within this table.

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 an option (or a pragma) cannot overrule the restrictions as described earlier.

See `#pragma jumptable_memory` in section 4.4, *Pragmas*.

3.22 PORTABLE C CODE

If you are developing C code for the DSP56xxx family using **c563**(or **c56**), 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 `c56.h`. This header file checks if `_C56` is defined, 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 DSP56xxx C intrinsic 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 DSP56xxx and link these functions with your application for simulation purposes.

3.23 EFFICIENT USE OF THE DSP56XXX TOOL SET

The following sections give you some guidelines and hints to get the best results from the DSP56xxx tool set.

3.23.1 CHAR AND SHORT TYPES

Avoid types smaller than `int` whenever possible, as they need more instructions for conversions and do not save data space. Types smaller than `int` are (see table 3–2):

DSP5600x `signed` and `unsigned char` and `short`

DSP563xx in 24–bit arithmetic mode: `signed` and `unsigned char` and `short`

in 16–bit arithmetic modes: `signed` and `unsigned char`

DSP566xx `signed` and `unsigned char`

3.23.2 UNSIGNED

Try to use the `unsigned` qualifier as little as possible, because unsigned comparisons require more code than signed comparisons. When retrieving unsigned values from memory the accumulator extension word can be set incorrectly. Therefore, the extension word has to be cleared before any comparison.

3.23.3 HARDWARE LOOPS

For the DSP5600x the loop counter register LC (16 bits) is smaller than an (`unsigned`) `int` (24 bits). The compiler only optimizes to a hardware DO loop when it is absolutely sure that the loop counter fits in the loop counter register. When the start and end value of the loop iteration counter in the C code is fixed and fits within a 16–bit value this is the case. Also when an (`unsigned`) `short` is used for loop iteration counter or loop conditions, the compiler is sure the loop counter register will not overflow. Therefore, it is recommended to use (`unsigned`) `short` for loops and for loop conditions on the DSP5600x.

When using global variables inside the loop the compiler may not be able to produce a hardware DO loop that can be optimized to a REP loop. For example:

```
_fract output=0.0;
_fract _X x[10];
_fract _Y c[10];
main()
{
     int i;
     for(i=0; i < 10; i++)
         output += x[i]*c[i];
}
```

In this example the loop cannot be optimized to a REP loop because the variable `output` is written in each iteration. You can use an automatic temporary variable to make it possible to optimize the hardware loop to a REP loop:

```
_fract output=0.0;
_fract _X x[10];
_fract _Y c[10];
main()
{
     int i;
     _fract tmp = output;   // copy to temporary variable
     for(i=0; i < 10; i++)
         tmp += x[i]*c[i];  // use temporary variable
     output = tmp;   // get value from temporary variable
}
```

To generate a hardware DO loop the compiler must recognize the variable used as loop counter in the C code. Incrementing the loop counter as a separate statement is easier to recognize by the compiler than when it is used in some expression or as an index of an array.

For example, in the following code the compiler does not recognize the loop counter `i`:

```
_fract output=0.0;
_fract _X x[10];
_fract _Y c[10];
main()
{
     int i=0;
     _fract tmp = output;
     do
     {
          tmp += x[0]*c[i++];  // i++ in expression
     } while(i<10);
     output = tmp;
}
```

When rewriting the do–while loop code to:

```
     do
     {
          tmp += x[0]*c[i];
          i++;          // i++ as separate statement
     } while(i<10);
```

the compiler recognizes `i` as the loop counter and generates a hardware loop.

In for statements containing multiple update statements, the loop count variable must be the last updated variable to allow the compiler to create a hardware loop. Example:

```
void f(unsigned short max)
{
     unsigned short i, j;
     for(i = 0, j = 0;  i < max;  i++, j++)
     {
          a[i] = b[j];
     }     /* no hardware loop created */
     for(i = 0, j = 0;  i < max;  j++, i++)
     {
          a[i] = b[j];
     }     /* hardware loop created */
}
```

3.23.4 SPEED VS. SIZE

The C compiler optimizations are by default tuned for code size. When execution speed is more important than code size you can use the **–O3** or **–O4** command line option to let the compiler optimize for speed. The optimization options can also be changed using the **#pragma optimize**.

For example:

```
#pragma optimize 4    // optimize for speed
     . . .            // some C code to be
     . . .            // optimized for speed
#pragma endoptimize   // back to optimization set
                      // before the pragma optimize
```

3.23.5 ASSEMBLY INTERFACING

The **#pragma asm** can be used to get access to specific DSP56xxx instructions or to write code that cannot be achieved using the C language. For interfacing the C language to assembly it is recommended to use a function call interface with the `_asmfunc` function qualifier. With **#pragma asm** it is unsafe to rely on the registers the compiler allocates for the variables. See section 3.12 *Inline Assembly* and section 3.12.4 *Linking with Separate Assembly Routines* for more information.

The compiler also features intrinsic functions that can be used to get access to some special DSP56xxx instructions. These functions have the advantage of being portable and they have a neat C interface. See also section 3.13 *Intrinsic Functions*.

When calling an assembly routine from C in the mixed or reentrant model, R7 must not be used in this function because this register is used as user stack pointer. When no more address registers are available in your assembly routine you can save the contents of R7 to some memory location on entry of the routine and restore it before returning to C. Note that interrupt functions in the reentrant model or interrupt functions declared `_reentrant` in the mixed model cannot be entered when R7 is not available as user stack pointer because these interrupt functions may want to save registers on the stack.

3.23.6 SELECTING THE MOST EFFICIENT MODEL

The DSP563xx compiler can generate code for 16–bit precision arithmetic in two models: the 16–bit model and the 16/24–bit model. Use the 16/24–bit model only if the code and/or data require more than 64k words of memory, because pointer arithmetic is much more efficient in the 16–bit model.

The DSP5600x compiler supports three models: Static, Mixed and Reentrant. Select the most efficient model for your application. The compiler generates the best code for the static model. The code density generated for the mixed model is close to the static model because it only cannot use R7 which is reserved for user stack pointer. The code density for the reentrant model is less than the static and mixed model. Use the static model for any function that does not need to be reentrant (i.e., not recursive and not called from (an) interrupt routine(s) and the main program simultaneously).

If you place the stack in L–memory, code becomes faster, but data memory use may slightly increase. If you select default memory to be P, the code will become very inefficient and slow; use this selection only as a last resort.

See also section 3.2.2 *Memory Models*.

3.23.7 MEMORY MAPPED I/O FROM C

Use the unions in the supplied header file for memory mapped I/O; optimal bit field operations are generated this way. A C header file is supplied for each DSP56xxx family derivative in the `include` subdirectory of the installed product.

The registers defined in the header file are created by a union of a structure (field `.B`) and an integer (field `.I`). The structure defines all bit fields in the register. The integer can be used to access the register as one word. When copying the full register contents it is recommended to use this one word `.I` field instead of copying the whole structure. Copying one field yields more efficient code than copying it as a structure.

Example:

```
#include <reg56002.h>      // include register
                           // header file
void main( void )
{
     sr_type a;
     a = SSISR;            // copy whole structure
     a.I = SSISR.I;        // preferred: copy .I field
}
```

The best place to add your own I/O devices in the DSP56xxx memory map is in the high addresses of Y memory. You can then access these I/O devices with the same efficient instructions as the built–in I/O devices. To access them from C code you can create a header file for your I/O system that is similar to the ones provided for the DSP device.

3.23.8 PARALLEL MOVES

For the DSP5600x and DSP563xx it is recommended to allocate data structures in Y memory whenever possible, to take advantage of parallel X/Y moves. Automatics are stored in X memory, so, this gives you a better balance.

3.23.9 SHIFTING FRACTIONAL DATA

The DSP56xxx compilers support shifting of fractional data values. You may want to use the method below, however, because this feature is not portable to floating point calculations. You can use multiplication/division with fractional powers of 2 to implement shifting of fractional data. For example:

```
_fract scale_down( _fract f )
{
     return ( f*0.25 )  /* f >>= 2  */
}
```

... Code ...

```
asr  a
asr  a
rts
```

```
_fract scale_up( _fract f )
{
     return ( f/0.125 )  /* f <<= 3  */
}
```

... Code ...

```
rep  #3
asl  a
rts
```

3.23.10 DYNAMIC SCALING

To scale dynamically, you can use a negative `_fract` scaling factor to avoid rounding errors at gain 1:

```
output = – ( input*gain ); /* –1.0 <= gain <= 0 */
```

... Code ...

```
move  x:Finput,x1
move  x:Fgain,y0
mpyr  –x1,y0,a
move  a,x:Foutput
```

3.23.11 REVIEWING THE OPTIMIZED CODE

The optimizations for a C program are partially done in the C compiler and partially in the assembler. This implies that the fully optimized result is only available after assembly. The assembler will produce a list file that shows how it rearranged the compiler generated source. The final result can also be reviewed using the disassembler option of the object reader **pr563** (**pr56** for DSP5600x). For example:

cc563 –c file.c	compile into object 'file.obj'
pr563 –iv4 file.obj	disassemble the object 'file.obj'

or

cc563 –c –l –s file.c	compile and produce an assembler listing with source merging

3.23.12 INTEGER AND FRACTIONAL TYPES

Converting integer to fractional types follows the rules for integer to float conversion. Due to the limited overlap between integer and fractional types there are only a few possibilities, listed in the following table. As these conversions are almost always inadvertent, a warning message is issued for them.

In combined integer and fractional operations the fractional value is converted to integer, as the integer type has the largest range.

From	Value	To	Result
signed char, short, int, long	<= –1	_fract, long _fract	–1.0
	> –1		0.0
unsigned char, short, int, long	>= 0	_fract, long _fract	0.0
_fract, long _fract	<= –1.0	signed char, short, int, long	–1
	> –1.0		0
_fract, long _fract	< = –1.0	unsigned char, short, int, long	max. value (all ones)
	>= –1.0		0

Table 3–8: Type conversions

The fractional value is truncated and not rounded.

Example:

```
int      i;
_fract   f;

i = 3;      f = i;    // f will be 0.0
. . .
i = -5;     f = i;    // f will be -1.0
. . .
f = 0.5;    i = f;    // i will be 0
. . .
f = -0.1;   i = f;    // i will be 0
. . .
f = -1.0;   i = f;    // i will be -1
```

The following code shows the result of a conversion from `_fract` to `int` and vice versa:

```
; 1  |extern int i;
; 2  |extern _fract f;

     . . .

; 6  |     i = f;    // _fract to int

     move x:Ff,a     ; get f into accu a
     neg  a          ; negate to set flags
     move #0,a       ; result in i is 0
     jle  L3         ; if value was >= 0 result in i is 0
     jes  L3         ; or if value was > –1.0 if extension
                     ; not in use  result in i is 0
     move #>–1,a     ; else result in i is –1
L3:  move a,x:Fi     ; store i

; 8  |    f = i;     // int to _fract

     move x:Fi,b     ; get i into accu b
     tst  b          ; check value of i
     move #0,b       ; result in f is 0.0
     jge  L4         ; if i >= 0 result in f is 0.0
     move #<$80,b    ; else result in f is –1.0
L4:  move b,x:Ff     ; store f
```

It is obvious that when integer and fractional types are mixed frequently, the above conversions also occur frequently. Therefore, it is more efficient to use only fractional types or only integer types in expressions. The above of course also applies to the types `char`, `short`, `long` and `long _fract`.

When you do not want the conversion as described above there are three ways to achieve this:

1. Use the `_int2fract` and `_fract2int` intrinsic functions. See section 3.13 *Intrinsic Functions*.
2. Use a `union` of the fractional and integer type. Initialization with hexadecimal constants is also possible, depending on which member is mentioned first.

For example:

```
typedef union
{
     _fract f;
     int i;
}  fract_int;

fract_int val = 0x123456;  // hex initialization

int f( void )
{
     val.f = 0.1;          // put fractional value 0.1
                           // in val.f
     return( val.i );      // return 0xCCCCC as integer
}
```

3. Use a pointer cast.

   For example:

```
int f( void )
{
     _fract f = 0.1;       // put fractional value 0.1
                           // in f
     return( *(int*)&f ); // return 0xCCCCC as integer
}
```

   It is recommended to use fractional types whenever possible. The compiler can generate the most efficient code for these types because this is the type which suites the DSP56xxx family the best.

   For example the compiler will use the MAC instruction whenever possible, also for integer multiplication. To be able to do this the integer must be scaled first.

```
; 1  |extern int i, j, k;

     . . .

; 10 |    k += i * j;

     move x:Fk,a0      ; get k into a0, this must
                       ; be a0 to do the MAC
     move x:Fi,x0      ; get i into x0
     move x:Fj,y0      ; get j into y0
     asl  a            ; prepare for mac on integer:
                       ; scaling
     mac x0,y0,a       ; mac!
     asr  a            ; get result of mac: scaling
     move a0,x:Fk      ; store k
```

When this was done using the fractional type the code would be more efficient:

```
; 1  |extern _fract i,j,k;

     . . .

; 10 |    k += i * j;

     move x:Fk,a       ; get k into accu a
     move x:Fi,x0      ; get i into x0
     move x:Fj,y0      ; get j into y0
     macr x0,y0,a      ; mac!
     move a,x:Fk       ; store k
```

3.23.13 INTERRUPT ROUTINES

Avoid using function calls in interrupt routines, as this will force the compiler to stack all registers. All registers are free to use in a function, so there is no way that the compiler can limit this. Instead, an `_inline` function or a macro can be used for repeated code if necessary. This will cause your code to become larger, but it will also avoid call/return overhead in the interrupt function.

CHAPTER 4

COMPILER USE

TASKING

4 CHAPTER

4.1 CONTROL PROGRAM

The control program **cc563** facilitates the invocation of the various components of the DSP563xx/DSP566xx family toolchain, from a single command line. **cc56** is the control program for the DSP5600x. The control program accepts source files and options on the command line in random order.

The invocation syntax of the control program is:

cc563 [[*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; the remaining options are passed to those programs in the toolchain that accept the option.
- 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 hand–written assembly source files which have to be 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 `.a` suffix are interpreted as library files and are passed to the linker.
- Arguments with a `.obj` suffix are interpreted as object files and are passed to the linker.
- Arguments with a `.cln` suffix are interpreted as COFF object files and are passed to the linker.
- Arguments with a `.clb` suffix are interpreted as COFF library files and are passed to the linker.

- 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.
- An argument with a `.dsc` suffix is interpreted as a locator description file and is passed to the linker and the locator.

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 programs:

Option	Description
–? or none	Display invocation syntax
–Mm	Mixed memory model (only allowed for **cc56**)
–Mr	Reentrant memory model (only allowed for **cc56**)
–Ms	Static memory model (only allowed for **cc56**)
–M24	24–bit memory model (only allowed for **cc563**)
–M1624	16/24–bit memory model (only allowed for **cc563**)
–M16	16–bit memory model (only allowed for **cc563**)
–M6	DSP566xx memory model (only allowed for **cc563**)
–S	Generate Motorola compatible assembly file with COFF debug, stops at `.asm`
–T*name*	target hardware
–V	Display version header only
–Wa*arg*	Pass argument directly to the assembler
–Wc*arg*	Pass argument directly to the C compiler
–Wcp*arg*	Pass argument directly to the C++ compiler
–Wlc*arg*	Pass argument directly to the locator
–Wlk*arg*	Pass argument directly to the linker
–Wpl*arg*	Pass argument directly to the C++ pre–linker
–c++	Force `.c` files to C++ mode
–c	Do not link: stop at `.obj`
–cc	Compile C++ files to `.c` and stop

Option	Description
–cl	Do not locate: stop at `.out`
–clas	Set locator output file format to CLAS compatible
–cs	Do not assemble: compile C and C++ files to `.src` and stop
–f *file*	Read arguments from *file* ("–" denotes standard input)
–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
–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

4.2 COMPILERS

The invocation syntax of the C compilers is:

c563 [[*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 compilers accept 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 compilers generate 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 **c563** 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[*flag*...]	Control language extensions
–C*flag*...	Control compatibility options
–D*macro*[**=***def*]	Define preprocessor *macro*
–E[**m**|**l**|**c**|**i**|**p**|**x**]	Preprocess options or emit dependencies
–H*file*	Include *file* before starting compilation
–I*directory*	Look in *directory* for include files
–L*number*	Specify depth of hardware stack use
–M[**m**|**s**|**r**][**x**|**y**|**l**|**p**][**L**]	Select memory model: mixed, static or reentrant. Select default memory space: X, Y, L or P. Specify stack not in L memory (only for **c56**)

Option	Description
–M[**24**|**1624**|**16**|**6**] [**n**][**x**|**y**|**l**|**p**][**L**]	Select memory model: 24–bit, 16/24–bit, 16–bit or DSP566xx. Do not use hardware stack extension. Select default memory space: X, Y, L or P. Specify stack not in L memory (only for **c563**).
–O*flag*...	Control optimization
–R[*dname*[**=***sname*]]	Change default section name
–U*macro*	Remove preprocessor *macro*
–V	Display version header only
–c*size*	Specify *size* of cache sectors for the DSP563xx
–e	Remove output file if compiler errors occur
–err	Send diagnostics to error list file (`.err`)
–f *file*	Read options from *file*
–g[**f**|**l**|**n**|**c**]...	Enable symbolic debug information (unless **–gn** is used)
–m*mask*	Compile for silicon *mask* (**c563** only)
–n	Send output to standard output
–o *file*	Specify name of output *file*
–p*page*	Specify number of Patriot memory pages
–r*register*	Reserve a register for external use
–s	Merge C source code with assembly output
–si	Merge C source code and included files with assembly output
–t	Display lines/min
–u	Treat all 'char' variables as signed
–w[*num*]	Suppress one or all warning messages
–wstrict	Suppress warning messages 196, 303
–z*pragma*	Identical to '#pragma *pragma*' in the C source

Table 4–2: Compiler options (alphabetical)

Description	Option
Include options	
Read options from *file*	**–f** *file*
Include *file* before starting compilation	**–H***file*
Look in *directory* for include files	**–I***directory*
Preprocess options	
Preprocess options or emit dependencies	**–E**[**m**|**l**|**c**|**i**|**p**|**x**]
Define preprocessor *macro*	**–D***macro*[**=***def*]
Remove preprocessor *macro*	**–U***macro*
Allocation control options	
Specify depth of hardware stack use	**–L***number*
Change default section name	**–R**[*dname*[**=***sname*]]
Specify *size* of cache sectors for the DSP563xx	**–c***size*
Reserve a register for external use	**–r***register*
Code generation options	
Select memory model: mixed, static or reentrant. Select default memory space: X, Y or P. Specify stack not in L memory (only for **c56**)	**–M**[**m**|**s**|**r**][**x**|**y**|**l**|**p**][**L**]
Select memory model: 24–bit, 16/24–bit, 16–bit or DSP566xx. Do not use hardware stack extension. Select default memory space: X, Y, L or P. Specify stack not in L memory (only for **c563**).	**–M**[**24**|**1624**|**16**|**6**] [**n**][**x**|**y**|**l**|**p**][**L**]
Control optimization	**–O***flag...*
Compile for silicon *mask*	**–m***mask*
Specify number of Patriot memory pages	**–p***page*
Identical to '#pragma *pragma*' in the C source	**–z***pragma*
Language control options	
Control language extensions	**–A**[*flag...*]
Control compatibility options	**–C***flag...*
Treat all 'char' variables as signed	**–u**
Output file options	
Remove output file if compiler errors occur	**–e**
Send output to standard output	**–n**

Description	Option
Specify name of output *file*	**–o** *file*
Merge C source code with assembly output	**–s**
Merge C source code and included files with assembly output	**–si**
Diagnostic options	
Display invocation syntax	**–?**
Display version header only	**–V**
Send diagnostics to error list file (`.err`)	**–err**
Enable symbolic debug information (unless **–gn** is used)	**–g**[**f**|**l**|**n**|**c**]...
Display lines/min	**–t**
Suppress one or all warning messages	**–w**[*num*]
Suppress warning messages 196, 303	**–wstrict**

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.

With options that can be set from within EDE, you will find a mouse icon that describes the corresponding action.

–?

Option:

–?

Description:

Display an explanation of options at stdout.

Example:

```
c563  –?
```

-A

Option:

Select the `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Language Extensions`. Select the `Advanced language extensions` radio button and enable one or more language exentensions.

–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 **–ACDEFKLOPRSTV** 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:

c 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"; }
```

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 '`BSC 1`' for '`const char i[1];`'.

e Default. Relaxed `char`/`short` arithmetic overflow masking. Do not force conversions on small types to prevent overflows.

E Select forced conversions on small types. Normally, the excess bits are not cleared or extended on a cast from a `char` or a `short` to an `int` or a `long`. For `signed` values, the ANSI specification does not prescribe such behavior, for `unsigned` values it requires overflow masking. By default the compiler avoids the overhead involved by code to prevent overflows, but this flag forces it to insert this code. Example:

```
short a_short;
long  a_long;

void f(void)
{
    a_long  = 0x123456;
    a_short = a_long;
}
```

Generated code with **-Ae** (default):

```
move      l:Fa_long,b
move      b0,x:Fa_short
```

Generated code with **-AE**:

```
move      l:Fa_long,b
move      b0,b
extract   #$10018,b,b
move      b0,b
move      b0,x:Fa_short
```

f Default. A constant in the range `[-1.0,1.0>` has type `_fract`.

F A constant in the range `[-1.0,1.0>` has type `float`.

k Default. Allow keyword language extensions such as `_fract`, `_X` and `_near`.

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.

o Default. Allow language extension keyword `_circ`.

O Disable language extension keyword `_circ`. This removes the handling of m–registers from the scope of the compiler. When the `_circ` keyword is not allowed, the compiler does not need to save the modifier registers in an interrupt routine, and set them to the linear mode. In applications that do not need circular pointers this will decrease the interrupt latency. Of course, the application may not use the modifier registers on the assembly level either, or the interrupt handling will fail. Example:

```
int a, *p;

void _long_interrupt(28) irq28(void)
{
        a = *p++;
}

/* code contains moves for m–register without –AO*/
```

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.

r Default. Enable the `restrict` keyword.

R Disable the `restrict` keyword.

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.

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.

0 – same as **–ACDEFKLOPRSTV** (disable all, strict ANSI–C)

1 – same as **–Acdefkloprstv** (default, enable all)

Example:

To disable C++ comments enter:

```
c563  -AP test.c
```

-C

Option:

Select the `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Output`. Enable or disable the `Generate Motorola compatible assembly` check box.

Select the `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Code Generation`. Enable or disable the following check boxes:

- `User stack pointer points to first free location`
- `All functions with _compatible calling convention`
- `Use R6 as user stack pointer`

–C[*flags*]

Arguments:

One or more compatibility option flags.

Default:

–CACGRS

Description:

You can use the **–C** command line option to switch global compatibility options. **–C** without any flags specified enables all compatibility options. This is the same as specifying **–Cacgrs** or **–C1**.

Flags which are controlled by a letter, can be switched on with the lower case letter and switched off with the uppercase letter. The following flags are allowed:

a Motorola assembler compatible output. The compiler will use conditional assembly to generate assembly sources that are compatible with the Motorola assembler while retaining all features of the TASKING toolchain. The output will be a little longer and somewhat harder to read. Use this option and the **–S** option of the TASKING assembler together with assembler optimization options to produce an optimized Motorola compatible assembler file.

A Default. TASKING assembler output.

c All functions are generated and called with Motorola compatible calling convention, including external functions. This also means that any libraries linked in (including the C library) must have been translated with this option, otherwise conflicts in the passing of parameters will result. In most cases using the `_compatible` qualifier for only those functions requiring it will be more convenient and gives more efficient compilation results.

C Default. TASKING calling convention.

g Create COFF style debug information.

G Default. IEEE–695 style debug information.

r User stack pointer register is R6 instead of R7. This option is required to link objects created by the Motorola C compiler.

R Default. Register R7 is used as the user stack pointer.

s Stack pointer points to the next item above the top–of–stack item (first unused location), as used by Motorola and by TASKING compilers prior to version 2.2r1.

S Default. Stack pointer points to the top–of–stack item (last used location).

0 – same as **–CACGRS** (disable all, default)

1 – same as **–Cacgrs** (all compatibility options)

Example:

To enable all compatibility options, enter:

```
c563  -C1 test.c
```

-c (c563 only)

Option:

–c*size*

Pragma:

cache_sector_size

Arguments:

The cache sector size of the used DSP563xx derivative. Only the values 128 and 256 are allowed.

Default:

–c128

Description:

Specify the size of the cache sectors for the used DSP563xx derivative. This size is used for the alignments generated for the pragma **cache_align_now** and for the `_cache_region` qualifier. The default cache size is 128.

Example:

To set the cache size to 256 enter:

```
c563  -c256 test.c
```

Section 3.16 *DSP563xx Cache Support*.

–D

Option:

Select the `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Preprocessing`. Define a macro (syntax: *macro*[**=***def*]) in the `Define user macros` field. 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`.

```
c563 –DNORAM –DPI=3.1416 test.c
```

–U

–E

Option:

Select the `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Preprocessing`.
Enable the `Preprocess only and capture output` check box.

–E[m|l|c|i|p|x]

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.

With the **–Em** option, the compiler generates dependency rules which can be used by a 'make' utility.

With the **–El** 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.

With the **–Ec** option, comments in the c file are preserved.

With the **–Ei** option, include libraries are not included in the c file while the #include statement remains present.

With the **–Ep** option, no #line numbers are added to the c file.

With the **–Ex** option, macros are not expanded while the macro definition and calls to macros remain present in the c file.

Examples:

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

```
c563 –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`.

```
c563 –El test.c –o multi
```

The following command generates dependency rules for the file `test.c` which can be used by **mk563** (the 'make' utility).

```
c563 -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:

```
c563 –e test.c
```

-err

Option:

In EDE this option is not useful.

–err

Description:

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

Example:

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

```
c563 –err test.c
```

–f

Option:

Select the `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Miscellaneous`.
Add the option to the `Additional options` field.

–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` containts the following line:

```
-err
test.c
```

The command line can now be:

```
c563 -f mycmds
```

–g

Option:

Select the `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Debug`. Enable `Generate symbolic debug information`. Optionally enable the `Include debug information for non referenced types` check box and/or disable the `Include lifetime info for all types` check box.

–g[**f**|**l**|**n**|**c**]...

Default:

Generate type checking information only.

Description:

Add directives to the output files for incorporating symbolic information. This facilitates high level debugging.

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

With **–gl** you disable the 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.

If you use **–gc**, code to the application is added that performs a run–time check on stack overflows. When calling a function, the stack usage of this function and the maximum number of parameters that will be passed on stack is checked. If too little reserved memory for the stack is left, a stack overflow occurs. The application halts and stays on the label _stack_error.

When the compiler is set to a high optimization level the debug comfort may decrease. Therefore, the following rules are applied:

- When **–g** is supplied and **–O** is not supplied the compiler switches **–O2** on, which enables higher debug comfort.
- When **–g** is supplied and **–O** with one or more flags is supplied, the compiler issues warning W555 when the debug comfort would be decreased.

Examples:

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

```
c563 -g test.c
```

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

```
c563 -gl test.c
```

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

```
c563 -gn test.c
```

–O

-H

Option:

Select the `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Preprocessing`. Enter a filename in the `Include this file before source` field.

–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:

```
c563 –Hstdio.h test.c
```

–I

–I

Option:

Select the `Project | 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 C56INC (for DSP5600x), C563INC (for DSP563xx). C56INC and C563INC 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 C56INC (C563INC for DSP563xx), and the relative path) are still searched.

Example:

```
c563 –I/proj/include test.c
```

Section *Include Files*.

-L

Option:

Select the `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Code Generation`. For **c563** only, first disable the `Use hardware stack extension` check box. Enter a stack level in the `Max. hardware stack use outside interrupt functions` field.

–L*number*

Arguments:

The amount of hardware stack space available in a function (1..15)

Default:

–L7

Description:

Control the amount of hardware stack space available in a function. The compiler generates hardware DO loops for the innermost loops only. If a (nested) loop contains a function call, no hardware loop is generated.

Example:

```
c563 -L4 example.c
```

The code in `example.c` is:

```
for( ... )                  /* depth 1 -> no hardware DO loop */
{
   for( ... )               /* depth 2 -> hardware DO loop */
   {                        /* stack level 2 */
      for( ... )            /* depth 3 -> hardware DO loop */
      {                     /* stack level 4 */
         ...
      }
   }
}
```

Section *Hardware DO and REP Loops* in chapter *Overview*.

-M

Option:

Select the `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Code Generation`. Select a `Memory Model` and a `Default Data Memory`.

–M[*model*][**n**][*mem*][**L**] (at least one argument must be specified)

Arguments:

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

s	static	(**c56** only)
m	mixed	(**c56** only)
r	reentrant	(**c56** only)
24	24–bit	(**c563** only)
1624	16/24–bit	(**c563** only)
16	16–bit	(**c563** only)
6	DSP566xx	(**c563** only)

and *mem* is one of:

x	X memory
y	Y memory
l	L memory
p	P memory

other arguments:

n do not use hardware stack extension (**c563** only)

L stack not in L memory (default X or Y only)

Default:

–Mmx for **c56**
–M24x for **c563**

Description:

Select memory model and default memory space to be used. With **–Mn** (**c563** only) the compiler selects pushing and popping the return address on the user stack (DSP563xx only). To circumvent hardware stack extension silicon problems on the DSP563xx, the compiler can avoid using the hardware stack for function calls by saving the return address on the user stack.

The use of default P data memory (**–Mp**) is not recommended because it leads to much more object code. It is meant for applications with special hardware layouts only.

Example:

```
c56 –Mr test.c
c563 –M16 test.c
```

Section *Memory Models*.

-m (c563 only)

Option:

Select the `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Miscellaneous`. Select a `chip mask`.

–m*mask*

Arguments:

A number indicating the silicon mask for a DSP563xx/DSP566xx processor:

Number	Mask
0	0F92R and 1F92R
1	3F48S

Description:

Specify the silicon mask to compile for.

Example:

```
c563 -m1 example.c
```

-n

Option:

–n

Description:

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

Example:

```
c563 –n test.c
```

-O

Option:

Select the `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Optimization`. Select an optimization level in the `Optimization` box.

If you select `Custom optimization` in the `Optimization` box, you can enable or disable individual optimizations in the `Custom optimization` list.

–O*flags*

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 **c563** is **–O1**, which is an optimization level to let **c563** 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 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. A `#pragma optimize` *`number`* must be specified outside a function body. The optimization level of each `optimize` pragma is described for each **–O** option.

An overview of the flags is given below.

a	– relax alias checking (needs **–Oc**)	(function)
c	– common subexpression elimination	(function)
e	– expression propagation (needs **–Oc**)	(function)
f	– code flow, order rearranging	(function)
g	– register allocation graph optimization	(function)
h	– hardware loops	(function)
i	– move invariant code outside loop (needs **–Oc**)	(function)
j	– cache global variables in functions	(function)
l	– fast loops (increases code size)	(function)
n	– nop insertion	(flow)
o	– move parallelization, nop reduction (assembler)	(flow)
p	– data flow, constant/copy propagation	(function)
r	– single instruction DO to REP	(flow)
s	– small code size	(flow)
t	– force jump table for switch statement	(flow)
u	– loop unrolling	(function)
v	– subscript strength reduction	(function)
w	– smart switch statement, table or chain	(flow)
x	– register contents tracking	(flow)
y	– peephole optimization	(flow)
z	– non sequential register allocation	(flow)
0	– same as **–OACEFGHIJLNOPRSTUVWXYZ**	(no optim)
1	– same as **–OacefghijLnoprsTUvwxyz**	(default, size)
2	– same as **–OacefghijLnOprsTUvwxyz** all optimizations which can be debugged, optimized for code size	(debug, size)
3	– same as **–OacefghijLnoprSTuvwxyz**	(speed)
4	– same as **–OacefghijLnOprSTuvwxyz** all optimizations which can be debugged, optimized for speed	(debug, speed)

Example:

```
c563 -OacefghijLnOprsTUvwxyz test.c
```

Pragma `optimize` in section *Pragmas*.

USAGE

-Onumber

Option:

Select the `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Optimization`. Select an optimization level in the `Optimization` box. Optionally enable the `Reduce optimization for debugging` check box.

–O*number*

Arguments:

A number in the range 0 – 4.

Default:

–O1

Description:

Control optimization. You can specify a single number in the range 0 – 4, to enable or disable optimization. The options are a combination of the other optimization flags:

–O0 – same as **–OACEFGHIJLNOPRSTUVWXYZ**
Switchable optimizations switched off

–O1 – same as **–OacefghijLnoprsTUvwxyz**
Default. Set optimization to let **c563** generate the smallest code.

–O2 – same as **–OacefghijLnOprsTUvwxyz**
Set optimization flags to let **c563** generate the smallest code, but switch off optimizations that affect the ability to debug.

–O3 – same as **–OacefghijLnoprSTuvwxyz**
Set optimization flags to let **c563** generate the fastest code.

–O4 – same as **–OacefghijLnOprSTuvwxyz**
Set optimization flags to let **c563** generate the fastest code, but switch off optimizations that affect the ability to debug.

The flags 0 to 4 cannot be concatenated with other flags. For example, **–Oa3c** is not allowd, **–OacE** is allowed.

Example:

To optimize for code size and debug information, enter:

```
c563 -O2 test.c
```

-g

-Oa / -OA

Option:

Select the `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Optimization`.
Select `Custom optimization` in the `Optimization` box.
Enable or disable `Relaxed alias checking (requires CSE)`.

–Oa / –OA

Pragma:

optimize a / optimize A (on function level)

Default:

–Oa

Description:

With **–Oa** you relax alias checking. If you specify this option, **c563** 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.

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
Pragma `optimize` in section *Pragmas*.

-Oc / -OC

Option:

Select the `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Optimization`.
Select `Custom optimization` in the `Optimization` box.
Enable or disable `Common subexpression elimination (CSE)`.

–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.

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);
}
```

Pragma `optimize` in section *Pragmas*.

-Oe / -OE

Option:

Select the `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Optimization`.
Select `Custom optimization` in the `Optimization` box.
Enable or disable `Expression propagation (requires CSE)`.

–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*.

USAGE

-Of / -OF

Option:

Select the `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Optimization`.
Select `Custom optimization` in the `Optimization` box.
Enable or disable `Code flow optimization and order rearranging`.

 **–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*.

-Og / -OG

Option:

Select the `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Optimization`.
Select `Custom optimization` in the `Optimization` box.
Enable or disable `Register allocation graph optimization`.

–Og / –OG

Pragma:

optimize g / optimize G (on function level)

Default:

–Og

Description:

With **–Og** you switch on optimizations in the register allocation graph. This will for one thing create instructions with the indexed addressing mode.

With **–OG** no optimizations on the register allocation graph are performed.

Example:

```
/*
 * Compile with -OG -O0
 * Compile with -Og -O0, register allocation
 * graph optimization
 */
typedef struct
{
     long l;
     int  i;
} STRUCT;

int f(STRUCT *s_p)
{
     return s_p->i;
}
```

-Oh / -OH

Option:

Select the `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Optimization`.
Select `Custom optimization` in the `Optimization` box.
Enable or disable `Hardware loop generation`.

–Oh / –OH

Pragma:

optimize h / optimize H (on function level)

Default:

–Oh

Description:

With **–Oh** you enable the use of hardware loops. Note that this option must be on to use the **–Or** option.

With **–OH c563** will not use hardware DO/REP loops.

Example:

```
/*
 * Compile with –OH –O0, software loop is used
 * Compile with –Oh –O0, hardware DO loop is used
 */
int cumu;

void
main( void )
{
      int i;
      for ( i = 0; i<1000; i++ )
      {
            cumu = cumu + i;
      }
}
```

Section *Hardware DO and REP Loops*.
Pragma `optimize` in section *Pragmas*.

-Oi / -OI

Option:

Select the `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Optimization`.
Select `Custom optimization` in the `Optimization` box.
Enable or disable `Move invariant code outside loop (requires CSE)`.

–Oi / –OI

Pragma:

optimize i / optimize I (on function level)

Default:

–Oi

Description:

With **–Oi** you move invariant code outside a loop. With **–OI** you disable moving invariant code outside a loop.

The option **–Oc** must be on to use the **–Oi** option.

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;
      }
}
```

USAGE

–Oc
Pragma `optimize` in section *Pragmas*.

-Oj / -OJ

Option:

Select the `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Optimization`.
Select `Custom optimization` in the `Optimization` box.
Enable or disable `Cache global variables in functions`.

–Oj / –OJ

Pragma:

optimize j / optimize J (on function level)

Default:

–Oj

Description:

With **–Oj** the compiler will cache a global variable in a register in some situations. The compiler detects when a global variable is accessed multiple times without being changed by other routines, and creates a local variable to temporarily contain the global. Afterwards, the local variable is written back to the global. This speeds up access to global variables.

With **–OJ** all updates of a global variable are directly written to memory.

Examples:

```
/*
 * Compile with -OJ -O0, 'sum' is updated twice
 * Compile with -Oj -O0, 'sum' is updated only once
 */
int sum;

void
add( int a, int b )
{
      sum += a;
      sum += b;
}
```

Pragma `optimize` in section *Pragmas*.

-Ol / -OL

Option:

Select the `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Optimization`.
Select `Custom optimization` in the `Optimization` box.
Enable or disable `Generate fast loops (increases code size)`.

The fast loop optimization is disabled when `Hardware loop generation` is enabled.

–Ol / –OL

Pragma:

optimize l / optimize L (on function level)

Default:

–OL

Description:

With **–Ol** 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.

The fast loop optimization is switched off when hardware loop generation is enabled (**–Oh**).

Example:

```
/*
 * Compile with –OL –O0
 * Compile with –Ol –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*.

-On / -ON

Option:

Select the `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Optimization`.
Select `Custom optimization` in the `Optimization` box.
Enable or disable `NOP insertion`.

–On / –ON

Pragma:

optimize n / optimize N (on flow level)

Default:

–On

Description:

With **–On** you enable NOP insertion. This ensures the generation of correct code. For some instructions, for example moves to and from address–type registers, a nop instruction must be inserted before the pending instruction to take care of the delay slot.

With **–ON** you disable NOP insertion. The assembler is instructed to insert nops instead.

Example:

```
/*
 * Compile with –ON, NOP is not inserted.
 * Compile with –On, NOP is inserted.
 */
int i;

void
main( void )
{
     int a;

     f();     /* call a function */

     for ( i = 0; i < 1; i++)
          a = i;
}
```

Section *Replacing NOPs*.
Pragma `optimize` in section *Pragmas*.

-Oo / -OO

Option:

Select the `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Optimization`.
Select `Custom optimization` in the `Optimization` box.
Enable or disable `Generate parallel moves`.

–Oo / –OO

Pragma:

optimize o / optimize O (on flow level)

Default:

–Oo

Description:

With **–Oo** the compiler generates the `opt op` option for the assembler to enable move parallelization and nop reduction.

With **–OO** you disable the assembler optimization.

Example:

To disable the assembler optimization, enter:

```
c563 -OO test.c
```

Section *Replacing NOPs* and section *Instruction Parallelization (parallel moves)* in chapter *Overview*.
Pragma `optimize` in section *Pragmas*.

-Op / -OP

Option:

Select the `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Optimization`.
Select `Custom optimization` in the `Optimization` box.
Enable or disable `Constant and copy propagation (data flow)`.

–Op / –OP

Pragma:

optimize p / optimize P (on function level)

Default:

–Op

Description:

With **–Op** 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 **–OP** you disable constant and copy propagation.

Example:

```
/*
 * Compile with –OP –O0, 'i' is actually assigned to 'j'
 * Compile with –Op –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*.

USAGE

-Or / -OR

Option:

Select the `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Optimization`.
Select `Custom optimization` in the `Optimization` box.
Enable or disable `Optimize single instruction hardware DO to REP loops`.

–Or / –OR

Pragma:

optimize r / optimize R (on flow level)

Default:

–Or

Description:

With **–Or** you specify to optimize single instruction DO loops to REP loops. Note that the option **–Oh** (enable hardware loops) must be on to use this option.

With **–OR c563** will not optimize to REP loops. This is advantageous for the interrupt latency as REP loops are not interruptible.

Example:

```
/*
 * Compile with –OR –Oh, a DO loop is used
 * Compile with –Or –Oh, optimize to REP loops
 */
void
f( void )
{
      int i, a;
      for ( i = 0; i<1000; i++ )
      {
            a = i;
      }
      return( a+i );
}
```

Section *Hardware DO and REP Loops*.
Pragma `optimize` in section *Pragmas*.

-Os / -OS

Option:

Select the `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Optimization`.
Select `Custom optimization` in the `Optimization` box.
Enable or disable `Favor code size above execution speed`.

–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 fewer instructions are used. Note that this may result in more instruction cycles.

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

Example:

```
int a, b;

void
main( void )
{
     a = ( b != 0 );
}
```

Compiling once with **–Os** and once **–OS** results in the following difference in code (**–Os** generates smaller code):

```
bset #0,b          ; -Os:  1 word,  4 cycles
move #>1,b         ; -OS:  2 words, 3 cycles
```

The number of cycles is calculated for zero wait state memory. the cycle count for the **–OS** option increases rapidly for non–zero wait states, so that small code will execute faster here.

	–Os	**–OS**	
1 wait:	5	5	cycles
2 wait:	6	7	cycles

Pragma `optimize` in section *Pragmas*.

-Ot / -OT

Option:

Select the `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Optimization`.
Select `Custom optimization` in the `Optimization` box.
Select an entry from the `Code generation for switch statement` box.

–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 -O0, generate jump chain.
 * Compile with -Ot -OW -O0, generate jump table.
 */
int i;
```

USAGE

```
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 `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Optimization`.
Select `Custom optimization` in the `Optimization` box.
Enable or disable `Loop unrolling`.

–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 `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Optimization`.
Select `Custom optimization` in the `Optimization` box.
Enable or disable `Subscript strength reduction`.

–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:

Select the `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Optimization`.
Select `Custom optimization` in the `Optimization` box.
Select an entry from the `Code generation for switch statement` box.

–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.

Overview:

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

Example:

```
/*
 * Compile with –OW –OT –O0, always generate jump chain.
 * Compile with –Ow –OT –O0, 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*.
–Ot / **–OT**

-Ox / -OX

Option:

Select the `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Optimization`.
Select `Custom optimization` in the `Optimization` box.
Enable or disable `Register contents tracking`.

–Ox / –OX

Pragma:

optimize x / optimize X (on function level)

Default:

–Ox

Description:

With **–Ox** you switch on register contents tracking.

With **–OX** you disable register contents tracking.

Example:

```
/*
 * Compile with –OX –O0
 * Compile with –Ox –O0, register contents tracking,
 * the overlay scratch section is removed
 */
int a, b, c;

void main(void)
{
      a = 2;
      switch(b)
      {
      case 1:
            c = 3;
            break;
      case 2:
      case 3:
            c = 0;
            break;
      }
}
```

-Oy / -OY

Option:

Select the `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Optimization`.
Select `Custom optimization` in the `Optimization` box.
Enable or disable `Peephole optimization`.

–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 redundent 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.
 *  asl  b
 *  add  a,b
 * can be combined to: addl a,b
 */

long a;
long f(void);

void
main( void )
{
      long b;
      b = f();
      a = (a << 1) + b;
}
```

Pragma `optimize` in section *Pragmas*.

-Oz / -OZ

Option:

Select the `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Optimization`.
Select `Custom optimization` in the `Optimization` box.
Enable or disable `Non-sequential register allocation`.

–Oz / –OZ

Pragma:

optimize z / optimize Z (on flow level)

Default:

–Oz

Description:

With **–Oz** the compiler tries to use a non sequential register allocation scheme. The compiler chooses one of the available registers in an arbitrary order. With this compiler optimization the assembler can perform a better optimization.

With **–OZ** the compiler uses a sequential register allocation scheme. The next available register is used. This can also be a previously allocated register that is no longer used.

Example:

```
/*
 * Compile with –OZ –O0, sequential register allocation, the
 *   same register is used for each allocation.
 * Compile with –Oz –O0, non sequential register allocation.
 *   Three different registers are used for the allocations.
 */
int a[3];

void
main( void )
{
      a[0] = 0;
      a[1] = 1;
      a[2] = 2;
}
```

Pragma `optimize` in section *Pragmas*.

-o

Option:

Select the `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Miscellaneous`.
Add the option to the `Additional options` field.

–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:

```
c563 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**.

-p

Option:

–p*page*

Arguments:

The number of Patriot memory pages (1..8).

Default:

–p2

Description:

Use this option to specify the number of Patriot memory pages. Page switch instructions depend on the specified number of memory pages.

Section 3.17, *Patriot Bank Switching Support*, in chapter *Language Implementation*.

Example:

To set the number of Patriot memory pages to five:

```
c563 –p5 test.c
```

–R

Option:

Select the `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Output`. Add a section name replacement to the `Replace default section names` field.

–R[*dname*[**=***sname*]]

Arguments:

Optionally the default section name and the new section name.

Description:

You can use the **–R** option to replace the default section names generated by the compiler. If you use the **–R** option alone, the compiler replaces all section names by a name that is related to the filename of the source module. If you want to replace some of the default section names, you can specify these default names with **–R**. The new names are related to the module name unless the new name is specified in an optional second part. The **–R** option may appear more than once.

dname is the default name of the section, normally used by the compiler. *sname* is the new section name the compiler must generate instead.

Example:

To generate the section name `mystring` instead of the default section name `.xstring`, enter:

```
c563 –R.xstring=mystring test.c
```

By specifying:

```
c563 –R.xstring test.c
```

the section name `.xstring` has been renamed to `.xstringtest`.

-r

Option:

Select the `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Code Generation`. Add one or more registers to the `Reserve address or offset register` field.

–r*register*

Arguments:

A register name. The *register* argument must be one of the following:

For the DSP5600x and DSP563xx/DSP566xx:

R0, R1, R2, R3, R4, R5, R6,
M0, M1, M2, M3, M4, M5, M6,
N0, N1, N2, N3, N4, N5, N6

In the static model (DSP5600x only) R7, M7 and N7 can also be used; in the other models these registers are used for the user stack pointer.

Description:

Reserve a register. The compiler avoids the use of the *register* while allocating registers. When you want to reserve a register that is to be used in, for example, an interrupt routine, it is recommended to do this application wide. This implies that you will have to specify the **–r** option for each module translated. If the register is not reserved in all modules, the compiler may allocate it and overwrite its contents.

Because an R and an M register are very close related, the compiler also reserves the M register when an R register is reserved. E.g., reserving R1 implies also reserving M1. You can reserve a list of registers by supplying a **–r** option to the compiler for each register to be reserved. Besides the user stack pointer registers, at least one combination of R and N registers must be kept free for compiler use.

Example:

The following example reserves r5, m5 and n5 while translating `example.c`.

```
c563 –rr5 –rn5 example.c
```

-s

Option:

Select the `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Output`. Enable the `Merge C source file with assembly output` check box.

–s[**i**]

Pragma:

source

Description:

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

With the optional '**i**' sub–option you can specify to merge the lines of included files as well. This is useful when included files generate program code. In general it will only create longer source files due to expansion of all header files.

Example:

```
c563 -s test.c
```

```
;             test.c:
; 1    |int i;
; 2    |
; 3    |int
; 4    |main( void )
; 5    |{

         extern  F_START
         global  Fmain
```

Pragmas `source` and `nosource` in section *Pragmas*.

-t

Option:

Select the `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Output`.
Enable the `Display lines/min` check box.

–t

Description:

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

Example:

```
c563 –t test.c

processed 180 lines at 8102 lines/min
```

-U

Option:

Select the `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Preprocessing`. Undefine one or more predefined macros by disabling the corresponding check box.

–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:

__FILE__	”current source filename”
__LINE__	current source line number (int type)
__TIME__	”hh:mm:ss”
__DATE__	”Mmm dd yyyy”
__STDC__	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, __STDC__ is set to 0 (zero).

When **c563** is invoked, also the following predefined symbols exist:

_C56	predefined symbol to identify the compiler. This symbol can be used to flag parts of the source which must be recognized by the TASKING **c56** or **c563** compiler only. It expands to the version number of the compiler.
_MODEL	identifies for which memory model the module is compiled. It expands to a single character (’r’ for reentrant, ’m’ for mixed, ’s’ for static, 16 for 16–bit 1624 for 16/24–bit or 24 for 24–bit) that can be tested by the preprocessor. See section *Memory Models* for details.

_DEFMEM identifies the default data memory. See section *Memory Models* for details.

_DSP identifies for which DSP processor type the module is compiled. It expands to a number. For example, '0' for the DSP5600x processor type, '3' for the DSP563xx and '6' for the DSP566xx processor types.

_STKMEM identifies the data memory used for the stack; this is either default data memory, or L memory. See section *Memory Models* for details.

_USP identifies the user stack pointer register from the **–Cr**/**–CR** option (r6 or r7). For r6 its value is '6', for r7 it is '7'.

_CACHE_SECTOR_SIZE
Expands to the value set with the option **–c***size* or with the pragma **cache_sector_size** (**c563** only).

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

Example:

```
c563 –U_MODEL test.c
```

–D

-u

Option:

Select the `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Code Generation`.
Enable the `Treat 'char' variables as unsigned` check box.

–u

Description:

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

Example:

With the following command `char` is treated as `signed char`:

```
c563 -u test.c
```

-V

Option:

Select the `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Miscellaneous`.
Add the option to the `Additional options` field.

–V

Description:

Display version information.

Example:

```
c56 –V

TASKING DSP5600x C compiler     vx.yrz Build nnn
Copyright 1995–year Altium BV   Serial# 00000000

c563 –V

TASKING DSP563xx/6xx compiler   vx.yrz Build nnn
Copyright 1996–year Altium BV   Serial# 00000000
```

-w

Option:

Select the `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Diagnostics`. Select `Display all warnings`, `Suppress all warnings` or `Suppress only certain warnings`.
If you select `Suppress only certain warnings`, type the numbers of the warnings you want to suppress in the corresponding field.

–w[*num*]
–wstrict

Arguments:

Optionally the warning number to suppress.

Description:

–w suppresses all warning messages. –**w***num* only suppresses the given warning. **–wstrict** suppresses all "strict" warning messages (196, 303).

Example:

To suppress warning 135, enter:

```
c563 file1.c –w135
```

-z

Option:

Select the `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Miscellaneous`.
Add the option to the `Additional options` field.

-z*pragma*

Arguments:

A pragma as listed in section *Pragmas*.

Description:

With this option you can give a pragma on the command line. This is the same as specifying '#pragma *pragma*' in the C source. Dashes ('–') on the command line in the pragma are converted to spaces. A dash prefixed by another dash or space is never translated, so it is still possible to specify a dash for negative numbers as pragma argument.

Example:

The following options

```
-zcache_sector_size-256
-zjumptable_memory-x
```

are equivalent with:

```
#pragma cache_sector_size 256
#pragma jumptable_memory x
```

Section *Pragmas*.

4.3 INCLUDE FILES

You may specify include files in two ways: enclosed in <> or enclosed in "". When an #include directive is seen, **c563** 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 '\' for PC, or a '/' for UNIX), the include file is searched for in the directory of the file containing the #include line. For example, in:

   PC:

   ```
   c563 ..\..\source\test.c
   ```

   UNIX:

   ```
   c563 ../../source/test.c
   ```

   c563 first searches in the directory `..\..\source` (`../../source` for UNIX) for include files.

   If you compile a source file in the directory where the file is located (**c563** 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:

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

   PC:

   ```
   c563 –I..\..\include demo.c
   ```

   UNIX:

   ```
   c563 –I../../include demo.c
   ```

3. Check if the environment variable C563INC exists (for the DSP563xx; use C56INC for the DSP5600x). 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 C563INC by using a separator character. Instead of using **–I** as in the example above, you can specify the same directory using C563INC:

PC (Command Prompt window):

```
set C563INC=..\..\include
c563 demo.c
```

UNIX:

if using the Bourne shell (sh)

```
C563INC=../../include
export C563INC
c563 demo.c
```

or if using the C–shell (csh)

```
setenv C563INC ../../include
c563 demo.c
```

4. When an include file is not found with the rules mentioned above, the compiler tries the subdirectory `include` in the directory that contains the subdirectory with the **c563** binary. For example:

   PC:

   c563.exe is installed in the directory `C:\C563\BIN`
   The directory searched for the include file is `C:\C563\INCLUDE`

   UNIX:

   c563 is installed in the directory `/usr/local/c563/bin`
   The directory searched for the include file is
   `/usr/local/c563/include`

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

The DSP5600x installation directory is `c56`.

A directory name specified with the **–I** option or in C563INC (C563INC for the DSP5600x) may or may not be terminated with a directory separator, because **c563** (or **c56** respectively) inserts this separator, if omitted.

When you specify more than one directory to the environment variable C563INC, you have to use one of the following separator characters:

PC:

; **,** *space*

USAGE

e.g. **set C563INC=..\..\include;\proj\include**

UNIX:

: **;** **,** *space*

e.g. **setenv C563INC ../../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 generator: command line options and keywords. The compiler acknowledges these three groups using the following rule:

Command line options can be overruled by keywords and pragmas. Keywords can be overruled by pragmas. So the pragma has 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.

c563 supports the following pragmas:

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 such as NOPs removal, parallel moves and 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 DSP56xxx features which could otherwise not be accessed by the C language.

For more information on intrinsic functions see section 3.13 *Intrinsic Functions*.

asm_noflush

Same as **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.

cache_align_now

Aligns current address at a cache boundary. This can be done only once per function. The alignment may introduce an unusable memory alignment gap at the beginning of the section.

cache_sector_size

Specify the size of the cache sectors for the used DSP563xx derivative. This size is used for the alignments generated for the pragma **cache_align_now** and for the `_cache_region` qualifier. The size is either 128 or 256 words. The default cache size is 128. This pragma is equivalent to the command line option **-c** and it is given in the form:

#pragma cache_sector_size *size*

cache_region_start

Mark start position of a cache region. The argument of the pragma must be a function pointer defined with the `_cache_region` qualifier.

cache_region_end

Mark end position of a cache region. The argument of the pragma must be a function pointer defined with the `_cache_region` qualifier.

The section 3.16 *DSP563xx Cache Support* contains more infomation about these pragmas.

iterate_at_least_once
no_iterate_at_least_once *(default)*

For the compiler it is not always possible to determine whether a loop condition is valid for the first iteration. In these cases the compiler will generate code to check the loop end condition for this first iteration. If it is certain that the loop will be executed at least once, this code is superfluous. The **iterate_at_least_once** pragma tells the compiler that the following loop(s) are iterated at least once and that it does not have to generate the code for checking the end condition for the first iteration. This pragma remains valid until the **no_iterate_at_least_once** pragma is specified.

If the **iterate_at_least_once** pragma is specified for a loop which is not iterated at least once, the results of the loop will not be as expected.

Example:

```
void foo(int i)
{
    while (i > 0)
    {
        printf("%d\n", i);
        i--;
    }
}

...Code...
Ffoo:   move    (r7)+
        do      a1,L11
        move    a1,n5
        move    a1,x:(r7-1)
        move    n5,a
        move    #L8,r0
        jsr     Fprintf
        move    x:(r7-1),a
        sub     #1,a
L11:    void    a, r0, n5
        move    (r7)-
        rts
```

The compiler does not know the start value of `i`, and therefore cannot predict whether the end condition is true for the first loop. The **iterate_at_least_once** pragma tells the compiler that the condition is valid for the first iteration and the compiler will not generate code to test the condition for the first iteration.

For do–while loops (which are always iterated at least once) the **iterate_at_least_once** pragma indicates that the do–while loop end condition is false for the first iteration.

Example:

```
void foo(int i)
{
    do
    {
        printf("%d\n", i);
        i--;
    } while (i > 0)
}
```

...Code...

```
Ffoo:   move    (r7)+
        do      a1,L5
        move    a1,r5
        move    a1,x:(r7-1)
        move    r5,a
        move    #L3,r0
        jsr     Fprintf
        move    x:(r7-1),a
        sub     #1,a
L5:     void    a, r0, r5
        move    (r7)-
        rts
```

jumptable_memory

Specifies in which memory the switch jump table is generated. The default is P memory. Please note that changing the default into X or Y memory will result in a (possibly large) entry in the copy table! This pragma is given in the form:

#pragma jumptable_memory *mem*

where, *mem* can be either P, X or Y.

optimize

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. This pragma is given in the form:

#pragma optimize *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. A **#pragma optimize** *number* must be specified outside a function body. 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.

pack_strings

After this pragma all string constants will be packed string constants. Using such a string as a not packed string (e.g., passing to a function with a not packed string argument type) will yield a type conflict error.

nopack_strings

After this pragma string constants will no longer be packed string constants.

The section 3.18 *Packed Strings* contains more information about packed strings.

source

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

nosource

Default. Disable generation of C source within assembly code.

4.5 ALIAS CHECKING

When alias checking is turned on (**–OA** option) 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 (**–Oa** option or **#pragma optimize a**). This is the default. Note that the option **–Oc** (or **#pragma optimize c**) 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:

```
typedef union
{
   struct
   {
      unsigned int exponent;
      _fract fraction;
   } binary;
   double value;
} _FDEF;

#define _FBIAS 127

double
frexp_excerpt( double value, int *exp )
{
   double loc_value = value;  /* prevent taking address of
                                 argument 'value' */

   *exp = ((_FDEF *)&loc_value)->binary.exponent - _FBIAS;
   ((_FDEF *)&loc_value)->binary.exponent = _FBIAS;

   if ( loc_value == -1.0 )
   {
      *exp += 1;
      return( -0.5 );
   }
   return( loc_value );
}
```

The compiler will make a CSE on `loc_value`, ignoring the write through the type casted pointer because it is of a different type.

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. **c563**'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 **#include**d 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

TASKING

5 CHAPTER

5.1 INTRODUCTION

c563 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, **c563** 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, **c563** displays the line number causing the error before the error message. **c563** 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

c563 returns an exit status to the operating system environment for testing.

For example,

in a BATCH–file you can examine the exit status of the program executed with ERRORLEVEL:

```
c563 –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:

```
c563 $*
case $? in
0)        echo ok ;;
1|2|3)    echo error ;;
esac
```

The exit status of **c563** 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' dirctive 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 (0x*hexnumber*)

The character with the hexadecimal ASCII value 0x*hexnumber* 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 contant must contain exactly one character. Empty character contants (`''`) are not allowed.

E 21 character constant overflow

A character contant 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 formatl 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 protoype 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 contstant 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 hexadicimal 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 unkown 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 a function should return something. 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 */
}
```

W 184 empty pragma name in –z option – ignored

The **–z** option requires a pragma name.

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 `123.12fl` 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 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 = 17000000L;  /* 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 "*name*" – 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 protoype 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 contstant expression with a value greater than zero.

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 303 variable '*name*' possibly uninitialized

Possibly an initialization statement is not reached, while a function should return something. 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.

F 347 Error in constraints for inline assembly: Wrong register–size or no memory or immediate possible

The variable does not fit into the register specified, or the variable can not be referenced in memory (**m** opererand constraint), or the expression does not yield an immediate value (**i** operand constraint).

F 348 Error in constraints for inline assemly: Numeric constraint (opr nbr *number*) out of range.

Operand number nbr is not defined in constraint.

F 349 Error in constraints for inline assembly: Second numeric constraint to the same output

W 358 User stack pointer register can not be reserved

Reserved user stack pointer will be ignored.

Backend

W 501 function qualifier used on non–function

A function qualifier can only be used on functions.

W 502 _fract constant saturation occurred

An overflow occurred. The following is an example of this warning:

```
_fract a;

int f(void)
{
   a = .75 + .5;  /* 1.25, overflow */
}
```

E 503 cache pragma requires function address

The argument of a `cache_region_start` and `cache_region_end` pragma must be a function pointer defined with the `_cache_region` qualifier.

W 504 24–bit pointer calculation requires mode switching

A pointer calculation is generated for the 16/24–bit model that requires switching to the 24–bit model temporarily and therefore gives inefficient coding.

W 508 function qualifier duplicated

Only one function qualifier is allowed. The duplicate qualifier is ignored.

F 509 invalid *name* option

The specified option is not valid. See the invocation syntax for a list of options and their suboptions.

F 510 illegal number in option *name*

For the **–c** option only the values 128 and 256 are allowed. The **–L** option only accepts numbers in the range 1..15. The **–m** option only accepts the values 0 and 1. See the invocation syntax for more information.

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*' illegal interrupt number (–1, or 0 to 63) – ignored

The interrupt vector number must be –1 or in the range 0 to 63. Any other number is illegal.

E 513 calling an interrupt routine, use '_swi()'

An interrupt function cannot be called directly, you must use the intrinsic function `_swi()`.

E 514 conflict in '*attribute_type*' attribute

The attributes of the current function qualifier declaration and the previous function qualifier declaration are not the same.

E 515 different '_long_interrupt | _fast_interrupt' number

The interrupt number of the current function qualifier declaration and the previous function qualifier declaration are not the same.

E 516 '*memory_type*' is illegal memory for function: program '_P' only

The storage type is not valid for this function.

W 517 conversion of long address to short address

The compiler converts a long address to a short address when you assign a value to an `_near` object.

W 518 conversion of fractional to integer type occurred

The compiler converts a fractional type to an integer type when _fracts are mixed in expressions. This can also occur when constants are folded. In most cases this is an error that can be avoided with a type cast.

W 519 conversion of integer to fractional type occurred

The compiler converts an integer type a fractional type in the code. This can also occur when constants are folded. In most cases this is an error.

W 520 conversion of _circular pointer does not preserve circular information.

If a `_circular` pointer is converted to a `long` or `unsigned long`, the circular information will be lost.

F 524 illegal memory model

The memory model you specified does not exist. See the **–M** option for a list of the available arguments.

W 525 function qualifier '_reentrant' ignored for static memory model

`_reentrant` is only allowed in non–static memory models.

E 526 illegal __asm() constraint *string*

The character used for a register type in an `__asm` intrinsic function is not valid.

E 527 illegal __asm() modifier *char* for register *reg*

The character used for a register modifier in an `__asm` intrinsic function is not valid.

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.

W 532 _at() has no effect on external declaration

When declared `extern` the variable is not allocated by the compiler.

W 533 c56 language extension keyword used as identifier

A language extension keyword is a reserved word, and reserved words cannot be used as an identifier.

E 534 hardware stack level must be in range 1..15/16

The **–L** option only accepts a hardware stack level depth in the range 1..15 (1..16 for 563xx). See the description of the **–L** option for more information.

E 536 illegal syntax used for default section name '*name*' in –R option

See the description of the **–R** option for the correct syntax.

E 537 default section name '*name*' not allowed

See the description of the **–R** option for the correct syntax.

W 538 default section name '*name*' already renamed to '*new_name*'

Only use one **–R** option for section *name* or use another name.

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`.

W 543 fast interrupt code section too large – long interrupt substituted

The section became too large to fit in a fast interrupt code section, so the compiler placed it in a long interrupt code section.

W 544 fast interrupt routine must be committed to a vector – ignored

A `_fast_interrupt` function must have a vector in the range 0..63.

W 546 illegal modifier for this type – ignored

For example, `short _fract` is illegal.

E 547 assignment of a circular object to a non–circular pointer

assignment of a non–circular object to a circular pointer

Only circular objects can be assigned to circular pointers.

E 549 only _circ pointers and arrays/structs/unions are allowed – ignored on '*name*'

only _circ pointers and arrays/structs/unions are allowed – ignored on struct/union member '*name*'

`_circ` is only allowed on structures and unions if its member is of type `_circ`.

W 551 recursion in non–reentrant function '*name*'

Recursion is only allowed on reentrant functions.

E 552 only 'long' type objects are allowed in _L memory space – ignored on '*symbol_name*'

The specified object is not of type 'long' and therefore is not allowed in the _L memory space.

E 553 only 'long' type struct/union members are allowed in _L memory space – ignored on '*symbol_name*'

The specified member is not of type 'long' and therefore is not allowed in the _L memory space.

W 555 current optimization level could reduce debugging comfort (–g); the –O2 or –O4 option may be used to improve this

You could have HLL debug conflicts with these optimization settings. The **–O2** and **–O4** option optimize for debug.

W 556 Register *name* cannot be reserved: invalid register

Register *name* cannot be reserved: is a function return register

If the first warning occurs you specified an invalid register name to the **–r** option. The second warning occurs if R0 or N0 is used, because these are function return registers and therefore cannot be reserved. See the **–r** option for a list of available register names.

W 557 Register *name* cannot be reserved: is used for dynamic stack

The specified register is used as a stack pointer register and therefore cannot be reserved. See the **–r** option for a list of available register names.

E 558 Error in reserving registers: not enough address registers left

All address registers have been used.

E 559 Intrinsic function call needs a reserved register

You have to provide a reserved register name as an argument of an intrinsic function.

E 560 Invalid operand in intrinsic function call

See section *Intrinsic Functions* for the syntax of the intrinsic function.

E 562 Cache sector size must be 128 or 256

For the **–c** option only the values 128 and 256 are allowed.

E 563 _circ pointer has modulo size 0

A pointer is used to retrieve a circular pointer, but the modulo part is undefined and therefore set to 0. This can occur when type casts are used incorrectly.

E 564 invalid intrinsic function for this DSP type

An intrinsic function for DSP is coded that is not available on the current DSP because of instruction set limitations.

W 565 intrinsic function is not supported on all silicon revisions

An intrinsic function is encountered that is not implemented on all DSPs of the current type.

W 566 packed strings not supported when generating Motorola assembler compatible output, #pragma ignored

Packed strings are not supported when you use the **–Ca** option (Motorola compatible assembly). Remove the pragma or generate TASKING compatible assembly.

E 567 '*mem*' is illegal memory for jumptable

An illegal memory type was specified for the **jumptable_memory** pragma. Please change this memory type into either X, Y or P.

W 568 Depth of hardware stack limited without disabling hardware stack extension

The **–L** option is best served in combination with the **–Mn** option. **–Mn** assures maximum hardware stack array capacity when stack extension is off. If stack extension is on, there is no reason to limit the stack array because it is automatically extended into data memory when exceeded.

W 569 Motorola compatibility mode selected with stack in L–memory; use –ML to disable

The Motorola debugger does not support placing the stack in L memory. So, this combination is of limited use.

E 570 _compatible function definition not allowed with new–style stackframe; use –Cs to disable

The Motorola compatibility calling convention requires the old–style stack layout (stack pointer points to first unused location).

F 571 Maximum number of nested loops exceeded

Control structures and compound statements have been nested too deeply.

F 572 *arg* of function qualifier '_bank' out of range; check –p option

The indicated argument in the `_bank` function qualifier is out of range. Use the command line option **–p***page* (*page* is 1..8) to specify the total number of pages.

W 573 Not possible to use different calling conventions on same function, function qualifier *qualifier* ignored.

You specified two calling conventions on the same function. The second function qualifier is ignored. Use either `_compatible` or `_callee_save`.

CHAPTER 6

LIBRARIES

TASKING

9 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. **c563** comes 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 (e.g. 32 bit signed divide). These operations are implemented as run–time library functions. The run–time library routines are added to the C library.

The C libraries supplied with the compiler are suitable for **–Mx** (the default data space X memory). In the following cases you must rebuild the libraries to avoid conflicts:

- if you change the default data space or change the data space of the user stack (default L memory) (**–ML**)
- if you change the user stack pointer (**–Cr**)
- if you select the default calling convention (**–Cc**, or `_compatible`)
- if you select the old–style/Motorola compatible stack frame (**–Cs**)

Use the same options for both rebuilding the libraries as building your application.

The sources of these libraries and a makefile are included in the compiler package.

6.2 REBUILDING LIBRARIES

Use the following procedure to rebuild the libraries for the **c563** compiler (similar steps can be followed for the **c56**):

1. Check if the `bin` directory of the installed **c563** product is included in your path environment setting. If it is not there you should add it.

2. In the directory `lib\src` you will find the library sources. For each type of library it also contains a subdirectory with a makefile for creating the specific library. Select the directory, depending on which library you want to build, and make this directory the working directory.

3. Edit the concerning makefile and add the new options to the line which defines `CFLAGS`. For example:

```
CFLAGS  = -I$(SRCDIR) -M24yn -Cacr -AF
```

4. Type:

```
mk563
```

   to build libraries for **c563**.

5. When building is finished you will find the library in the current directory. You can copy the library to the `lib\563xx` or `lib\566xx` directory of the product. But before you do so, you may want to make a backup copy of the original library in that directory.

   When the hardware stack extension has to be disabled on the DSP, you also have to retranslate `start.asm` with the `NOESTACK` macro defined. When retranslating the libraries, you can set this option also in the concerning makefile, by adding **-DNOESTACK** to the `ASFLAGS` definition.

6.3 LIBRARIES OVERVIEW

The table below lists the libraries included in the DSP5600x (**c56**) and DSP563xx/6xx (**c563**) toolchains. What libraries are to be linked depends on the memory model selected. The Control Program and EDE will automatically select the correct libraries depending on the memory model specified.

The `lib` directory contains the subdirectories with the library files for the DSP5600x, DSP563xx and DSP566xx.

Compiler	Processor	Model	Libraries		
			C	Run–time	Floating point
c56	DSP5600x	Static	libcs.a	librt.a	libfp.a
		Reentrant	libcr.a		
		Mixed	libcm.a		
c563	DSP563xx	16 bit	libc16.a	librt16.a	libfp16.a
		16/24 bit	libc1624.a		
		24 bit	libc24.a	librt24.a	libfp24.a
	DSP566xx	–	libc6.a	librt6.a	libfp6.a

Table 6–1: Overview of Libraries

The **lk563** linker uses this naming convention when specifying the **–l** option. For example, with **–lc16** the linker looks for `libc16.a` in the DSP type specific subdirectory of 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 always 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.

6.4 INPUT/OUTPUT FUNCTIONS

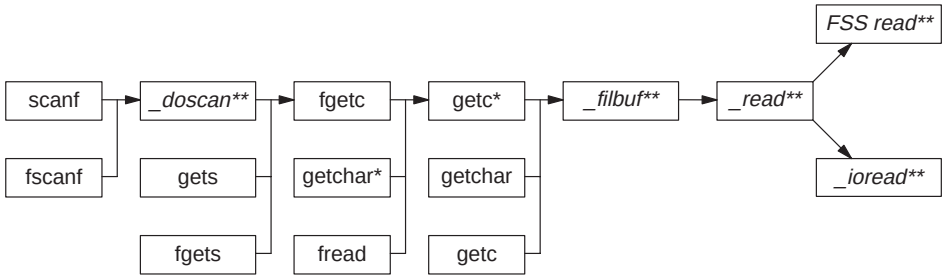


Figure 6–1: Calling mechanism of input function in C library

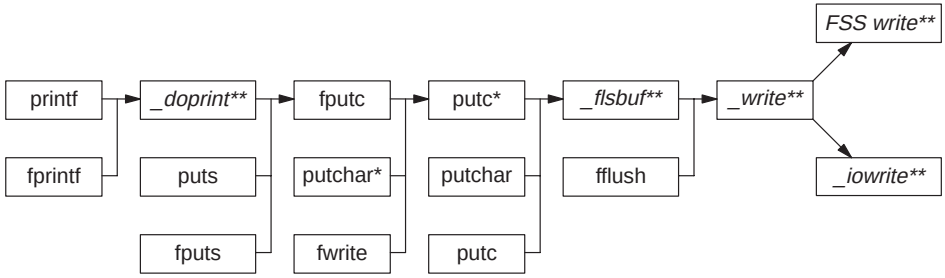


Figure 6–2: Calling mechanism of output function in C library

* = function is implemented as a macro (if `stdio.h` is included); the C libraries contain them as well, for example, in cases where a function pointer needs the address of such a function.

** = internal function.

6.5 HEADER FILES

The following header files are delivered with the C compiler:

<**assert.h**> assert

<**c56.h**> Special file with **c563** definitions. No C functions. Can be used for prototyping your application on a host using a standard C compiler.

<**conio.h**> _insize, kbhit().

<**ctype.h**> isalnum, isalpha, isascii, iscntrl, 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**> Constants related to floating point arithmetic.

<**limits.h**> Limits and sizes of integral types. No C functions.

<**locale.h**> localeconv, setlocale. Delivered as skeletons.

<**math.h**> acos, asin, atan, atan2, ceil, cos, cosh, exp, fabs, floor, fmod, frexp, ldexp, log, log10, modf, pow, sin, sinh, sqrt, tan, tanh

<**reg56xxx.h**>
Include files with all special function register definitions for the DSP56*xxx*.

<**setjmp.h**> longjmp, setjmp

<**signal.h**> raise, signal. Functions are delivered as skeletons.

<**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, _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, _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, _packsize, _packstr, _pstr_get, _pstr_put, strcat, strchr, strcmp, strcol, strcpy, strcspn, strerror, strlen, strncat, strncmp, strncpy, strpbrk, strrchr, strspn, strstr, strtok, strxfrm, _unpackstr, _unpstrlen

<**time.h**> asctime, clock, ctime, difftime, gmtime, localtime, mktime, strftime, time.

6.6 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.

6.6.1 C LIBRARY IMPLEMENTATION DETAILS

A detailed description of the delivered C library is shown in the following list.

Explanation :

Y – Fully implemented
I – Implemented, using file system simulation
L – Delivered as a skeleton

File	Imple–mented	Routine name	Description / Reason
assert.h	Y	'assert()' macro	Macro definition
conio.h	Y	_insize kbhit()	

File	Imple–mented	Routine name	Description / Reason
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	acos	
	Y	asin	
	Y	atan	
	Y	atan2	
	Y	ceil	
	Y	cos	
	Y	cosh	
	Y	exp	
	Y	fabs	
	Y	floor	
	Y	fmod	
	Y	frexp	
	Y	ldexp	
	Y	log	
	Y	log10	
	Y	modf	
	Y	pow	
	Y	sin	
	Y	sinh	
	Y	sqrt	
	Y	tan	
	Y	tanh	
setjmp.h	Y		
	Y	longjmp	
	Y	setjmp	
signal.h	Y		
	L	raise	No OS present
	L	signal	No OS present
stdarg.h	Y		
	Y	va_arg	
	Y	va_end	
	Y	va_start	
stddef.h	Y		Only Macros
stdio.h	Y		Due to 'stdarg.h/varargs.h' conflicts, the routines 'vprintf()', 'vfprintf()', 'vsprintf()' are not ANSI yet.
	Y	clearerr	
	I	fclose	Needs _fclose
	Y	feof	
	Y	ferror	
	I	fflush	Needs _write
	I	fgetc	Needs _read

File	Imple–mented	Routine name	Description / Reason
	I	fgetpos	Needs _lseek
	I	fgets	Needs _read
	I	fopen	Needs _fopen
	I	fprintf	Needs _write
	I	fputc	Needs _write
	I	fputs	Needs _write
	I	fread	Needs _read
	I	freopen	Needs _fclose/_fopen
	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
	L	remove	
	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	
	I	_close	Low level file close routine
	I	_fopen	Low level file open routine
	I	_lseek	Low level file positioning routine
	I	_open	Low level file open routine
	I	_read	Low level input routine
	I	_write	Low level output routine

File	Imple–mented	Routine name	Description / Reason
stdlib.h	Y		
	Y	abort	Calls _exit() in cstart
	Y	abs	
	Y	atexit	
	Y	atof	
	Y	atoi	
	Y	atol	
	Y	bsearch	
	Y	calloc	
	Y	div	
	Y	exit	Calls _exit() in cstart
	Y	free	
	L	getenv	No OS present
	Y	labs	
	Y	ldiv	
	Y	malloc	
	Y	qsort	
	Y	strtod	
	Y	strtol	
	Y	strtoul	
	Y	rand	
	Y	realloc	
	Y	srand	
	L	system	No OS present
	L	mblen	wide chars not supported
	L	mbstowcs	wide chars not supported
	L	mbtowc	wide chars not supported
	L	wcstombs	wide chars not supported
	L	wctomb	wide chars not supported

File	Imple–mented	Routine name	Description / Reason
string.h	Y		
	Y	memchr	
	Y	memcmp	
	Y	memcpy	
	Y	memmove	
	Y	memset	
	Y	strcat	
	Y	strchr	
	Y	strcmp	
	L	strcoll	wide chars not supported
	Y	strcpy	
	Y	strcspn	
	Y	strerror	
	Y	strlen	
	Y	strncat	
	Y	strncmp	
	Y	strncpy	
	Y	strpbrk	
	Y	strrchr	
	Y	strspn	
	Y	strstr	
	Y	strtok	
	L	strxfrm	wide chars not supported
time.h	Y		real time clock not supported, but the DSP timer is used to maintain relative time
	Y	asctime	
	Y	clock	
	Y	ctime	
	Y	difftime	
	Y	gmtime	
	Y	localtime	
	Y	mktime	
	Y	strftime	
	Y	time	

6.6.2 C LIBRARY INTERFACE DESCRIPTION

_close

```
#include <stdio.h>
int _close( int fd );
```

Low level file close function. _close is used by the function fclose. The given file descriptor should be properly closed, any buffer is already flushed. The delivered library is targeted at TASKING or Motorola FSS (File System Simulation). The source file contains versions for TASKING and Motorola FSS, and an 'empty' function. This function must be customized to perform I/O on different file systems. See the file `_close.c` in the `lib\src` directory for the example implementations of this low level close function.

_filbuf

```
#include <stdio.h>
int    _filbuf  ( FILE * );
```

Low level file input function. Read a character from a file, buffering when necessary. Filling the buffer is done through calls to _read.

Returns the character written, or EOF.

_flsbuf

```
#include <stdio.h>
int    _flsbuf  ( int, FILE * );
```

Low level file output function. Write a character to a file, flushing the buffer when necessary. Flushing is done through calls to _write.

Returns the character written, or EOF.

_fopen

```
#include <stdio.h>
FILE _fopen( const char *filename, const char *mode,
             FILE *iop );
```

Low level file open function. Opens a file with name `filename` and access type mode in `iop`. _fopen is used by the functions fopen and freopen, and in turn passes control to _open. Note that for text modes the file functions will work on character basis, whereas for binary modes they will work on memory–cell–wide objects (integers in most models). Files opened in binary mode will read/write full DSP words, whereas files opened in text mode will read/write the low 8 bits only.

Returns the descriptor of the file opened, or NULL if an error occurred.

_insize

```
#include <conio.h>
long _insize( int fd );
```

Low level function for investigating the size of stdin.

Returns queue size for stdin.

_ioread

```
int _ioread( FILE *fp );
```

Low level input function. This function reads a character from a file. _ioread is used by all input functions (scanf, getc, gets, etc.). The delivered library is targeted at TASKING or Motorola FSS (File System Simulation). The source file contains versions for TASKING and Motorola FSS, and an 'empty' function. This function must be customized to perform I/O on different file systems. See the file `_ioread.c` in the `lib\src` directory for the example implementations of this low level input function.

Returns the character read, or EOF if an error occurred.

_iowrite

```
int _iowrite( int c, FILE *fp );
```

Low level output function. This function writes a character to a file. _iowrite is used by all output functions (printf, putc, puts, etc.). The delivered library is targeted at TASKING or Motorola FSS (File System Simulation). The source file contains versions for TASKING and Motorola FSS, and an 'empty' function. This function must be customized to perform I/O on different file systems. See the file `_iowrite.c` in the `lib\src` directory for the example implementations of this low level output function.

Returns the character written, or EOF if an error occurred.

_lseek

```
#include <stdio.h>
long
_lseek( int fd, long offset, int origin );
```

Low level file positioning function. This function sets the file position of a file. _lseek is used by all file positioning functions (fgetpos, fseek, fsetpos, ftell, rewind). The delivered library is targeted at TASKING or Motorola FSS (File System Simulation). The source file contains versions for TASKING and Motorola FSS, and an 'empty' function. This function must be customized to perform I/O on different file systems. See the file `_lseek.c` in the `lib\src` directory for the example implementations of this low level file function.

Returns the new file position, or EOF if an error occurred.

_open

```
#include <stdio.h>
int _open( const char * filename, int flags );
```

Low level file open function. This function opens a file with name filename and access type flags. _open is called from the function _fopen and from system initialization (for stdin, stdout and stderr). With FSS (File System Simulation), this function opens a file on the host system. The source file contains versions for TASKING and Motorola FSS, and an 'empty' function. This function can be adapted to work on different file systems; the given stream should be properly opened.

Returns the descriptor of the file opened, or –1 if an error occurred.

_packsize

```
#include <string.h>
size_t _packsize( const char * p );
```

Returns the size of a string when it is packed.

_packstr

```
#include <string.h>
_packed char *_packstr( _packed char * p,
                const char * unp );
```

Pack string pointed to by *unp* in the buffer pointed to by *p*.

Returns a pointer to the packed string.

_pstr_get

```
char _pstr_get( _packed char * p,
                size_t idx );
```

This function should be used to obtain one character from a packed string. The packed string is supplied via the *p* argument. The index in the string is supplied via the *idx* argument. This index is the count in bytes starting at zero.

Returns the character at the given index in the packed string.

Example:

```
int printpstring( _packed char *p )
{
     int idx = 0;
     char c;
     while( c = _pstr_get( p, idx++ ) )
          putchar( c );
     return( idx );
}
```

See also section section 3.18 *Packed Strings*.

_pstr_put

```
void _pstr_put( _packed char * p,
                size_t idx, char c );
```

This function updates one character (one byte) in a packed string pointed to by *p*. The character is supplied in the argument *c* and the index in *idx*. This index is the count in bytes starting at zero.

Returns nothing.

Example:

```
#include <stdio.h>

_packed char p[10];

void main( void )
{
     int idx;
     char c = 'a';
     for ( idx = 0; idx < 10; idx++, c++ )
          _pstr_put( p, idx, c );
          // build string "abcd ..."
     printf( "%S\n", p );  // print packed string
}
```

See also section section 3.18 *Packed Strings*.

_read

```
#include <stdio.h>
size_t
_read( int fd, char *base, size_t size );
```

Low level block input function. This function reads a block of data from the given stream. It is used by all input functions. The delivered library is targeted at TASKING or Motorola FSS (File System Simulation). The source file contains versions for TASKING and Motorola FSS, and an 'empty' function. This function must be customized to perform I/O on different file systems. When not customized it will use _ioread().

Returns the number of characters read.

_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.

_unpackstr

```
#include <string.h>
char *_unpackstr( char * unp,
                  const _packed char * p );
```

Unpack string pointed to by *p* in the buffer pointed to by *unp*.

Returns a pointer to unpacked string.

_unpstrlen

```
#include <string.h>
size_t _unpstrlen( const _packed char *p );
```

Returns the length in number of characters of the packed string pointed to by *p*. This is the number of characters when the string would be unpacked.

_write

```
#include <stdio.h>
size_t
_write( int fd, char *base, size_t size );
```

Low level block output function. This function writes a block of data to the given stream. It is used by all output functions. The delivered library is targeted at TASKING or Motorola FSS (File System Simulation). The source file contains versions for TASKING and Motorola FSS, and an 'empty' function. This function must be customized to perform I/O on different file systems. When not customized it will use _iowrite().

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 $[-\pi/2, \pi/2]$.

atan2

```
#include <math.h>
double atan2( double y, double x );
```

Returns the result of: $\tan^{-1}(y/x)$ in the range $[-\pi, \pi]$ where `y/x` are coordinates. `x` and `y` cannot be both zero.

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.

See also ”free()”.

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 number of seconds since the last reset, assuming a 66 MHz cpu.

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 calender 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. By including a ’8’ the file will also be opened in binary mode but the read/write uses only the lower 8 bits as in text mode. 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.

Files opened in binary mode will read/write full DSP words (except when you used '8'), whereas files opened in text mode will read/write the low 8 bits only.

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` and 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 0.5·2^3. The function returns 0.5, and 3 is stored in var.

See also "ldexp()".

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 calender 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.

iscntrl

```
#include <ctype.h>
int iscntrl( 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]).

isgraph

```
#include <ctype.h>
int isgraph( int c );
```

Returns a non–zero value when `c` is printable, but not a space.

islower

```
#include <ctype.h>
int islower( int c );
```

Returns a non–zero value when `c` is a lowercase character ([a–z]).

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 );
```

See also ”frexp()”.

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 calender 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 `log10(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.

Because of the low–level nature of this routine it will not work correctly with the **–Cc** switch (all functions compatible). With minor changes to adapt it to the compatible calling convention this can be changed, but it cannot run under both calling conventions. On the DSP563xx, this routine will also fail when the stack extension is in use and the required stack level is located in the extension area.

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.

See also ”free()”.

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, S	char * or _packed char * respectively, the characters from the string or packed string respectively are printed until a NULL character is found. When the given precision is met before, printing will also stop
f	double
e, E	double; [–]*m.ddddd*e±*xx* or [–]*m.ddddd*E±*xx*, where the number of *d*'s is specified by the precision.

Character	Printed as
g, G	double; uses %e or %E if the exponent is less than –4 or greater than or equal to the precision; otherwise uses %f.
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 6–2: Printf conversion characters

To print fractional numbers, the value can be cast to a `float` and printed as a `float`. For single–precision fractional numbers the printed value is exact, for `long _fract` it is rounded. If you need more precision when printing `long _fract` values you can print them in hexadecimal form:

```
#include <stdio.h>

_fract fvalue = 0.987654321;
long _fract lfvalue = (float)1.0/3;

void main(void)
{
    printf("fvalue is: %8.6f\n", (float) fvalue);
    printf("lfvalue is: %8.6f\n", (float) lfvalue);
    printf("lfvalue in hex is: %12lx\n", _lfract2long(lfvalue));
}
```

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`.

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.

See also ”srand()”.

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()”.

In this implementation the returned address will be the same as the original address if realloc() is successfully used to decrease the size of a buffer. The minimal amount of extra memory is allocated when a buffer size is increased.

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.


See also ”free()”.

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.

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 build 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 precede 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 precede 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).
s, S	char * or _packed char * respectively, a string of non white space characters. The argument should point to an array of characters or packed characters respectively, large enough to hold the string and a terminating NULL character.
f	float
e, E	float; [–]*m.dddddd*e±*xx* or [–]*m.dddddd*E±*xx*, where the number of *d*'s is specified by the precision.
g, G	float; uses %e or %E if the exponent is less than –4 or greater than or equal to the precision; otherwise uses %f.
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 6–3: 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)

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.

Because of the low–level nature of this routine it will not work correctly with the **–Cc** switch (all functions compatible). With minor changes to adapt it to the compatible calling convention this can be changed, but it cannot run under both calling conventions. On the DSP563xx, this routine will also fail when the stack extension is in use and the required stack level is located in the extension area.

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
SIGILLillegal 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.

See also "rand()".

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 where 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. Its 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.

6.6.3 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 precision specifier '.' cannot be used

The same applies to all `scanf` type functions, which all call the function `_doscan()`.

The formatters included in the libraries are LARGE. You can select different formatters by linking separate objects of `_doscan()` and `_doprint()` with your application. The following objects are included:

For the DSP563xx:

`lib/563xx/libc16`
- `_doprnts.obj` `_doprint()`, 16–bit model, SMALL formatter
- `_doprntm.obj` `_doprint()`, 16–bit model, MEDIUM formatter
- `_doscans.obj` `_doscan()`, 16–bit model, SMALL formatter

`lib/563xx/libc1624`
- `_doprnts.obj` `_doprint()`, 16/24–bit model, SMALL formatter
- `_doprntm.obj` `_doprint()`, 16/24–bit model, MEDIUM formatter
- `_doscans.obj` `_doscan()`, 16/24–bit model, SMALL formatter

`lib/563xx/libc24`
- `_doprnts.obj` `_doprint()`, 24–bit model, SMALL formatter
- `_doprntm.obj` `_doprint()`, 24–bit model, MEDIUM formatter
- `_doscans.obj` `_doscan()`, 24–bit model, SMALL formatter

For the DSP566xx:

`lib/566xx/libc6`
- `_doprnts.obj` `_doprint()`, SMALL formatter
- `_doprntm.obj` `_doprint()`, MEDIUM formatter
- `_doscans.obj` `_doscan()`, SMALL formatter

For the DSP5600x:

`lib/5600x/libcm`
- `_doprnts.obj` `_doprint()`, mixed model, SMALL formatter
- `_doprntm.obj` `_doprint()`, mixed model, MEDIUM formatter
- `_doscans.obj` `_doscan()`, mixed model, SMALL formatter

`lib/5600x/libcr`
- `_doprnts.obj` `_doprint()`, reentrant model, SMALL formatter
- `_doprntm.obj` `_doprint()`, reentrant model, MEDIUM formatter
- `_doscans.obj` `_doscan()`, reentrant model, SMALL formatter

`lib/5600x/libcs`
- `_doprnts.obj` `_doprint()`, static model, SMALL formatter
- `_doprntm.obj` `_doprint()`, static model, MEDIUM formatter
- `_doscans.obj` `_doscan()`, static model, SMALL formatter

Example (PC):

```
cc563 -M24 hello.obj c:\c563\lib\563xx\libc24\_doprntm.obj
```

6.7 RUN-TIME LIBRARY

Some basic operations take up a lot of memory if generated as inline code. For example, code to perform a 48–bit integer division. The compiler can generate code with calls to library functions for these situations. This depends on the optimization level (optimize for size or speed). These routines are stored in the run–time library. The run–time library also contains some default versions of routines required by the C library (e.g., `exit`).

Because **c563** generates assembly code (and not object code) it prepends an 'F' for the names of (public) C variables to distinguish these symbols from internal symbols. So if you use a function name starting with an underscore, the assembly label for this function will start with 'F_'. Run–time library functions have no prepended 'F', they have a prepended 'R'.

The routines from the run–time library are stored in special sections (`.rttext`, `.petext` and `.zerotext`) to allow precise locating of these routines.

6.8 FLOATING POINT LIBRARY

The DSP56xxx does not have a hardware floating point unit, so floating point calculations must be emulated in software. The floating–point routines are stored in the floating–point library. For a detailed description of the floating–point format, see chapter 7. The floating point library is reentrant and independent of the memory model, except for the data precision (16 or 24–bit). All routines from the floating–point library are stored in a special section, `.fptext`, to allow precise locating of these routines. Floating–point library functions have no prepended 'F', their names start with 'Rfp'.

CHAPTER 7

RUN–TIME ENVIRONMENT

TASKING

1 CHAPTER

7.1 STARTUP CODE

When linking your C modules with the C library, you can automatically link the object module containing the C startup code. The name of this module depends on the selected execution environment (target selection in EDE, or command line option **–T** of the control program). A large set of predefined target modules is supplied already. See below for information how to create support files for a user–designed target board. All of these modules define a list of values used to initialize the hardware correctly, and include the file `cstart.inc` which contains the actual code.

Because these modules specifiy the run–time environment of your DSP56xxx C application, you might want to edit them to match your needs. Therefore, these modules and the file `cstart.inc` are delivered in assembly source in the `src` subdirectory of the `lib` directory. Typically, you will copy the startup code file `start.asm` (and rename it to, for example, `mystart.asm`) and copy the file `cstart.inc` to your own directory and edit it. The file `cstart.inc` contains macro preprocessor symbols to tune the code. Include this file in your own startup code. You can check the predefined target startup code files *target_name*`.asm` to see what information you can put in your own startup code file.

To use the changed file, you must add it to the file list (EDE) or makefile (command line). You must also select ”User supplied, no library startup code” (EDE) or **–T** on the command line to avoid linking with a library startup file. If you would link your own startup file with a library startup file, you would get conflicts on all symbols in the startup files.

The invocation (using the control program) is:

```
cc56 –c mystart.asm –DNODSTACK
```

for the static model (DSP5600x only), or

```
cc563 –c mystart.asm
```

for all other models.

In the C startup code an absolute code section is defined for setting up the power on vector and the DSP56xxx C environment. The power–on vector contains a jump to the F_START label. This global label must not be removed, since it is referred to by the C compiler. It is also used as the default start address of the application (see the `start` keyword in the locator description language DELFEE). The code space for all non–used interrupt vectors may be occupied by small user code sections. When this is not desirable, you should include default versions of the interrupt vectors by including the startup code in your project file list and removing the comments in the `cstart.inc` file. The default interrupt vectors will jump to the `abort()` function when they are inadvertently called.

The stack is defined in the locator description file (`.dsc` in directory `etc`) with the keyword `stack`, which results in a section called `stack`. Except in the static model of the DSP5600x, a user stack is always required.

See the section *Stack* for detailed information on the stack.

The stack pointer can be switched between r6 and r7. To initialize the correct address register with the base of the stack the linker retrieves a module from the library that defines the symbol R_STKINIT that contains the opcode for the register move. This avoids the need for two versions of every startup definition file.

The heap is defined in the description file with the keyword `heap`, which results in as section called `heap`.

See the section *Heap* for detailed information on heap management.

An important task of the startup code is to configure the interface of the DSP to external memory. The following registers are used for this:

```
5600x:         BCR
563xx: SR OMR BCR AAR0 AAR1 AAR2 AAR3 DCR
566xx: SR OMR BCR
```

A download of a program that uses external memory can only take place after the external memory interface has been initialized correctly. CrossView Pro uses the values of symbols with coded names R_*xxxx*VALUE (e.g., R_AAR0VALUE) for the above registers to do this. The startup code assembly file defines and exports these symbols; the code in the `cstart.inc` file sets them during program start, so the code can also run without CrossView Pro (e.g., from an EPROM).

The startup code also takes care of initialized C variables, residing in the different RAM areas. Each memory type has a unique name for both the ROM and the RAM section. The startup code copies the initial values of initialized C variables from ROM to RAM, using these special sections and some run–time library functions. When initialization of C variables is not needed, you can translate the startup file with **–DNOCOPY**. See also the `table` keyword in the locator description language DELFEE.

When everything described above has been executed, your C application is called, using the global label `Fmain`, which has been generated by **c563** for the C function `main()`.

When the C application 'returns', which is not likely to happen in an embedded environment, the program ends with a DEBUG instruction, using the assembly label `F_exit`. When using a debugger, it can be useful to set a breakpoint on this label, indicating that the program has reached the end, or that the library function `exit()` has been called.

The following macros can be used to control the functionality of `cstart.inc`:

Macro	Description
NODSTACK	Do NOT initialize the dynamic stack pointer (only in static model for the DSP5600x).
NOESTACK	Do NOT initialize stack extension hardware (DSP563xx only).
NOCACHE	Do NOT enable cache (DSP563xx only).
NOCOPY	Do NOT produce code to clear BSS sections and initialize DATA sections.
NOARGV	Do NOT produce code to provide a dummy argument vector to function main().
PLLVALUE=*val*	Set clock phase locked loop register to *val*. If not defined, the PLL is not initialized.
BCRVALUE=*val*	Set external memory waitstates to *val*. If not defined, zero wait states is selected for all memory (not on DSP563xx).
LOW_DYNAMIC	Select 16–bit arithmetic mode (DSP563xx only).
NARROW_BUS	Select 16–bit address bus mode (DSP563xx only).
USP=R6	Select register R6 for user stack pointer.

Table 7–1: Macros used in `cstart.inc`

The `cstart.inc` file declares some labels external, that will be resolved by the linker. The label '`start`' will link the object from the file *`start`*`.asm` that contains a jump to the label `F_START` and will generate the start vector in the interrupt vector table. Likewise, some labels named **irq***xx*, with *xx* in the range 1 to 63 are declared external or can be declared external and will create default interrupt vectors in the interrupt vector table. These default vectors will always jump to abort(), because they indicate that either a system error has occurred (e.g. hardware stack overflow), or that an unexpected interrupt has occurred. By default most interrupt vectors are left uninstalled in order to have as much internal program memory available as possible. If an interrupt is created in the C program this will overrule the default vector by defining the label **irq***xx*; if you create interrupt handlers in assembly you can suppress the default vector in the same way. The symbols `null_x` and `null_y` are declared external in order to link a one–word absolute section at X:$0 and another one at Y:$0. This is done to allow safe checking of NULL pointers: by occupying these addresses we know for sure that if a pointer returns NULL it indicates an error condition. The `_at()` modifier can still be used to force variables at address 0.

7.2 REGISTER USAGE

The compiler will use all available registers that fit for storing variables and intermediates. Registers used are A, B, X0, Y0, X1, Y1, R0–R7, N0–N7. The modifier registers M0–M7 are only used in conjunction with the associated R0–R7 register to act as a circular pointer. When not in use as a circular pointer they must remain in the reset state (–1, linear addressing). Of course registers PC, SR and CCR are used implicitly, as is the hardware stack.

7.3 CALLING CONVENTIONS

The compiler will try to use the available registers as efficiently as possible. The compiler uses a flexible register allocation scheme, which implies that any change to the C code may result in a different register usage.

For passing parameters to functions the compiler uses the following scheme:

- Arithmetic–type arguments: Float arguments are passed via AB, X and Y; long arguments are passed via A, B, X and Y; integers are passed via A, B, X0, Y0, X1 and Y1. The order of the arguments from left to right is A, B, X0, Y0, X1, Y1. Arguments requiring the whole X or Y register (e.g., a long) are allocated in the A, B, X and Y register before other arguments which only need X0, X1, Y0 or Y1.
- Structure–type arguments: Single–word structures are passed and returned in the same registers as integers, double–word structures in the same registers as floats. Longer structures are passed on the stack. The processor status flags are undefined upon return.
- Pointer–type arguments: Pointers are passed in registers R0, R4, R1, R5, R2, (R6), R3, (R7) (and the corresponding modifier registers in case of circular pointers). Except for the static model, R6 or R7 is not used because it is reserved as user stack pointer.
- Variable argument lists are passed via the stack. All other arguments, except the argument that immediately preceeds the variable argument list, are stored using default register parameter passing.

```
extern int f(int,...);
extern int g(int,int,...);
extern int h(int,int,int,...);

int a0;
int a1;
int a2;

void foo(void)
{
     f(a0, "a0 on stack as is this string");
     g(a0, a1, "a0 via accumulator – remainder on stack");
     h(a0, a1, a2, "a0 via A, a1 via B – remainder on stack");
}
```

- When there are too many arguments to be passed in the registers the remaining arguments are passed on the stack. If the `_asmfunc` qualifier is used, the compiler will issue an error message.

For C function return types, the following registers are used:

Return type	Register	Description
char	A	accumulator
short/int	A	accumulator
long	A	accumulator
_fract	A	accumulator
long _fract	A	accumulator
float	AB	floating point stack (see section 7.8 for floating point information)
pointer	R0	scratch address registers
spointer	R0	address registers
circular pointer	R0/M0	address registers

Table 7–2: Function return types

The address register R7 (or R6) is used as user stack pointer (not for the DSP5600x in the static memory model). R6 is used if the corresponding compatibility switch (**–Cr**) is selected.

During the execution of the called function ('callee') all registers can be used. The caller saves all registers that must be preserved over the function call ('save–by–caller' calling convention) with the exception of the modifier registers. Upon return, the callee must reset all modifier registers to linear addressing (except M0 if it is used to return a circular pointer). Of course the stacks must remain balanced between call and return.

In the compatible calling convention, entered with either the keyword `_compatible` or with the global option **–Cc**, the parameter passing changes to the convention used by the Motorola C compiler. The first two parameters are passed in registers A and B and the return value is always in register A. Upon return of a value the processor status flags are set by the callee according to the return value. The registers Y0, Y1, R2, N2, R3, N3, R7 and N7 are preserved by the callee. To be completely compatible with the Motorola compiler the stack pointer must be chosen to be R6 (**–Cr**) and the right default memory space must be selected.

The compiler uses a convention to pass parameters in registers and on the stack, to save certain registers over a function call and to return values in certain registers or on the stack. In the `_compatible` calling convention several registers must be preserved in a function. The set of registers is different for the members of the DSP56xxx family, as shown in the table below.

Family Member	Parameter Registers	Registers to be preserved by callee	Return Register
DSP5600x	None (all on stack) When returning a struct/union, A points to the address to store it.	B1, B0, X1, X0, Y1, Y0, R0..R5, R7 (R7 is replaced by R6 if R7 is used for user stack pointer)	A, except when returning a struct/union; in that case A points to the address where it is stored
DSP563xx DSP566xx	A,B (rest on stack) When returning a struct/union, R7 points to the address to store it.	Y1, Y0, R2/N2, R3/N3, R7/N7 (R7/N7 are replaced by R6/N6 if R7 is used for user stack pointer)	A, except when returning a struct/union; in that case R7 points to the address where it is stored

Table 7–3: Compatible calling convention

In the save–by–callee calling convention, selected with the `_callee_save` qualifier, all registers are preserved by the callee. The compiler tries to limit the number of registers used in the function when this has been selected. The parameter passing convention is the same as the default. Although this can have advantages in some cases, the default calling convention in combination with the flexible register allocation guarantees the best performance.

An example of the `_callee_save` qualifier is listed below.

```
#ifndef _CALLEE_SAVE
        #define _who_saves
#else
        #define _who_saves _callee_save
#endif

int i;

_who_saves void __inc(void)
{
        i++;
}
```

Default it will compile with the standard save–by–caller calling convention and the assembly will look as follows:

```
F__inc:  move    x:Fi,r3
         move    (r3)+
         move    r3,x:Fi
         rts
```

But when you compile with the define _CALLEE_SAVE, the callee itself becomes responsible for preserving R3 and the assembly will change to:

```
F__inc:  move    (r7)+
         move    r3,x:(r7)
         move    x:Fi,r3
         move    (r3)+
         move    r3,x:Fi
         move    x:(r7)-,r3
         rts
```

7.4 SECTION USAGE

c563 uses a number of sections. For a section the compiler generates an ORG directive in the assembler output file. The following list gives an overview of sections that may be generated by **c563**:

Section Name	Possible Attributes	Comment
.p[n][i|e]text	near, internal, external	code
.fptext		code from the floating point library
.l[n][i|e]const	near, internal, external	constant initialized L data, not copied from copy table
.p[n][i|e]const	near, internal, external	constant initialized P data, not copied from copy table
.x[n][i|e]const	near, internal, external	constant initialized X data, not copied from copy table
.y[n][i|e]const	near, internal, external	constant initialized Y data, not copied from copy table
.lovl		overlayed area in L memory for non–reentrant (static) functions
.povl		overlayed area in P memory for non–reentrant (static) functions

Section Name	Possible Attributes	Comment
.xovl		overlayed area in X memory for non–reentrant (static) functions
.yovl		overlayed area in Y memory for non–reentrant (static) functions
.l[n][i|e]data	near, internal, external	initialized L data, copied from copy table in P
.p[n][i|e]data	near, internal, external	initialized P data, copied from copy table in P
.x[n][i|e]data	near, internal, external	initialized X data, copied from copy table in P
.y[n][i|e]data	near, internal, external	initialized Y data, copied from copy table in P
.l[n][i|e]bss	near, internal, external	cleared L data
.p[n][i|e]bss	near, internal, external	cleared P data
.x[n][i|e]bss	near, internal, external	cleared X data
.y[n][i|e]bss	near, internal, external	cleared Y data

Table 7–4: Section names

The [n][i|e] part of the section names has the following meaning:

n near

i internal

e external

ni near, internal

ne near, external

7.5 COMPILER HARDWARE ENVIRONMENT

In the compiled code it is assumed that all processor mode registers like OMR, SR and SP are in the state they have after reset, except when they are set up differently in the startup code. Although other settings of these registers may work just as well, care is needed. In any case, different settings must be compatible with the hardware connected and the memory layout selected. For some changes the startup code and/or the locator description files must be updated as well (e.g. if you want to place the stack extension in Y memory).

7.5.1 OPERATING MODE REGISTER

OMR bit 0–3: chip operating mode.
Normally the compiled code runs in expanded mode (default), but different settings may work. The interrupt vectors and startup vector are placed from address 0 by the compiler, so some modes will need special precautions.

OMR bit 16–20: stack extension settings (DSP563xx only).
Do not change these bits after the startup code.

7.5.2 STATUS REGISTER

The compiler assumes all status register bits are in the reset state, unless they have been set up otherwise in the startup code (`lib/src/cstart.inc`). This does not mean that the compiled code does not work with other settings, but care is needed. In summary (some of these bits are not present on all DSP56xxx family members):

SR bit 0–7: condition code register.
These bits are changed by just about any instruction. The L–bit (limiting occurred) is also used by the floating point library to indicate an overflow/underflow condition.

SR bit 8/9: interrupt mask.
Can be set to any value, but if the system timer is used (for `delay()` for instance) its function may be affected (see `lib/src/clock.c`). No other library functions use interrupts.

SR bit 10/11: scaling mode.
Never used in compiled code. These bits can be changed locally in assembly code, but they must remain zero during execution of compiled code. These bits are cleared during interrupts and restored afterwards.

SR bit 12: Reserved.

SR bit 13: Sixteen bit compatibility mode (DSP563xx only).
Set up in 16–bit compilation model. This bit can be changed locally in assembly code, but must be restored to its startup value during execution of compiled code.

SR bit 14: Double precision multiply mode.
Never used in compiled code. This bit can be changed locally in assembly code, but must remain zero during execution of compiled code.

SR bit 15: DO–loop flag.
Never used in compiled code. Not very useful, do not touch in compiled code.

SR bit 16: DO–forever flag.
Never used in compiled code. Not very useful, do not touch in compiled code.

SR bit 17: Sixteen bit arithmetic mode.
Used in 16–bit compiled code. This bit can be changed locally in assembly code, but must be restored to its startup value during execution of compiled code. This bit is cleared during interrupts and restored afterwards; in the 16–bit models the compiler will force it on again in interrupt routines.

SR bit 18: Reserved.

SR bit 19: Cache Enable.
Switched on in startup code normally. Can be changed in the application, but the cache intrinsic functions cannot be called when the cache is switched off. Care must be taken to avoid using the cache area for code when switching the cache mode at run–time.

SR bit 20: Arithmetic saturation mode.
Never used in compiled code. This bit can be changed locally in assembly code, but must remain zero during execution of compiled code. The overflow behavior of integers is affected by this bit, so compiled code may behave strangely when this bit is set. Long division might not work anymore either. As register–register and register–memory moves of (long) `_fract`s result in saturation anyway, it is not very useful in compiled code. This bit is not changed during interrupt routines; if it is set anywhere in the code, it must be turned off in all C compiled interrupt routines using inline assembly.

SR bit 21: Rounding mode.
Never used in compiled code. This bit can be changed in compiled code to get the effect described in the processor manual. This bit is cleared during interrupts and restored afterwards.

SR bit 22–23: Core priority.
Never used in compiled code. These bits can be changed in compiled code to get the effect described in the processor manual.

The execution of compiled interrupt routines must be avoided as well where the summary mentions that bits must remain unchanged in compiled code. So, the interrupts may have to be switched off locally as well.

7.5.3 OTHER REGISTERS

LA, LC, SP: do not change their value to avoid disrupting the program flow.

SSL, SSH: the hardware stack is used to store return addresses and hardware loop information. If required, return addresses are popped from the hardware stack to avoid overflows (DSP5600x). The hardware stack can be used as long as it is not exhausted and stack balance is preserved. In most cases using the user stack is a faster and easier method.

EP, SZ, SC (DSP563xx only):
the first two registers are set up in the startup code if the hardware stack extension is enabled and must not be changed during program execution. The stack count register could be used by a task switching kernel, but is of little use otherwise and should be left untouched.

7.6 STACK

The DSP563xx/DSP566xx processors have a **system stack** (hardware stack) with 16 locations, 15 for the DSP5600x, divided in a system stack high word (SSH) and system stack low word (SSL). This system stack is used by the DSP for function calls, long interrupts and program looping (DO and REP loops). For a C program the system stack size is not sufficient. A deeply nested C program that also uses DO and REP loops and where long interrupts may occur, would soon generate a system stack overflow, because the system stack is also used by the DSP for return addresses of function calls. Therefore, all memory models except the static model (**c56** only) use a **user stack** to store C variables, common subexpressions, temporary results, and to pass parameters when all registers are occupied. The mixed model for functions that are not explicitly declared `_reentrant`, and the static model use overlayable sections for these purposes.

When hardware stack extension is not available (DSP5600x) or is not enabled, the hardware stack size is very limited. Therefore the changed hardware stack contents is transferred to the user stack during the function execution. The compiler generates code that pops the return address from the system stack and pushes it on the user stack (reentrant function) or saves it in a **static area** on function entry (static function). Before returning from the function it reverses this operation. A leaf function does not move the return address from the system stack.

The following diagrams show the structure of the stack. The first diagram reflects the system stack. The second diagram shows the user stack when using reentrant functions.

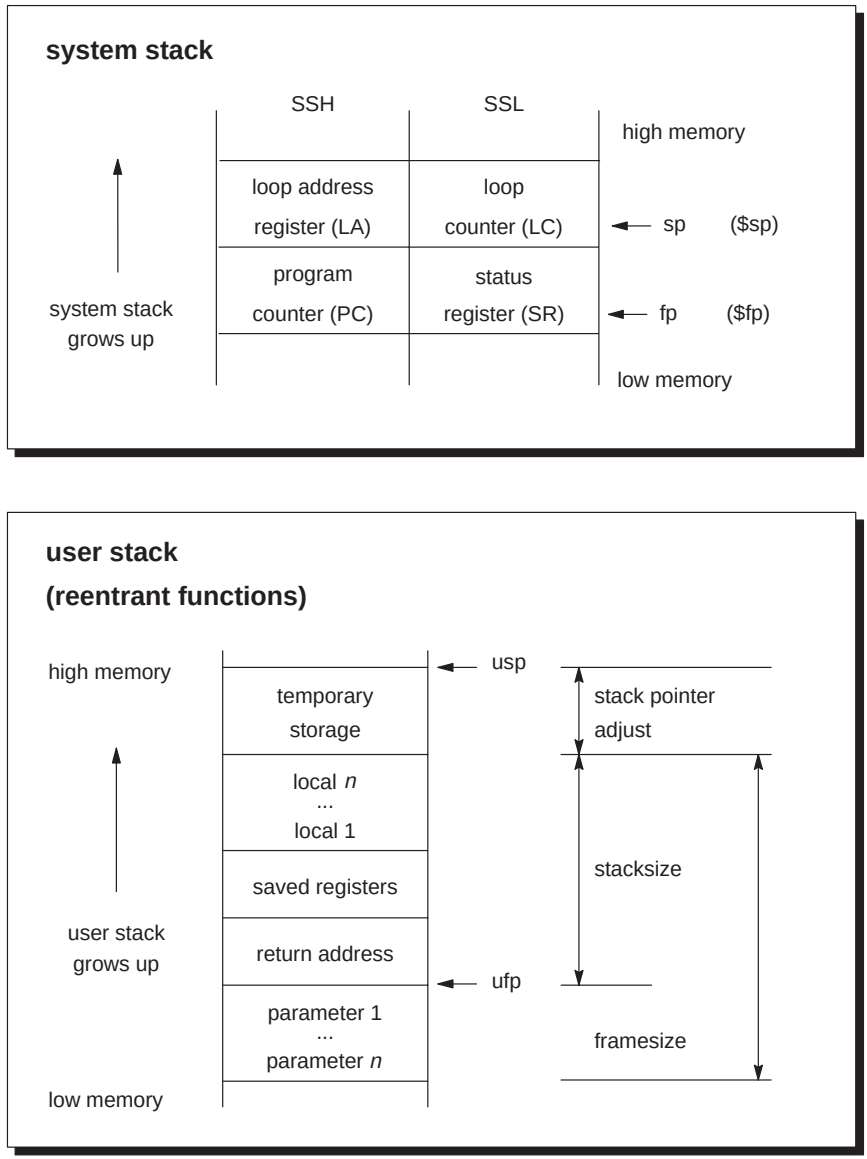


Figure 7–1: Stack diagrams

The user stack is defined in the locator description file (`.dsc` in directory `etc`) with the keyword `stack`, which results in a section called `stack`. The description file tells the locator where to allocate the user stack.

The user 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 RAM for the stack, as done in the startup code. You can use the locator defined labels `F_lc_bs` and `F_lc_es` in your application to retrieve the begin 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 `F_lc_bs` or `F_lc_es`. (This is usually done in the startup code.) Remember that there must be enough space allocated for the stack, which grows upwards.

For non–reentrant functions, (non–register) automatics and (non–register) parameters are allocated in a **static area** and therefore do not use any stack space.

For reentrant functions, a **user stack** is used in memory. Automatics and parameters are all accessed using a user stack pointer register, allocated as a 16–bit pointer (`USP`). The stack pointer `USP` points to the last occupied location on the stack. If the compatibility option **–Cs** is used, the stack pointer `USP` points to the first free location on the stack. The stack frame also contains a so–called user frame pointer (`UFP`). The saved registers are also accessed using a user stack pointer. The user stack pointer (`USP`) is maintained in register R7 or R6 if the compatibility option is used. The stack must be placed in default memory, or at least contain default memory, so L memory can be used for it when default memory is X or Y.

The `UFP` is always relative to the `USP`. To save registers the `UFP` is not maintained in a register, but is calculated from the `USP` with an offset.

7.6.1 STACK EXTENSION

Stack extension is a mechanism that allows larger stack sizes than supported by the internal hardware mechanism. When stack extension is enabled and the internal hardware stack has reached its maximum capacity, the Least Recently Used (LRU) internal hardware stack location is copied to data memory to create a new stack entry on the internal stack.

The toolchain works from the assumption that stack extension is required and initializes EP and OMR in the startup code. Locator label `F_lc_ub_se` is loaded into EP and afterwards stack extension is enabled from the OMR.

You can set the size of the stack extension as follows:

Select the `Project | Project Options...` menu item. Expand the `Linker/Locator` entry and select `Control File`. Type the size of the stack extension in the `Stack extension size` field.

Or with the locator command line option:

–emSESIZE=*size*

The locator uses this macro to preprocess the following line in the description file (which results in the locator label `F_lc_ub_se`):

```
reserved label=se length=SESIZE;
```

If you do not want to use stack extension, you must add the macro NOESTACK to the assembler preprocessor options and make sure the startup code is added to your project. In this case you need to be sure that there will not be more than 16 function calls or do–loops; the number of available internal hardware stack entries.

To avoid the calculation for this, you can also choose not to use the stack extension:

Select the `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Code Generation`. Disable the `Use hardware stack extension` check box.

Or with the compiler command line option:

–Mn

This 'ignoring' of the stack extension does not set a control register bit. Only the function return address is saved/restored to/from the *user stack*. (see section 7.6, *Stack*).
The effect is that the internal hardware stack is bypassed for function calls and as such cannot overflow. Do–loops cannot be bypassed but you can limit them:

Select the `Project | Project Options...` menu item. Expand the `C Compiler` entry and select `Code Generation`. Disable the `Use hardware stack extension` check box and type a number in the `Max. hardware stack use outside interrupt functions` field.

Or with the compiler option:

–L*number*

Ignoring the hardware stack extension decrease execution speed because user calls must be saved and restored on the user stack as well. Furthermore, nested hardware do–loops become restricted. Since function calls are being preserved, the user stack size will naturally increase, as will code size. All this pleads for using the hardware stack extension.

7.7 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 default memory; it cannot be placed in a different memory type because the library functions handling it rely on the memory type. 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 default 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 X memory for the heap. The locator defined labels `F_lc_bh` and `F_lc_eh` (begin and end of heap) are used by the library function `sbrk()`, which is called by `malloc()` when memory is needed from the heap.

7.8 FLOATING POINT

This section describes the definition and implementation of the TASKING Software Floating Point Library for the Motorola DSP56xxx Family of Digital Signal Processors.

7.8.1 SOFTWARE FLOATING POINT IMPLEMENTATION

7.8.1.1 CHARACTERISTICS OF FLOATING TYPES

There are three floating types defined by the ANSI C standard document, designated as `float`, `double` and `long double`. The characteristics for the `double` and `long double` types are equal to the `float` type, as described in the standard definition include file `<float.h>`.

7.8.1.2 FLOATING POINT CONSTANTS

Floating point constants conform to the ANSI C standard, except that an unsuffixed floating point constant has type `float`. If suffixed by the letter `f` or `F`, it has type `float`. If suffixed by the letter `l` or `L`, it also has type `float` because the characteristics for the `double` and `long double` types are chosen to be equal to the `float` type. Floating point constants in the range `<-1,1>` are interpreted as a fractional type (switchable with the compiler **–AF** option). Semantics, the type of a float constant is the first in which its value can be represented, first `_fract` then `float`. This allows fixed point arithmetic with fractional constants without suffixes. See also section 3.3.1 *The Fractional Type***.**

7.8.1.3 USUAL ARITHMETIC CONVERSIONS

Promotions conform to the ANSI C pattern of usual arithmetic conversions. This pattern is extended for `_fract` and `long _fract`.

First, if either operand has a fractional type and the other operand has a non–fractional type, then the operand with the fractional type is converted to the non–fractional type.

Otherwise, if either operand has type `long _fract`, the other is converted to `long _fract`.

Otherwise, both operands have type `_fract`.

Remember that floating point constants in `<-1,1>` are interpreted as a fractional type. See previous section *Floating Point Constants*.

7.8.1.4 SINGLE PRECISION FLOATING POINT FORMAT

Floating point number – (m,e) including mantissa sign

Decimal value = $m * (2^{e - fbias})$

Bit number	23 0	23 0
Binary encoding	s.mmm.mmmm.mmmm.mmmm.mmmm.mmmm	0000.0000.0000.0000.eeee.eeee
Bit Weight (2^n)	0 −1 −23	7 0
s = sign bit, m = mantissa bit, e = exponent bit		

Table 7–5: 2–Complement Format for 24–bit data models

Bit number	15 0	15 0
Binary encoding	s.mmm.mmmm.mmmm.mmmm	0000.0000.eeee.eeee
Bit Weight (2^n)	0 −1 −15	7 0
s = sign bit, m = mantissa bit, e = exponent bit		

Table 7–6: 2–Complement Format for 16–bit data models

m = 24–bit mantissa or 16–bit mantissa (16 bit models) (two's complement, normalized fraction). 23–bit or 15–bit (16–bit models) mantissa precision plus 1–bit mantissa sign gives precision of approximately 7 or 4 (16–bit models) decimal digits. A minimum of 6 decimal digits is prescribed by the ANSI C standard for single precision floating point. The 24–bit mantissa (24–bit models) or 16–bit mantissa (16–bit models) was chosen to maximize precision with efficient use of the MPY and MAC instructions. A hidden leading 1 is not implemented in this format.

e = 8–bit exponent (unsigned integer, biased by fbias = +127) stored as a 24–bit or 16–bit (16–bit models) unsigned integer with 16 or 8 (16–bit models) leading zeros.

Largest positive mantissa		
24–bit data models	$7FFFFF	+0.9999998807907104492187 5
16–bit data models	$7FFF	+0.99996948242875
Smallest positive mantissa		
24–bit data models	$400000	+0.5
16–bit data models	$4000	+0.5
Floating point zero mantissa		
24–bit data models	$000000	0.0
16–bit data models	$0000	0.0
Smallest negative mantissa		
24–bit data models	$BFFFFF	–0.5000001192092895507812 5
16–bit data models	$BFFF	–0.500030517578125
Largest negative mantissa		
24–bit data models	$800000	–1.0
16–bit data models	$8000	–1.0
Reserved mantissas		
24–bit data models	$000001 through $3FFFFF $C00000 through $FFFFFF	
16–bit data models	$0001 through $3FFF $C000 through $FFFF	

Table 7–7: Supported mantissas

All reserved mantissas are illegal since they represent denormalized mantissas. Denormalized numbers are not supported.

Assumed fixed point exponent		2^{+0} = +1.0
24–bit data models	$00007F	
16–bit data models	$007F	
Smallest exponent		2^{-127}
24–bit data models	$000000	
16–bit data models	$0000	
Largest exponent		2^{+128}
24–bit data models	$0000FF	
16–bit data models	$00FF	
Reserved exponents		
24–bit data models	$000100 through $FFFFFF	
16–bit data models	$0100 through $FFFF	

Table 7–8: Supported exponents

If bit weight 2^8 is set, exponent overflow has occurred.

If bit weight 2^9 is set, exponent underflow has occurred.

No distinct exponents are reserved for plus infinity, minus infinity, Not–a–Number (IEEE NaN), minus zero or denormalized numbers.

7.8.1.5 SINGLE PRECISION FLOATING POINT NUMBER RANGE

Floating point number	Mantissa	Exponent	Decimal Value
Largest positive			
24–bit data model	$7FFFFF	$0000FF	+3.402823E+38
16–bit data model	$7FFF	$00FF	+3.403E+38
Smallest positive			
24–bit data model	$400000	$000000	+2.938736E–39
16–bit data model	$4000	$0000	+2.939E–39
Floating point zero			
24–bit data model	$000000	$000000	+0.0
16–bit data model	$0000	$0000	+0.0
Smallest negative			
24–bit data model	$BFFFFF	$000000	–2.938736E–39
16–bit data model	$BFFF	$0000	–2.939E–39
Largest negative			
24–bit data model	$800000	$0000FF	–3.402823E+38
16–bit data model	$8000	$00FF	–3.403E+38

Table 7–9: Floating point number range

Note that the two's complement mantissa does not have equal positive and negative ranges. Only sign–magnitude formats possess this property. These ranges should be checked after most arithmetic operations.

7.8.1.6 COMPARISON TO IEEE-754 STANDARD FOR BINARY FLOATING POINT ARITHMETIC

Since the IEEE Floating Point Arithmetic Standard is well publicized, it is useful to compare these two floating point formats. This floating point format is compared to the single precision IEEE format and it differs from the IEEE standard primarily in its handling of floating point exceptions. Other differences are noted in the table below. Conversion between the IEEE standard format and this format is straight–forward.

Characteristic	2–Complement Format	IEEE Format
Mantissa Precision 24–bit models 16–bit models	 23 bits 15 bits	24 bits
Hidden Leading One	No	Yes
Mantissa Format 24–bit models 16–bit models	 24–bit Two's Complement Fraction 16–bit Two's Complement Fraction	23 bit Unsigned Magnitude Fraction
Exponent Width	8 bits	8 bits (single)
Maximum Exponent	+128	+127 (single)
Minimum Exponent	–127	–127 (single)
Exponent Bias	+127	+127 (single)
Format Width 24–bit models 16–bit models	 48 bits 32 bits	32 bits (single)
Rounding	Round to Nearest	Round to Nearest (default) Round to +/–Infinite Round to Zero
Infinity Arithmetic	Saturation Limiting	Affine Operations
Denormalized Numbers	No (Forced to Zero)	Yes (With Minimum Exponent)
Exceptions	Divide by Zero Overflow Underflow	Divide by Zero Overflow Underflow Invalid Operations Inexact Arithmetic

Table 7–10: IEEE 754 Comparison

As shown in the table, the 2–complement floating point mantissa precision is one bit (24–bit models) or nine bits (16–bit models) less than the IEEE single precision format. This is a result of using two's complement arithmetic.

If exponent overflow occurs, the result is limited to the maximum representable floating point number of the correct sign. If exponent underflow occurs, the result is limited to the minimum representable floating point number, which is zero. Although this format does not provide the arithmetic safety offered by the IEEE standard, it avoids extensive error checking and exceptions in favor of real–time execution speed and efficient implementation.

All exception conditions are handled "in–line" according to predefined rules. This accepts the fact that real–time systems have no choice but to provide an output with some amount of error if an exception occurs. It is not possible to stop execution until the application program determines a solution to the problem and fixes it.

One major difference is the use of affine arithmetic in the IEEE standard versus the use of saturation arithmetic in this 2–complement floating point format. Affine arithmetic gives separate identity to plus infinity, minus infinity, plus zero and minus zero. In operations involving these values, finite quantities remain finite and infinite quantities remain infinite. In contrast, this format gives special identity only to unsigned zero.

This format performs saturation arithmetic such that any result out of the representable floating point range is replaced with the nearest floating point representation. In the analog world, overflow is analogous to an analog opamp output clamping at the power supply rails.

The IEEE floating point standard provides extensive error handling required by affine arithmetic, denormalized numbers, signaling Not a Number (NaNs) and quiet NaNs. It postpones introducing computation errors by using internal signaling and user traps to process each exception condition. Computational errors will be introduced by the application program if the calculation is completed instead of aborting the program. This format introduces computation errors when an exception occurs in order to maintain real–time execution. An error flag (L bit in CCR) is set to inform the application program that an exception has occurred. This bit will remain set until reset by the application program.

7.8.1.7 SINGLE PRECISION FLOATING POINT MEMORY USAGE

The floating point mantissa and exponent may be stored in any locations in any memory space. The input and output register values are organized so that the long (L:) addressing mode may be used to load/store both the mantissa and exponent with one instruction. If the long addressing mode is used, the mantissa is in X memory and the exponent is in Y memory at the same address.

7.8.2 SOFTWARE FLOATING POINT INTERFACING

This section describes how a floating point operation has to be performed using the DSP56xxx floating point library functions. This contains the basic floating point operations, floating point accumulator format and floating point interface functions.

This section does not describe the algorithms used or the implementaion considerations, nor does it give a thorough explanation of the floating point routines themselves.

7.8.2.1 THE BASIC FLOATING POINT OPERATIONS

The basic operations of the floating point library are specified below and consist of arithmetic operations and conversion operations. These operations are implemented to support the 2–Complement Format.

For each floating point operation, function calls are specified in single precision. It is also specified whether a floating point function needs one, two or three floating point operands as input, an integer operand as input, a fractional operand as input and if it returns a floating, integer or fractional value.

Floating point operations are performed on so–called floating point accumulators. These accumulators are located in predefined registers and contain the floating point value(s) passed to the floating point operation. Section 7.8.2.2 *The Floating Point Accumulators* describes the format of these accumulators. The first floating point operand has to be loaded in accumulator fac and (if necessary) the second and third operand in accumulator ftm1 or ftm2. A floating point result always resides in the floating point accumulator fac. An integer, long or fractional operand is passed via accumulator register A.

The following tables list all supported functions and their function names.

The actual functions are prefixed by the letters **Rfp** to meet the compiler run–time library function calling convention.

Operation	Function	Input operand(s)	Result
Add	addf2	fac, ftm1	fac
Subtract	subf2	fac, ftm1	fac
Multiply	mulf2	fac, ftm1	fac
Divide	divf2	fac, ftm1	fac
Multiply–Accumulate +	macpf2	fac, ftm1, ftm2	fac
Multiply–Accumulate –	macnf2	fac, ftm1, ftm2	fac
Compare	cmpf2	fac, ftm1	CCR
Negate	negf2	fac	fac

Table 7–11: Floating point arithmetic operations

Operation	Function	Input operand(s)	Result
Signed Integer to Float	cif12	A	fac
Signed Long to Float	cif22	A	fac
Unsigned Integer to Float	cuf12	A	fac
Unsigned Long to Float	cuf22	A	fac
Fract to Float	crf12	A	fac
Long Fract to Float	crf22	A	fac
Float to Signed Integer	cfi21	fac	A
Float to Signed Long	cfi22	fac	A
Float to Unsigned Integer	cfu21	fac	A
Float to Unsigned Long	cfu22	fac	A
Float to Fract	cfr21	fac	A
Float to Long Fract	cfr22	fac	A

Table 7–12: Floating point conversion operations

7.8.2.2 THE FLOATING POINT ACCUMULATORS

The software floating point libraries for the DSP56xxx are based on the 2–Complement Format, which is fully optimized for fast and efficient floating point operations. The floating point values are stored in so–called floating point accumulators.

Three accumulators are necessary to perform all the floating point operations, they are called fac, ftm1 and ftm2. Accumulator fac is used for passing operand 1, result values, intermediate results and for internal calculations. Accumulators ftm1 and ftm2 are used for passing operands and internal calculations. The accumulators are located in registers.

	Mantissa	Exponent
FAC	A2 – sign extension of A1 (unused) A1 – mantissa A0 – zero	B2 – sign extension of B1 (unused) B1 – exponent B0 – zero
FTM1	X1	X0
FTM2	Y1	Y0

Table 7–13: Accumulator formats

7.8.2.3 STORAGE 2-COMPLEMENT FORMAT VALUES

The following table shows the memory storage implementation used by the software floating point libraries for single precision floating point values.

Address	+0000	+0001
Binary encoding	0000.0000.0000.0000.eeee.eeee	s.mmm.mmmm.mmmm.mmmm.mmmm.mmmm
s = sign bit, m = mantissa bit, e = exponent bit		

Table 7–14: Memory Layout for 24–bit data models

Address	+0000	+0001
Binary encoding	0000.0000.eeee.eeee	s.mmm.mmmm.mmmm.mmmm
s = sign bit, m = mantissa bit, e = exponent bit		

Table 7–15: Memory Layout for 16–bit data models

7.8.2.4 INTERNAL REGISTER USAGE

The software floating point arithmetic and conversion functions use a set of registers. Some are used for parameter passing and others are free for internal use. If you use some of these registers in your own assembly function you have to save them before a floating point function can be performed.

The registers that are modified by each function are described in the following tables.

Function	Modified register(s)
addf2	A, B, X, R0, N0
subf2	A, B, X, Y, R0, N0
mulf2	A, B, X0, R0, N0 (DSP563xx) A, B, X0, R0 (other)
divf2	A, B, X, R0, N0 (DSP563xx) A, B, X, R0 (other)
macpf2	A, B, X, Y, R0, N0 (DSP563xx) A, B, X, Y, R0 (other)
macnf2	A, B, X, Y, R0, N0 (DSP563xx) A, B, X, Y, R0 (other)
cmpf2	none
negf2	A, B, R0, N0 (DSP563xx) A, B, R0 (other)

Table 7–16: Floating point arithmetic operations register usage

Function	Modified register(s)
cif12	A, B, R0, N0 (DSP563xx) A, B, R0 (other)
cif22	A, B, R0, N0 (DSP563xx) A, B, R0 (other)
cuf12	A, B, X, R0, N0 (DSP563xx) A, B, X, R0 (other)
cuf22	A, B, X, R0, N0 (DSP563xx) A, B, X, R0 (other)
crf12	A, B, R0, N0 (DSP563xx) A, B, R0 (other)
crf22	A, B, R0, N0 (DSP563xx) A, B, R0 (other)
cfi21	A, B, Y1 (DSP563xx) A, B, Y, R0, N0 (other)
cfi22	A, B, Y1 (DSP563xx) A, B, Y, R0, N0 (other)
cfu21	A, B, Y, R0, N0
cfu22	A, B, Y1

Function	Modified register(s)
cfr21	A, B, Y, R0, N0
cfr22	A, B, Y, R0, N0

Table 7–17: Floating point conversion operations register usage

7.8.3 FLOATING POINT CODE GENERATION

This section describes, using some examples, the basics of floating point code generation. It is impossible to describe here all possible code generation combinations with all the floating point operations, because the number of possible floating point expression is almost infinite. So, if you want to write your own floating point expression in assembly it is profitable to write it first in C and then use the code generated by the C compiler in your own assembly function.

The following example will illustrate a floating point expression using two floating point values returning a floating point value.

```
c = a + b;

move x:Fa,b      ; pass floating point a in fac
move x:Fa+1,a
move x:Fb,x0     ; pass floating point b in ftm1
move x:Fb+1,x1
jsr  Rfpaddf2    ; perform add
move a,x:Fc+1    ; store result from fac in c
move b,x:Fc
```

Integer and long operands and integer and long results are passed via accumulator register A. The following example illustrates a conversion from long to float.

```
float  a;
long   b;

a = b;

move x:Fb+1,a    ; pass long b in A
move x:Fb,a0
jsr  Rfpcif22    ; convert from long to float
move a,x:Fa+1    ; store result from fac in a
move b,x:Fa
```

For more comprehensive floating point expressions it is not needed to store the floating result of a previous floating point operation and load it again for the next floating point operation. The result of the previous floating point operations remains in the accumulator fac and will be used in the next floating point operation. This is called intermediate result optimization. Only the second operand must be loaded in accumulator ftm1. The following example illustrates this.

```
d = a + b – c;

move x:Fa,b      ; pass floating point a in fac
move x:Fa+1,a
move x:Fb,x0     ; pass floating point b in ftm1
move x:Fb+1,x1
jsr  Rfpaddf2    ; perform add
                 ; The intermediate floating point
                 ; result stays in fac !
move x:Fc,x0     ; pass floating point c in ftm 1
move x:Fc+1,x1
jsr  Rfpsubf2    ; perform subtract
move a,x:Fd+1    ; store result from fac in d
move b,x:Fd
```

The floating point mechanism is based on the fact that when the floating point accumulator is loaded with a floating point operand and a next operand must be loaded in it, then the current contents of fac is saved on the user stack.

Next example is for the DSP5600x and illustrates the use of the user stack in a floating point expression, which needs to subtract two intermediate results. The first intermediate floating point result is stored on the user stack and the second can be hold in the accumulator fac. To perform the subtraction the intermediated result is popped from the user stack and loaded in accumulator ftm1. This example shows generation of reentrant code.

```
e = (a + b) – (c * d)

move (r7)+          ; reserve user stack space
move (r7)+
move x:Fa,b         ; pass floating point a in fac
move x:Fa+1,a
move x:Fb,x0        ; pass floating point b in ftm1
move x:Fb+1,x1
jsr  Rfpaddf2       ; perform add
move #–2,n7         ; user stack offset
move (r7)+          ; first stack element
move a,x:(r7+n7)    ; push result mantissa from
                    ; fac on stack
move (r7)–          ; second stack element
move b,x:(r7+n7)    ; push result exponent from
                    ; fac on stack
move x:Fc,b         ; pass floating point c in fac
move x:Fc+1,a
move x:Fd,x0        ; pass floating point d in ftm1
move x:Fd+1,x1
jsr  Rfpmulf2       ; perform multiply
move b,x0           ; store result from fac in ftm1
move a,x1
move (r7)+          ; first stack element
move x:(r7+n7),a    ; pop mantissa result from
                    ; stack to fac
move (r7)–          ; second stack element
move x:(r7+n7),b    ; pop exponent result from
                    ; stack to fac
jsr  Rfpsubf2       ; perform subtract
move a,x:Fe+1       ; store result from fac in e
move b,x:Fe
move (r7)+n7        ; free reserved user stack space
```

CHAPTER 8

SUPPORT FOR USER–DESIGNED TARGET BOARDS

TASKING

8 CHAPTER

This chapter contains the steps you have to take to support user–designed target boards.

To create support files for a user–designed target board:

1. Create a `mytarget.asm` file in the `lib\src\` directory by copying one of the startup files (for example, `def_targ.asm`).

2. Change the new file to contain adequate values for the AARx registers etcetera for the board. Refer to the DSP manual and the hardware manual of the board for the correct settings.

3. Add the new startup file to your project:

   Using EDE:

   – add `mytarget.asm` to the project file list.

   Using the command line:

   – add `mytarget.asm` to the makefile.

   Alternatively, you can add the new startup file to the libraries. From the command line, in the product directory, add the new file to the libraries with (example for the DSP563xx):

```
bin\as563 lib\src\mytarget.asm –O              >NUL
bin\ar563 –r lib\563xx\libc24.a mytarget.obj   >NUL
bin\as563 lib\src\mytarget.asm –M16 –O         >NUL
bin\ar563 –r lib\563xx\libc16.a mytarget.obj   >NUL
bin\ar563 –r lib\563xx\libc1624.a mytarget.obj >NUL
del mytarget.obj
```

4. Create a correct memory description of your board for the locator.

   Memory descriptions are in the product `etc` directory (`*.mem`) for several targets. They are included in the `*.dsc` files. If you created a `mytarget.asm` file, name these files `mytarget.dsc` and `mytarget.mem`. `mytarget.dsc` can be a copy of any of the `*.dsc` files, just replace the `.mem` include with `mytarget.mem`, and the `.cpu` include with the correct cpu type.

Create a `mytarget.mem` file with the correct sizes of the external memory. One of the existing `.mem` files can be used as a starting point for this. Take care that your bus structure is correct: some files have a unified bus (X/Y/P maps to the same physical memory, so sections are placed after each other), others have separate buses (X/Y/P map to different chips, so sections can be placed in parallel). Take the file that resembles your board, and then just change the memory sizes in the chips section of the file.

5. If you added a new startup file to your project (in step 3), disable the startup file from the libraries and specify the correct startup code label and description file:

   Using EDE:

   – Select the `Project | Project Options...` menu item. Expand the `Linker/Locator` entry and select `Control File`. Select `User supplied, no library startup code` in the `Target` list box. In the `User defined target name` field, fill in "`mytarget`". This will tell the tools to include the startup file `mytarget.asm` from the library, and to use the file `mytarget.dsc` for the locating process.

   Using the command line:

   – Add **–Tmytarget** to the control program (**cc563**) command line.

6. If you added a new startup file to the libraries (in step 3), specify the correct description file:

   Using EDE:

   – Select the `Project | Project Options...` menu item. Expand the `Linker/Locator` entry and select `Control File`. Select `User supplied target definition` in the `Target` list box. Specify `mytarget.dsc` in the field of the `Use project specific linker/locator control file (.dsc)` radio button.

   Using the command line:

   – Add **–T mytarget.dsc** to the control program (**cc563**) command line.

APPENDIX A

FLEXIBLE LICENSE MANAGER (FLEXlm)

TASKING

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:

- 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 *server*.

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 (*lfpath*) with a '**;**' (on UNIX also '**:**') :

Windows:

set LM_LICENSE_FILE=*lfpath*[**;***lfpath*]...

UNIX:

setenv LM_LICENSE_FILE *lfpath*[**:***lfpath*]...

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.

lmhostid

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, encription 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

feature Any feature this daemon supports.

vendor The name of the vendor daemon (such as **Tasking**).

new–file 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

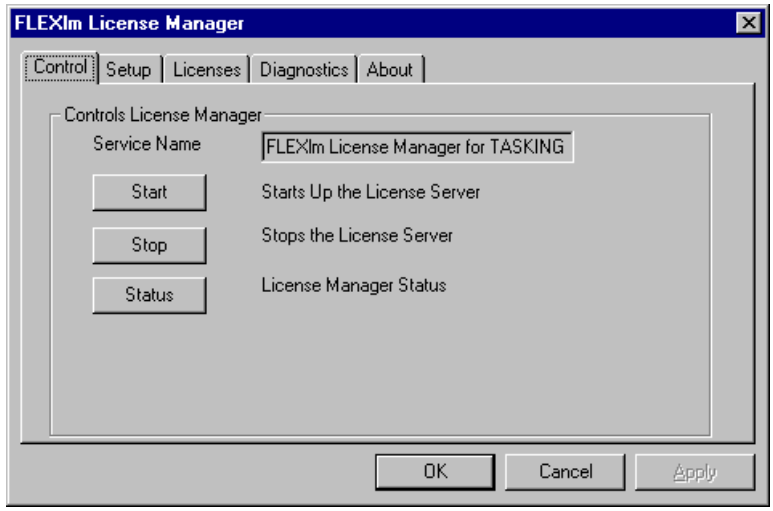
3.11.1 LMTOOLS FOR WINDOWS

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.

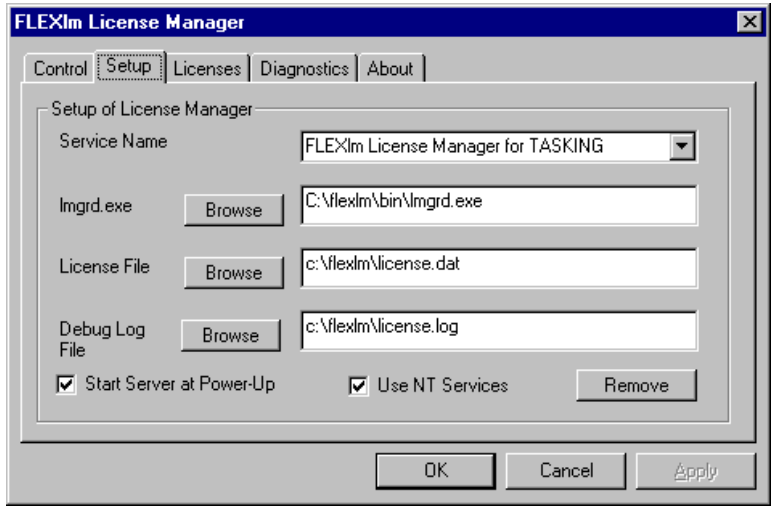
The `HOSTID` button displays the hostid's for the computer on which the program is running. The `TIME` button prints out the system's internal time settings, intended to diagnose any time zone problems. The `TCP Settings` button is intended to fix a bug in the Microsoft TCP protocol stack which has a 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 nnn
EXITING with code nnn

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 nnn

The license daemon received fatal signal *nnn*.

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 nnn)

Vendor daemon *xxx* was restarted at internet port *nnn*.

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 nnn: 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 = nnn

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 SWiiiiii-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 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, 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

MOTOROLA COMPATABILITY

TASKING

3 APPENDIX

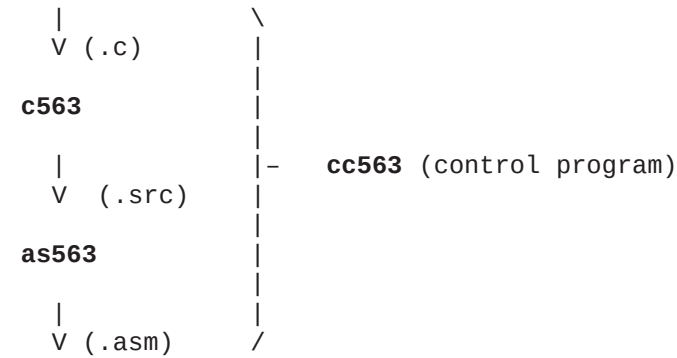
1 INTRODUCTION

This appendix describes the interoperability between the TASKING and Motorola tool sets. That means, how to create an application with the TASKING DSP56xxx C compiler, which can be debugged with the Motorola debugger; or how to link Motorola object files and library files with the TASKING linker.

2 CREATING A MOTOROLA COFF OBJECT FILE

The TASKING tools do not have the capability to generate objects in the Motorola CLAS COFF object format with debug information. In order to overcome this limitation it is possible to generate an assembly file with the TASKING tools, which can then be assembled with the Motorola assembler.

Generating an assembly file with the TASKING tools uses the following path:



The control program calls both the compiler and the assembler. The compiler generates a source file. This source file generated by the compiler contains COFF style debug information that will be converted by the Motorola assembler. The generated source file is taken through the TASKING assembler to further optimize this assembly code. Among the optimizations performed by the TASKING assembler are move parallelization, jump and branch optimizations and DO into REP conversion. The assembler generates an assembly file that is compatible with the Motorola assembler.

It is possible to use the source file from the compiler (`.src`) in the Motorola assembler directly. However, in doing so optimizations will get lost.

In order to support the generation of Motorola compatible assembly code with COFF debug information, the following command line options must be supplied to the TASKING tools:

Tool	Option
cc563	**–S**
c563	**–Cg**
as563	**–S**

cc563

The **–S** option tells the 563xx control program to generate a Motorola compatible assembly file. It does so by taking the following steps:

1. call the compiler with the **–C1** option (full Motorola compatibility mode)
2. call the assembler with the **–S** option (generate an assembly file instead of an object file)

The control program stops after generating the assembly file. The linker and locator will not be invoked. The generated assembly file is generated by default by adding a `.asm` extension to the input file. You can specify the name of the generated assembly file with the **–o** option of the control program.

If, for any reason, you do not want to use the full compatibility mode, you have to specify the compiler and assembler options yourself. For instance:

```
cc563 -g -Wc-Cg -Wa-S -c -o file.asm file.c
```

will only generate COFF debug information, but will skip the other compatibility options.

c563

The **–Cg** option tells the compiler to generate COFF style debug information instead of the normal SYMB directives. The COFF debug information is only generated when the **–g** option is specified as well. It is highly advisable to use the **–Ca** option as well in order to generate Motorola assembler compatible output.

When using the **–Ca** option it is not possible to use packed strings. The `_packed` keyword will then be ignored.

as563

The **–S** option instructs the assembler to generate an assembly file instead of an object file. It is still possible to generate a list file as well.

Using the assembler for optimization can result in synchronization loss between the generated code and the debugging information. Using the **–OG** option will prevent this, but has negative effects on the optimization performed.

Example:

This example compiles the file `demo.c` with the TASKING compiler, and uses the Motorola assembler and linker to generate the final COFF absolute file. The **–g** switch is given in order to get debug information.

```
cc563 –S –g demo.c
g563c demo.asm –g –o demo.cld –mx–memory
```

or if you do not want to use the different control programs:

```
c563 –g –C1 demo.c
as563 –S demo.src
asm56300 –c –Bdemo.cln demo.asm
dsplnk –g –c –Bdemo.cld crt0563x.cln demo.cln –Llib563cx.clb
```

The same examples, now using Y memory:

```
cc563 –S –g –My demo.c
g563c demo.asm –g –o demo.cld
```

or

```
c563 –C1 –g –My demo.c
as563 –S demo.src
asm56300 –c –Bdemo.cln demo.asm
dsplnk –g –c –Bdemo.cld crt0563y.cln demo.cln –Llib563cy.clb
```

The Motorola assembler needs the **–c** option in order to work correctly.

Known restrictions:

- Packed strings are not supported when generating Motorola compatible assembly. The `_packed` keyword is ignored because the Motorola assembler does not have directives to support this.
- The TASKING tools can move variables around between registers and/or stack in order to get optimal code. To allow the debugger to keep track of these moves the compiler generates lifetime information, telling the debugger in which register the variable is located. The COFF debug format cannot handle lifetime information. Therefore, this information is lost. Watching an automatic variable in a debugger could result in looking at the wrong place. This problem can be overcome by turning off the optimizations with **–O0**.
- The default memory model for the TASKING tools is X memory. The Motorola tools use Y memory. You need to specify another memory model for one of the tool chains. You can either specify the **–My** option to the TASKING control program for generating programs which use Y memory, or you can specify the **–mx–memory** option to the Motorola control program.

The COFF libraries as supplied with the Tasking tools use the X memory.

- Floating point variables generated by the Motorola C compiler have a format that differs from the TASKING format. Restrict the use of floating point variables to one compilation system.

3 USING LIBRARY FUNCTIONS

Due to certain differences between the TASKING compiler and the Motorola compiler it is normally not possible to use the standard Motorola libraries with COFF objects generated with the TASKING tools. Therefore, the TASKING libraries are supplied in both IEEE object format and in COFF object format. These COFF libraries fully support Motorola's file system support. The libraries support the 563xx in 24–bit mode (`librt24.clb`, `libfp24.clb`, `libc24.clb`), the 566xx (`librt6.clb`, `libfp6.clb`, `libc6.clb`) and the 5600x (`librt.clb`, `libfp.clb`, `libcm.clb`, `libcs.clb`, `libcr.clb`).

Even when using the TASKING COFF libraries it is necessary to use the crt0 startup files as provided by Motorola. This is necessary because this file contains numerous variables that are filled in by the Motorola linker.

When using the `malloc()` functions from the TASKING COFF libraries, you must use a linker memory control file in order to specify the heap. Normally theTASKING linker/locator allocates the heap and generates symbols for the start and end of it.. You have to specify the symbols `F_lc_bh` and `F_lc_eh`, where `F_lc_bh` indicates the beginning of the heap area and the `F_lc_eh` the end of the heap area.

The heap must reside in default memory space.

An example of a memory control file is given here:

```
reserve                 y:$3000..$3fff
symbol  F_lc_bh         y:$3000
symbol  F_lc_eh         y:$4000
```

This will define the Y memory from `$3000` till `$4000` as heap space which the `malloc()` routine can use for allocating memory.

4 LINKING MOTOROLA CLAS/COFF

The TASKING linker has the capability to read in Motorola CLAS/COFF object files and library files. It will convert debug information in these files into that of the TASKING IEEE–695 object format. This allows debugging a program compiled with the Motorola tools with CrossView Pro.

The TASKING linker looks at the filename extension to determine what kind of input file it is. In order for the linker to recognize a file as a Motorola CLAS/COFF file the file must have the extension `.cln`. A Motorola CLAS/COFF library file must have the extension `.clb`.

Example:

This example links the CLAS COFF object file `util.cln` with the program `prog.c`. The Motorola library is specified as well in order to resolve any run–time routines which might be called from `util.cln`. The control program automatically links with the TASKING libraries.

```
cc563 -g prog.c util.cln lib563x.clb
```

Known restrictions:

- The Motorola linker/locator uses internally defined symbols like DSIZE. This symbol defines the beginning of the dynamic stack. The TASKING linker/locator does not know the DSIZE symbol. Therefore, this will remain an unresolved symbol. This will only happen if you link with the Motorola startup code (`crt0563x.cln`). Use the TASKING startup code instead.
- The TASKING libraries are translated with a different calling convention than the Motorola compiler uses. So, any calls to a TASKING library function from a program compiled by the Motorola compiler will fail. It is possible to recompile the libraries with the **–Ccr** option to create libraries that use the Motorola calling convention.
- The TASKING libraries are compiled for using X memory by default. The Motorola compiler generates programs that use Y memory by default. Either recompile all the TASKING libraries for Y memory (**–M**y), or specify **–mx–memory** to the Motorola tools.

5 RUNNING EXAMPLES FROM EDE

The TASKING DSP56xxx toolchain now provides a special EDE file to generate code in Motorola COFF format with debug information. The following installation steps are necessary to allow this to work:

1. Install the Motorola executables (at least the assembler and linker, and optionally the debugger) in a single directory. Add the path to this directory to the PATH environment variable, as prescribed by the Motorola installation instructions (adding it to `Project | Directories | Executable Files Path` will not work).

2. In the `Project | Select Toolchain...` dialog, select the correct toolchain that is shown as "with Motorola tools". The `Project | Project Options...` dialog will now show options for the Motorola tools instead of the TASKING linker and locator. To revert to the TASKING tools later, select the correct toolchain without the additional "with Motorola tools".

3. Add the path to the Motorola library directory to `Project | Directories | Library Files Path` to allow the tools to find the Motorola startup files.

4. Select an EDE option, for instance `Project | Project Options | Motorola Debugger`, and press `OK` to force the tools to rebuild the makefile.

5. Press the `Rebuild` button to build the project.

6. Press the `Debug` button to start the Motorola debugger. The debugger will automatically load the executable.

7. To run an example that writes to `stdout`, like "hello world", enter the following commands on the debugger command line:

```
streams enable
redirect stdout file.txt
go
```

8. The output of the program can now be found in *`file`*`.txt` in the current working library.

   This method works for the 'hello','sieve' and 'whet' examples. For the 'dhry_1' and 'dhry_2' examples, a linker control file must be added to the project to create a heap for the malloc() function. The following contents can be used for this file, but it may require changes for different memory sizes on actual hardware:

```
reserve             x$2000..$3fff
symbol F_lc_bh      x:$2000
symbol F_lc_eh      x:$3000
```

   Add the linker control file to the linker options with the **–R***filename* command. Running the examples in the `bench` directory may take a very long time on the simulator. Other examples cannot be run because they contain assembly files that are not Motorola compliant.

MOTOROLA COMPATIBILITY

INDEX

INDEX

TASKING

INDEX

Symbols

Numbers

A

B

C

INDEX

D

E

F

G

H

J

K

L

M

N

Q

R

S

T

U

V

W

INDEX