MA0060–024–00–00
Doc. ver.: 1.2

# TriCore v2.0

## C Compiler, Assembler, Linker User's Guide

TASKING
Embedded software development from Altium

A publication of

Altium BV

Documentation Department



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

CONTENTS

# TABLE OF CONTENTS

TASKING

# CONTENTS

## TRICORE C LANGUAGE 3-1



CONTENTS

## TRICORE ASSEMBLY LANGUAGE 4-1



## USING THE COMPILER 5-1

## MANUAL PURPOSE AND STRUCTURE

### *Windows Users*

The documentation explains and describes how to use the TriCore toolchain to program a TriCore DSP. The documentation is primarily aimed at Windows users. You can use the tools either with the graphical Embedded Development Environment (EDE) or from the command line in a command prompt window.

### *UNIX Users*

For UNIX the toolchain works the same as it works for the Windows command line.

Directory paths are specified in the Windows way, with back slashes as in `.\include`. Simply replace the back slashes by forward slashes for use with UNIX: `./include`.

Some characters have a special meaning in a UNIX shell. In such cases you must escape the special characters. For example, '**`-?`**' must be specified as '**`-\?`**' in some shells. See your UNIX documentation for more information.

### *Structure*

The toolchain documentation consists of a User's Guide (this manual) which includes a Getting Started section and a separate Reference Guide.

First you need to install the software and make it run under the licence manager FLEXlm. This is described in Chapter 1, *Software Installation and Configuration*

After installation you are ready to follow the *Getting Started* in Chapter 2.

Next, move on with the other chapters which explain how to use the compiler, assembler, linker and the various utilities.

Once you are familiar with these tools, you can use the Reference Guide to lookup specific options and details to make full use of the TriCore toolchain.

## SHORT TABLE OF CONTENTS



MANUAL STRUCTURE

***Appendix A: Flexible Licence Manager (FLEXlm)***

TASKING products are licensed through FLEXlm. This chapter provides information about this license system and how to solve possible problems.

## CONVENTIONS USED IN THIS MANUAL

### *Notation for syntax*

The following notation is used to describe the syntax of command line input:

| | |
|---|---|
| **bold** | Type this part of the syntax literally. |
| *italics* | Substitute the italic word by an instance. For example:<br>*filename*<br>Type the name of a file in place of the word *filename*. |
| { } | Encloses a list from which you must choose an item. |
| [ ] | Encloses items that are optional. For example<br>**ctc** [ **–?** ]<br>Both **`ctc`** and **`ctc –?`** are valid commands. |
| \| | Separates items in a list. Read it as OR. |
| ... | You can repeat the preceding item zero or more times. |
| ,... | You can repeat the preceding item zero or more times, separating each item with a comma. |

### *Example*

**ctc** [*option*]... *filename*

You can read this line as follows: enter the command **ctc** with or without an option, follow this by zero or more options and specify a *filename*. The following input lines are all valid:

```
ctc test.c
ctc –g test.c
ctc –g –E test.c
```

Not valid is:

```
ctc –g
```

According to the syntax description, you have to specify a filename.

### *Icons*

The following illustrations are used in this manual:

Note: notes give you extra information.

Warning: read the information carefully. It prevents you from making serious mistakes or from loosing information.

This illustration indicates actions you can perform with the mouse. Such as EDE menu entries and dialogs.

Command line: type your input on the command line.

Reference: follow this reference to find related topics.

## RELATED PUBLICATIONS

***C Standards***

- The C Programming Language (second edition) by B. Kernighan and D. Ritchie (1988, Prentice Hall)
- ISO/IEC 9899:1999(E), Programming languages – C [ISO/IEC]
  More information on the standards can be found at
  `http://www.ansi.org`
- DSP–C, An Extension to ISO/IEC 9899:1999(E),
  Programming languages – C [TASKING, TK0071–14]

***MISRA C***

- Guidelines for the Use of the C Language in Vehicle Based Software [MISRA]
  See also `http://www.misra.org.uk`

***TASKING Tools***

- TriCore C Compiler, Assembler, Linker Reference Guide
  [TASKING, MB060–024–00–00]
- TriCore C++ Compiler User's Guide
  [TASKING, MA60–012–00–00]
- TriCore CrossView Pro Debugger User's Guide
  [TASKING, MA060–043–00–00]

***TriCore***

- TriCore 1 Unified Processor Core v1.3 Architecture Manual, Doc v1.3.3 [2002–09, Infineon]
- TriCore2 Architecture Overview Handbook [2002, Infineon]
- TriCore Embedded Application Binary Interface [2000, Infineon]

CHAPTER 1

# SOFTWARE INSTALLATION AND CONFIGURATION

TASKING

# 1 CHAPTER

## 1.1 INTRODUCTION

This chapter guides you through the procedures to install the software on a Windows system or on a Linux or UNIX host.

The software for Windows has two faces: a graphical interface (Embedded Development Environment) and a command line interface. The Linux and UNIX software has only a command line interface.

After the installation, it is explained how to configure the software and how to install the license information that is needed to actually use the software.

## 1.2 SOFTWARE INSTALLATION

### 1.2.1 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 1.4, *Licensing TASKING Products*.

### 1.2.2 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:

```
SWproduct-version-RPMrelease.i386.rpm
swproduct_version-release_i386.deb
SWproduct-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.

***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.

### *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`.

### *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 SW***product–version***.tar.gz –C /usr/local**

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

### 1.2.3 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 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 1.4, *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 <=
   ```

Answer **y** (yes) to continue with the installation. The last message will be:

```
Installation of SWxxxxxx xxxx.xxxx completed.
```

5. If you purchased a protected TASKING product, license the software product as explained in section 1.4, *Licensing TASKING Products*.

## 1.3 SOFTWARE CONFIGURATION

Now you have installed the software, you can configure both the Embedded Development Environment and the command line environment for Windows, Linux and UNIX.

### 1.3.1 CONFIGURING THE EMBEDDED DEVELOPMENT ENVIRONMENT

After installation, the Embedded Development Environment is automatically configured with default search paths to find the executables, include files and libraries. In most cases you can use these settings. To change the default settings, follow the next steps:

1. Double–click on the EDE icon on your desktop to start the Embedded Development Environment (EDE).
2. From the **Project** menu, select **Directories...**

   *The Directories dialog box appears.*
3. Fill in the following fields:
   - In the **Executable Files Path** field, type the pathname of the directory where the executables are located. The default directory is `c:\ctc\bin`.
   - In the **Include Files Path** field, add the pathnames of the directories where the compiler and assembler should look for include files. The default directory is `c:\ctc\include`. Separate pathnames with a semicolon (;).

     The first path in the list is the first path where the compiler and assembler look for include files. To change the search order, simply change the order of pathnames.

- In the **Library Files Path** field, add the pathnames of the directories where the linker should look for library files. The default directory is `c:\ctc\lib`. Separate pathnames with a semicolon (;).

  The first path in the list is the first path where the linker looks for library files. To change the search order, simply change the order of pathnames.

Instead of typing the pathnames, you can click on the **Configure...** button.

A dialog box appears in which you can select and add directories, remove them again and change their order.

### 1.3.2 CONFIGURING THE COMMAND LINE ENVIRONMENT

To facilitate the invocation of the tools from the command line (either using a Windows command prompt or using Linux or UNIX), you can set *environment variables*.

You can set the following variables:

| Environment Variable | Description |
|---|---|
| PATH | With this variable you specify the directory in which the executables reside (default: `c:\ctc\bin`). This allows you to call the executables when you are not in the `bin` directory.<br>Usually your system already uses the PATH variable for other purposes. To keep these settings, you need to add (rather than replace) the path. Use a semicolon (;) to separate pathnames. |
| CTCINC | With this variable you specify one or more additional directories in which the C compiler **ctc** looks for include files. The compiler first looks in these directories, then always looks in the default `c:\ctc\include` directory. |
| ASTCINC | With this variable you specify one or more additional directories in which the assembler **astc** looks for include files. The assembler first looks in these directories, then always looks in the default `c:\ctc\include` directory. |
| ASPCPINC | With this variable you specify one or more additional directories in which the assembler **aspcp** looks for include files. The assembler first looks in these directories, then always looks in the default `c:\ctc\include` directory. |
| CCTCBIN | With this variable you specify the directory in which the control program **cctc** looks for the executable tools. The path you specify here should match the path that you specified for the PATH variable. |
| CCTCOPT | With this variable you specify options and/or arguments to each invocation of the control program **cctc**. The control program processes these arguments before the command line arguments. |

| Environment Variable | Description |
|---|---|
| LIBTC1V1_2<br>LIBTC1V1_3<br>LIBTC2 | With this variable you specify one or more alternative directories in which the linker **ltc** looks for library files for a specific core. The linker first looks in these directories, then always looks in the default `c:\ctc\lib` directory. |
| LM_LICENSE_FILE | With this variable you specify the location of the license data file. You only need to specify this variable if your host uses the FLEXlm licence manager. |
| TMPDIR | With this variable you specify the location where programs can create temporary files. Usually your system already uses this variable. In this case you do not need to change it. |

*Table 1–1: Environment variables*

The following examples show how to set an environment variable using the PATH variable as an example.

### *Example for Windows 95/98*

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

```
set PATH=%path%;c:\ctc\bin
```

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

### *Example for Windows NT*

1. Right–click on the `My Computer` icon on your desktop and select **Properties** from the menu.

   *The System Properties dialog appears.*

2. Select the **Environment** tab.

3. In the list of **System Variables** select **Path**.

4. In the **Value** field, add the path where the executables are located to the existing path information. Separate pathnames with a semicolon (;). For example: `c:\ctc\bin`.

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

***Example for Windows XP / 2000***

1. Right–click on the `My Computer` icon on your desktop and select **Properties** from the menu.

   *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 list of **System variables** select **Path**.

5. Click on the **Edit** button.

   *The Edit System Variable dialog appears.*

6. In the **Variable value** field, add the path where the executables are located to the existing path information. Separate pathnames with a semicolon (;). For example: `c:\ctc\bin`.

7. Click on the **OK** button to accept the changes and close the dialogs.

***Example for UNIX***

Enter the following line (C–shell):

```
setenv PATH $PATH:/usr/local/ctc/bin
```

## 1.4 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 Appendix A, *Flexible License Manager (FLEXlm)*, for more information.

### 1.4.1 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 1.4.7, *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 1.4.7, *How to Determine the Hostid* and section 1.4.8, *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.

## 1.4.2 INSTALLING NODE-LOCKED LICENSES

Keep your "License Information Form" ready. If you do not have such a form read section 1.4.1, *Obtaining License Information*, before continuing.

***Step 1***

Install the TASKING software product following the installation procedure described in section 1.2.1, *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 1.4.6, *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 1.4.6, *Modifying the License File Location*, for additional information.

The software product and license file are now properly installed.

See Appendix A, *Flexible License Manager (FLEXlm)*, for more information.

### 1.4.3 INSTALLING FLOATING LICENSES

Keep your "License Information Form" ready. If you do not have such a form read section 1.4.1, *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 1.4.5, *Setting Up the License Daemon to Run Automatically*.

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

***Step 3***

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

***Step 4***

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

If you wish to install the license file in a different directory, see section 1.4.6, *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 1.4.6, *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 1.4.6, *Modifying the License File Location*, for more information.

### *Step 6*

Now all license information is entered, the license manager must be started (see section section 1.4.4). 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 1.4.4 on how to start the daemon and read section 1.4.5 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 Appendix A, *Flexible License Manager (FLEXlm)*, for more information.

### 1.4.4 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. From the Windows **Start** menu, select **Programs –> TASKING FLEXlm –> FLEXlm License Manager**.

   *The license manager tool appears.*

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 Appendix A, *Flexible License Manager (FLEXlm)*, for more information.

### 1.4.5 SETTING UP THE LICENSE DAEMON TO RUN AUTOMATICALLY

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

***Windows***

1. From the Windows **Start** menu, select **Programs –> TASKING FLEXlm –> FLEXlm License Manager**.

   *The license manager tool appears.*

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 appropriate sequence number. Refer to you operating system documentation for more information.

### 1.4.6 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 Appendix A, *Flexible License Manager (FLEXlm)*, for detailed information.

### 1.4.7 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**<br>(or use **lmhostid**) | 0800200055327 |

*Table 1–2: 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.

### 1.4.8 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 1–3: Determine the hostname*

CHAPTER 2

# GETTING STARTED

TASKING

# 2 CHAPTER

## 2.1 INTRODUCTION

With the TASKING TriCore suite you can write, compile, assemble, link and locate applications for the several TriCore cores. The TASKING TriCore suite conforms to Infineon's *TriCore Embedded Applications Binary Interface* (EABI), which defines a set of standards to ensure interoperability between software components.

### *Embedded Development Environment*

The TASKING Embedded Development Environment (EDE) is a Windows application that facilitates working with the tools in the toolchain and also offers project management and an integrated editor.

EDE has three main functions: *Edit / Project management*, *Build* and *Debug*. The figure below shows how these main functionalities relate to each other.

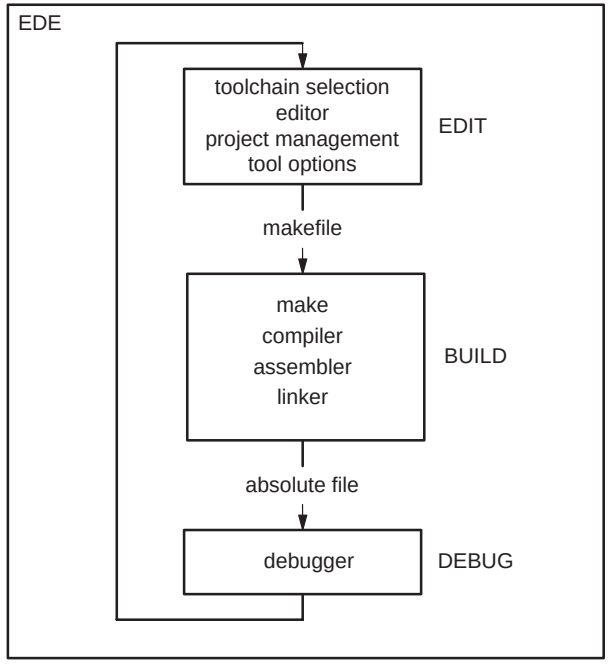



*Figure 2–1: EDE development flow*

In the **Edit** part you make all your changes:

- create a project space
- create and maintain one or more projects in a project space
- add, create and edit source files in a project
- set the options for each tool in the toolchain
- select another toolchain if you want to create an application for another target than the TriCore.

In the **Build** part you build your files:

- a makefile (created by the Edit part) is used to invoke the needed toolchain components, resulting in an absolute object file.

In the **Debug** part you can debug your project:

- call the TASKING debugger "CrossView Pro" with the generated absolute object file.

This *Getting Started* Chapter guides you step–by–step through the most important features of EDE

The TASKING EDE is an *embedded* environment and differs from a *native* program development.

A *native* program development environment is often used to develop applications for systems where the host system and the target are the same. Therefore, it is possible to run a compiled application directly from the development environment.

In an *embedded* environment, however, a simulator or target hardware is required to run an application. TASKING offers a number of simulators and target hardware debuggers.

### *Toolchain overview*

You can use all tools in the toolchain from the embedded development environment (EDE) and from the command line in a Command Prompt window or a UNIX shell.

The next illustration shows all components of the TriCore toolchain with their input and output files.

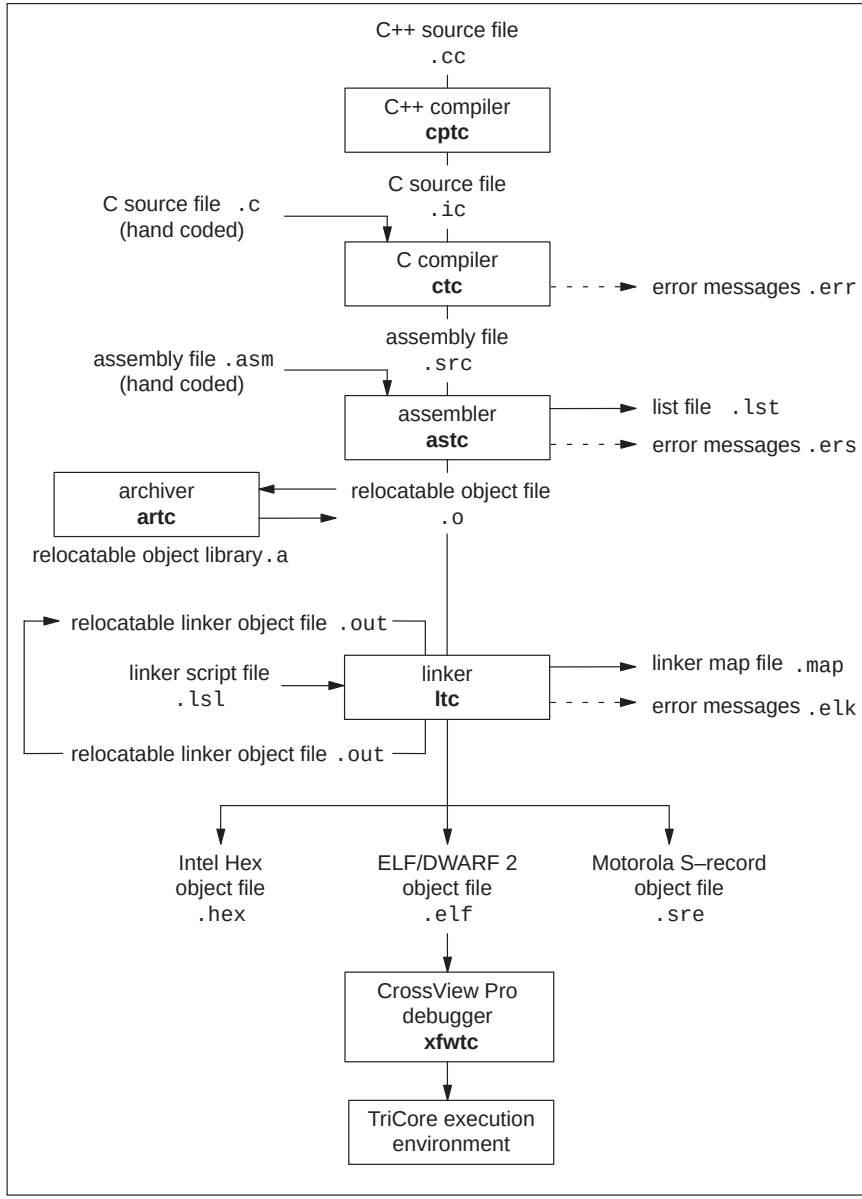



*Figure 2–2: TriCore toolchain*

The following table lists the file types used by the TriCore toolchain.

| Extension | Description |
|---|---|
| **Source files** | |
| .cc | C++ source file, input for the C++ compiler |
| .c | C source file, input for the C compiler |
| .asm | Assembler source file, hand coded |
| .lsl | Linker script file using the Linker Script Language |
| .i | Locating control file with defines for linker script file |
| **Generated source files** | |
| .ic | C source file, generated by the C++ compiler, input for the C compiler |
| .src | Assembler source file, generated by the C compiler, does not contain macros |
| **Object files** | |
| .o | ELF/DWARF relocatable object file, generated by the assembler |
| .a or .elb | Archive with ELF/DWARF object files |
| .abs | IEEE–695 absolute object file, generated by the locating part of the linker |
| .out | Relocatable linker output file |
| .elf | ELF/DWARF absolute object file, generated by the locating part of the linker |
| .hex | Absolute Intel Hex object file |
| .sre | Absolute Motorola S–record object file |
| **List files** | |
| .lst | Assembler list file |
| .map | Linker map file |
| .mcr | MISRA C report file |
| **Error list files** | |
| .err | Compiler error messages file |
| .ers | Assembler error messages file |
| .elk | Linker error messages file |

*Table 2–1: File extensions*

## 2.2 WORKING WITH PROJECTS IN EDE

EDE is a complete project environment in which you can create and maintain project spaces and projects. EDE gives you direct access to the tools and features you need to create an application from your project.

A *project space* holds a set of projects and must always contain at least one project. Before you can create a project you have to setup a project space. All information of a project space is saved in a *project space file* (`.psp`):

- a list of projects in the project space
- history information

Within a project space you can create *projects*. Projects are bound to a target! You can create, add or edit files in the project which together form your application. All information of a project is saved in a *project file* (`.pjt`):

- the target for which the project is created
- a list of the source files in the project
- the options for the compiler, assembler, linker and debugger
- the default directories for the include files, libraries and executables
- the build options
- history information

When you build your project, EDE handles file dependencies and the exact sequence of operations required to build your application. When you push the **Build** button, EDE generates a makefile, including all dependencies, and builds your application.

### *Overview of steps to create and build an application*

1. Create a project space
2. Add one or more projects to the project space
3. Add files to the project
4. Edit the files
5. Set development tool options
6. Build the application

## 2.3 START EDE

***Start EDE***

- Double–click on the EDE shortcut on your desktop.

  – or –

  Launch EDE via the program folder created by the installation program. Select **Start –> Programs –> TASKING *toolchain* –> EDE**.

*Figure 2–3: EDE icon*

The EDE screen contains a menu bar, a toolbar with command buttons, one or more windows (default, a window to edit source files, a project window and an output window) and a status bar.

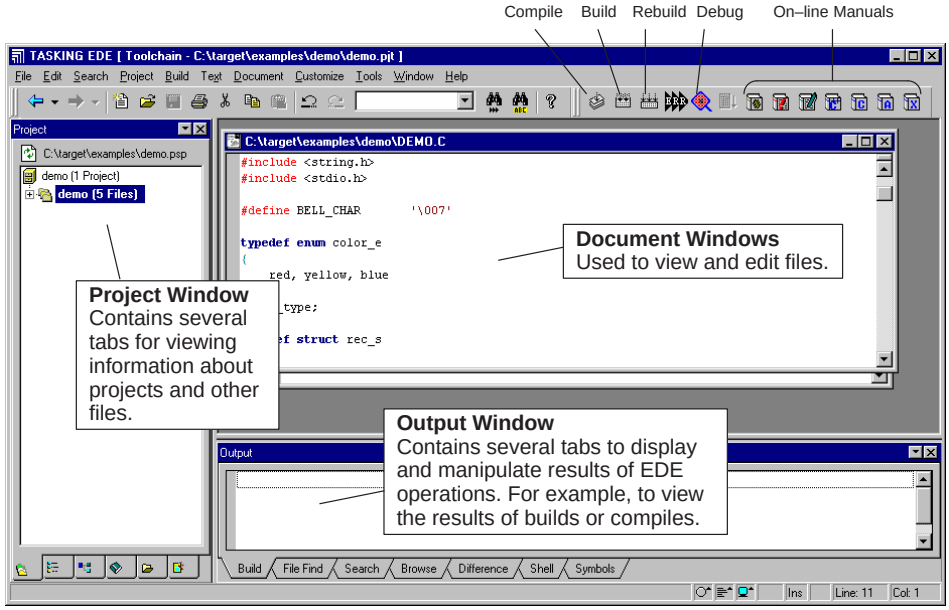



*Figure 2–4: EDE desktop*

## 2.4 USING THE SAMPLE PROJECTS

When you start EDE for the first time (see section 2.3, *Start EDE*), EDE opens with a ready defined project space that contains several sample projects. Each project has its own subdirectory in the `examples` directory. Each directory contains a file `readme.txt` with information about the example. The default project is called `demo.pjt` and contains a CrossView Pro debugger example.

### *Select a sample project*

To select a project from the list of projects in a project space:

1. In the Project Window, right–click on the project you want to open.

   *A menu appears.*

2. Select **Set as Current Project**.

   *The selected project opens.*

3. Read the file `readme.txt` for more information about the selected sample project.

### *Building a sample project*

To build the currently active sample project:

- Click on the **Execute 'Make' command** button.

*Once the files have been processed you can inspect the generated messages in the **Build** tab of the **Output** window.*

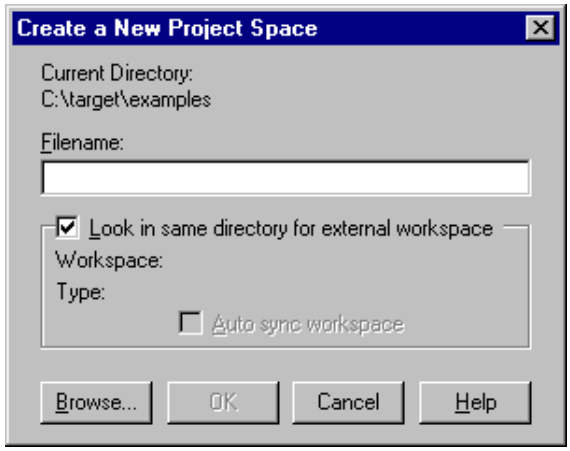
## 2.5 CREATE A NEW PROJECT SPACE WITH A PROJECT

Creating a project space is in fact nothing more than creating a project space file (`.psp`) in an existing or new directory.

### *Create a new project space*

1. From the **File** menu, select **New Project Space...**

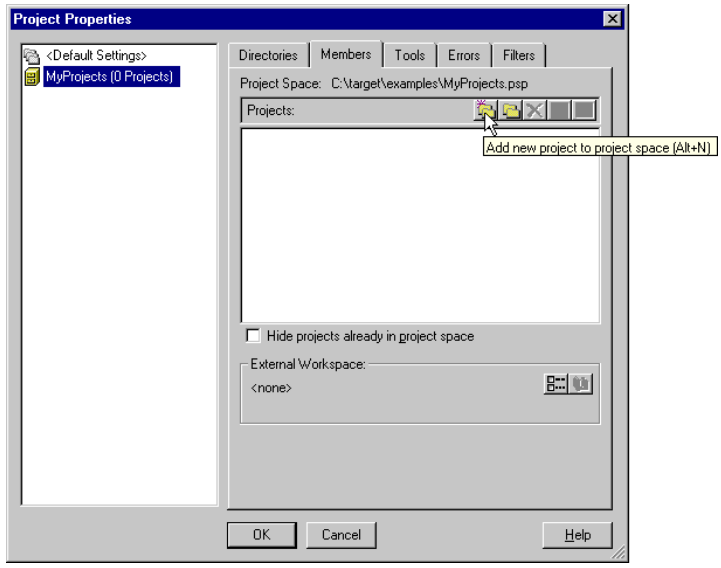
   *The Create a New Project Space dialog appears.*




2. In the the **Filename** field, enter a name for your project space (for example `MyProjects`). Click the **Browse** button to select a directory first and enter a filename.

3. Check the directory and filename and click **OK** to create the `.psp` file in the directory shown in the dialog.

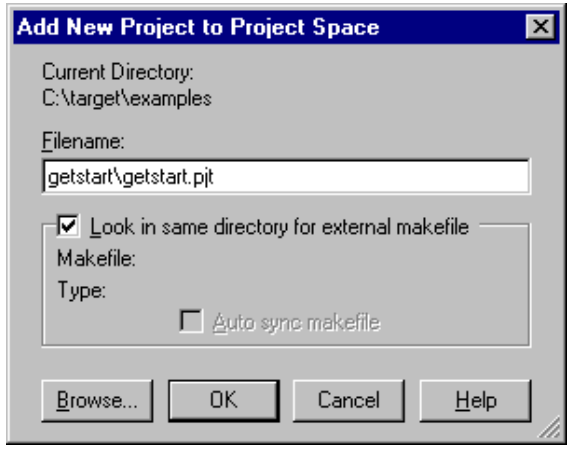
   *A project space information file with the name* `MyProjects.psp` *is created and the Project Properties dialog box appears with the project space selected.*




### *Add a new project to the project space*

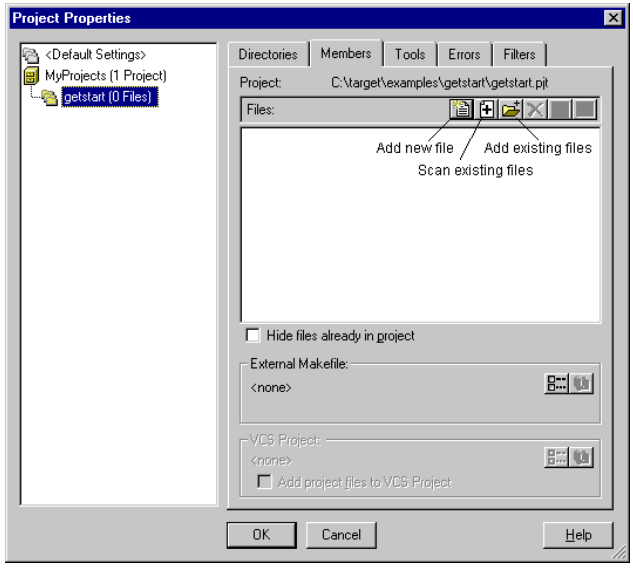
4. In the Project Properties dialog, click on the **Add new project to project space** button (see previous figure).

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



5. Give your project a name, for example `getstart\getstart.pjt` (a directory name to hold your project files is optional) and click **OK**.

   *A project file with the name* `getstart.pjt` *is created in the directory* `getstart`*, which is also created. The Project Properties dialog box appears with the project selected.*

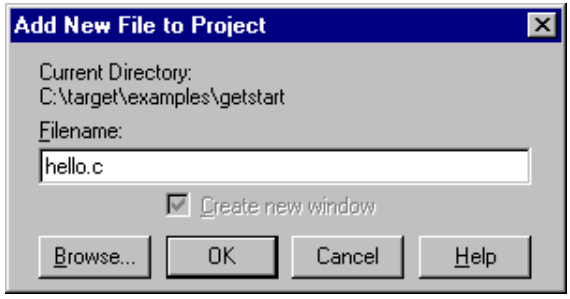



### *Add new files to the project*

Now you can add all the files you want to be part of your project.

6. Click on the **Add new file to project** button.

   *The Add New File to Project dialog appears.*



7. Enter a new filename (for example `hello.c`) and click **OK**.

   *A new empty file is created and added to the project. Repeat steps 6 and 7 if you want to add more files.*

8. Click **OK**.

   *The new project is now open. EDE loads the new file(s) in the editor in separate document windows.*

   EDE automatically creates a *makefile* for the project (in this case `getstart.mak`). This file contains the rules to build your application. EDE updates the makefile every time you modify your project.

### *Edit your files*

9. As an example, type the following C source in the `hello.c` document window:

```
#include <stdio.h>

void main(void)
{
    printf("Hello World!\n");
}
```

10. Click on the **Save the changed file <Ctrl–S>** button.

*EDE saves the file.*

## 2.6 SET OPTIONS FOR THE TOOLS IN THE TOOLCHAIN

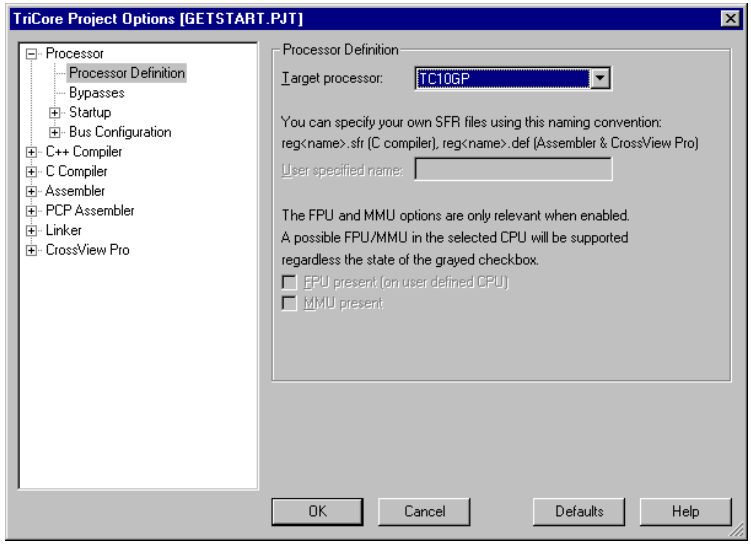
The next step in the process of building your application is to select a target processor and specify the options for the different parts of the toolchain, such as the C and/or C++ compiler, assembler, linker and debugger.

### *Select a target processor*

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

   *The Project Options dialog appears.*

2. Expand the **Processor** entry and select **Processor Definition**.




3. In the **Target processor** list select (for example) **TC10GP**.

4. **C**lick **OK** to accept the new project settings.

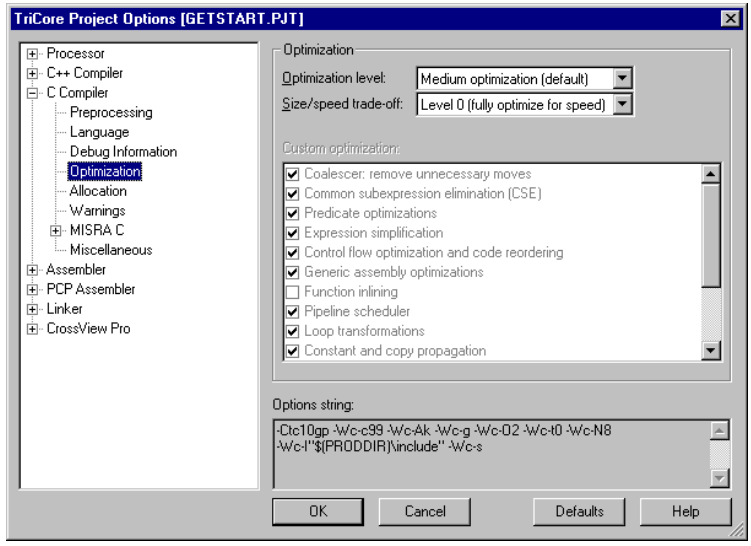
### *Set tool options*

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

   *The Project Options dialog appears. Here you can specify options that are valid for the entire project. To overrule the project options for the currently active file instead, from the **Project** menu select **Current File Options...***

2. Expand the **C Compiler** entry.

   *The C Compiler entry contains several pages where you can specify C compiler settings.*




3. For each page make your changes. If you have made all changes click **OK**.

The **Cancel** button closes the dialog without saving your changes. With the **Defaults** button you can restore the default project options (for the current page, or all pages in the dialog).

4. Make your changes for all other entries (Assembler, Linker, CrossView Pro) of the Project Options dialog in a similar way as described above for the C compiler.

If available, the **Options string** field shows the command line options that correspond to your graphical selections.

## 2.7 BUILD YOUR APPLICATION

If you have set all options, you can actually compile the file(s). This results in an absolute ELF/DWARF object file which is ready to be debugged.

### *Build your Application*

To build the currently active project:

- Click on the **Execute 'Make' command** button.

*The file is compiled, assembled, linked and located. The resulting file is* `getstart.elf`.

The build process only builds files that are out–of–date. So, if you click **Make** again in this example nothing is done, because all files are up–to–date.

### *Viewing the Results of a Build*

Once the files have been processed, you can see which commands have been executed (and inspect generated messages) by the build process in the **Build** tab of the **Output** window.

This window is normally open, but if it is closed you can open it by selecting the **Output** menu item in the **Window** menu.

### *Compiling a Single File*

1. Select the window (document) containing the file you want to compile or assemble.
2. Click on the **Execute 'Compile' command** button. The following button is the execute Compile button which is located in the toolbar.

*If you selected the file* `hello.c`*, this results in the compiled and assembled file* `hello.o`.

***Rebuild your Entire Application***

If you want to compile, assemble and link/locate all files of your project from scratch (regardless of their date/time stamp), you can perform a rebuild.

- Click on the **Execute 'Rebuild' command** button. The following button is the execute Rebuild button which is located in the toolbar.

## 2.8 HOW TO BUILD YOUR APPLICATION ON THE COMMAND LINE

If you are not using EDE, you can build your entire application on the command line. The easiest way is to use the *control program* **cctc**.

1. In a text editor, write the file `hello.c` with the following contents:

```
#include <stdio.h>

void main(void)
{
    printf("Hello World!\n");
}
```

2. Build the file `getstart.elf`:

```
cctc -ogetstart.elf hello.c -v
```

*The control program calls all tools in the toolchain. The* ***-v*** *option shows all the individual steps. The resulting file is* `getstart.elf`.

## 2.9 DEBUG GETSTART.ELF

The application `getstart.elf` is the final result, ready for execution and/or debugging. The debugger uses `getstart.elf` for debugging but needs symbolic debug information for the debugging process. This information must be included in `getstart.elf` and therefore you need to compile and assemble `hello.c` once again.

```
cctc -g -ogetstart.elf hello.c
```

Now you can start the debugger with `getstart.elf` and see how it executes.

***Start CrossView Pro***

- Click on the **Debug application** button.

*CrossView Pro is launched. CrossView Pro will automatically download the file* `getstart.elf` *for debugging.*

See the *CrossView Pro Debugger User's Guide* for more information.

CHAPTER 3

# TRICORE C LANGUAGE

TASKING

# CHAPTER 3

## 3.1 INTRODUCTION

The TASKING C cross–compiler (**ctc**) fully supports the ISO C standard and adds extra possibilities to program the special functions of the TriCore.

In addition to the standard C language, the compiler supports the following:

- extra data types, like `__fract`, `__laccum` and `__packb`
- intrinsic (built–in) functions that result in TriCore specific assembly instructions
- pragmas to control the compiler from within the C source
- predefined macros
- the possibility to use assembly instructions in the C source
- keywords to specify memory types for data and functions
- attributes to specify alignment and absolute addresses

All non–standard keywords have two leading underscores (__).

In this chapter the TriCore specific characteristics of the C language are described, including the above mentioned extensions.

## 3.2 DATA TYPES

### 3.2.1 FUNDAMENTAL DATA TYPES

The TriCore architecture defines the following fundamental data types:

- An 8–bit byte
- A 16–bit short
- A 32–bit word
- A 64–bit double word

The next table shows the mapping between these fundamental data types and the C language data types.

| Type | Keyword | Size (bit) | Align (bit) | Ranges |
|---|---|---|---|---|
| Boolean | `_Bool` | 8 | 8 | 0 or 1 |
| Character | `char`<br>`signed char` | 8 | 8 | $-2^7$ .. $2^7-1$ |
| | `unsigned char` | 8 | 8 | 0 .. $2^8-1$ |
| Integral | `short`<br>`signed short` | 16 | 16 | $-2^{15}$ .. $2^{15}-1$ |
| | `unsigned short` | 16 | 16 | 0 .. $2^{16}-1$ |
| | `int`<br>`signed int`<br>`long`<br>`signed long` | 32 | 16 | $-2^{31}$ .. $2^{31}-1$ |
| | `unsigned int`<br>`unsigned long` | 32 | 16 | 0 .. $2^{32}-1$ |
| | `enum` | 8<br>16<br>32 | 8<br>16 | $-2^7$ .. $2^7-1$<br>$-2^{15}$ .. $2^{15}-1$<br>$-2^{31}$ .. $2^{31}-1$ |
| | `long long`<br>`signed long long` | 64 | 32 | $-2^{63}$ .. $-2^{63}-1$ |
| | `unsigned long long` | 64 | 32 | 0 .. $2^{64}-1$ |
| Pointer | `pointer to data`<br>`pointer to func` | 32 | 32 | 0 .. $2^{32}-1$ |
| Floating Point | `float` | 32 | 16 | $-3.402e^{38}$ .. $-1.175e^{-38}$<br>$1.175e^{-38}$ .. $3.402^{e38}$ |
| | `double`<br>`long double` | 64 | 32 | $-1.797e^{308}$ .. $-2.225e^{-308}$<br>$2.225e^{-308}$ .. $1.797e^{308}$ |

*Table 3–1: Data Types*

When you use the `enum` type, the compiler will use the smallest sufficient integer type, unless you use compiler option **--integer-enumeration** (always use 32–bit integers for enumeration).

See also the *TriCore Embedded Applications Binary Interface (EABI)*.

### 3.2.2 FRACTIONAL DATA TYPES

The TASKING TriCore C compiler **ctc** additionally supports the following *fractional* types:

| Type | Keyword | Size (bit) | Align (bit) | Ranges |
|---|---|---|---|---|
| Fract | `__sfract` | 16 | 16 | [–1, 1> |
| | `__fract` | 32 | 32 | [–1, 1> |
| Accum | `__laccum` | 64 | 64 | [–131072,131072> |

*Table 3-2: Fractional Data Types*

The `__sfract` type has 1 sign bit + 15 mantissa bits
The `__fract` type has 1 sign bit + 31 mantissa bits
The `__laccum` type has 1 sign bit + 17 integral bits + 46 mantissa bits.

The `_accum` type is only included for compatibility reasons and is mapped to `__laccum`.

The TASKING C compiler **ctc** fully supports fractional data types which allow you to use normal expressions:

```
__fract f, f1, f2   /* Declaration of fractional variables    */

f1 = 0.5;           /* Assignment of a fractional constants   */
f2 = 0.242;

f = f1 * f2         /* Multiplication of two fractionals      */
```

The TriCore instruction set supports most basic operation on fractional types directly. To obtain more portable code, you can use several intrinsic functions that use fractional types. Fractional values are automatically saturated.

Section 3.5, *Intrinsic Functions* explains intrinsic functions.

Section 1.5.2, *Fractional Arithmetic Support* in Chapter *TriCore C Language* of the *Reference Guide* lists the intrinsic functions.

***Promotion rules***

For the three fractional types, the promotion rules are similar to the promotion rules for `char`, `short`, `int`, `long` and `long long`. This means that for an operation on two different fractional types, the smaller type is promoted to the larger type before the operation is performed.

When you mix a fractional type with a `float` or `double` type, the fractional number is first promoted to `float` respectively `double`.

When you mix an integer type with the `__laccum` type, the integer is first promoted to `__laccum`.

Because of the limited range of `__sfract` and `__fract`, only a few operations make sense when combining an integer with an `__sfract` or `__fract`. Therefore, the TriCore compiler only supports the following operations for integers combined with fractional types:

| left | oper | right | result |
|---|---|---|---|
| fractional | * | integer | fractional |
| integer | * | fractional | fractional |
| fractional | / | integer | fractional |
| integer | / | fractional | integer |
| fractional | << | integer | fractional |
| fractional | >> | integer | fractional |
| fractional: __sfract, __fract<br>integer: char, short, int, long, long long | | | |

*Table 3–3: Fractional operations for integers with fractional types*

### 3.2.3 BIT DATA TYPE

The TASKING TriCore C compiler **ctc** additionally supports the *bit* data type:

| Type | Keyword | Size (bit) | Align (bit) | Range |
|---|---|---|---|---|
| Bit | `__bit` | 8 | 8 | 0 or 1 |

*Table 3–4: Bit Data Type*

The TriCore instruction set supports some operations of the `__bit` type directly.

The following rules apply to `__bit` type variables:

- A `__bit` type variable is always unsigned.
- A `__bit` type variable can be exchanged with all other type–variables. The compiler generates the correct conversion.

  A `__bit` type variable is like a boolean. Therefore, if you convert an `int` type variable to a `__bit` type variable, it becomes 1 (true) if the integer is not equal to 0, and 0 (false) if the integer is 0. The next two C source lines have the same effect:

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

- Pointer to `__bit` is **not** allowed when it has the `__atbit()` qualifier.
- The `__bit` type is allowed as a structure member.
- A `__bit` type variable is allowed as a parameter of a function.
- A `__bit` type variable is allowed as a return type of a function.
- A `__bit` typed expression is allowed as switch expression.
- The `sizeof` of a `__bit` type is 1.
- Global or static `__bit` type variable can be initialized.
- A `__bit` type variable can be declared absolute using the `__atbit` attribute. See section 3.3.2 *Declare a Data Object at an Absolute Address: __at() and __atbit()* for more details.
- A `__bit` type variable can be declared volatile.

### *Promotion Rules*

For the `__bit` type, the promotion rules are similar to the promotion rules for `char`, `short`, `int`, `long` and `long long`.

### 3.2.4 PACKED DATA TYPES

The TASKING TriCore C compiler **ctc** additionally supports the following *packed* types:

| Type | Keyword | Size (bit) | Align (bit) | Ranges |
|---|---|---|---|---|
| Packed | `__packb`<br>`signed __packb` | 32 | 16 | 4x: $-2^7$ .. $2^7-1$ |
| | `unsigned __packb` | 32 | 16 | 4x: 0 .. $2^8-1$ |
| | `__packhw`<br>`signed __packhw` | 32 | 16 | 2x: $-2^{15}$ .. $2^{15}-1$ |
| | `unsigned __packhw` | 32 | 16 | 2x: 0 .. $2^{16}-1$ |

*Table 3–5: Fractional Data Types*

A `__packb` value consists of four signed or unsigned `char` values.
A `__packhw` value consists of two signed or unsigned `short` values.

The TriCore instruction set supports a number of arithmetic operations on packed data types directly. For example, the following function:

```
__packb  add4 ( __packb a, __packb b )
{
     return a + b;
}
```

results into the following assembly code:

```
add4:
     add.b   d2,d4,d5
     ret16
```

Section 3.5, *Intrinsic Functions* explains intrinsic functions.

Section 1.5.3, *Packed Data Type Support* in Chapter *TriCore C Language* of the *Reference Guide* lists the intrinsic functions.

#### *Halfword Packed Unions and Structures*

To minimize space consumed by alignment padding with unions and structures, elements follow the minimum alignment requirements imposed by the architecture. The TriCore arichitecture supports access to 32–bit integer variables on halfword boundaries.

Because only doubles, circular buffers, `__laccum` or pointers require the full word access, structures that do not contain members of these types are automatically halfword (2–bytes) packed.

Structures and unions that are divisible by 64–bit or contain members that are divisible by 64–bit, are word packed to allow efficient access through `LD.D` and `ST.D` instructions. These load and store operations require word aligned structures that are divisible by 64–bit. If necessary, 64–bit divisible structure elements are aligned or padded to make the structure 64–bit accessible.

With **#pragma pack 2** you can disable the `LD.D/ST.D` structure and union copy optimization to ensure halfword structure and union packing when possible. This ”limited” halfword packing only supports structures and unions that do not contain double, circular buffer, `__laccum` or pointer type members and that are not qualified with **#pragma align** to get an alignment larger than 2–byte. With **#pragma pack 0** you turn off halfword packing again.

```
#pragma pack 2
typedef struct {
     unsigned char  uc1;
     unsigned char  uc2;
     unsigned short us1;
     unsigned short us2;
     unsigned short us3;
} packed_struct;
#pragma pack 0
```

When you place a **#pragma pack 0** before a structure or union, its alignment will not be changed:

```
#pragma pack 0
packed_struct pstruct;
```

The alignment of data sections and stack can also affect the alignment of the base address of a halfword packed structure. A halfword packed structure can be aligned on a halfword boundary or larger alignment. When located on the stack or at the beginning of a section, the alignment becomes a word, because of the minimum required alignment of data sections and stack objects. A stack or data section can contain any type of object. To avoid wrong word alignment of objects in the section, the section base is also word aligned.

## 3.3 MEMORY QUALIFIERS

You can use static memory qualifiers to allocate static objects in a particular part of the addressing space of the processor.

In addition, you can place variables at absolute addresses with the keyword `__at()`. If you declare an integer at an absolute address, you can declare a single bit of that variable as bit variable with the keyword `__atbit()`.

### 3.3.1 DECLARE A DATA OBJECT IN A SPECIAL PART OF MEMORY

With a memory qualifier you can declare a variable in a specific part of the addressing space. You can use the following memory qualifiers:

**__near** The declared data object will be located in the first 16 kB of a 256 MB block. These parts of memory are directly addressable with the absolute addressing mode (see section 4.4.1, *Operands and Addressing Modes*, in Chapter *TriCore Assembly Language*).

**__far** The data object can be located anywhere in the indirect addressable memory region.

If you do not specify `__near` or `__far`, the compiler chooses where to place the declared object. With the compiler option **-N** (maximum size in bytes for data elements that are default located in `__near` sections) you can specify the size of data objects which the compiler then by default places in near memory.

**__a0** The data object is located in a section that is addressable with a sign–extended 16–bit offset from address register `A0`.

**__a1** The data object is located in a section that is addressable with a sign–extended 16–bit offset from address register `A1`.

**__a8** The data object is located in a section that is addressable with a sign–extended 16–bit offset from address register `A8`.

**__a9** The data object is located in a section that is addressable with a sign–extended 16–bit offset from address register `A9`.

Address registers A0, A1, A8, and A9 are designated as system global registers. They are not part of either context partition and are not saved/restored across calls. They can be protected against write access by user applications.

By convention, A0 and A1 are reserved for compiler use, while A8 and A9 are reserved for OS or application use. A0 is used as a base pointer to the *small* data section, where global data elements can be accessed using base + offset addressing. A0 is initialized by the execution environment.

A1 is used as a base pointer to the *literal data section*. The literal data section is a read–only data section intended for holding address constants and program literal values. Like A0, it is initialized by the execution environment.

As noted, A8 and A9 are reserved for OS use, or for application use in cases where the application and OS are tightly coupled.

All these memory qualifiers (__near, __far, __a0, __a1, __a8 and __a9) are related to the object being defined, they influence *where* the object will be located in memory. They are not part of the type of the object defined. Therefore, you cannot use these qualifiers in typedefs, type casts or for members of a struct or union.

***Examples:***

To declare a fast accessible integer in directly addressable memory:

```
int __near Var_in_near;
```

To allocate a pointer in far memory (the compiler will not use absolute addressing mode):

```
__far int *Ptr_in_far;
```

To declare and initialize a string in A0 memory:

```
char __a0 string[] = "TriCore";
```

If you use the __near memory qualifier, the compiler generates faster access code for those (frequently used) variables. Pointers are always 32–bit.

Functions are by default allocated in ROM. In this case you can omit the a memory qualifier. You cannot use memory qualifiers for function return values.

Some examples of using memory qualifiers:

```
int __near *p;     /* pointer to int in __near memory
                      (pointer has 32-bit size)      */
int __far *g;      /* pointer to int in __far memory
                      (pointer has 32-bit size)      */

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

You cannot use memory qualifiers in structure declarations:

```
struct S {
   __near int i;  /* put an integer in near
                     memory: Incorrect !       */
   __far int *p;  /* put an integer pointer in
                     far memory: Incorrect !   */
   }
```

If a library function declares a variable in near memory and you try to redeclare the variable in far memory, the linker issues an error:

```
extern int _near foo;   /* extern int in near memory*/

int __far foo;          /* int in far memory        */
```

The usage of the variables is always without a storage specifier:

```
char __near example;
example = 2;
```

The generated assembly would be:

```
mov16 d15,2
st.b  example,d15
```

All allocations with the same storage specifiers are collected in units called 'sections'. The section with the `__near` attribute will be located within the first 16 kB of of each 256 MB block.

With the linker it is possible to control the location of sections manually. See Chapter 7 Linker.

### 3.3.2 DECLARE A DATA OBJECT AT AN ABSOLUTE ADDRESS: __at() AND __atbit()

Just like you can declare a variable in a specific *part* of memory, you can also place an object at an *absolute address* in memory. This may be useful to interface with other programs using fixed memory schemes, or to access special function registers.

With the attribute `__at()` you can specify an absolute address.

***Examples***

```
int myvar __at(0x100);
```

The variable `myvar` is placed at address 0x100.

```
unsigned char Display[80*24] __at( 0x2000 )
```

The array `Display` is placed at address 0x2000. In the generated assembly, an absolute section is created. On this position space is reserved for the variable `Display`.

***Restrictions***

Take note of the following restrictions if you place a variable at an absolute address:

- You can place only global variables at absolute addresses. Parameters of functions, or automatic variables within functions cannot be placed at absolute addresses.
- 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, because an assembler external object cannot specify an absolute address.
- When the variable is declared `static`, no public symbol will be generated (normal C behavior).
- You cannot place functions at absolute addresses.
- Absolute variables cannot overlap each other. If you declare two absolute variables at the same address, the assembler and / or linker issues an error. The compiler does not check this.
- When you declare the same absolute variable within two modules, this produces conflicts during link time (except when one of the modules declares the variable 'extern').

### *Declaring a bit variable with __atbit()*

If you have defined a 32–bits base variable (`int`, `long`) you can declare a single bit of that variable as a bit variable with the keyword `__atbit()`. The syntax is:

**__atbit(** *name*, *offset* **)**

*name* is the name of an integer variable in which the bit is located. *offset* (range 0–31) is the bit–offset within the variable.

If you have defined an absolute integer variable with the keyword `__at()`, you can declare a single bit of that variable as an absolute bit variable with `__atbit()`.

#### *Example*

```
int          bw   __at(0x100);
__bit        myb  __atbit( bw, 3 );
```

Note that the keyword `__bit` is used to declare the variable `myb` as a bit, and that the keyword `__atbit()` is used to declare that variable at an absolute offset in variable `bw`.

See also section 3.2.3, *Bit Data Type*.

#### *Restrictions*

- You can only use the `__atbit()` qualifier on variables of type `__bit`.
- When a variable is `__atbit()` qualified it represents an alias of a bit in another variable. Therefore, it cannot be initialized.
- You can only use the `__atbit()` qualifier on variables which have either a global scope or file scope.

## 3.4 DATA TYPE QUALIFIERS

### 3.4.1 CIRCULAR BUFFERS: __circ

The TriCore core has support for implementing specific DSP tasks, such as finite impulse response (FIR) and infinite impulse response (IIR) filters and fast Fourier transforms (FFTs). For the FIR and IIR filters the TriCore architecture supports the circular addressing mode and for the FFT the bit–reverse addressing mode. The TriCore C compiler supports *circular buffers* for these DSP tasks. This way, the TriCore C compiler makes hardware features available at C source level instead of at assembly level only.

A *circular buffer* is a linear (one dimensional) array that you can access by moving a pointer through the data. The pointer can jump from the last location in the array to the first, or vice–versa (the pointer wraps–around). This way the buffer appears to be continuous. The TriCore C compiler supports the `__circ` keyword (circular addressing mode) for this type of buffer.

***Example: __circ***

```
__fract __circ circbuffer[10];
__fract __circ *ptr_to_circbuffer = circbuffer;
```

Here, `circbuffer` is declared as a circular buffer. The compiler aligns the base address of the buffer on the access width (in this example an `int`, so 4 bytes). The compiler keeps the buffer size and uses it to control pointer arithmetic of pointers that are assigned to the buffer later.

You can perform operations on circular pointers with the usual C pointer arithmetic with the difference that the pointer will wrap. When you acces the circular buffer with a circular pointer, it wraps at the buffer limits. Circular pointer variables are 64 bits in size.

Example:

```
while( *Pptr_to_circbuf++ );
```

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

Example:

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

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

If you want to initialize a circular pointer with a dynamically allocated buffer at run–time, you should use the intrinsic function `__initcirc()`:

```
#define N 100
unsigned short s = sizeof(__fract);
__fract *ptr_to_circbuf = calloc( N, s );
circbuf = __initcirc( ptr_to_circbuf, N * s, 0 * s );
```

### 3.4.2 DECLARE AN SFR BIT FIELD: __sfrbit16 AND __sfrbit32

With the data type qualifiers `__sfrbit16` and `__sfrbit32` you can declare bit fields in special function registers.

According to the *TriCore Embedded Applications Binary Interface*, 'normal' bit fields are accessed as `char`, `short` or `int`. Thus:

- fields with a width of 8–bits or less impose only byte alignments
- fields with a width from 9 to 16 bits impose halfword alignment
- fields with a width from 17 to 32 bits impose word alignment

If you declare bit fields in special function registers, this behavior is not always desired: some special function registers require 16–bit or 32–bit access. To force 16–bit or 32–bit access, you can use the data type qualifiers `__sfrbit16` and `__sfrbit32`.

For each supported target, a special function register file (`reg`*`cpu_name`*`.sfr`) is delivered with the TriCore toolchain. In normal circumstances you should not need to declare SFR bit fields.

***Example***

The next example is part of an SFR file and illustrates the declaration of a special function register using the data type qualifier `__sfrbit32`:

```
typedef volatile union
{
   struct
   {
   unsigned __sfrbit32 SRPN : 1;  /* BCU Service Priority
                                        Number            */
   unsigned int : 2;
   unsigned __sfrbit32 TOS  : 2;  /* BCU Type-of-Service
                                        Control           */
   unsigned __sfrbit32 SRE  : 1;  /* BCU Service Request
                                        Enable Control    */
   unsigned __sfrbit32 SRR  : 1;  /* BCU SerService Request
                                        Flag              */
   unsigned __sfrbit32 CLRR : 1;  /* BCU Request Clear Bit */
   unsigned __sfrbit32 SETR : 1;  /* BCU Request Set Bit   */
   unsigned int : 16;
   } B;

   int I;
} BCU_SRC_type;

#define BCU_SRC (*(BCU_SRC_type*)(0xF00002FC))
                        /* BCU Service Request Node         */
```

You can now access the register and bit fields as follows:

```
#include <regtc10gp.sfr>

BCU_SRC.I |= 0xb32a;   /* access BCU Service Request
                          Control register as a whole      */

BCU_SRC.B.SRE = 0x1;   /* access SRE bit field of BCU
                          Service Request Control register */
```

### *Restrictions*

You can use the `__sfrbit32` and `__sfrbit16` data type qualifiers only for `int` types. The compiler issues an error if you use for example `__sfrbit32 char x : 8;`

When you use the `__sfrbit32` and `__sfrbit16` data type qualifiers for other types than a bit field, the compiler ignores this without a warning. For example, `__sfrbit32 int global;` is equal to `int global;`.

Structure or unions that contain a member qualified with `__sfrbit16`, are zero padded to complete a halfword if necessary. The structure or union will be halfword aligned.
Structures or unions that contain a member qualified with `__sfrbit32`, are zero padded to complete a full word if necessary. The structure or union will be word aligned.

## 3.5 INTRINSIC FUNCTIONS

Some specific TriCore assembly instructions have no equivalence in C. *Intrinsic functions* give the possibility to use these instructions. Intrinsic functions are predefined functions that are recognized by the compiler. The compiler then generates the most efficient assembly code for these functions.

The compiler always inlines the corresponding assembly instructions in the assembly source rather than calling the function. This avoids unnecessary parameter passing and register saving instructions which are normally necessary when a function is called.

Intrinsic functions produce very efficient assembly code. Though it is possible to inline assembly code by hand, registers are used even more efficient by intrinsic functions. At the same time your C source remains very readable.

You can use intrinsic functions in C as if they were ordinary C (library) functions. All intrinsics begin with a double underscore character. The following example illustrates the use of an intrinsic function and its resulting assembly code.

```
x = __min( 4,5 );
```

The resulting assembly code is inlined rather than being called:

```
mov16  d2,#4
min    d2,d2,#5
```

The intrinsics cover the following subjects:

- Minimum and maximum of (short) integers
- Fractional data type support
- Packed data type support
- Interrupt handling
- Insert single assembly instruction
- Register handling
- Insert / extract bitfields and bits
- Miscellaneous

For extended information about all available intrinsic functions, refer to section 1.5, *Intrinsic Functions*, in Chapter *TriCore C Language* of the *Reference Guide*.

## 3.6 USING ASSEMBLY IN THE C SOURCE: __asm()

With the `__asm()` keyword you can use assembly instructions in the C source and pass C variables as operands to the assembly code. Be aware that C modules that contain assembly are not portable and harder to compile in other environments.

The compiler does not interpret assembly blocks but passes the assembly code to the assembly source file. Possible errors can only be detected by the assembler.

***General syntax of the* `__asm` *keyword***

```
__asm( "instruction_template"
       [ : output_param_list
       [ : input_param_list
       [ : register_save_list]]] );
```

| | |
|---|---|
| *instruction_template* | Assembly instructions that may contain parameters from the input list or output list in the form: **%***parm_nr* |
| **%***parm_nr*[**.***regnum*] | Parameter number in the range 0 .. 9. With the optional **.***regnum* you can access an individual register from a register pair or register quad. For example, with register pair `d0/d1`, `.0` selects register `d0`. |
| *output_param_list* | [[ **"=**[**&**]*constraint_char***"(***C_expression***)**],...] |
| *input_param_list* | [[ **"***constraint_char***"(***C_expression***)**],...] |
| **&** | Says that an output operand is written to before the inputs are read, so this output must not be the same register as any input. |
| *constraint _char* | Constraint character: the type of register to be used for the *C_expression*. (see table 3–6) |
| *C_expression* | Any C expression. For output parameters it must be an *lvalue*, that is, something that is legal to have on the left side of an assignment. |
| *register_save_list* | [[**"***register_name***"**],...] |
| *register_name* | Name of the register you want to reserve. |

### *Typical example: multiplying two C variables using assembly*

```
int a,b,result;

void main( void )
{
    __asm("mul\t%1,%2,%0" : "=d"(result) : "d"(a), "d"(b) );
}
```

generated code:

```
ld.w    d15,a
ld.w    d0,b
mul     d15,d0,d15
st.w    result,d15
```

`%0` corresponds to the first C variable, `%1` corresponds to the second and so on. The escape sequence `\t` generates a tab.

### *Specifying registers for C variables*

With a *constraint character* you specify the register *type* for a parameter. In the example above, the `d` is used to force the use of data registers for the parameters `a`, `b` and `result`.

You can reserve the registers that are used in the assembly instructions, either in the parameter lists or in the reserved register list (*register_save_list*). The compiler takes account of these lists, so no unnecessary register saves and restores are placed around the inline assembly instructions.

| Constraint character | Type | Operand | Remark |
|---|---|---|---|
| a | Address register | a0 .. a15 | |
| d | Data register | d0 .. d15 | |
| e | Data register pair | e0 .. e7 | |
| m | Memory | *variable* | Stack or memory operand |
| *number* | Type of operand it is associated with | same as %*number* | Indicates that %*number* and *number* are the same register. |

*Table 3–6: Available input/output operand constraints*

### *Loops and conditional jumps*

The compiler does not detect loops with multiple `__asm` statements or (conditional) jumps across `__asm` statements 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. The same counts for (conditional) jumps. As a rule of thumb, all references to a label in an `__asm` statement must be in that same statement.

### *Example 1: no input or output*

A simple example without input or output parameters. You can just output any assembly instruction:

```
__asm( "nop" );
```

Generated code:

```
nop
```

### *Example 2: using output parameters*

Assign the result of inline assembly to a variable. With the constraint `d` a data register is chosen for the parameter; the compiler decides which data register it uses. The `%0` in the instruction template is replaced with the name of this data register. Finally, the compiler generates code to assign the result to the output variable.

```
int result;

void main( void )
{
    __asm( "mov %0,#0xFF" : "=d"(result));
}
```

Generated assembly code:

```
mov   d15,#0xFF
st.w  result,d15
```

### *Example 3: using input and output parameters*

Multiply two C variables and assign the result to a third C variable. Data type registers are necessary for the input and output parameters (constraint `d`, `%0` for `result`, `%1` for `a` and `%2` for `b` in the instruction template). The compiler generates code to move the input expressions into the input registers and to assign the result to the output variable.

```
int a, b, result;

void multiply( void )
{
    __asm( "mul %1, %2, %0": "=d"(result): "d"(a), "d"(b) );
}

void main(void)
{
    multiply();
}
```

Generated assembly code:

```
multiply:
    ld.w   d15,a
    ld.w   d0,b
    mul    d15, d0, d15
    st.w   result,d15

main:
    jg     multiply
```

### *Example 4: reserve registers*

If you use registers in the `__asm` statement, reserve them. Same as *Example 3*, but now register `d0` is a reserved register. You can do this by adding a reserved register list (`: "d0"`) (sometimes referred to as 'clobber list'). As you can see in the generated assembly code, register `d0` is not used (the first register used is `d1`).

```
int a, b, result;

void multiply( void )
{
 __asm( "mul %1, %2, %0": "=d"(result): "d"(a), "d"(b) : "d0" );
}
```

Generated assembly code:

```
ld.w   d15,a
ld.w   d1,b
mul    d15, d1, d15
st.w   result,d15
```

### *Example 5: input and output are the same*

If the input and output must be the same you must use a number constraint. The following example inverts the value of the input variable `ivar` and returns this value to `ovar`. Since the assembly instruction `not` uses only one register, the return value has to go in the same place as the input value. To indicate that `ivar` uses the same register as `ovar`, the constraint '0' is used which indicates that `ivar` also corresponds with %0.

```
int ovar;

void invert(int ivar)
{
    __asm ("not %0": "=d"(ovar): "0"(ivar) );
}

void main(void)
{
    invert(255);
}
```

Generated assembly code:

```
invert:
    not   d4
    st.w  ovar,d4

main:
    mov   d4,#255
    jg    invert
```

### *Example 6: writing your own intrinsic function*

Because you can use any assembly instruction with the `__asm` keyword, you can use the `__asm` keyword to create your own intrinsic functions. The essence of an intrinsic function is that it is inlined.

First write a function with assembly in the body using the keyword `__asm`. We use the multiply routine from *Example 3*.

Next make sure that the function is inlined rather than being called. You can do this with the function qualifier `inline`. This qualifier is discussed in more detail in section 3.9.1, *Inlining Functions*.

```
int a, b, result;

inline void __my_mul( void )
{
    __asm( "mul %1, %2, %0": "=d"(result): "d"(a), "d"(b) );
}

void main(void)
{
    // call to function __my_mul
    __my_mul();
}
```

Generated assembly code:

```
main:
    ; __my_mul code is inlined here
    ld.w  d15,a
    ld.w  d0,b
    mul   d15, d0, d15
    st.w  result,d15
```

As you can see, the generated assembly code for the function `__my_mul` is inlined rather than called.

### *Example 7: accessing individual registers in a register pair*

You can access the individual registers in a register pair by adding a '.' after the operand specifier in the assembly part, followed by the index in the register pair.

```
int f1, f2;

void foo(double d)
{
    __asm ("ld.w %0, %2.0\n"
        "   ld.w %1, %2.1"
          : "=&d"(f1), "=d"(f2): "e"(d) );
}
```

The first `ld.w` instruction uses index #0 of argument 2 (which is a double placed in a DxDx register) and the second `ld.w` instruction uses index #1. The input operand is located in register pair d4/d5. The assembly output becomes:

```
ld.w   d15, d4
ld.w   d0, d5
st.w   f1,d15
st.w   f2,d0
ret16
```

If the index is not a valid index (for example, the register is not a register pair, or the argument has not a register constraint), the '.' is passed into the assembly output. This way you can still use the '.' in assembly instructions.

## 3.7 CONTROLLING THE COMPILER: PRAGMAS

*Pragmas* are keywords in the C source that control the behavior of the compiler. Pragmas sometimes overrule compiler options and keywords. In general pragmas give directions to the code generator of the compiler.

For example, you can set a compiler option to specify which optimizations the compiler should perform. With the `#pragma optimize` *`flags`* you can set an optimization level for a specific part of the C source. This overrules the general optimization level that is set in the compiler options dialog (command line option **-O**).

Some pragmas have an equivalent command line option. This is useful if you want to overrule certain keywords in the C source without the need to change the C source itself.

See section 4.1, *Compiler Options*, in Chapter 4, *Tool Options*, of the *Reference Guide*.

The compiler recognizes the following pragmas, other pragmas are ignored.

```
#pragma align n
#pragma align restore
```

(See compiler option **--align** in section *Compiler Options* in Chapter *Tool Options* of the *Reference Guide*.)

```
#pragma clear
#pragma noclear
```

Performs 'clearing' or no 'clearing' of non–initialized static/public variables.

```
#pragma default_a0_size [value]
```

(See compiler option **-Z** in section *Compiler Options* in Chapter *Tool Options* of the *Reference Guide*.)

```
#pragma default_near_size [value]
```

(See compiler option **-N** in section *Compiler Options* in Chapter *Tool Options* of the *Reference Guide*.)

```
#pragma inline
#pragma noinline
#pragma smartinline
```

(See section 3.9.1, *Inlining Functions*.)

```
#pragma optimize flags
#pragma endoptimize
#pragma optimize restore
```

(See section 5.3, *Compiler Optimizations* in Chapter *Using the Compiler*)

```
#pragma pack 2
#pragma pack 0
```

(See section 3.2.4, *Packed Data Types*)

```
#pragma section all "section_name"
#pragma section section_type "section_name"
#pragma section code_init
#pragma section data_overlay
```

(See section 3.10, *Compiler Generated Sections* and compiler option **-R** in section *Compiler Options* in Chapter *Tool Options* of the *Reference Guide*.)

```
#pragma source
#pragma nosource
```

(See compiler option **-s** in section *Compiler Options* in Chapter *Tool Options* of the *Reference Guide*.)

```
#pragma switch auto
#pragma switch jumptab
#pragma switch linear
#pragma switch lookup
#pragma switch restore
```

(See section 3.11, *Switch Statement*)

## 3.8 PREDEFINED MACROS

In addition to the predefined macros required by the ISO C standard, the TASKING TriCore C compiler supports the predefined macros as defined in Table 3–7. The macros are useful to create conditional C code.

| Macro | Description |
|---|---|
| __DOUBLE_FP__ | Defined when you do not use compiler option **–F** (Treat double as float) |
| __SINGLE_FP__ | Defined when you use compiler option **–F** (Treat double as float) |
| __FPU__ | Defined when you use compiler option **––fpu–present** (Use hardware floating point instructions) |
| __CTC__ | Identifies the compiler. You can use this symbol to flag parts of the source which must be recognized by the **ctc** compiler only. It expands to the version number of the compiler. |
| __TASKING__ | Identifies the compiler as the TASKING TriCore compiler. It expands to 1. |
| __DSPC__ | Indicates conformation to the DSP–C standard. It expands to 1. |
| __DSPC_VERSION__ | Expands to the decimal constant 200001L. |

*Table 3–7: Predefined macros*

***Example***

```
#ifdef __CTC__
/* this part is for the TriCore compiler */
...
#endif
```

## 3.9 FUNCTIONS

### 3.9.1 INLINING FUNCTIONS: INLINE

You can use the `inline` keyword to tell the compiler to inline the function body instead of calling the function. Use the `__noinline` keyword to tell the compiler *not* to inline the function body.

You must define inline functions in the same source module as in which you call the function, because the compiler only inlines a function in the module that contains the function definition. When you need to call the inline function from several source modules, you must include the definition of the inline function in each module (for example using a header file).

The compiler inserts the function body at the place the function is called. If the function is not called at all, the compiler does not generate code for it.

***Example: inline***

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

inline int add( int a, int b )
{
     int i = 4;
     return( a + b );
}

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

The function `add()` is defined before it is called. The compiler inserts (optimized) code for both calls to the `add()` function. The generated assembly is:

```
main:
    mov16  d15,#3
    st.w   w,d15

    ld.w   d15,x
    ld.w   d0,y
    add16  d0,d15
    st.w   z,d0
```

***Example: #pragma inline / #pragma noinline***

Instead of the `inline` qualifier, you can also use `#pragma inline` and `#pragma noinline` to inline a function body:

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

#pragma inline
int add( int a, int b )
{
     int i=4;
     return( a + b );
}
#pragma noinline

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

If a function has an `inline`/`__noinline` function qualifier, then this qualifier will overrule the current pragma setting.

***#pragma smartinline***

Default, small fuctions that are not too often called, are inlined. This reduces execution speed at the cost of code size (compiler option **–Oi**).

With the `#pragma noinline / #pragma smartinline` you can temporarily disable this optimization.

With the compiler options **--inline–max–incr** and **--inline–max–size** you have more control over the function inlining process of the compiler.

See for more information of these options, section *Compiler Options* in Chapter *Tool Options* of the *TriCore Reference Guide*.

***Combining inline with __asm to create intrinsic functions***

With the keyword `__asm` it is possible to use assembly instructions in the body of an inline function. Because the compiler inserts the (assembly) body at the place the function is called, you can create your own intrinsic function.

See section 3.6, *Using Assembly in the C Source*, for more information about the `__asm` keyword.
*Example 6* in that section shows how in combination with the `inline` keyword an intrinsic function is created.

## 3.9.2 INTERRUPT AND TRAP FUNCTIONS

The TriCore C compiler supports a number of function qualifiers and keywords to program interrupt service routines (ISR) or trap handlers. Trap handlers may also be defined by the operating system if your target system uses one.

An *interrupt service routine* (or: interrupt function, or: interrupt handler) is called when an interrupt event (or: *service request*) occurs. This is always an external event; peripherals or external inputs can generate an interrupt signals to the CPU to request for service.
Unlike other interrupt systems, each interrupt has a unique *interrupt request priority number* (IRPN). This number is (0 to 255) is set as the *pending interrupt priority number* (PIPN) in the interrupt control register (ICR) by the interrupt control unit. If multiple interrupts occur at the same time, the priority number of the request with the hightest priority is set, so this interrupt is handled.

The TriCore vector table provides an entry for each pending interrupt priority number, not for a specific interrupt source. A request is handled if the priority number is higher then the CPU priority number (CCPN). An interrupt service routine can be interrupted again by another interrupt request with a higher priority. Interrupts with priority number 0 are never handled.

A *trap service routine* (or: trap function, or: trap handler) is called when a trap event occurs. This is always an event generated within or by the application. For example, a devide by zero or an invalid memory access.

With the following function qualifiers you can declare an interrupt handler or trap handler:

```
__interrupt()   __interrupt_fast()
__trap()        __trap_fast()
```

There is one special type of trap function which you can call manually, the system call exception (trap class 6). See section 3.9.2.3, *Defining a Trap Service Routine Class 6*.

```
__syscallfunc()
```

During the execution of an interrupt service routine or trap service routine, the system blocks the CPU from taking further interrupt requests. With the following keywords you can enable interrupts again, immediately after an interrupt or trap function is called:

```
__enable_       __bisr_()
```

### 3.9.2.1 DEFINING AN INTERRUPT SERVICE ROUTINE

Interrupt functions cannot accept arguments and do not return anything:

```
void __interrupt( vector ) isr( void )
{
...
}
```

The argument *vector* identifies the entry into the interrupt vector table (0..255). Unlike other interrupt systems, the priority number (PIPN) of the interrupt now being serviced by the CPU identifies the entry into the vector table.

For an extensive description of the TriCore interrupt system, see the *TriCore 1 Unified Processor Core v1.3 Architecture Manual, Doc v1.3.3* [2002–09, Infineon]

The compiler generates an interrupt service frame for interrupts. The difference between a normal function and an interrupt function is that an interrupt function ends with an RFE instruction instead of a RET, and that the lower context is saved and restored with a pair of SVLCX / RSLCX instructions when one of the lower context registers is used in the interrupt handler.

When you define an interrupt service routine with the `__interrupt()` qualifier, the compiler generates an entry for the interrupt vector table. This vector jumps to the interrupt handler.

When you define an interrupt service routine with the `__interrupt_fast()` qualifier, the interrupt handler is directly placed in the interrupt vector table, thereby eliminating the jump code. You should only use this when the interrupt handler is very small, as there is only 32 bytes of space available in the vector table. The compiler does not check this restriction.

***Example***

The next example illustrates the function definition for a function for a software interrupt with vector number 0x30:

```
int c;

void __interrupt( 0x30 ) transmit( void )
{
     c = 1;
}
```

### 3.9.2.2 DEFINING A TRAP SERVICE ROUTINE

The definition of a trap service routine is similar to the definition of an interrupt service routine. Trap functions cannot accept arguments and do not return anything:

```
void __trap( class ) tsr( void )
{
...
}
```

The argument *class* identifies the entry into the trap vector table. TriCore defines eight classes of trap functions. Each class has its own trap handler.

When a trap service routine is called, the `d15` register contains the so–called *Trap Identification Number* (TIN). This number identifies the cause of the trap. In the trap service routine you can test and branch on the value in `d15` to reach the sub–handler for a specific TIN.

The next table shows the classes supported by TriCore.

| Class | Description |
|---|---|
| Class 0 | Reset |
| Class 1 | Internal Protection Traps |
| Class 2 | Instruction Errors |
| Class 3 | Context Management |
| Class 4 | System Bus and Peripheral Errors |
| Class 5 | Assertion Traps |
| Class 6 | System Call |
| Class 7 | Non–Maskable Interrupt |

For a complete overview of the trap system and the meaning of the trap identification numbers, see the *TriCore 1 Unified Processor Core v1.3 Architecture Manual, Doc v1.3.3* [2002–09, Infineon]

Analogous to interrupt service routines, the compiler generates a trap service frame for interrupts.

When you define a trap service routine with the `__trap()` qualifier, the compiler generates an entry for the interrupt vector table. This vector jumps to the trap handler.

When you define a trap service routine with the `__trap_fast()` qualifier, the trap handler is directly placed in the trap vector table, thereby eliminating the jump code. You should only use this when the trap handler is very small, as there is only 32 bytes of space available in the vector table. The compiler does not check this restriction.

### 3.9.2.3 DEFINING A TRAP SERVICE ROUTINE CLASS 6: `__syscallfunc()`

A special kind of trap service routine is the system call trap. With a system call the trap service routine of class 6 is called. For the system call trap, the trap identification number (TIN) is taken from the immediate constant specified with the function qualifier `__syscallfunc()`:

```
__syscallfunc(TIN)
```

The TIN is a value in the range 0 and 255. You can only use `__syscallfunc()` in the function declaration. A function body is useless, because when you call the function declared with `__syscallfunc()`, a trap class 6 occurs which calls the corresponding trap service routine.

In case of the other traps, when a trap service routine is called, the *system* places a trap identification number in `d15`.

Unlike the other traps, a class 6 trap service routine can contain arguments and return a value. Arguments that are passed via the stack, remain on the stack of the caller because it is not possible to pass arguments from the user stack to the interrupt stack on a system call. This restriction, caused by the TriCore's run–time behavior, cannot be checked by the compiler.

The next example illustrates the definition of a class 6 trap service routine and the corresponding system call:

***Example***

```
__syscallfunc(1) int syscall_a( int, int );
__syscallfunc(2) int syscall_b( int, int );

int x;

void main( void )
{
     x = syscall_a(1,2);   // causes a trap class 6 with TIN = 1
     x = syscall_b(4,3);   // causes a trap class 6 with TIN = 2
}

int __trap( 6 ) trap6( int a, int b )    // trap class 6 handler
{
     int tin;
     __asm(st.w tin,d15);   // put d15 in C variable 'tin'

     switch( tin )
     {
     case 1:
          a += b;
          break;
     case 2:
          a –= b;
          break;
     default:
          break;
     }
     return a;
}
```

### 3.9.2.4 ENABLING INTERRUPT REQUESTS: __enable_, __bisr_()

***Enabling interrupt service requests***

During the execution of an interrupt service routine or trap service routine, the system blocks the CPU from taking further interrupt requests. You can immediately re–enable the system to accept interrupt requests:

```
__interrupt(vector) __enable_ isr( void )
__trap(class) __enable_ tsr( void )
```

The compiler generates an `enable` instruction as first instruction in the routine. The `enable` instruction sets the interrupt enable bit (ICR.IE) in the interrupt control register.

You can also generate the `enable` instruction with the `__enable()` intrinsic function, but it is not guaranteed that it will be the *first* instruction in the routine.

***Enabling interrupt service requests and setting CPU priority number***

The function qualifier `__bisr_()` also re–enables the system to accept interrupt requests. In addition, the *current CPU priority number* (CCPN) in the interrupt control register is set:

```
__interrupt(vector) __bisr_(CCPN) isr( void )
__trap(class) __bisr_(CCPN) tsr( void )
```

The argument *CCPN* is a number between 0 and 255. The system accepts all interrupt requests that have a higher pending interrupt priority number (PIPN) than the current CPU priority number. So, if the CPU priority number is set to 0, the system accepts all interrupts. If it is set to 255, no interrupts are accepted.

The compiler generates a `bisr` instruction as first instruction in the routine. The `bisr` instruction sets the interrupt enable bit (ICR.IE) and the current CPU priority number (ICR.CCPN) in the interrupt control register.

You can also generate the `bisr` instruction with the `__bisr()` intrinsic function, but it is not guaranteed that it will be the *first* instruction in the routine.

***Setting the CPU priority number in a Class 6 trap service routine***

The `bisr` instruction saves the lower context so passing and returning arguments is not possible. Therefore, you cannot use the function qualifier `__bisr_()` for class 6 traps.

Instead, you can use the function qualifier `__enable_` to set the ICR.IE bit, and the intrinsic function `__mtcr( int, int )` to set the ICR.CCPN value at the beginning of a class 6 trap service routine (or use the intrinsic function `__mtcr()` to set both the ICR.IE bit and the ICR.CCPN value).

### 3.9.3 FUNCTION CALLING MODES: __indirect

Functions are default called with a single word direct call. However, when you link the application and the target address appears to be out of reach (+/– 16 MB from the `callg` or `jg` instruction), the linker generates an error. In this case you can use the `__indirect` keyword to force the less efficient, two and a half word indirect call to the function:

```
int __indirect foo( void )
{
...
}
```

With compiler option **--indirect** you tell the compiler to generate far calls for *all* functions.

### 3.9.4 PARAMATER PASSING AND THE STACK MODEL: __stackparm

The parameter registers D4..D7 and A4..A7 are used to pass the initial function arguments. Up to 4 arithmetic types and 4 pointers can be passed this way. A 64–bit argument is passed in an even/odd data register pair. Parameter registers skipped because of alignment for a 64–bit argument are used by subsequent 32–bit arguments. Any remaining function arguments are passed on the stack. Stack arguments are pushed in reversed order, so that the first one is at the lowest address. On function entry, the first stack parameter is at the address (SP+0).

All function arguments passed on the stack are aligned on a multiple of 4 bytes. As a result, the stack offsets for all types except `float` are compatible with the stack offsets used by a function declared without a prototype.

Structures up to eight bytes are passed via a data register or data register pair. Larger structures are passed via the stack.

Arithmetic function results of up to 32 bits are returned in the D2 register. 64–bit arithmetic types are returned in the register pair D2/D3. Pointers are returned in A2, and circular pointers are returned in A2/A3.

When the function return type is a structure, it is copied to a "return area" that is allocated by the caller. The address of this area is passed as an implicit first argument in A4.

### *Stack Model: __stackparm*

The function qualifier `__stackparm` changes the standard calling convention of a function into a convention where all function arguments are passed via the stack, conforming a so–called stack model. This qualifier is only needed for situations where you need to use an indirect call to a function for which you do not have a valid prototype.

The compiler sets the least significant bit of the function pointer when you take the address of a function declared with the `__stackparm` qualifier, so that these function pointers can be identified at run–time. The least significant bit of a function pointer address is ignored by the hardware.

***Example***

```
void             plain_func ( int );
void __stackparm stack_func ( int );

void call_indirect ( unsigned int fp, int arg )
{
     typedef __stackparm void (*SFP)( int );
     typedef void (*RFP)( int );

     SFP     fp_stack;
     RFP     fp_reg;

     if ( fp & 1 )
     {
          fp_stack = (SFP) fp;
          fp_stack( arg );
     }
     else
     {
          fp_reg = (RFP) fp;
          fp_reg( arg );
     }
}

void main ( void )
{
     call_indirect( (unsinged int) plain_func, 1 );
     call_indirect( (unsinged int) stack_func, 2 );
}
```

## 3.10 COMPILER GENERATED SECTIONS

The compiler generates code and data in several types of sections. The compiler uses the following section naming convention:

*prefix.module–name.function–or–object–name*

The prefix depends on the type of the section and determines if the section is initialized, constant or uninitialized and which addressing mode is used.

| Type | Name prefix | Description |
|---|---|---|
| code | .text | program code |
| neardata | .zdata | initialized __near data |
| fardata | .data | initialized __far data |
| nearrom | .zrodata | constant __near data |
| farrom | .rodata | constant __far data |
| nearbss | .zbss | uninitialized __near data (cleared) |
| farbss | .bss | uninitialized __far data (cleared) |
| nearnoclear | .zbss | uninitialized __near data |
| farnoclear | .bss | uninitialized __far data |
| a0data | .sdata | initialized __a0 data |
| a0rom | .srodata | constant __a0 data |
| a0bss | .sbss | uninitialized __a0 data (cleared) |
| a1rom | .ldata | constant __a1 data |
| a8data | .data_a8 | initialized __a8 data |
| a8rom | .rodata_a8 | constant __a8 data |
| a8bss | .bss_a8 | uninitialized __a8 data (cleared) |
| a9data | .data_a9 | initialized __a9 data |
| a9rom | .rodata_a9 | constant __a9 data |
| a9bss | .bss_a9 | uninitialized __a9 data (cleared) |

*Table 3–8: Section types and name prefixes*

### *Rename sections*

You can change the default section names with one of the following pragmas:

`#pragma section` *`type "string"`*

All sections of the specified *type* will be named "*prefix*.*string*". For example,

```
#pragma section neardata "where"
```

all sections of type neardata have the name "`.zdata.where`".

`#pragma section` *`type`* will restore the default section naming for sections of this type.

`#pragma section` *`type`* `restore` will restore the previous setting of `#pragma section` *`type`*.

`#pragma section all` *`"string"`*

All sections will be named "*prefix*.*string*", unless you use a type specific renaming pragma. For example,

```
#pragma section all "here"
```

all sections have the syntax "*prefix*`.here`". For example, sections of type neardata have the name "`.zdata.here`".

`#pragma section all` will restore the default section naming (not for sections that have a type specific renaming pragma).

`#pragma section all restore` will restore the previous setting of `#pragma section all`.

Example:

```
#pragma section all "rename_1"
 //  .text.rename_1
 //  .data.rename_1

#pragma section code "rename_2"
 //  .text.rename_2
 //  .data.rename_1
```

See also compiler option **–R** in section *Compiler Options* in Chapter *Tool Options* of the *Reference Guide*.

### *Influence section definition*

The following pragmas also influence the section definition:

`#pragma section code_init`

The code section is copied from ROM to RAM at program startup.

`#pragma section data_overlay`

The `nearnoclear` and `farnoclear` sections can be overlaid by other sections with the same name. Since default section naming never leads to sections with the same name, you must force the same name by using one of the section renaming pragmas. To get noclear sections instead of BSS sections you must also use **#pragma noclear**.

## 3.11 SWITCH STATEMENT

**ctc** supports three ways of code generation for a switch statement: a jump chain (linear switch), a jump table or a lookup table.

A *jump chain* is comparable with an if/else–if/else–if/else construction. A *jump table* is a table filled with target addresses for each possible switch value. The switch argument is used as an index within this table. A *lookup table* is a table filled with a value to compare the switch argument with and a target address to jump to. A binary search lookup is performed to select the correct target address.

By default, the compiler will automatically choose the most efficient switch implementation based on code and data size and execution speed. You can influence the selection of the switch method with compiler option **–t** (**––tradeoff**), which determines the speed/size tradeoff.

It is obvious that, especially for large switch statements, the jump table approach executes faster than the lookup table 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. However, when the case labels are distributed far apart, the jump table becomes sparse, wasting code memory. The compiler will not use the jump table method when the waste becomes excessive.

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

### *How to overrule the default switch method*

You can overrule the compiler chosen switch method with a pragma:

```
#pragma switch linear    /* force jump chain code */
#pragma switch jumptab   /* force jump table code */
#pragma switch lookup    /* force lookup table code */
#pragma switch auto      /* let the compiler decide
                            the switch method used */
#pragma switch restore   /* restore previous switch
                            method */
```

Pragma `switch auto` is also the default of the compiler.

On the command line you can use compiler option **––switch**.

## 3.12 LIBRARIES

The compiler **ctc** comes with standard C libraries (ISO/IEC 9899:1999) and header files with the appropriate prototypes for the library functions. The standard C libraries are available in object format and in C or assembly source code.

A number of standard operations within C are too complex to generate inline code for. These operations are implemented as *run–time* library functions.

The lib directory contains subdirectories with separate libraries for the TriCore 1 and the TriCore 2. Furthermore, protected libraries are available for the TC112 and TC113 functional problems.

The protected library sets provide software bypasses for all TC112 and TC113 supported CPU functional problems. They must be used in conjunction with the appropriate C compiler workarounds for CPU functional problems. For more details refer to Chapter 8, *CPU Functional Problems* in the *Reference Guide*.

The directory structure is:

```
\ctc\lib\
         tc1\          TriCore 1 libraries
         tc2\          TriCore 2 libraries
         p\
              tc112    Protected libraries for TC112 problems
              tc113    Protected libraries for TC113 problems
```

### 3.12.1 OVERVIEW OF LIBRARIES

Table 3–9 lists the libraries included in the TriCore (**ctc**) toolchain.

| Library to link | Description |
|---|---|
| libc.a | C library<br>(With full printf/scanf functionality. Some functions require the floating point library. Also includes the startup code.) |
| libcs.a | C library single precision (compiler option **–F**)<br>(With full printf/scanf functionality. Some functions require the floating point library. Also includes the startup code.) |
| libcs_fpu.a | C library single precision with FPU instructions (compiler option **–F** and **––fpu–present**) |
| libfp.a | Floating point library (non–trapping) |
| libfpt.a | Floating point library (trapping)<br>(Control program option **–fptrap**) |
| libfp_fpu.a | Floating point library (non–trapping, with FPU instructions)<br>(Compiler option **––fpu–present**) |
| libfpt_fpu.a | Floating point library (trapping, with FPU instructions)<br>(Control program option **–fptrap**, compiler option **––fpu–present**) |
| librt.a | Run–time library |

*Table 3–9: Overview of libraries*

See section 2.1.2, *Library Functions*, in Chapter *Libraries* of the *Reference Guide* for an extensive description of all standard C library functions.

### 3.12.2 PRINTF AND SCANF FORMATTING ROUTINES

The C library functions `printf()`, `fprintf()`, `vfprintf()`, `vsprintf()`, ... call one single function, `_doprint()`, that deals with the format string and arguments. 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 need all the possibilities of the format specifiers, you can use a smaller `_doprint()`. Three different versions exist:

| | |
|---|---|
| LARGE | the full formatter, no restrictions (used in `libc.a`) |
| MEDIUM | floating point printing is not supported |
| SMALL | floating point printing and the precision specifier '.' are not supported (for example, `%10.10s`) |

The same applies to all `scanf` type functions, which call the function `_doscan()`.

If you want to use the MEDIUM or SMALL formatters you must rebuild the C library `libc.a` as described in section 3.12.3, *Rebuilding Libraries*.

### *Fixed point format specifiers*

The `printf` and `scanf` type functions support two additional format specifiers for the conversion of fixed–point types (fractional and accumulator types).

For `printf` type functions:

| | |
|---|---|
| %lR | An `__laccum` argument representing a fixed–point accumulator number is converted to decimal notation in the style [**–**]*ddd.ddd*, where the number of digits after the decimal–point character is equal to the precision specification. |
| %r | A `__fract` argument representing a fixed–point fractional number is converted to decimal notation in the style [**–**]*d.ddd*, where there is one digit (which is non–zero if the argument is –1.0) before the decimal point character and the number of digits after it is equal to the precision. |

For `scanf` type functions:

| | |
|---|---|
| %lR | Matches an optionally signed fixed–point accumulator number. The corresponding argument shall be a pointer to `__laccum`. |
| %r | Matches an optionally signed fixed–point fractional number. The corresponding argument shall be a pointer to `__fract`. |

Example:

```
#include <stdio.h>

__fract  fvalue = 1.0/3;
__laccum lacvalue = 1.234;

void main(void)
{
    printf("fvalue is: %r\n", fvalue);
    printf("lacvalue is: %lR\n", lacvalue);
}
```

### 3.12.3 REBUILDING LIBRARIES

If you have manually changed one of the standard C library functions, or you want to use the MEDIUM or SMALL printf and scanf routines, you need to recompile the standard C libraries.

The sources of the libraries are present in the `lib\src` directory. This directory also contains subdirectories with a `makefile` for each type of library:

```
lib\src\
        p\
            tc112\
                  libc\makefile
                  libcs\makefile
                  libcs_fpu\makefile
            tc113\
                  libc\makefile
                  libcs\makefile
                  libcs_fpu\makefile
        tc1\
            libc\makefile
            libcs\makefile
            libcs_fpu\makefile
        tc2\
            libc\makefile
            libcs\makefile
            libcs_fpu\makefile
```

To rebuild the libraries, follow the steps below. As an example the instructions are given to rebuild a C library with MEDIUM sized printf routines for the TriCore 2.

First make sure that the `bin` directory for the TriCore toolchain is included in your PATH environment variable. (See section 1.3.2, *Configuring the Command Line Environment*.

1. Make the directory `lib\src\tc2\libc` the current working directory.

   *This directory contains a* `makefile` *which also uses the default make rules from* `mktc.mk` *from the* `ctc\etc` *directory.*

2. Edit the `makefile` as follows to set the macro MEDIUM:

```
CC = $(PRODDIR)\bin\ctc -DMEDIUM
```

See section 8.3, *Make Utility*, in Chapter *Utilities* for an extensive description of the make utility and makefiles.

3. Assuming the `lib\src\tc2\libc` directory is still the current working directory, type:

   **`mktc`**

   to build the library.

   *The new library is created in the* `lib\src\tc2\libc` *directory. All routines with conditional code are compiled for the MEDIUM printf/scanf formatters.*

4. Make a backup copy of the original library and copy the new library to the `lib\tc2` directory of the product.

CHAPTER 4

# TRICORE ASSEMBLY LANGUAGE

TASKING

# 4 CHAPTER

## 4.1 INTRODUCTION

In this chapter the most important aspects of the TriCore assembly language are described. For a complete overview of the TriCore2 architecture, refer to the *TriCore2 Architecture Overview Handbook* [2002, Infineon].

## 4.2 ASSEMBLY SYNTAX

An assembly program consists of zero or more statements. A statement may optionally be followed by a comment. Any source statement can be extended to more lines by including the line continuation character (\) as the last character on the line. The length of a source statement (first line and continuation lines) is only limited by the amount of available memory.

Mnemonics and directives are case insensitive. Labels, symbols, directive arguments, and literal strings are case sensitive.

The syntax of an assembly *statement* is:

[*label*[**:**]] [*instruction* | *directive* | *macro_call*] [**;***comment*]

*label* A label can consist of letters, digits and underscore characters (**_**). The first character cannot be a digit. A label which is prefixed by whitespace (spaces or tabs) has to be followed by a colon (:). The size of an identifier is only limited by the amount of available memory.

Examples:

```
      LAB1:  ; This label is followed by a colon and
               can start with a space or tab
  LAB1       ; This label has to start at the beginning
               of a line
```

*instruction* An instruction consists of a mnemonic and zero, one, two, three, four or five operands. Operands are described in section 4.4, *Operands*. The instructions are described in the *TriCore Architecture Manuals*.

Examples:

```
ret                         ; No operand
call    label               ; One operand
mov     D0,#1               ; Two operands
jne     D0,#0,loop          ; Three operands
madd    D2,D3,D0,D1         ; Four operands
insert  D1,D2,#3,#16,#2     ; Five operands
```

*directive* With directives you can control the assembler from within the assembly source. Directives are described in section 4.8, *Directives*.

*macro_call* A call to a previously defined macro. Macros are described in section 4.9 *Macro Operations*.

You can use empty lines or lines with only comments.

## 4.3 ASSEMBLER SIGNIFICANT CHARACTERS

Some characters have a special meaning to the assembler. Special characters associated with expression evaluation are described in section 4.6.3, *Expression Operators*. Other special assembler characters are:

**;** – Comment delimiter

\ – Line continuation character or
Macro dummy argument concatenation operator

**?** – Macro value substitution operator

**%** – Macro hex value substitution operator

**^** – Macro local label override operator

**"** – Macro string delimiter or
Quoted string **.DEFINE** expansion character

@ – Function delimiter

***** – Location counter substitution

# – Constant number

++ – String concatenation operator

[] – Substring delimiter

## 4.4 OPERANDS

In an instruction, the mnemonic is followed by zero, one, two, three, four or five operands. An operand has one of the following types:

| Operands | Description |
|---|---|
| *label* | A label reference as described in section 4.2, *Assembly Syntax*. |
| *register* | Any valid register or a register pair, register quad, register extension, register part or special function register. |
| *symbol* | A symbolic name as described in section 4.5, *Symbol Names*. Symbols can also occur in expressions. |
| *expression* | Any valid expression as described in the section 4.6, *Expressions*. |
| *address* | A combination of *expression*, *register* and *symbol*. |

### 4.4.1 OPERANDS AND ADDRESSING MODES

The TriCore assembly language has several addressing modes. These are listed below with a short description. For details see the *TriCore 1 Unified Processor Core v1.3 Architecture Manual, Doc v1.3.3* [2002–09, Infineon]

***Absolute addressing***

The instruction uses an 18–bit constant as the memory address. The full 32–bit address results from moving the most significant 4 bits of the 18–bit constant to the most significant bits of the 32–bit address. The other bits are zero filled.

Syntax:

*constant18*

### *Base+offset*

The effective address is the sum of an address register and the sign–extended 10–bit or 16–bit offset.

Syntax:

[A*n*]*offset10*
[A*n*]*offset16*

### *Pre–increment/decrement*

This addressing mode uses the sum of the address register and the offset both as the effective address and as the value written back into the address register. Use the minus sign for a pre–decrement.

Syntax:

[+A*n*]*offset10*

### *Post–increment/decrement*

This addressing mode uses the value of the address register as the effective address, and then updates this register by adding the sign–extended 10–bit offset to its previous value. Use the minus sign for a post–decrement.

Syntax:

[A*n*+]*offset10*

### *Circular addressing*

This addressing mode is used for accessing data values in circular buffers. It uses an address register pair to hold the state it requires. The even register is always a base address (B). The most–significant half of the odd register is the buffer size (L). The least significant half holds the index into the buffer (I). The effective address is (B+I). The buffer occupies memory from addresses B to B+L–1. The 10–bit offset is specified in the instruction word and is a byte–offset that can be either positive or negative.

Syntax:

[A*n*+c]*offset10*

### *Bit–reverse addressing*

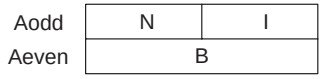
Bit reverse addressing is used to access arrays used in FFT algorithms. Bit–reverse addressing uses an address register pair to hold the required state. The even register is the base address of the array (B), the least–significant half of the odd register is the index into the array (I), and the most–significant half is the modifier (M) which is added to I after every access. The effective address is B+reverse(I). The reverse() function exchanges bit n with bit (15–n) for n = 0, ..., 7. The index, I, is post–incremented and its new value is (I + M), where M is the most significant half of the odd register.

Syntax:

[A*n*+r]

### *Indexed addressing*

The indexed addressing mode uses an address register pair to hold the required state. The $A_{even}$ register is the base address of the array (B). The $A_{odd}$ register is divided equally between the index into the array (I), and the modifier (N) which is added to I after every access.




All load (LD.xxx) instructions, all store(ST.xxx except ST.T) instructions, the load/modify/store (SWAP.W,LDMST) instructions and the cache management (CACHEA.xxx) instructions are able to use the indexed addressing mode.

Syntax:

[A*a*/A*b*+i]

```
ld.w d0,[a0/a1+i] ; load word indexed addressing mode
st.w [a2/a3+i],d0 ; store word indexed addressing mode
```

## 4.4.2 PCP ADDRESSING MODES

The PCP assembly language has several addressing modes. These addressing modes are used for FPI addressing, PRAM data indirect addressing or flow control destination addressing. For details see the PCP/DMA Architecture manual from Siemens.

## 4.5 SYMBOL NAMES

A symbol can consist of letters, digits and underscore characters (_). The first character cannot be a digit. The size of an identifier is limited to 4000 characters.

Symbol names and other identifiers beginning with a period (.) are reserved for the system. Names of assembler directives are also reserved.

The following symbols are predefined:

| Symbol | Description |
|---|---|
| __ASTC__ | Contains the name of the assembler ("astc") |
| __ASPCP__ | Contains the name of the PCP assembler ("aspcp") |
| __FPU__ | Defined when you use assembler option **--fpu–present** (allow use of FPU instructions) |
| __MMU__ | Defined when you use assembler option **--mmu–present** (allow use of MMU instructions) |
| __TC2__ | Defined when you use assembler option **--is–tricore2** (allow use of TriCore 2 instructions) |

*Table 4–1: Predefined symbols*

***Examples:***

| Valid symbol names | Invalid symbol names | |
|---|---|---|
| `loop_1` | `1_loop` | (starts with a number) |
| `ENTRY` | `loop.e` | (reserved name) |
| `a_B_c` | `.loop` | (reserved name) |
| `_aBC` | | |

## 4.6 EXPRESSIONS

An expression is a combination of symbols, constants, operators, and parentheses which represent a value that is used as an operand of an assembler instruction (or directive).

Expressions can contain user–defined labels (and their associated integer or floating–point values), and any combination of integers, floating–point numbers, or ASCII literal strings.

Expressions follow the conventional rules of algebra and boolean arithmetic.

Expressions that can be evaluated at assembly time are called *absolute expressions*. Expressions where the result is unknown until all sections have been combined and located, are called *relocatable* or *relative expressions*.

When any operand of an expression is relocatable, the entire expression is relocatable. Relocatable expressions are emitted in the object file and are evaluated by the linker. Relocatable expressions can only contain integral functions; floating point functions and numbers are not supported by the ELF/DWARF object format. An error is emitted when during object creation non–ELF/DWARF relocatable expressions are found.

The assembler evaluates expressions with 64–bit precision in two's complement.

The syntax of an *expression* can be any of the following:

- *numeric contant*
- *string*
- *symbol*
- *expression binary_operator expression*
- *unary_operator expression*
- **(** *expression* **)**
- *function call*

All types of expressions are explained below and in the following sections.

**( )** You can use parentheses to control the evaluation order of the operators. What is between parentheses is evaluated first.

### 4.6.1 NUMERIC CONSTANTS

Numeric constants can be used in expressions. If there is no prefix, the assembler assumes the number is a decimal number.

| Base | Description | Example |
|---|---|---|
| Binary | '**0B**' or '**0b**' followed by binary digits (0,1). | `0B1101`<br>`0b11001010` |
| Hexadecimal | '**0X**' or '**0x**' followed by a hexadecimal digits (0–9, A–F, a–f). | `0X12FF`<br>`0x45`<br>`0x9abc` |
| Decimal, integer | Decimal digits (0–9). | `12`<br>`1245` |
| Decimal, floating point | Includes a decimal point, or an '**E**' or '**e**' followed by the exponent. | `6E10`<br>`.6`<br>`2.7e10` |

### 4.6.2 STRINGS

ASCII characters, enclosed in single (') or double (") quotes constitue a literal ASCII string. Both type of strings can contain escape characters.

Strings constants in expressions are evaluated to a number. Strings in expressions can have the size of a long word (first 4 characters); any subsequent characters in the string are ignored. In this case the assembler issues a warning. An exception to this rule is when a string longer than 4 characters is used in a .BYTE assembler directive; in that case all characters result in a constant byte. Null strings have a value of 0.

Square brackets (**[ ]**) delimit a substring operation in the form:

**[***string*,*offset*,*length***]**

*offset* is the start position within *string*. *length* is the length of the desired substring. Both values may not exceed the size of *string*.

***Examples***

```
'ABCD'              ; ($41424344)

'''79'              ; to enclose a quote double it

"A\"BC"             ; or to enclose a quote escape it

'A'+1               ; ($00000042)

''                  ; null string

'abcdef'            ; ($61626364)

'abc'++'de'         ; you can concatenate
                    ; two strings with the '++' operator.
                    ; This results in 'abcde'

['TriCore',0,3]  ; results in the substring 'Tri'
```

### 4.6.3 EXPRESSION OPERATORS

The next table shows the assembler operators. They are ordered according to their precedence. Operators of the same precedence are evaluated left to right. Expressions between parentheses have the highest priority (innermost first).

Valid operands include numeric constants, literal ASCII strings and symbols.

Most assembler operators can be used with both integer and floating–point values. If one operand has an integer value and the other operand has a floating–point value, the integer is converted to a floating–point value before the operator is applied. The result is a floating–point value.

| Type | Operator | Name | Description |
|---|---|---|---|
| | ( ) | parentheses | Expressions enclosed by parenthesis are evaluated first. |
| Unary | + | plus | Returns the value of its operand. |
| | – | minus | Returns the negative of its operand. |
| | ~ | complement | Returns complement, integer only |
| | ! | logical negate | Returns 1 if the operands' value is 1; otherwise 0. For example, if `buf` is 0 then `!buf` is 1. |
| Arithmetic | * | multiplication | Yields the product of two operands. |
| | / | division | Yields the quotient of the division of the first operand by the second.<br>With integers, the divide operation produces a truncated integer. |
| | % | mod | Used with integers: yields the remainder from a division of the first operand by the second.<br>Used with floats:<br>Y%Z = Y if Z= 0<br>Y%Z = Y–(*integer**Z) |
| | + | addtion | Yields the sum of its operands. |
| | – | subtraction | Yields the difference of its operands. |
| Shift | << | shift left | Integer only: shifts the left operand to the left (zero–filled) by the number of bits specified by the right operand. |
| | >> | shift right | Integer only: shifts the left operand to the right (sign bit extended) by the number of bits specified by the right operand. |
| Relational | < | less than | If the indicated condition is:<br>– True: result is an integer 1<br>– False: result is an integer 0<br>Be cautious when you use floating point values in an equality test; rounding errors can cause unexpected results. |
| | <= | less or equal | |
| | > | more than | |
| | >= | mor or equal | |
| | == | equal | |
| | != | not equal | |

| Type | Operator | Name | Description |
|---|---|---|---|
| Bitwise | & | AND | Integer only: yields bitwise AND |
| | \| | OR | Integer only: yields bitwise OR |
| | ^ | exclusive OR | Integer only: yields bitwise exlusive OR |
| Logical | && | logical AND | Returns an integer 1 if both operands are nonzero; otherwise, it returns an integer 0. |
| | \|\| | logical OR | Returns an integer 1 if either of the operands is nonzero; otherwise, it returns an integer 1 |

*Table 4–2: Assembly expression operators*

## 4.7 BUILT-IN ASSEMBLY FUNCTIONS

The assembler has several built–in functions to support data conversion, string comparison, and math computations. You can use functions as terms in any expression. Functions have the following syntax:

*@function_name***(**[*argument*[**,***argument*]...]**)**

Functions start with the '@' character and have zero or more arguments, and are always followed by opening and closing parentheses. White space (a blank or tab) is not allowed between the function name and the opening parenthesis and between the (comma–separated) arguments.

The built–in assembler functions are grouped into the following types:

- **Mathematical functions** comprise, among others, transcendental, random value, and min/max functions.
- **Conversion functions** provide conversion between integer, floating point, and fixed point fractional values.
- **String functions** compare strings, return the length of a string, and return the position of a substring within a string.
- **Macro functions** return information about macros.
- **Address calculation functions** return the high or low part of an address.
- **Assembler mode functions** relating assembler operation.

The following tables provide an overview of all built–in assembler functions. For a detailed description of these functions, see section 3.2, *Built–in Assembly Function*, in Chapter *Assembly Language* of the *Reference Guide*.

### *Overview of mathematical functions*

| Function | Description |
|---|---|
| @ABS(*expr*) | Absolute value |
| @ACS(*expr*) | Arc cosine |
| @ASN(*expr*) | Arc sine |
| @AT2(*expr1*,*expr2*) | Arc tangent |
| @ATN(*expr*) | Arc tangent |
| @CEL(*expr*) | Ceiling function |
| @COH(*expr*) | Hyperbolic cosine |
| @COS(*expr*) | Cosine |
| @FLR(*expr*) | Floor function |
| @L10(*expr*) | Log base 10 |
| @LOG(*expr*) | Natural logarithm |
| @MAX(*expr*,[,...,*exprN*]) | Maximum value |
| @MIN(*expr*,[,...,*exprN*]) | Minimum value |
| @POW(*expr1*,*expr2*) | Raise to a power |
| @RND() | Random value |
| @SGN(*expr*) | Returns the sign of an expression as –1, 0 or 1 |
| @SIN(*expr*) | Sine |
| @SNH(*expr*) | Hyperbolic sine |
| @SQT(*expr*) | Square root |
| @TAN(*expr*) | Tangent |
| @TNH(*expr*) | Hyperbolic tangent |
| @XPN(*expr*) | Exponential function (raise e to a power) |

### *Overview of conversion functions*

| Function | Description |
|---|---|
| @CVF(*expr*) | Convert integer to floating–point |
| @CVI(*expr*) | Convert floating–point to integer |
| @FLD(*base*,*value*, *width*[,*start*]) | Shift and mask operation |
| @FRACT(*expr*) | Convert floating–point to 32–bit fractional |
| @SFRACT(*expr*) | Convert floating–point to 16–bit fractional |
| @LNG(*expr*) | Concatenate to double word |
| @LUN(*expr*) | Convert long fractional to floating–point |
| @RVB(*expr1*[,*expr2*]) | Reverse order of bits in field |
| @UNF(*expr*) | Convert fractional to floating–point |
| @LSB(*expr*) | Get least significant byte of a word |
| @MSB(*expr*) | Get most significant byte of a word |

### *Overview of string functions*

| Function | Description |
|---|---|
| @CAT(*str1*,*str2*) | Concatenate strings |
| @LEN(*string*) | Length of string |
| @POS(*str1*,*str2*[,*strt*]) | Position of substring in string |
| @SCP(*str1*,*str2*) | Returns 1 if two strings are equal |
| @SUB(*str1*,*expr*,*expr*) | Returns substring in string |

### *Overview of macro functions*

| Function | Description |
|---|---|
| @ARG({*symbol*\|*expr*}) | Test if macro argument is present |
| @CNT() | Return number of macro arguments |
| @MAC(*symbol*) | Test if macro is defined |
| @MXP() | Test if macro expansion is active |

***Overview of address calculation functions***

| Function | Description |
|---|---|
| @HI(*expr*) | Returns upper 16 bits of expression value |
| @HIS(*expr*) | Returns upper 16 bits of expression value, adjusted for signed addition |
| @LO(*expr*) | Returns lower 16 bits of expression value |
| @LOS(*expr*) | Returns lower 16 bits of expression value, adjusted for signed addition |

***Overview of assembler mode functions***

| Function | Description |
|---|---|
| @ASPCP() | Returns the name of the pcp assembler executable |
| @ASTC() | Returns the name of the assembler executable |
| @CPU(*string*) | Test if CPU type is selected |
| @DEF(*symbol*) | Returns 1 if symbol has been defined |
| @EXP(*expr*) | Expression check |
| @INT(*expr*) | Integer check |
| @LST() | LIST control flag value |

## 4.8 DIRECTIVES AND CONTROLS

An assembler directive is simply a message to the assembler. Assembler directives are not translated into machine code because they are not instructions. There are three types of assembler directives.

- Assembler directives that tell the assembler how to go about translating instructions into machine code. This is the most typical form of assembly directives. Typically they tell the assembler where to put a program in memory, what space to allocate to variables, and allow you to preset memory with data. When code is assembled, a location counter in the assembler keeps track of where the code and data is to go in memory.

- Directives that are processed by the macro processor. These directives in fact tell the macro processor how to manipulate your assembly code before it is actually being assembled. You can use these directives to write macros and to write conditional source code. Parts of the code that do not match the condition, will not be assembled at all.
- Some directives act as assembler options and most of them indeed do have an equivalent assembler (command line) option. The advantage of using a directive is that with such a directive you can overrule the assembler option for a particular part of the code. Directives of this kind are called *controls*. A typical example is to tell the assembler with an option to generate a list file while with the controls $LIST ON and $LIST OFF you overrule this option for a part of the code that you do *not* want to appear in the list file. Controls always start with a '$' sign.

Assembler directives are grouped in the following categories:

- Assembly control directives
- Symbol definition directives
- Data definition / Storage allocation directives
- Macro and conditional assembly directives
- Debug directives

Assembler controls are grouped in the following categories:

- Assembler listing controls
- Miscellaneous controls

Each assembler directive or control has its own syntax. You can use assembler directives and controls in the assembly code as pseudo instructions.

### 4.8.1 OVERVIEW OF ASSEMBLER DIRECTIVES

The following tables provide an overview of all assembler directives. For a detailed description, see section 3.3.2, *Detailed Description of Assembler Directives*, in Chapter *Assembly Language* of the *Reference Guide*.

### *Overview of assembly control directives*

| Directive | Description |
|---|---|
| .COMMENT | Start comment lines. You cannot use this directive in .IF/.ELSE/.ENDIF constructs and .MACRO/.DUP definitions. |
| .DEFINE | Define substitution string |
| .END | End of source program |
| .FAIL | Programmer generated error message |
| .INCLUDE | Include secondary file |
| .MESSAGE | Programmer generated message |
| .NAME | Identification for object file (instead of file name) |
| .ORG | Initialize memory space and location counters to create a nameless section |
| .SDECL | Declare a section with name, type and attributes |
| .SECT | Activate a declared section |
| .UNDEF | Undefine .DEFINE symbol |
| .WARNING | Programmer generated warning |

### *Overview of symbol definition directives*

| Function | Description |
|---|---|
| .EQU | Assigns permanent value to a symbol |
| .EXTERN | External symbol declaration |
| .GLOBAL | Global symbol declaration |
| .LOCAL | Local symbol declaration |
| .SET | Set temporary value to a symbol |
| .SIZE | Set size of symbol in the ELF symbol table |
| .TYPE | Set symbol type in the ELF symbol table |

### *Overview of data definition / storage allocation directives*

| Function | Description |
|---|---|
| .ALIGN | Define alignment |
| .ACCUM | Define 64–bit constant in 18 + 46 bits format |
| .ASCII / .ASCIIZ | Define ASCII string without / with ending NULL byte |
| .BYTE | Define constant byte |
| .FLOAT / .DOUBLE | Define a 32–bit / 64–bit floating point constant |
| .FRACT / .SFRACT | Define a 16–bit / 32–bit constant fraction |
| .SPACE | Define storage |
| .WORD / .HALF | Define a word / half–word constant |

### *Overview of macro and conditional assembly directives*

| Function | Description |
|---|---|
| .DUP | Duplicate sequence of source lines |
| .DUPA | Duplicate sequence with arguments |
| .DUPC | Duplicate sequence with characters |
| .DUPF | Duplicate sequence in loop |
| .ENDM | End of macro or duplicate sequence |
| .EXITM | Exit macro |
| .IF/.ELIF/.ELSE/.ENDIF | Conditional assembly |
| .MACRO | Define macro |
| .PMACRO | Purge macro definition |

### *Overview of debug directives*

| Function | Description |
|---|---|
| .CALLS | Passes call information to object file. Used by the linker to build a call graph. |
| .SYMB | Passes debug information to object file. Used by CrossView Pro for debugging. |

### 4.8.2 OVERVIEW OF ASSEMBLER CONTROLS

The following tables provide an overview of all assembler controls. For a detailed description, see section 3.3.4, *Detailed Description of Assembler Controls*, in Chapter *Assembly Language* of the *Reference Guide*.

***Overview of assembler listing controls***

| Function | Description |
|---|---|
| $LIST ON/OFF | Generation of assembly list file temporary ON/OFF |
| $LIST "*flags*" | Exclude / include lines in assembly list file |
| $PAGE | Generate formfeed in assembly list file |
| $PAGE *settings* | Define page layout for assemly list file |
| $PRINT | Specify alternative name for assembly list file |
| $NOPRINT | Disable list file generation |
| $PRCTL | Send control string to printer |
| $STITLE | Set program subtitle in header of assembly list file |
| $TITLE | Set program title in headerof assembly list file |

***Overview of miscellaneous assembler controls***

| Function | Description |
|---|---|
| $CASE ON/OFF | Case sensitive user names ON/OFF |
| $DEBUG ON/OFF | Generation of symbolic debug ON/OFF |
| $DEBUG "*flags*" | Generation of symbolic debug ON/OFF |
| $FPU | Allow single precision floating point instructions |
| $HW_ONLY | Prevent substitution of assembly instructions by smaller or faster instructions |
| $IDENT LOCAL/GLOBAL | Assembler treats labels by default as local or global |
| $MMU | Allow memory management instructions |
| $OBJECT | Alternative name for the generated object file |
| $TC*defect* ON/OFF | Enable/disable assembler check for specified functional problem |
| $TC2 | Allow TriCore 2 instructions |
| $WARNING OFF [*num*] | Suppress one or all warnings |

## 4.9 MACRO OPERATIONS

Macros provide a shorthand method for handling a repeated pattern of code or group of instructions. Yuo can define the pattern as a macro, and then call the macro at the points in the program where the pattern would repeat.

Some patterns contain variable entries which change for each repetition of the pattern. Others are subject to conditional assembly for a given occurrence of the instruction group. In either case, macros provide a shorthand notation for handling these instruction patterns.

When a macro is called, the assembler generates in–line source statements. The generated statements may contain substitutable arguments. The statements produced by a macro can be any processor instruction, almost any assembler directive, or any previously–defined macro. Source statements resulting from a macro call are subject to the same conditions and restrictions as any other statements.

The assembler is able to process *nested* macro calls at expansion time only. However, the nested macro definition is not processed until the primary macro is expanded.

### 4.9.1 DEFINING A MACRO

The first step in using a macro is to define it in the source file. The definition of a macro consists of three parts:

- *Header*, which assigns a name to the macro and defines the dummy arguments.
- *Body*, which contains the code or instructions to be inserted when te macro is called.
- *Terminator*, which indicates the end of the macro definition (ENDM directive).

A macro definition takes the following form:

```
Header:      macro_name .MACRO [dumarg[,dumarg...] [; comment]
                        .
Body:                   source statements
                        .
Terminator:             .ENDM
```

If the macro name is the same as an existing assembler directive or mnemonic opcode, the assembler replaces the directive or mnemonic opcode with the macro and issues a warning. This replacement does not occur for definitions from macro libraries.

The dummy arguments are symbolic names that the macro processor replaces with the literal arguments when the macro is expanded (called). Each dummy argument must follow the same rules as global symbol names. Dummy argument names cannot start with an underscore (_).

***Example***

The macro definition:

```
CONSTD  .MACRO  reg,value                    ;header
    mov.u   reg,#lo(value)                   ;body
    addih   reg,reg,#hi(value)
    .ENDM                                    ;terminator
```

The macro call:

```
    .SDECL  "data",DATA
    .SECT   "data"

    CONSTD  d4,0x12345678

    .END
```

The macro expands as follows:

```
    mov.u   d4,#lo(0x12345678)
    addih   d4,d4,#hi(0x12345678)
```

## 4.9.2 CALLING A MACRO

To invoke a macro, construct a source statement with the following format:

```
[label] macro_name [arg[,arg...]]          [; comment]
```

where:

*label* — An optional label that corresponds to the value of the location counter at the start of the macro expansion.

*macro_name* — The name of the macro. This must be in the operation field.

*arg* One or more optional, substitutable arguments. Multiple arguments must be separated by commas.

*comment* An optional comment.

The following applies to macro arguments:

- Each argument must correspond one–to–one with the dummy arguments of the macro definition. If the macro call does not contain the same number of arguments as the macro definition, the assembler issues a warning.
- If an argument has an embedded comma or space, you must surround the argument by single quotes (').
- You can declare a macro call argument as NULL in four ways:
  - enter delimiting commas in succession with no intervening spaces

    ```
    macroname ARG1,,ARG3 ; the second argument
                           is a NULL argument
    ```

  - terminate the argument list with a comma, the arguments that normally would follow, are now considered NULL

    ```
    macroname ARG1,      ; the second and all following
                           arguments are NULL
    ```

  - declare the argument as a NULL string
- No character is substituted in the generated statements that reference a NULL argument.

### 4.9.3 USING OPERATORS FOR DUMMY ARGUMENTS

The assembler recognizes certain text operators within macro definitions which allow text substitution of arguments during macro expansion. You can use these operators for text concatenation, numeric conversion, and string handling.

| Operator | Name | Description |
|---|---|---|
| \ | Macro argument concatenation | Concatenates a macro dummy argument with adjacent alphanumeric characters. |
| ? | Return decimal value of symbol | Substitutes the ?*symbol* sequence with a character string that represents the decimal value of the symbol. |
| % | Return hex value of symbol | Substitutes the ?*symbol* sequence with a character string that represents the hexadecimal value of the symbol. |
| " | Macro string delimiter | Allows the use of macro arguments as literal strings. |
| ^ | Macro local label override | Causes local labels in its term to be evaluated at normal scope rather than at macro scope. |

***Example: Argument Concatenation Operator – ***

Consider the following macro definition:

```
SWAP_MEM .MACRO REG1,REG2          ;swap memory contents
     LD.W  D0,[A\REG1]             ;use D0 as temp
     LD.W  D1,[A\REG2]             ;use D1 as temp
     ST.W  [A\REG1],D1
     ST.W  [A\REG2],D0
     .ENDM
```

The macro is called as follows:

```
     SWAP_MEM  0,1
```

The macro expands as follows:

```
     LD.W  D0,[A0]
     LD.W  D1,[A1]
     ST.W  [A0],D1
     ST.W  [A1],D0
```

The macro processor would substitutes the character '0' for the dummy argument REG1, and the character '1' for the dummy argument REG2. The concatenation operator (\) indicates to the macro processor that the substitution characters for the dummy arguments are to be concatenated with the character 'A'.

### *Example: Decimal value Operator – ?*

Instead of substituting the dummy arguments with the macro call arguments, you can also use the *value* of the macro call arguments.

Consider the following source code that calls the macro SWAP_SYM after the argument AREG has been set to 0 and BREG has been set to 1.

```
AREG .SET      0
BREG .SET      1
     SWAP_SYM  AREG,BREG
```

If you want to replace the dummy arguments with the *value* of AREG and BREG rather than with the literal strings 'AREG' and 'BREG', you can use the **?** operator and modify the macro as follows:

```
SWAP_SYM   .MACRO  REG1,REG2        ;swap memory contents
     LD.W  D0,_lab\?REG1            ;use D0 as temp
     LD.W  D1,_lab\?REG2            ;use D1 as temp
     ST.W  _lab\?REG1,D1
     ST.W  _lab\?REG2,D0
     .ENDM
```

The macro first expands as follows:

```
     LD.W  D0,_lab\?AREG
     LD.W  D1,_lab\?BREG
     ST.W  _lab\?AREG,D1
     ST.W  _lab\?BREG,D0
```

Then **?**AREG is replaced by '0' and **?**BREG is replaced by '1':

```
     LD.W  D0,_lab\1
     LD.W  D1,_lab\2
     ST.W  _lab\1,D1
     ST.W  _lab\2,D0
```

Finally, the strings are concatenated because of the '\' operator '\':

```
LD.W  D0,_lab1
LD.W  D1,_lab2
ST.W  _lab1,D1
ST.W  _lab2,D0
```

### *Example Hex Value Operator – %*

The percent sign (**%**) is similar to the standard decimal value operator except that it returns the hexadecimal value of a symbol.

Consider the following macro definition:

```
GEN_LAB    .MACRO  LAB,VAL,STMT
LAB\%VAL   STMT
     .ENDM
```

A symbol with the name NUM is set to 10 and the macro is called with NUM as argument:

```
NUM .SET       10
    GEN_LAB    HEX,NUM,NOP
```

The macro expands as follows:

```
HEXA NOP
```

The %VAL dummy argument is replaced by the character 'A' which represents the hexadecimal value 10 of the dummy argument VAL.

### *Example: Dummy Argument String Operator – "*

To generate a literal string, enclosed by single quotes ('), you must use the argument string operator (") in the macro definition.

Consider the following macro definition:

```
STR_MAC    .MACRO  STRING
    .BYTE  "STRING"
    .ENDM
```

The macro is called as follows:

```
  STR_MAC  ABCD
```

The macro expands as follows:

```
    .BYTE  'ABCD'
```

Within double quotes .DEFINE directive definitions can be expanded. Take care when using constructions with quotes and double quotes to avoid inappropriate expansions. Since a .DEFINE expansion occurs before a macro substitution, all .DEFINE symbols are replaced first within a macro dummy argument string:

```
    DEFINE LONG  'short'
STR_MAC    .MACRO  STRING
    .MESSAGE 'This is a LONG STRING'
    .MESSAGE "This is a LONG STRING"
    ENDM
```

If the macro is called as follows:

```
    STR_MAC  sentence
```

The macro expands as:

```
   .MESSAGE 'This is a LONG STRING'
   .MESSAGE 'This is a short sentence'
```

Single quotes prevent expansion so the first .MESSAGE is not stated as is. In the double quoted .MESSAGE, first the define LONG is expanded to 'short' and then the argument STRING is substituted by 'sentence'.

### *Macro Local Label Override Operator – ^*

The macro ^–operator prevents name mangling on the ADDR argument if the argument has a leading underscore. If there is no leading underscore on the actual argument the the ^–operator has no effect.

Consider the following macro definition:

```
LOAD   .MACRO  ADDR
       LD.W   D0,^ADDR
       .ENDM
```

The macro is called as follows:

```
_LOCAL  LOAD  _LOCAL
```

The macro expands as:

```
_LOCAL
       LD.W  D0,_LOCAL
```

Without the ^ operator, the macro processor would choose another name for _LOCAL because the label already exists. The macro then would expand like:

```
_LOCAL
       LD.W  D0,_LOCAL_M_0
```

### 4.9.4 USING THE .DUP, .DUPA, .DUPC, .DUPF DIRECTIVES AS MACROS

The .DUP, .DUPA, .DUPC, and .DUPF directives are specialized macro forms. You can think of them as a simultaneous definition and call of an unnamed macro. The source statements between the .DUP, .DUPA, .DUPC, and .DUPF directives and the .ENDM directive follow the same rules as macro definitions.

For a detailed description of these directives, see section 3.3, *Assembler Directives*, in Chapter *Assembly Language* of the *Reference Guide*.

### 4.9.5 CONDITIONAL ASSEMBLY: .IF, .ELIF AND .ELSE DIRECTIVES

With the conditional assembly directives you can create a part of conditional assembly code. The assembler assembles only the code that it reaches.

You can specify assembly conditions with arguments in the case of macros, or through definition of symbols via the `.DEFINE`, `.SET`, and `.EQU` directives.

The built–in functions of the assembler provide a versatile means of testing many conditions of the assembly environment

You can use conditional directives also within a macro definition to check at expansion time if arguments fall within a certain range values. In this way macros become self–checking and can generate error messages to any desired level of detail.

The conditional assembly directive **.IF** has the following form:

```
.IF  expression
 .
 .
[.ELSE]     ;(the .ELSE directive is optional)
 .
 .
[.ELIF]     ;(the .ELIF directive is optional)
 .
 .
.ENDIF
```

The *expression* must evaluate to an absolute integer and cannot contain forward references. If *expression* evaluates to zero, the **.IF**–condition is considered FALSE. Any non–zero result of *expression* is considered as TRUE.

For a detailed description of these directives, see section 3.3, *Assembler Directives*, in Chapter *Assembly Language* of the *Reference Guide*.

CHAPTER 5

# USING THE COMPILER

TASKING

# 5 CHAPTER

## 5.1 INTRODUCTION

EDE uses a *makefile* to build your entire project, from C source till the final ELF/DWARF object file which serves as input for the debugger.

Although in EDE you cannot run the compiler separately from the other tools, this chapter discusses the options that you can specify for the compiler.

On the command line it is possible to call the compiler separately from the other tools. However, it is recommended to use the control program **cctc** for command line invocations of the toolchain (see section 8.2, *Control Program*, in Chapter *Using the Utilities*). With the control program it is possible to call the entire toolchain with only one command line.

The compiler takes the following files for input and output:

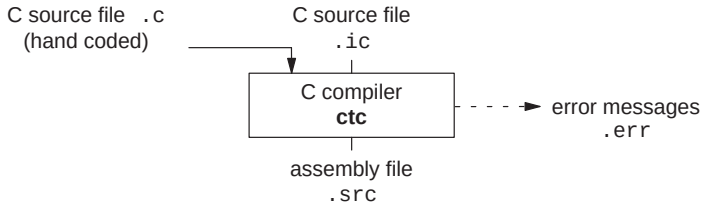



*Figure 5–1: C compiler*

This chapter first describes the compilation process which consists of a *frontend* and a *backend* part. During compilation the code is optimized in several ways. The various optimizations are described in the second section. Third it is described how to call the compiler and how to use its options. An extensive list of all options and their descriptions is included in the section 4.1, *Compiler Options*, in Chapter 4, *Tool Options*, of the *Reference Guide*. Finally, a few important basic tasks are described.

## 5.2 COMPILATION PROCESS

During the compilation of a C program, the compiler **ctc** runs through a number of phases that are divided into two groups: *frontend* and *backend*.

The backend part is not called for each C statement, but starts after a complete C module or set of modules has been processed by the frontend (in memory). This allows better optimization.

The compiler requires only one pass over the input file which results in relative fast compilation.

***Frontend phases***

1. The preprocessor phase:

   The preprocessor includes files and substitutes macros by C source. It uses only string manipulations on the C source. The syntax for the preprocessor is independent of the C syntax but is also described in the ISO/IEC 9899:1999(E) standard.

2. The scanner phase:

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

3. 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. This code is called MIL (Medium level Intermediate Language).

4. The frontend optimization phase:

   Target processor *independent* optimizations are performed by transforming the intermediate code.

***Backend phases***

5. Instruction selector phase:

   This phase reads the MIL input and translates it into Low level Intermediate Language (LIL). The LIL objects correspond to a TriCore processor instruction, with an opcode, operands and information used within the compiler.

6. Peephole optimizer/instruction scheduler/software pipelining phase:

   This phase replaces instruction sequences by equivalent but faster and/or shorter sequences, rearranges instructions and deletes unnecessary instructions.

7. Register allocator phase:

   This phase chooses a physical register to use for each virtual register.

8. The backend optimization phase:

   Performs target processor *independent* and *dependent* optimizations which operate on the Low level Intermediate Language.

9. The code generation/formatter phase:

   This phase reads through the LIL operations to generate assembly language output.

## 5.3 COMPILER OPTIMIZATIONS

The compiler has a number of optimizations which you can enable or disable. To enable or disable optimizations:

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

   *The Project Options dialog box appears.*

2. Expand the **C Compiler** entry and select **Optimization**.

3. Select an optimization level in the **Optimization level** box.

   or:

   In the **Optimization level** box, select **Custom optimization** and enable the optimizations you want in the **Custom optimization** box.

If you specify a certain optimization, all code in the module is subject to that optimization. Within the C source file you can overrule the compiler options for optimizations with `#pragma optimize` *`flag`* and `#pragma endoptimize`. Nesting is allowed:

```
#pragma optimize e    /* Enable expression
...                      simplification              */
... C source ...
...
#pragma optimize c    /* Enable common expression
...                      elimination. Expression
... C source ...         simplification still enabled */
...
#pragma endoptimize   /* Disable common expression
...                      elimination                 */
#pragma endoptimize   /* Disable expression
...                      simplification              */
```

The compiler optimizes the code between the pragma pair as specified.

You can enable or disable the optimizations described below. The command line option for each optimization is given in brackets.

See also option **–O** (**––optimize**) in section 4.1, *Compiler Options*, of Chapter *Tool Options* of the *TriCore Reference Guide*.

### *Generic optimizations (frontend)*

#### *Common subexpression elimination (CSE)* *(option **–Oc**/**–OC**)*

The compiler detects repeated use of the same (sub–)expression. Such a "common" expression is replaced by a variable that is initialized with the value of the expression to avoid recomputation. This method is called *common subexpression elimination* (CSE).

#### *Expression simplification* *(option **–Oe**/**–OE**)*

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 (for example, array subscription).

#### *Constant propagation* *(option **–Op**/**–OP**)*

A variable with a known constant value is replaced by that value.

***Function Inlining*** *(option* ***–Oi/–OI****)*

Small functions that are not too often called, are inlined. This reduces execution speed at the cost of code size.

***Control flow simplification*** *(option* ***–Of/–OF****)*

A number of techniques to simplify the flow of the program by removing unnecessary code and reducing the number of jumps. For example:

*Switch optimization:*
A number of optimizations of a switch statement are performed, such as removing redundant case labels or even removing an entire switch.

*Jump chaining:*
A (conditional) 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 speeds up execution.

*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.

*Dead code elimination:*
Code that is never reached, is removed. The compiler generates a warning messages because this may indicate a coding error.

***Subscript strength reduction*** *(option* ***–Os/–OS****)*

An array of pointer subscripted with a loop iterator variable (or a simple linear function of the iterator variable), is replaced by the dereference of a pointer that is updated whenever the iterator is updated.

***Loop transformations*** *(option* ***–Ol/–OL****)*

Temporarily transform a loop with the entry point at the bottom, to a loop with the entry point at the top. This enables *constant propagation* in the initial loop test and code motion of loop invariant code by the *CSE* optimization.

***Forward store*** *(option* ***–Oo/–OO****)*

A temporary variable is used to cache multiple assignments (stores) to the same non–automatic variable.

### *Core specific optimizations (backend)*

#### *Coalescer* *(option* ***–Oa****/****–OA****)*

The coalescer seeks for possibilities to reduce the number of moves (MOV instruction) by smart use of registers. This optimizes both speed as code size.

#### *Peephole optimizations* *(option* ***–Oy****/****–OY****)*

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

#### *Instruction Scheduler* *(option* ***–Ok****/****–OK****)*

Instructions are rearranged with the following purposes:

- Pairing a L/S instruction with a data arithmetic instruction in order to fill both pipelines as much as possible.
- Avoiding structural hazards by inserting another non–related instruction.

#### *Software pipelining* *(option* ***–Ow****/****–OW****)*

A number of techniques to optimize loops. For example, within a loop the most efficient order of instructions is chosen by the *pipeline scheduler* and it is examined what instructions can be executed parallel.

#### *Use of SIMD instructions* *(option* ***–Om****/****–OM****)*

The iteration counts of loops are reduced where possible by taking advantage of the TriCore SIMD instructions. This optimizes speed, but may cause a slight increase in code size.

#### *Generic assembly optimizations* *(option* ***–Og****/****–OG****)*

A set of target independent optimizations that increase speed and decrease code size.

### 5.3.1 OPTIMIZE FOR SIZE OR SPEED

You can tell the compiler to focus on execution speed or code size during optimizations. You can do this by specifying a size/speed trade–off level from 0 (speed) to 4 (size). This trade–off does not turn optimization phases on or off. Instead, its level is a weight factor that is used in the different optimization phases to influence the heuristics. The higher the level, the more the compiler focuses on code size optimization.

To specify the size/speed trade–off optimization level:

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

   *The Project Options dialog box appears.*

2. Expand the **C Compiler** entry and select **Optimization**.

3. Select a **Size/speed trade–off** level.

See also option **–t** (**––tradeoff**) in section 4.1, *Compiler Options*, in Chapter *Tool Options* of the *TriCore Reference Guide*.

## 5.4 CALLING THE COMPILER

EDE uses a *makefile* to build your entire project. This means that you cannot run the compiler only. If you compile a single C source file from within EDE, the file is also automatically assembled. However, you can set options specific for the compiler. After you have build your project, the output files of the compilation step are available in your project directory.

To compile your program, click either one of the following buttons:

Compiles and assembles the currently selected file. This results in a relocatable object file (`.o`).

Builds your entire project but looks whether there are already files available that are needed in the building process. If so, these files will not be generated again, which saves time.

Builds your entire project unconditionally. All steps necessary to obtain the final `.elf` file are performed.

To access the TriCore processor options:

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

*The Project Options dialog box appears.*

2. Expand the **Processor** entry, fill in the **Processor Definition** page and optionally the **Startup** page and click **OK** to accept the processor options.

   *Processor options affect the invocation of all tools in the toolchain. In EDE you only need to set them once. The corresponding options for the compiler are listed in table 5–1.*

To specify the search path and include directories:

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

   *The Directories dialog box appears.*

2. Fill in the directory path settings and click **OK**.

To get access to the compiler options:

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

   *The Project Options dialog box appears.*

2. Expand the **C Compiler** entry, fill in the various pages and click **OK** to accept the compiler options.

   *The command line variant is shown simultaneously.*

The following processor options are available:

| EDE options | Command line |
|---|---|
| **Target** | |
| Target processor<br>User defined TriCore 2<br>FPU present | **–C***cpu*<br>**––is–tricore2**<br>**––fpu–present** |
| **Bypasses** | |
| CPU functional problem bypasses | **––silicon–bug=** *bug* |
| **Startup** | |
| Automatically add cstart.asm to your project | *EDE only* |
| **Bus Configuration** | |
| Initialize bus configuration registers in startup code | *EDE only* |

*Table 5–1: Processor options*

The following project directories are available:

| EDE options | Command line |
|---|---|
| **Directories** | |
| Executable files path | *$PATH environment* |
| Include files path | **–I***dir* |
| Library files path | *linker option* **–L***dir* |

*Table 5–2: Project directories*

The following compiler options are available:

| EDE options | Command line |
|---|---|
| **Preprocessing** | |
| Store the C compiler preprocess output (*file*`.pre`)<br>Keep comments<br>Strip #line source position info | **–E***flag*<br>**–Ec**<br>**–Ep** |
| Automatic inclusion of '.sfr' file | **––no–tasking–sfr** |
| Define preprocessor *macro* | **–D***macro*[**=***def*] |
| Predefine the macro __TASKING__ to the value 1<br>(Undefine the macro __TASKING__) | *no option*<br>**–U__TASKING__** |

| EDE options | Command line |
|---|---|
| Predefine the macro __CTC__ to compiler version<br>(Undefine the macro __CTC__)<br>Include an extra file at the beginning of the C source | *no option*<br>**–U__CTC__**<br>**–H***file* |
| **Language** | |
| ISO C standard 90 or 99 (default: 99)<br>Language extensions<br>Allow language extension keywords<br>Allow C++ style comments in C source | **–c**{**90**\|**99**}<br>**–A***flag*<br>**–Ak**<br>**–Ap** |
| **Debug Information** | |
| Generate symbolic debug information | **–g** |
| **Optimization** | |
| No optimization<br>Few optimizations<br>Medium optimizations (default)<br>Full optimization<br>Custom optimization<br>Size/speed trade–off (default: speed (0)) | **–O0**<br>**–O1**<br>**–O2**<br>**–O3**<br>**–O***flag*<br>**–t**{**0**\|**1**\|**2**\|**3**\|**4**} |
| **Allocation** | |
| Threshold for __near allocation<br>Threshold for __a0 allocation | **–N***threshold*<br>**–Z***threshold* |
| **Warnings** | |
| Report all warnings<br>Suppress all warnings<br>Suppress specific warnings<br>Treat warnings as errors | *no option –w*<br>**–w**<br>**–w***num*[,*num*]...<br>**––warnings–as–errors** |
| **MISRA C** | |
| MISRA C rules<br>Produce MISRA C report file | **––misrac=**{**all**\|*nr*[**–***nr*]**,**...}<br>*linker option*<br>**––misra–c–report** |
| **Miscellaneous** | |
| Merge C source code with assembly in output file (`.src`)<br>Treat 'double' as 'float' | **–s**<br>**–F** |

| EDE options | Command line |
|---|---|
| Use hardware single precision floating point instructions | **––fpu–present** |
| Treat 'char' variables as unsigned instead of signed | **–u** |
| Call functions indirect | **––indirect** |
| Additional command line options | *options* |

*Table 5–3: Compiler options*

The following options are only available on the command line:

| Description | Command line |
|---|---|
| Display invocation syntax | **–?** |
| Specify alignment (same as #pragma align *n*) | **––align=***n* |
| Make all addresses available for CSE | **––cse–all–addresses** |
| Show description of diagnostic(s) | **––diag=**[*fmt***:**]{**all**\|*nr*,...} |
| Redirect diagnostic messages to a file | **––error–file**[**=***file*] |
| Read options from file | **–f *file*** |
| Maximum size increment inlining (in %) (default: 25) | **––inline–max–incr=** *value* |
| Maximum size for function to always inline (default: 10) | **––inline–max–size=** *value* |
| Always use 32–bit integers for enumeration | **––integer–enumeration** |
| Keep output file after errors | **–k** |
| Send output to standard output | **–n** |
| Specify name of output file | **–o** *file* |
| Display version header only | **–V** |

*Table 5–4: Compiler options only available on the command line*

The invocation syntax on the command line is:

```
ctc [option]... [file]
```

The input file must be a C source file (`.c` or `.ic`).

```
ctc test.c
```

This compiles the file `test.c` and generates the file `test.src` which serves as input for the assembler.

For a complete overview of all options with extensive description, see section 4.1, *Compiler Options*, of Chapter *Tool Options* of the *TriCore Reference Guide*.

## 5.5 SPECIFYING A TARGET PROCESSOR

Before you call the compiler, you need to tell the compiler for which target processor it needs to compile. Based on the CPU type, the compiler includes a *special function register file*. This is a regular include file which enables you to use virtual registers that are located in memory.

### *Select a predefined target processor*

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

   *The Project Options dialog appears.*

2. Expand the **Processor** entry and select **Processor Definition**.

3. In the **Target processor** list select the target processor.

4. **C**lick **OK** to accept the new project settings.

   *The compiler includes the register file* `regcpu.sfr`.

### *Define a user defined target processor*

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

   *The Project Options dialog box appears.*

2. Expand the **Processor** entry and select **Processor Definition**.

3. In the **Target processor** list, select one of the **(user defined ...)** entries.

4. Specify (part of) the name of the user defined SFR files.

   *The compiler uses this name to include the register file* `regname.sfr`.

5. (Optional) Specify if your user defined target processor has an FPU (Floating–Point Unit) and/or an MMU (Memory Management Unit).

6. **C**lick **OK** to accept the new project settings.

The settings in EDE affect your whole project. If you use the command line, you must specify the same options to the assembler when assembling the file, or you can use the control program.

```
ctc -Ctc2 test.c
```

Instead of using the **–C** option, you can also include the special function register file in the C source with the line:

```
#include "regtc2.sfr"
```

## 5.6 HOW THE COMPILER SEARCHES INCLUDE FILES

When you use include files, you can specify their location in several ways. The compiler searches the specified locations in the following order:

1. If the #include statement contains a pathname, the compiler looks for this file. If no path is specified, the compiler looks in the same directory as the source file. This is only possible for include files that are enclosed in "".

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

2. When the compiler did not find the include file, it looks in the directories that are specified in the **Directories** dialog (**–I** option).

3. When the compiler did not find the include file (because it is not in the specified include directory or because no directory is specified), it looks which paths were set during installation. You can still change these paths.

See section 1.3.1, *Configuring the Embedded Development Environment* and environment variable CTCINC in section 1.3.2, *Configuring the Command Line Environment*, in Chapter *Software Installation*.

4. When the compiler still did not find the include file, it finally tries the default `include` directory relative to the installation directory.

## 5.7 COMPILING FOR DEBUGGING

Compiling your files is the first step to get your application ready to run on a target. However, during development of your application you first may want to debug your application.

To create an object file that can be used for debugging, you must instruct the compiler to include *symbolic debug information* in the source file.

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

   *The Project Options dialog box appears.*

2. Expand the **C Compiler** entry and select **Debug Information**.
3. Enable the option **Generate symbolic debug information**.
4. Click **OK** to accept the new project settings.

```
ctc -g
```

Due to different compiler opimizations, it might be possible that certain debug information is optimized away. Therefore, it is best to specify **No optimization** (**-O0**) when you want to debug your application.

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

   *The Project Options dialog box appears.*

2. Expand the **C Compiler** entry and select **Optimization**.
3. In the **Optimization level** box, select **No optimization**.

## 5.8 C CODE CHECKING: MISRA C

The C programming language is a standard for high level language programming in embedded systems, yet it is considered somewhat unsuitable for programming safety–related applications. Through enhanced code checking and strict enforcement of best practice programming rules, TASKING MISRA C code checking helps you to produce more robust code.

MISRA C specifies a subset of the C programming language which is intended to be suitable for embedded automotive systems. It consists of a set of 127 rules, defined in the document "Guidelines for the Use of the C Language in Vehicle Based Software" published by "Motor Industry Research Association" (MISRA).

For a complete overview of all MISRA C rules, see Chapter 9, *MISRA C Rules*, in the *Reference Guide*.

The MISRA C implementation in the compiler supports 117 of the 127 rules. Some MISRA C rules address documentation, run–time behavior, or other issues that cannot be checked by static source code inspection. Therefore, the following rules are not implemented: 2, 4, 6, 15, 41, 116, 117. In addition, the rules 23, 25 and 27 are not implemented in the compiler, because they cannot be checked without an application–wide overview.

During compilation of the code, violations of the enabled MISRA C rules are indicated with error messages and the build process is halted.

Note that not all MISRA C violations will be reported when other errors are detected in the input source. For instance, when there is a syntax error, all semantic checks will be skipped, including some of the MISRA C checks. Also note that some checks cannot be performed when the optimizations are switched off.

To ensure compliance to the MISRA C rules throughout the entire project, the TASKING TriCore linker can generate a MISRA C Quality Assurance report. This report lists the various modules in the project with the respective MISRA C settings at the time of compilation. You can use this in your company's quality assurance system to provide proof that company rules for best practice programming have been applied in the particular project.

***Apply MISRA C code checking to your application***

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

   *The Project Options dialog box appears.*

2. Expand the **C Compiler** entry and select **MISRA C**.

3. Select a MISRA C configuration. Select a predefined configuration for conformance with the required rules in the MISRA C guidelines.

4. (Optional) In the **MISRA C Rules** entry, specify the individual rules.

```
ctc --misrac={all | number [-number],...}
```

See compiler option **--misrac** in section 4.1, *Compiler Options* in Chapter *Tool Options* of the *TriCore Reference Guide*.

See linker option **--misra-c-report** in section 4.3, *Linker Options* in Chapter *Tool Options* of the *TriCore Reference Guide*.

## 5.9 C COMPILER ERROR MESSAGES

The **csc** compiler reports the following types of error messages:

***F Fatal errors***

After a fatal error the compiler immediately aborts compilation.

***E Errors***

Errors are reported, but the compiler continues compilation. No output files are produced unless you have set the compiler option **--keep-output-files** (the resulting output file may be incomplete).

***W Warnings***

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. You can control warnings in the **C Compiler | Warnings** page of the **Project | Project Options...** menu (compiler option **-w**).

***I Information***

Information messages are always preceded by an error message. Information messages give extra information about the error.

***S System errors***

System errors occur when internal consistency checks fail and should never occur. When you still receive the system error message

```
S9##: internal consistency check failed - please report
```

please report the error number and as many details as possible about the context in which the error occurred. The following helps you to prepare an e-mail using EDE:

1. From the **Help** menu, select **Technical Support -> Prepare Email...**

   *The Prepare Email form appears.*

2. Fill out the the form. State the error number and attach relevant files.

3. Click the **Copy to Email client** button to open your email application.

   *A prepared e-mail opens in your e-mail application.*

4. Finish the e–mail and send it.

***Display detailed information on diagnostics***

1. In the **Help** menu, enable the option **Show Help on Tool Errors**.
2. In the **Build** tab of the **Output** window, double–click on an error or warning message.

   *A description of the selected message appears.*

**ctc --diag=**[*format*:]{**all** | *number*,...}

See compiler option **--diag** in section 4.1, *Compiler Options* in Chapter *Tool Options* of the *TriCore Reference Guide*.

CHAPTER 6

# USING THE ASSEMBLER

TASKING

# 9 CHAPTER

## 6.1 INTRODUCTION

The assembler converts hand–written or compiler–generated assembly language programs into machine language, using the Executable and Linking Format (ELF) for object files.

The assembler takes the following files for input and output:

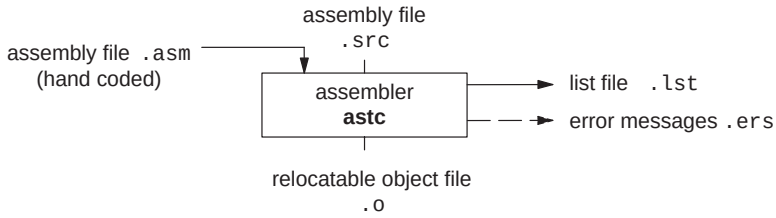



*Figure 6–1: Assembler*

This chapter first describes the assembly process. The various assembler optimizations are described in the second section. Third it is described how to call the assebmler and how to use its options. An extensive list of all options and their descriptions is included in the *Reference Guide*. Finally, a few important basic tasks are described.

## 6.2 ASSEMBLY PROCESS

The assembler generates relocatable output files with the extension `.o`. These files serve as input for the linker.

### *Phases of the assembly process*

1. Preprocess directives
2. Check syntax of instructions
3. Instruction grouping and reordering
4. Optimization (instruction size and generic instructions)
5. Generation of the relocatable object file and optionally a list file

The assembler integrates file inclusion and macro facilities. See section 4.9, *Macro Operations*, in Chapter *TriCore Assembly Language* for more information.

## 6.3 ASSEMBLER OPTIMIZATIONS

The **astc** assembler performs various optimizations to reduce the size of the assembled applications. There are two options available to influence the degree of optimization. To enable or disable optimizations:

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

   *The Project Options dialog box appears.*

2. Expand the **Assembler** entry and select **Optimization**.

You can enable or disable the optimizations described below. The command line option for each optimization is given in brackets.

See also option **–O** (**––optimize**) in section 4.2, *Assembler Options*, in Chapter *Tool Options* of the *TriCore Reference Guide*.

***Allow generic instructions*** *(option **–Og**/**–OG**)*

When this option is enabled, you can use generic instructions in your assembly source. The assembler tries to replace the generic instructions by faster or smaller instructions. For example, the instruction `jeq d0,#0,label1` is replaced by `jz d0,label1`.
By default this option is enabled. Because shorter instructions may influence the number of cycles, you may want to disable this option when you have written timed code. In that case the assembler encodes all instructions as they are.

***Optimize instruction size*** *(option **–Os**/**–OS**)*

When this option is enabled, the assembler tries to find the shortest possible operand encoding for instructions. By default this option is enabled.

## 6.4 CALLING THE ASSEMBLER

EDE uses a *makefile* to build your entire project. You can set options specific for the assembler. After you have build your project, the output files of the assembling step are available in your project directory.

To assemble your program, click either one of the following buttons:

Assembles the currently selected assembly file (`.asm` or `.src`). This results in a relocatable object file (`.o`).

Builds your entire project but looks whether there are already files available that are needed in the building process. If so, these files will not be generated again, which saves time.

Builds your entire project unconditionally. All steps necessary to obtain the final `.elf` file are performed.

To access the TriCore processor options:

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

   *The Project Options dialog box appears.*

2. Expand the **Processor** entry, fill in the **Processor Definition** page and optionally the **Startup** page and click **OK** to accept the processor options.

   *Processor options affect the invocation of all tools in the toolchain. In EDE you only need to set them once. The corresponding options for the assembler are listed in table 6–1.*

To get access to the assembler options:

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

   *The Project Options dialog box appears.*

2. Expand the **Assembler** entry, fill in the various pages and click **OK** to accept the compiler options.

   *The command line variant is shown simultaneously.*

The following processor options are available:

| EDE options | Command line |
|---|---|
| **Target** | |
| Target processor<br>User defined TriCore 2 | **–C***cpu*<br>**––is–tricore2** |
| FPU present | **––fpu–present** |
| MMU present | **––mmu–present** |
| **Bypasses** | |
| CPU functional problem bypasses | **––silicon–bug=** *bug* |
| **Startup** | |
| Automatically add cstart.asm to your project | *EDE only* |
| **Bus Configuration** | |
| Initialize bus configuration registers in startup code | *EDE only* |

*Table 6–1: Processor options*

The following assembler options are available:

| EDE options | Command line |
|---|---|
| **Preprocessing** | |
| Select TASKING preprocessor or no preprocessor | **–m**{**t**\|**n**} |
| Define preprocessor *macro* | **–D***macro*[**=***def*] |
| Include '.def' file | **––no–tasking–sfr** |
| **List File** | |
| Generate list file | **–l** |
| Include section summary in list file | **–tl** |
| *Suboptions for the* ***Generate list file*** *option* | **–L***flags* |
| **Debug Information** | |
| No debug information<br>Automatic HLL or assembly level debug information<br>Custom debug information | **–gAHLS**<br>**–gs**<br>**–g***flag* |
| **Optimization** | |
| Allow generic instructions<br>Optimize instruction size | **–Og**/**–OG** (= on/off)<br>**–Os**/**–OS** |

| EDE options | Command line |
|---|---|
| **Warnings** | |
| Report all warnings | *no option –w* |
| Suppress all warnings | **–w** |
| Suppress specific warnings | **–w***num*[,*num*]... |
| Treat warnings as errors | **––warnings–as–errors** |
| **Miscellaneous** | |
| Assemble case sensitive | **–c** |
| Allow memory management instructions | **––mmu–present** |
| Allow hardware floating–point instructions | **––fpu–present** |
| Labels are by default: | |
| local (default) | **–il** |
| global | **–ig** |
| Additional command line options | *options* |

*Table 6–2: Assembler options*

The following options are only available on the command line:

| Description | Command line |
|---|---|
| Display invocation syntax | **–?** |
| Show description of diagnostic(s) | **––diag=**[*fmt*:]{**all**\|*nr*,...} |
| Redirect diagnostic messages to a file | **––error–file**[**=***file*] |
| Read options from file | **–f *file*** |
| Keep output file after errors | **–k** |
| Specify name of output file | **–o** *file* |
| Display version header only | **–V** |

*Table 6–3: Assembler command line options*

The invocation syntax on the command line is:

```
astc [option]... [file]
```

The input file must be an assembly source file (`.asm` or `.src`).

```
astc test.asm
```

This assembles the file `test.asm` for and generates the file `test.o` which serves as input for the linker.

For a complete overview of all options with extensive description, see section 4.2, *Assembler Options*, of Chapter *Tool Options* of the *TriCore Reference Guide*.

## 6.5 SPECIFYING A TARGET PROCESSOR

Before you call the assembler, you need to tell the assembler for which target processor it needs to assemble. Based on the processor type, the assembler includes a *special function register file*. This is a regular include file which enables you to use virtual registers that are located in memory.

The settings in EDE affect your whole project. If you already specified these settings, you do not need to specify them again for the assembler. When you use the command line, you must specify the same options to the assembler as you did for the compiler.

***Select a predefined target processor***

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

   *The Project Options dialog appears.*

2. Expand the **Processor** entry and select **Processor Definition**.

3. In the **Target processor** list select the target processor.

4. **C**lick **OK** to accept the new project settings.

   *The assembler includes the register file* `reg`*`cpu`*`.def`.

***Define a user defined target processor***

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

   *The Project Options dialog box appears.*

2. Expand the **Processor** entry and select **Processor Definition**.

3. In the **Target processor** list, select one of the **(user defined ...)** entries.

4. Specify (part of) the name of the user defined SFR files.

   *The assembler uses this name to include the register file* `reg`*`name`*`.def`.

5. (Optional) Specify if your user defined target processor has an FPU (Floating–Point Unit) and/or an MMU (Memory Management Unit).

6. **C**lick **OK** to accept the new project settings.

```
astc –Ctc2 test.src
```

## 6.6 HOW THE ASSEMBLER SEARCHES INCLUDE FILES

When you use include files, you can specify their location in several ways. The assembler searches the specified locations in the following order:

1. If the `.INCLUDE` directive contains a pathname, the assembler looks for this file. If no path is specified, the assembler looks in the same directory as the source file.

2. When the assembler did not find the include file, it looks in the directories that are specified in the **Directories** dialog (**–I** option).

3. When the assembler did not find the include file (because it is not in the specified include directory or because no directory is specified), it looks which paths were set during installation. You can still change these paths if you like.

See section 1.3.1, *Configuring the Embedded Development Environment* and environment variable ASTCINC in section 1.3.2, *Configuring the Command Line Environment*, in Chapter *Software Installation*.

4. When the assembler still did not find the include file, it finally tries the default `include` directory relative to the installation directory.

## 6.7 GENERATING A LIST FILE

The list file is an additional output file that contains information about the generated code. With the options in the **List File** page of the **Assembler** entry in the **Project Options** dialog you choose to generate a list file or to skip it (**–l** option). You can also customize the amount and form of information (**–L** option).

If the assembler generates errors or warnings, these are reported in the list file just below the source line that caused the error or warning.

See section 5.1, *Assembler List File Format*, in Chapter *List File Formats* of the *Reference Guide* for an explanation of the format of the list file.

***Example:***

```
astc -l test.src
```

With this command the list file `test.lst` is created.

## 6.8 ASSEMBLER ERROR MESSAGES

The assembler produces error messages of the following types:

***F Fatal errors***

After a fatal error the assembler immediately aborts the assembling process.

***E Errors***

Errors are reported, but the assembler continues assembling. No output files are produced unless you have set the assembler option **--keep-output-files** (the resulting output file may be incomplete).

***W Warnings***

Warning messages do not result into an erroneous assembly output file. They are meant to draw your attention to assumptions of the assembler for a situation which may not be correct. You can control warnings in the **Assembler | Warnings** page of the **Project | Project Options...** menu (assembler option **-w**).

***Display detailed information on diagnostics***

1. In the **Help** menu, enable the option **Show Help on Tool Errors**.
2. In the **Build** tab of the **Output** window, double-click on an error or warning message.

   *A description of the selected message appears.*

**astc --diag=**[*format*:]{**all** | *number*,...}

See assembler option **--diag** in section 4.2, *Assembler Options* in Chapter *Tool Options* of the *TriCore Reference Guide*.

CHAPTER 7

# USING THE LINKER

TASKING

# 1 CHAPTER

## 7.1 INTRODUCTION

The linker **ltc** is a combined linker/locator. The linker phase combines relocatable object files (`.o` files, generated by the assembler), and libraries into a single *relocatable linker object file* (`.out`). The locator phase assigns absolute addresses to the linker object file and creates an absolute object file which you can load into a target processor. From this point the term *linker* is used for the combined linker/locator.

The linker takes the following files for input and output:

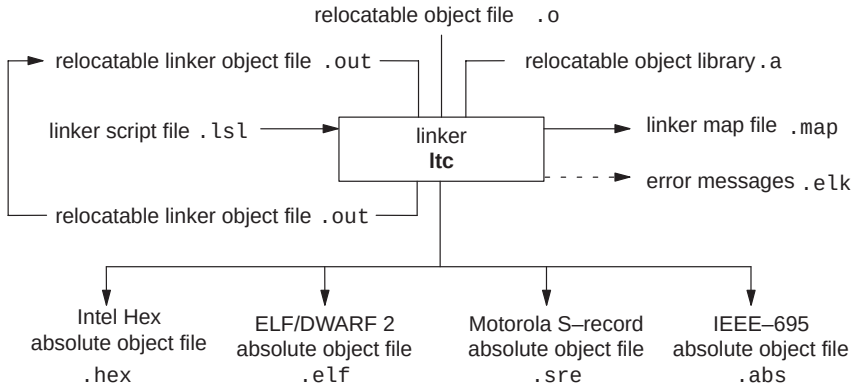



*Figure 7–1: ltc Linker*

This chapter first describes the linking process. Then it describes how to call the linker and how to use its options. An extensive list of all options and their descriptions is included in section 4.3, *Linker Options*, of the *Reference Guide*.

To gain even more control over the link process, you can write a script for the linker. This chapter shortly describes the purpose and basic principles of the *Linker Script Language* (LSL) on the basis of an example. A complete description of the LSL is included in Chapter 7, *Linker Script Language*, of the *Reference Guide*.

Finally, a few important features are described such as overlaying and choosing among various output formats.

## 7.2 LINKING PROCESS

The linker combines and transforms relocatable object files (`.o`) into a single absolute object file. This process consists of two phases: the linking phase and the locating phase.

In the first phase the linker combines the supplied relocatable object files and libraries into a single relocatable object file. In the second phase, the linker assigns absolute addresses to the object file so it can actually be loaded into a target.

### *Glossary of terms*

| Term | Definition |
|---|---|
| Absolute object file | Object code in which addresses have fixed absolute values, ready to load into a target. |
| Address | A specification of a location in an address space. |
| Address space | The set of possible addresses in a given space. A core can support multiple spaces, for example in a Harvard architecture the addresses that identify the location of an instruction refer to code space, whereas addresses that identify the location of a data object refer to a data space. |
| Architecture | A description of the characteristics of a core that are of interest for the linker. This encompasses the logical address space(s) and the internal bus structure. Given this information the linker can convert logical addresses into physical addresses. |
| Copy table | A section created by the linker. This section contains data that specifies how the startup code initializes the data and BSS sections. For each section the copy table contains the following fields:<br>– action: defines whether a section is copied or zeroed<br>– destination: defines the section's address in RAM.<br>– source: defines the sections address in ROM, zero for BSS sections<br>– length: defines the size of the section in MAUs |
| Core | An instance of one or more core architectures. |
| Derivative | The design of a processor. A description of one or more core architectures including internal memory and any number of buses. |
| Library | Collection of relocatable object files. Usually each object file in a library contains one symbol definition (for example, a function). |

| Term | Definition |
|---|---|
| Logical address | An address as encoded in an instruction word, an address generated by a core (CPU). |
| LSL file | The set of linker script files that are passed to the linker. |
| MAU | Minimum Addressable Unit. For a given processor the amount of memory loaded between an address and the next address. This is not necessarily a byte or a word. |
| Object code | The binary machine language representation of the C source. |
| Physical address | An addresses generated by the memory system. |
| Processor | An instance of one or more derivatives. Usually implemented as a (custom) chip, but can also be implemented on an FPGA, in which case the derivative can be designed by the developer. |
| Relocatable object file | Object code in which addresses are represented by symbols and thus relocatable. |
| Relocation | The process of assigning absolute addresses. |
| Relocation information | Information about how the linker must modify the machine code instructions when it relocates addresses. |
| Section | A group of instructions and/or data objects that occupy a contiguous range of addresses. |
| Section attributes | Attributes that define how the section should be linked or located. |
| Target | The hardware board on which an application is executing. A board contains at least one processor. However, a complex target may contain multiple processors and external memory and may be shared between processors. |
| Unresolved reference | A reference to a symbol for which the linker did not find a definition yet. |

*Table 7–1: Glossary of terms*

### 7.2.1 PHASE 1: LINKING

The linker takes one or more relocatable object files and/or libraries as input. A relocatable object file, as generated by the assembler, contains the following information:

- *Header information*: Overall information about the file, such as the code size, name of the source file it was assembled from, and creation date.
- *Object code*: Binary code and data, divided into various named sections. Sections are contiguous chunks of code or data that have to be placed in specific parts of the memory. The program addresses start at zero for each section in the object file.
- *Symbols*: Some symbols are exported – defined within the file for use in other files. Other symbols are imported – used in the file but not defined (external symbols). Generally these symbols are names of routines or names of data objects.
- *Relocation information*: A list of places with symbolic references that the linker has to replace with actual addresses. When in the code an external symbol (a symbol defined in another file or in a library) is referenced, the assembler does not know the symbol's size and address. Instead, the assembler generates a call to a preliminary relocatable address (usually 0000), while stating the symbol name.
- *Debug information*: Other information about the object code that is used by a debugger. The assembler optionally generates this information and can consist of line numbers, C source code, local symbols and descriptions of data structures.

The linker's first task is to resolve the external references between the supplied relocatable object files and/or libraries. Its second task is to combine the supplied relocatable object files into a single relocatable linker object file.

The linker starts its task by scanning all specified relocatable object files and libraries. If the linker encounters an unresolved symbol, it remembers its name and continues scanning. The symbol may be defined elsewhere in the same file, or in one of the other files or libraries that you specified to the linker. If the symbol is defined in a library, the linker extracts the object file with the symbol definition from the library. This way the linker collects all definitions and references of all of the symbols.

With this information, the linker combines the object code of all relocatable object files. The linker combines sections with the same section name and attributes into single sections, starting each section at address zero. The linker also substitutes (external) symbol references by (relocatable) numerical addresses where possible. At the end of the linking phase, the linker either writes the results to a file (a single relocatable object file) or keeps the results in memory for further processing during the locating phase.

The resulting file of the linking phase is a single relocatable object file (`.out`). If this file contains unresolved references, you can link this file with other relocatable object files (`.o`) or libraries (`.a`) to resolve the remaining unresolved references.

With the linker command line option **`--link–only`**, you can tell the linker to only perform this linking phase and skip the locating phase. The linker complains if any unresolved references are left.

### 7.2.2 PHASE 2: LOCATING

In the locating phase, the linker assigns absolute addresses to the object code, placing each section in a specific part of the target memory. The linker also replaces references to symbols by the actual address of those symbols. The resulting file is an absolute object file which you can actually load into a target memory. Optionally, when the resulting file should be loaded into a ROM device the linker creates a so–called copy table section which is used by the startup code to initialize the data and BSS sections.

#### *Code modification*

When the linker assigns absolute addresses to the object code, it needs to modify this code according to certain rules or *relocation expressions* to reflect the new addresses. These relocation expressions are stored in the relocatable object file. Consider the following snippet of x86 code that moves the contents of variable `a` to variable `b` via the `eax` register:

```
A1 3412 0000 mov a,%eax   (a defined at 0x1234, byte reversed)
A3 0000 0000 mov %eax,b   (b is imported so the instruction refers to
                           0x0000 since its location is unknown)
```

Now assume that the linker links this code so that the section in which `a` is located is relocated by 0x10000 bytes, and `b` turns out to be at 0x9A12. The linker modifies the code to be:

```
A1 3412 0100 mov a,%eax   (0x10000 added to the address)
A3 129A 0000 mov %eax,b   (0x9A12 patched in for b)
```

These adjustments affect instructions, but keep in mind that any pointers in the data part of a relocatable object file have to be modified as well.

The linker can produce its output in different file formats. The default ELF/DWARF2 format (`.elf`) contains an image of the executable code and data, and can contain additional debug information. The Intel–Hex format (`.hex`) and Motorola S–record format (`.sre`) only contain an image of the executable code and data. You can specify a format with the options **–F** (**––format**) and **–c** (**––chip–format**).

Via a so–called *linker script file* you can gain complete control over the linker. The script language used to describe these features is called the *Linker Script Language* (LSL). You can define:

- The types of memory that are installed in the embedded target system:

  To assign locations to code and data sections, the linker must know what memory devices are actually installed in the embedded target system. For each physical memory device the linker must know its start–address, its size, and whether the memory is read–write accessible (RAM) or read–only accessible (ROM).

- How and where code and data should be placed in the physical memory:

  Embedded systems can have complex memory systems. If for example on–chip and off–chip memory devices are available, the code and data located in internal memory is typically accessed faster and with dissipating less power. To improve the performance of an application, specific code and data sections should be located in on–chip memory. By writing your own LSL file, you gain full control over the locating process.

- The underlying hardware architecture of the target processor.

  To perform its task the linker must have a model of the underlying hardware architecture of the processor you are using. For example the linker must know how to translate an address used within the object file (a logical address) into an offset in a particular memory device (a physical address). In most linkers this model is hard coded in the executable and can not be modified. For the **ltc** linker this hardware model is described in the linker script file. This solution is chosen to support configurable cores that are used in system–on–chip designs.

Instead of writing your own linker script file, you can use the standard linker script files with architecture descriptions for the TriCore.

See also section 7.6, *Controlling the Linker with a Script*.

### 7.2.3 LINKER OPTIMIZATIONS

During the linking and locating phase, the linker looks for opportunities to optimize the object code. Both code size and execution speed can be optimized. To enable or disable optimizations:

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

   *The Project Options dialog box appears.*

2. Expand the **Linker** entry and select **Optimization**.

You can enable or disable the optimizations described below. The command line option for each optimization is given in brackets.

See also option **–O** (**––optimize**) in section 4.3, *Linker Options*, in Chapter *Tool Options* of the *TriCore Reference Guide*.

***First fit decreasing*** *(option **–Ol**/**–OL**)*

When the physical memory is fragmented or when address spaces are nested it may be possible that a given application cannot be located although the size of available physical memory is larger than the sum of the section sizes. Enable the first–fit–decreasing optimization when this occurs and re–link your application.

The linker's default behavior is to place sections in the order that is specified in the LSL file. This also applies to sections within an unrestricted group. If a memory range is partially filled and a section must be located that is larger than the remainder of this range, then the section and all subsequent sections are placed in a next memory range. As a result of this gaps occur at the end of a memory range.

When the first–fit–decreasing optimization is enabled the linker will first place the largest sections in the smallest memory ranges that can contain the section. Small sections are located last and can likely fit in the remaining gaps.

***Copy table compression*** *(option **–Ot**/**–OT**)*

The startup code initializes the application's data and BSS areas. The information about which memory addresses should be zeroed (bss) and which memory ranges should be copied from ROM to RAM is stored in the copy table.

When this optimization is enabled the linker will try to locate sections in such a way that the copy table is as small as possible thereby reducing the application's ROM image.

## 7.3 CALLING THE LINKER

EDE uses a *makefile* to build your entire project. This means that you cannot run only the linker. However, you can set options specific for the linker. After you have build your project, the output files of the linking step are available in your project directory.

To link your program, click either one of the following buttons:

Builds your entire project but only updates files that are out–of–date or have been changed since the previous build, which saves time.

Builds your entire project unconditionally. All steps necessary to obtain the final `.elf` file are performed.

To get access to the linker options:

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

   *The Project Options dialog box appears.*

2. Expand the **Linker** entry. Select the subentries and set the options in the various pages.

   *The command line variant is shown simultaneously.*

The following linker options are available:

| EDE options | Command line |
|---|---|
| **Output Format** | |
| Output formats | **–F***format*<br>**–c***format*[**:***addr_size*] |
| **Script File** | |
| Select linker script file | **–d***file* |
| **Map File** | |
| Generate a map file (.map)<br>*Suboptions for the **Generate a map file** option* | **–M**<br>**–m***flags* |
| **Libraries** | |
| Link default C libraries<br>Use non–trapping floating point library<br>Use trapping floating point library<br>Rescan libraries to solve unresolved exernals<br>Libraries | **–l***x*<br>**–lfp**<br>**–lfpt**<br>**––no–rescan**<br>*library files* |
| **Optimization** | |
| Use a 'first fit decreasing' algorithm<br>Emit smart restrictions to reduce copy table size | **–Ol**/**–OL** (= on/off)<br>**–Ot**/**–OT** |
| **Warnings** | |
| Report all warnings<br>Suppress all warnings<br>Suppress specific warnings<br>Treat warnings as errors | *no option –w*<br>**–w**<br>**–w***num*[,*num*]...<br>**––warnings–as–errors** |
| **Miscellaneous** | |
| Include symbolic debug information<br>Print the name of each file as it is processed<br>Link case sensitive (required for C language)<br>Dump processor and memory info from LSL file<br>Additional command line options | **–S** (strip debug)<br>**–v**<br>**––case–sensitive**<br>**––lsl–dump**[**=***file*]<br>*options* |

*Table 7–2: Linker options*

The following options are only available on the command line:

| Description | Command line |
|---|---|
| Display invocation syntax | **–?** |
| Define preprocessor *macro* | **–D***macro*[**=***def*] |
| Show description of diagnostic(s) | **––diag=**[*fmt***:**]{**all**\|*nr*,...} |
| Specify a symbol as unresolved external | **–e***symbol* |
| Redirect errors to a file with extension `.elk` | **––error–file**[**=***file*] |
| Read options from file | **–f *file*** |
| Scan libraries in given order | **––first–library–first** |
| Search only in **–L** directories, not in default path | **––ignore–default–library–path** |
| Keep output files after errors | **–k** |
| Link only, do not locate | **––link–only** |
| Check LSL file(s) and exit | **––lsl–check** |
| Do not generate ROM copy | **–N** |
| Locate all ROM sections in RAM | **––non–romable** |
| Name of the resulting output file | **–o** *file* |
| Link incrementally | **–r** |
| Display version header only | **–V** |

*Table 7–3: Linker command line options*

The invocation syntax on the command line is:

```
ltc [option]... [file]... ]...
```

When you are linking multiple files (either relocatable object files (`.o`) or libraries (`.a`), it is important to specify the files in the right order. This is explained in Section 7.4.1, *Specifying Libraries to the Linker*

For a complete overview of all options with extensive description, see section 4.3, *Linker Options*, of the *Reference Guide*.

## 7.4 LINKING WITH LIBRARIES

There are two kinds of libraries: system libraries and user libraries.

### *System library*

The system libraries are installed in subdirectories of the `c:\ctc\lib` directory. An overview of the system libraries is given in the following table.

| Library to link | Description |
|---|---|
| libc.a | C library<br>(With full printf/scanf functionality. Some functions require the floating point library. Also includes the startup code.) |
| libcs.a | C library single precision (compiler option **–F**)<br>(With full printf/scanf functionality. Some functions require the floating point library. Also includes the startup code.) |
| libcs_fpu.a | C library single precision with FPU instructions (compiler option **–F** and **––fpu–present**) |
| libfp.a | Floating point library (non–trapping) |
| libfpt.a | Floating point library (trapping)<br>(Control program option **–fptrap**) |
| libfp_fpu.a | Floating point library (non–trapping, with FPU instructions)<br>(Compiler option **––fpu–present**) |
| libfpt_fpu.a | Floating point library (trapping, with FPU instructions)<br>(Control program option **–fptrap**, compiler option **––fpu–present**) |
| librt.a | Run–time library |

*Table 7–4: Overview of libraries*

For more information on these libraries see section 3.12, *Libraries*, in Chapter *TriCore C Language*.

When you want to link system libraries, you must specify this with the option **–l**. With the option **–lc** you specify the system library `libc.a`.

### *User library*

You can also create your own libraries. Section 8.4, *Archiver*, in Chapter *Using the Utilities*, describes how you can use the archiver to create your own library with object modules. To link user libraries, specify their filenames on the command line.

### 7.4.1 SPECIFYING LIBRARIES TO THE LINKER

In EDE you can specify both system and user libraries.

***Link a system library with EDE***

To specify to link the default C libraries:

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

   *The Project Options dialog box appears.*

2. Expand the **Linker** entry and select **Libraries**.
3. Select **Link default C libraries**.
4. Select a floating–point library: **non–trapping** or **trapping**.
5. Click **OK** to accept the linker options.

The invocation syntax on the command line is for example:

```
ltc –lc start.o
```

***Link a user library in EDE***

To specify your own libraries:

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

   *The Project Options dialog box appears.*

2. Expand the **Linker** entry and select **Libraries**.
3. Add your libraries to the **Libraries** field and click **OK** to accept the linker options.

The invocation syntax on the command line is for example:

```
ltc start.o mylib.a
```

If the library resides in a subdirectory, specify that directory with the library name:

```
ltc start.o mylibs\mylib.a
```

***Library order***

The order in which libraries appear on the command line is important. By default the linker processes object files and libraries in the order in which they appear at the command line. With the option **––first–library–first** you can tell the linker to scan the libraries from left to right, and extract symbols from the first library where the linker finds it.

Example:

```
ltc ––first–library–first a.a test.o b.a
```

If the file `test.o` calls a function which is both present in `a.a` and `b.a`, normally the function in `b.a` would be extracted. With this option the linker first tries to extract the symbol from the first library `a.a`.

## 7.4.2 HOW THE LINKER SEARCHES LIBRARIES

***System libraries***

You can specify the location of system libraries (specified with option **–l**) in several ways. The linker searches the specified locations in the following order:

1. The linker first looks in the directories that are specified in the **Directories** dialog (**–L** option). If you specify the **–L** option without a pathname, the linker stops searching after this step.
2. When the linker did not find the library (because it is not in the specified library directory or because no directory is specified), it looks which paths were set during installation. You can still change these paths if you like.

See environment variables LIBTC1V1_2, LIBTC1V1_3 and LIBTC2 in section 1.3.2, *Configuring the Command Line Environment*, in Chapter *Software Installation*.

3. When the linker did not find the library, it tries the default `ctc\lib` directory which was created during installation (or a processor specific sub–directory).

***User library***

If you use your own library, the linker searches the library in the current directory only.

### 7.4.3 HOW THE LINKER EXTRACTS OBJECTS FROM LIBRARIES

A library built with **artc** always contains an index part at the beginning of the library. The linker scans this index while searching for unresolved externals. However, to keep the index as small as possible, only the defined symbols of the library members are recorded in this area.

When the linker finds a symbol that matches an unresolved external, the corresponding object file is extracted from the library and is processed. After processing the object file, the remaining library index is searched. If after a complete search unresolved externals are introduced, the library index will be scanned again.

The **–v** option shows how libraries have been searched and which objects have been extracted.

#### *Resolving symbols*

If you are linking from libraries, only the objects that contain symbol definition(s) to which you refer, are extracted from the library. This implies that if you invoke the linker like:

```
ltc mylib.a
```

nothing is linked and no output file will be produced, because there are no unresolved symbols when the linker searches through `mylib.a`.

It is possible to force a symbol as external (unresolved symbol) with the option **–e**:

```
ltc –e main mylib.a
```

In this case the linker searches for the symbol `main` in the library and (if found) extracts the object that contains `main`. If this module contains new unresolved symbols, the linker looks again in `mylib.a`. This process repeats until no new unresolved symbols are found.

## 7.5 INCREMENTAL LINKING

With the TriCore linker **ltc** it is possible to link *incrementally*. Incremental linking means that you link some, but not all `.o` modules to a relocatable object file `.out`. In this case the linker does not perform the locating phase. With the second invocation, you specify both new `.o` files and the `.out` file you had created with the first invocation.

Incremental linking is only possible on the command line.

```
ltc –r test1.o –otest.out
ltc test2.o test.out
```

This links the file `test1.o` and generates the file `test.out`. This file is used again and linked together with `test2.o` to create the file `task1.elf` (the default name if no output filename is given in the default ELF/DWARF 2 format).

With incremental linking it is normal to have unresolved references in the output file until all `.o` files are linked and the final `.out` or `.elf` file has been reached. The option **–r** for incremental linking also suppresses warnings and errors because of unresolved symbols.

## 7.6 CONTROLLING THE LINKER WITH A SCRIPT

With EDE or the options on the command line you can control the linker to a certain degree. However, when you exactly want to determine where your sections will be located, how much memory is available, which sorts of memory are available, and so on, you can write a script.

The language for the script is called the *Linker Script Language*, or shortly LSL. You can specify the script file to the linker, which reads it and locates your application exactly as defined in the script. If you do not specify your own script file, the linker always reads a standard script file which is supplied with the toolchain.

### 7.6.1 PURPOSE OF THE LINKER SCRIPT LANGUAGE

The Linker Script Language (LSL) serves three purposes:

1. It provides the linker with a definition of the target's core architecture and its *internal* memory (this is called the derivative). These definitions are written by Altium and supplied with the toolchain.

2. It provides the linker with a specification of the *external* memory attached to the target processor. The template `extmem.lsl` is supplied with the toolchain.

3. It provides the linker with information on how your application should be located in memory. This gives you, for example, the possibility to create overlaying sections.

The linker accepts multiple LSL files. You can use the specifications of the TriCore architectures and derivatives that Altium has supplied in the `include.lsl` directory. Do not change these files.

If you attached external memory to a derivative you must specify this in a separate LSL file and pass both the LSL file that describes the derivative's architecture and your LSL file that contains the memory specification to the linker. Next you may also want to specify how sections should be located and overlaid.

LSL has its own syntax. In addition, you can use the standard ANSI C preprocessor keywords because the linker sends the script file first to the C preprocessor before it starts interpreting the script.

The complete syntax is described in Chapter 7, *Linker Script Language*, in the *Reference Guide*.

### 7.6.2 EDE AND LSL

In EDE you can specify the size of the stack and heap; the physical memory attached to the processor; identify that particular address ranges are reserved; and specify which sections are located where in memory. EDE translates your input into an LSL file that is stored in the project directory under the name `project.lsl` and passes this file to the linker.

If you want to learn more about LSL you can inspect the generated file *project*.lsl. To change the LSL settings:

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

   *The Project Options dialog box appears.*

2. Expand the **Linker** entry and select **Script File**.

3. Make your changes.

Each time you close the Project Options dialog the file *project*.lsl is updated and you can immediately see how your settings are encoded in LSL.

Note that EDE supports ChromaCoding (applying color coding to text) and template expansion when you edit LSL files.

### 7.6.3 STRUCTURE OF A LINKER SCRIPT FILE

A script file consists of several definitions. The definitions can appear in any order.

***The architecture definition (required)***

In essence an *architecture definition* describes how the linker should convert virtual addresses into physical addresses for a given type of core. If the core supports multiple address spaces, then for each space the linker must know how to perform this conversion. In this context a physical address is an offset on a given internal or external bus. Additionally the architecture definition contains information about items such as the (hardware) stack and the interrupt vector table.

This specification is normally written by Altium. For each TriCore core architecture, a separate LSL file is provided. These are tc1v1_2.lsl, tc1v1_3.lsl, and tc2.lsl.

The architecture definition of the LSL file should not be changed by you unless you also modify the core's hardware architecture. If the LSL file describes a multi–core system an architecture definition must be available for each different type of core.

### *The derivative definition (required)*

The *derivative definition* describes the configuration of the internal (on–chip) bus and memory system. Basically it tells the linker how to convert offsets on the busses specified in the architecture definition into offsets in internal memory. A derivative definition must be present in an LSL file. Microcontrollers and DSPs often have internal memory and I/O sub–systems apart from one or more cores. The design of such a chip is called a *derivative*.

Altium provides LSL descriptions of supported derivatives, along with "SFR files", which provide easy access to registers in I/O sub–systems from C and assembly programs. When you build an ASIC or use a derivative that is not (yet) supported by the TASKING tools, you may have to write a derivative definition.

### *The processor definition*

The *processor definition* describes an instance of a derivative. Typically the processor definition instantiates one derivative only (single–core processor). A processor that contains multiple cores having the same (homogeneous) or different (heterogeneous) architecture can also be described by instantiating multiple derivatives of the same or different types in separate processor definitions.

If for a derivative 'A' no processor is defined in the LSL file, the linker automatically creates a processor named 'A' of derivative 'A'. This is why for single–processor applications it is enough to specify the derivative in the LSL file, for example with **–dtc1920a.lsl**.

### *The memory and bus definitions (optional)*

Memory and bus definition are used within the context of a derivative definition to specify internal memory and on–chip busses. In the context of a board specification the memory and bus definitions are used to define external (off–chip) memory and busses. Given the above definitions the linker can convert a logical address into an offset into an on–chip or off–chip memory device.

### *The board specification*

The processor definition and memory and bus definitions together form a *board specification*. LSL provides language constructs to easily describe single–core and heterogeneous or homogeneous multi–core systems. The board specification describes all characteristics of your target board's system busses, memory devices, I/O sub–systems, and cores that are of interest to the linker. Based on the information provided in the board specification the linker can for each core:

- convert a virtual address to a physical addresses (offsets within a memory device)
- locate sections in physical memory
- maintain an overall view of the used and free physical memory within the whole system while locating

### *The section layout definition (optional)*

The optional *section layout definition* enables you to exactly control where input sections are located. Features are provided such as: the ability to place sections at a given load–address or run–time address, to place sections in a given order, and to overlay code and/or data sections.

### *Example: Skeleton of a Linker Script File*

A linker script file that defines a derivative "X" based on the TC1V1.3 architecture, its external memory and how sections are located in memory, may have the following skeleton:

```
architecture TC1V1.3
{
    // Specification of the TC1v1.3 core architecture.
    // Written by Altium.
}
```

```
derivative X           // derivative name is arbitrary
{
    // Specification of the derivative.
    // Written by Altium.
    core tc            // always specify the core
    {
       architecture = TC1V1.3;
    }

    bus fpi_bus        // internal bus
    {
       // maps to fpi_bus in "tc" core
    }

    // internal memory
}

processor proc1        // processor name is arbitrary
{
    derivative = X;

    // You can omit this part, except if you use a
    // multi-core system.
}

memory ext_name
{
    // external memory definition
}

section_layout proc1:tc:linear    // section layout
{
    // section placement statements

    // sections are located in address space 'linear'
    // of core 'tc' of processor 'proc1'
}
```

### 7.6.4 THE ARCHITECTURE DEFINITION: SELF-DESIGNED CORES

Although you will probably not need to program the architecture definition (unless you are building your own processor core) it helps to understand the Linker Script Language and how the definitions are interrelated.

Within an architecture definition the characteristics of a target processor core that are important for the linking process are defined. These include:

- space definitions: the logical address spaces and their properties
- bus definitions: the I/O busses of the core architecture
- mappings: the address translations between logical address spaces, the connections between logical address spaces and busses and the address translations between busses

#### *Address spaces*

A logical address space is a memory range for which the core has a separate way to encode an address into instructions. For example, the Tricore's 32–bit linear address space encloses 16 24–bit sub–spaces and 16 14–bit sub–spaces. See also the *Tricore Architecture Manual* sections "*Memory Model*" and "*Addressing Model*".

Most microcontrollers and DSPs support multiple address spaces. An address space is a range of addresses starting from zero. Normally, the size of an address space is to $2^N$, with N the number of bits used to encode the addresses.

The relation of an address space with another address space can be one of the following:

- one space is a subset of the other. These are often used for "small" absolute, and relative addressing.
- the addresses in the two address spaces represent different locations so they do not overlap. This means the core must have separate sets of address lines for the address spaces. For example, in Harvard architectures we can identify at least a code and a data memory space.

Address spaces (even nested) can have different minimal addressable units (MAU), alignment restrictions, and page sizes.

### *The TriCore architecture in LSL notation*

The best way to program the architecture definition, is to start with a drawing. The following figure shows a part of the TriCore architecture:

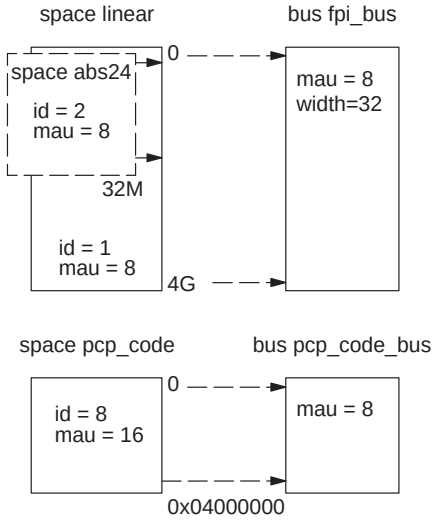



*Figure 7–2: Scheme of (part of) the TriCore architecture*

The figure shows three address spaces called `linear`, `abs24` and `pcp_code`. The address space `abs24` is a subset of the address space `linear`. All address spaces have attributes like a number that identifies the logical space (id), a MAU and an alignment. In LSL notation the definition of these address spaces looks as follows:

```
space linear
{
     id = 1;
     mau = 8;

     map (src_offset=0x00000000, dest_offset=0x00000000,
          size=4G, dest=bus:fpi_bus);
}
```

```
space abs24
{
     id = 2;
     mau = 8;

     map (src_offset=0x00000000, dest_offset=0x00000000,
          size=2M, dest=space:linear);
     map (src_offset=0x10000000, dest_offset=0x10000000,
          size=2M, dest=space:linear);
     map (src_offset=0x20000000, dest_offset=0x20000000,
          size=2M, dest=space:linear);
   //...
}

space pcp_code
{
     id = 8;
     mau = 16;
     map (src_offset=0x00000000, dest_offset=0,
          size=0x04000000, dest=bus:pcp_code_bus);
}
```

The keyword `map` corresponds with the arrows in the drawing. You can map:

- address space => address space
- address space => bus
- memory => bus (not shown in the drawing)
- bus => bus (not shown in the drawing)

Next the two internal busses, named `fpi_bus` and `pcp_code_bus` must be defined in LSL:

```
bus fpi_bus
{
     mau = 8;
     width = 32;  // there are 32 data lines on the bus
}

bus pcp_code_bus
{
     mau = 8;
     width = 8;
}
```

This completes the LSL code in the architecture definition. Note that all code above goes into the architecture definition, thus between:

```
architecture TC1V1.3
{
    All code above goes here.
}
```

### 7.6.5 THE DERIVATIVE DEFINITION: SELF-DESIGNED PROCESSORS

Although you will probably not need to program the derivative definition (unless you are building your own processor) it helps to understand the Linker Script Language and how the definitions are interrelated.

A derivative is the design of a processor, as implemented on a chip. It can be an ASIC or a generic design and comprises one or more cores and on–chip memory. The derivative definition includes:

- core definition: the core architecture
- bus definition: the I/O busses of the core architecture
- memory definitions: internal (or on–chip) memory

***Core***

Each derivative must have a specification of its core architecture in LSL:

```
core tc
{
   architecture = TC1V1.3;
}
```

***Bus***

Each derivative must contain a bus definition for connecting external memory. In this example, the bus `fpi_bus` maps to the bus `fpi_bus` defined in the architecture definition of core `tc`:

```
bus fpi_bus
{
   mau = 8;
   width = 32;
   map (dest=bus:tc:fpi_bus, dest_offset=0, size=4G);
}
```

***Memory***

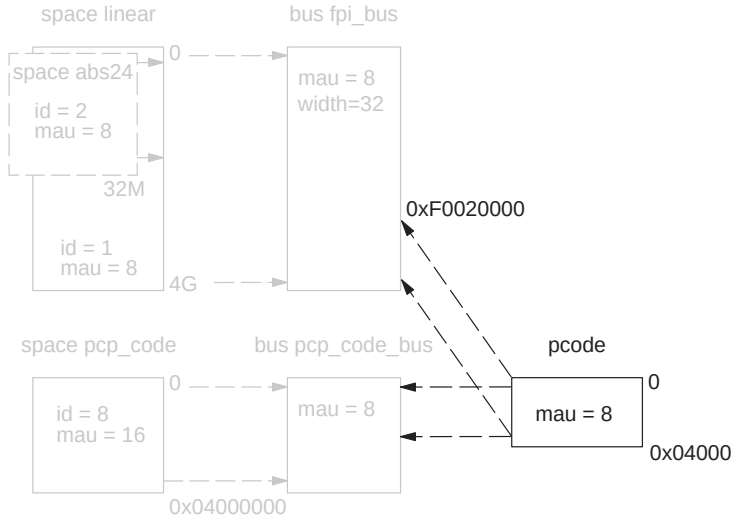



*Figure 7–3: Internal memory definition for a derivative*

According to the drawing, the TriCore contains internal memory called `pcode` with a size `0x04000` (16k). This is physical memory which is mapped to the internal bus `pcp_code_bus` and to the `fpi_bus`, so both the tc unit and the pcp can access the memory:

```
memory pcode
{
     mau = 8;
     size = 16k;
     type = ram;
     map (dest=bus:tc:fpi_bus, dest_offset=0xF0020000,
          size=16k);
     map (dest=bus:tc:pcp_code_bus, size=16k);
}
```

This completes the LSL code in the derivative definition. Note that all code above goes into the derivative definition, thus between:

```
derivative X    // name of derivative
{
    All code above goes here.
}
```

If you want to create a custom derivative and you want to use EDE to select sections, the core of the derivative must be called "tc", since EDE uses this name in the generated LSL file. If you want to specify external memory in EDE, the custom derivative must contain a bus named "fpi_bus" for the same reason. In EDE you have to define a target processor as specified in section 5.5, *Specifying a Target Processor*, in Chapter *Using the Compiler*.

### 7.6.6 THE MEMORY DEFINITION: DEFINING EXTERNAL MEMORY

Once the processor is defined in LSL, you may want to extend the processor with external (or off–chip) memory. You need to specify the location and size of the physical external memory devices in the target system.

The principle is the same as defining the core's architecture but now you need to fill the memory definition:

```
memory name
{
    External memory definitions.
}
```

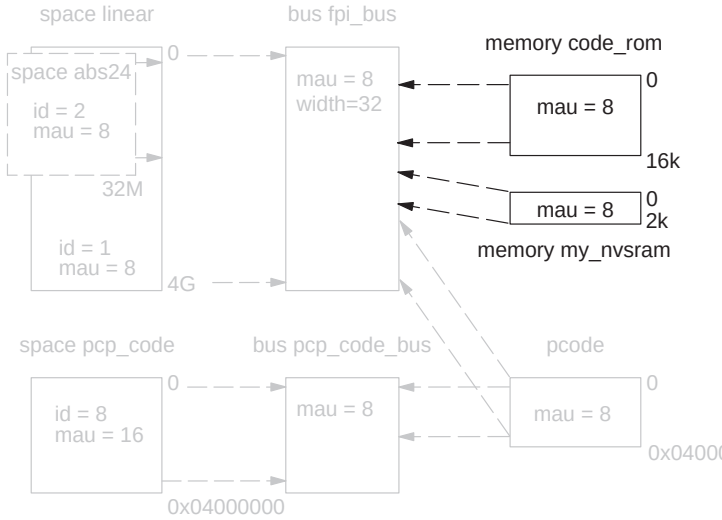



*Figure 7–4: Adding external memory to the TriCore architecture*

Suppose your embedded system has 16k of external ROM, named `code_rom` and 2k of external NVRAM, named `my_nvsram`. (See figure above.) Both memories are connected to the bus `fpi_bus`. In LSL this looks like follows:

```
memory code_rom
{
     type = rom;
     mau = 8;
     size = 16k;
     map (dest=bus:X:fpi_bus, dest_offset=0xa0000000,
          size=16k);
}
```

The memory `my_nvsram` is connected to the bus with an offset of 0xc0000000:

```
memory my_nvsram
{
        mau = 8;
        size = 2k;
        type = ram;
        map (dest=bus:X:fpi_bus, dest_offset=0xc0000000,
             size=2k);
}
```

### 7.6.7 THE SECTION LAYOUT DEFINITION: LOCATING SECTIONS

Once you have defined the internal core architecture and optional external memory, you can actually define where your application must be located in the physical memory.

During compilation, the compiler divides the application into sections. Sections have a name, an indication in which address space it should be located and attributes like writable or read–only.

In the section layout definition you can exactly define how input sections are placed in address spaces, relative to each other, and what their absolute run–time and load–time addresses will be. To illustrate section placement the following example of a C program is used:

### *Example: section propagation through the toolchain*

To illustrate section placement, the following example of a C program (`bat.c`) is used. The program prints the number of times it has been executed.

```
#define BATTERY_BACKUP_TAG  0xa5f0
#include <stdio.h>

int  uninitialized_data;
int  initialized_data = 1;
#pragma section all  ”non_volatile”
int  battery_backup_tag;
int  battery_backup_invok;
#pragma section all

void main (void)
{
     if (battery_backup_tag != BATTERY_BACKUP_TAG )
     {
         // battery back–upped memory area contains invalid data
         // initialize the memory
         battery_backup_tag = BATTERY_BACKUP_TAG;
         battery_backup_invok = 0;
     }
     printf( ”This application has been invoked %d times\n”,
              battery_backup_invok++);
}
```

The compiler assigns names and attributes to sections. With the `#pragma section all "`*`name`*`"` the compiler's default section naming convention is overruled and a section with the name `non_volatile` is defined. In this section the battery back–upped data is stored.

By default the compiler creates the section `.zbss.bat.uninitialized_data` to store uninitialized data objects. The section prefix ".zbss" tells the linker to locate the section in address space `abs18` and that the section content should be filled with zeros at startup.

As a result of the `#pragma section all "non_volatile"`, the data objects between the pragma pair are placed in `.zbss.non_volatile`. Note that ".zbss" sections are cleared at startup. However, battery back–upped sections should not be cleared and therefore we will change this section attribute using the LSL.

The generated assembly may look like:

```
        .extern printf
        .name   ”bat”
        .extern _START

        .sdecl  ”.text.bat.main”,CODE
        .sect   ”.text.bat.main
        .align  4
        .global main

; Function main
main:
        sub16.a a10,#8
        mov.u   d15,#42480
        ld.w    d0,battery_backup_tag
        jeq     d15,d0,L_2
        .
        .
        .
        jg      printf
        ; End of function
        ; End of section

        .sdecl  ”.zbss.bat.uninitialized_data”,DATA
        .sect   ”.zbss.bat.uninitialized_data”
        .global uninitialized_data
        .align  2
uninitialized_data:
        .space  4
        ; End of section

        .sdecl  ”.zdata.bat.initialized_data”,DATA
        .sect   ”.zdata.bat.initialized_data”
        .global initialized_data
        .align  2
initialized_data:
        .word   1
        ; End of section

        .sdecl  ”.zbss.non_volatile”,DATA
        .sect   ”.zbss.non_volatile”
        .global battery_backup_tag
        .align  2
battery_backup_tag:
        .space  4
        .global battery_backup_invok
        .align  2
battery_backup_invok:
        .space  4
        ; End of section
```

```
        .sdecl  ".rodata.bat..1.ini",DATA,ROM
        .sect   ".rodata.bat..1.ini"
        .align  2
_1_ini:
        .byte     84   /* T */
        .byte     104  /* h */
        ; This application has been invoked %d times\n
        .byte     10   /* \n   */
        .byte     0    /* NULL */

        ; End of section
        ; Module end
```

### *Section placement*

The number of invocations of the example program should be saved in *non–volatile* (battery back–upped) memory. This is the memory `my_nvsram` from the example in the previous section.

To control the locating of sections, you need to write one or more section definitions in the LSL file. At least one for each address space where you want to change the default behavior of the linker. In our example, we need to locate sections in the address space `abs18`:

```
section_layout ::abs18
{
    Section placement statements
}
```

To locate sections, you must create a group in which you select sections from your program. For the battery back–up example, we need to define one group, which contains the section `.zbss_non_volatile`. All other sections are located using the defaults specified in the architecture definition. Section `.zbss_non_volatile` should be placed in non–volatile ram. To achieve this, the run address refers to our non–volatile memory called `my_nvsram`. Furthermore, the section should *not* be cleared and therefore the attribute `s` (scratch) is assigned to the group:

```
group ( ordered, run_addr = mem:my_nvsram, attributes = s )
{
    select ".zbss.non_volatile";
}
```

This completes the LSL file for the sample architecture and sample program. You can now call the linker with this file and the sample program to obtain an application that works for this architecture.

For a complete description of the Linker Script Language, refer to Chapter 7, *Linker Script Language*, in the *Reference Guide*.

### 7.6.8 THE PROCESSOR DEFINITION: USING MULTI-PROCESSOR SYSTEMS

The processor definition is only needed when you write an LSL–file for a multi–processor embedded system. In the processor definition you can instantiate one or more cores and couple them to one of the architectures as defined in the architecture definition.

## 7.7 LINKER LABELS

The linker creates labels that you can use to refer to from within the application software. Some of these labels are real labels at the beginning or the end of a section. Other labels have a second function, these labels are used to address generated data in the locating phase. The data is only generated if the label is used.

Linker labels are labels starting with **_lc_**. The linker assigns addresses to the following labels when they are referenced:

| Label | Description |
|---|---|
| `_lc_cp` | Start of copy table. Same as `_lc_ub_table`. The copy table gives the source and destination addresses of sections to be copied. This table will be generated by the linker only if this label is used. |
| `_lc_bh` | Begin of heap. Same as `_lc_ub_heap`. |
| `_lc_eh` | End of heap. Same as `_lc_ue_heap`. |
| `_lc_u_`*`name`* | User defined label. The label must be defined in the LSL file. For example,<br>`"_lc_u_int_tab" = (INTTAB);` |
| `_lc_ub_`*`name`*<br>`_lc_b_`*`name`* | Begin of section *name*. Also used to mark the begin of the stack or heap or copy table. |
| `_lc_ue_`*`name`*<br>`_lc_e_`*`name`* | End of section *name*. Also used to mark the begin of the stack or heap. |
| `_lc_cb_`*`name`* | Start address of an overlay section in ROM. |
| `_lc_ce_`*`name`* | End address of an overlay section in ROM. |
| `_lc_gb_`*`name`* | Begin of group *name*. This label appears in the output file even if no reference to the label exist in the input file. |
| `_lc_ge_`*`name`* | End of group *name*. This label appears in the output file even if no reference to the label exist in the input file. |

*Table 7–5: Linker labels*

The linker only allocates space for the stack and/or heap when a reference to either of the section labels exists in one of the input object files.

***Example***

Suppose in an LSL you have defined a user stack section with the name "`ustack`" (with the keyword **`stack`**). You can refer to the begin and end of the stack from your C source as follows:

```
#include <stdio.h>
extern char *_lc_ub_ustack;
extern char *_lc_ue_ustack;
int main()
{
  printf( "Size of user stack is %d\n",
          _lc_ue_ustack - _lc_ub_ustack );
}
```

From assembly you can refer to the end of the user stack with:

```
.extern _lc_ue_ustack   ; user stack end
```

## 7.8 GENERATING A MAP FILE

The map file is an additional output file that contains information about the location of sections and symbols. With the options in the **Map File** page of the **Linker** entry in the **Project Options** dialog you choose to generate a map file or to skip it (**–M** option). You can also customize the type of information that should be included in the map file (**–m** option).

See section 5.2, *Linker Map File Format*, in Chapter *List File Formats* of the *Reference Guide* for an explanation of the format of the map file.

***Example:***

```
ltc –Mtest.map test.o
```

With this command the list file `test.map` is created.

## 7.9 LINKER ERROR MESSAGES

The linker produces error messages of the following types:

***F Fatal errors***

After a fatal error the linker immediately aborts the link/locate process.

***E Errors***

Errors are reported, but the linker continues linking and locating. No output files are produced unless you have set the linker option **--keep-output-files**.

***W Warnings***

Warning messages do not result into an erroneous output file. They are meant to draw your attention to assumptions of the linker for a situation which may not be correct. You can control warnings in the **Linker | Warnings** page of the **Project | Project Options...** menu (linker option **-w**).

***I Information***

Verbose information messages do not indicate an error but tell something about a process or the state of the linker. To see verbose information, use the linker option **-v**.

***S System errors***

System errors occur when internal consistency checks fail and should never occur. When you still receive the system error message

```
S6##: message
```

please report the error number and as many details as possible about the context in which the error occurred. The following helps you to prepare an e-mail using EDE:

1. From the **Help** menu, select **Technical Support -> Prepare Email...**

   *The Prepare Email form appears.*

2. Fill out the the form. State the error number and attach relevant files.

3. Click the **Copy to Email client** button to open your email application.

   *A prepared e-mail opens in your e-mail application.*

4. Finish the e–mail and send it.

***Display detailed information on diagnostics***

1. In the **Help** menu, enable the option **Show Help on Tool Errors**.
2. In the **Build** tab of the **Output** window, double–click on an error or warning message.

   *A description of the selected message appears.*

**ltc --diag=**[*format*:]{**all** | *number*,...}

See linker option **--diag** in section 4.3, *Linker Options* in Chapter *Tool Options* of the *TriCore Reference Guide*.

CHAPTER 8

# USING THE UTILITIES

TASKING

# 8 CHAPTER

## 8.1 INTRODUCTION

The TASKING toolchain for the TriCore processor family comes with a number of utilities. The utilities are not actually part of the toolchain provide useful extra features. The utilities are only available as command line tools.

**cctc** A control program for the TriCore toolchain. The control program invokes all tools in the toolchain and lets you quickly generate an absolute object file from C source input files.

**mktc** A utility program to maintain, update, and reconstruct groups of programs. The make utility looks whether files are out of date, rebuilds them and determines which other files as a consequence also need to be rebuild.

**artc** An ELF archiver. With this utility you create and maintain object library files.

## 8.2 CONTROL PROGRAM

The control program **cctc** is a tool that invokes all tools in the toolchain for you. It provides a quick and easy way to generate the final absolute object file out of your C sources without the need to invoke the compiler, assembler and linker manually.

### 8.2.1 CALLING THE CONTROL PROGRAM

You can only call the control program from the command line. The invocation syntax is

```
cctc [ [option]... [file]... ]...
```

For example:

```
cctc –v test.c
```

The control program calls all tools in the toolchain and generates the absolute object file `test.elf`. With the control program option **–v** you can see how the control program calls the tools:

```
+ c:\ctc\bin\ctc –o cc7b40ab.src test.c
+ c:\ctc\bin\astc –o test.o cc7b40ab.src
+ c:\ctc\bin\ltc –r test.o –lc –lfp –lrt
–Lc:\ctc\lib\tc1 –occ7b40af.out
+ c:\ctc\bin\ltc –FELF –M –dtc.lsl cc7b40af.out
–otest.elf
```

The control program produces unique filenames for intermediate steps in the compilation process (such as `cc7b40ab.src` in the example above) which are removed afterwards. The control program prevents preprocessing of the compiler generated assembly file because the compiler does not generate any preprocessor symbols. Normally, (hand–written) assembly input files are preprocessed first.

#### *Recognized input files*

The control program recognizes the following input files:

- Files with a `.cc`, `.cxx` or `.cpp` suffix are interpreted as C++ source programs and are passed to the C++ compiler.
- Files with a `.c` suffix are interpreted as C source programs and are passed to the compiler.

- Files with a `.asm` suffix are interpreted as hand–written assembly source files which have to be passed to the assembler.
- Files with a `.src` suffix are interpreted as compiled assembly source files. They are directly passed to the assembler.
- Files with a `.a` or `.elb` suffix are interpreted as library files and are passed to the linker.
- Files with a `.o` suffix are interpreted as object files and are passed to the linker.
- Files with a `.out` suffix are interpreted as linked object files and are passed to the locating phase of the linker. The linker accepts only one `.out` file in the invocation.
- An argument with a `.lsl` suffix is interpreted as a linker script file and is passed to the linker.

### *Options of the control program*

The following control program options are available:

| Description | Option |
|---|---|
| Display options | **–?** |
| Display version header | **–V** |
| Verbose option: show commands invoked | **–v** |
| Verbose option: show commands without executing | **–v0** |
| **Input files** | |
| Force C files to C++ mode | **–c++** |
| Force C++ files to C mode | **–noc++** |
| Read options from file | **–f** *file* |
| **Process** | |
| Stop after C++ files are compiled to intermediate C (.ic) | **–cc** |
| Stop after C++ files or C files are compiled to assembly (.src) | **–cs** |
| Stop after the files are assembled to objects (.o) | **–c** |
| Stop after the files are linked to a linker object file (.out) | **–cl** |
| Force invocation of the C++ muncher | **–cm** |
| Force invocation of the C++ linker | **–cp** |
| **Libraries** | |
| Do not link with the standard libraries | **–nolib** |
| Use floating–point library with support for trapping | **–fptrap** |

| Description | Option |
|---|---|
| **Code generation** | |
| Select CPU type | **–C***cpu* |
| Generate symbolic debug information | **–g** |
| CPU functional problem bypasses | **––silicon–bug=***bug,...* |
| **Output files** | |
| Do not generate a linker map file | **–nomap** |
| Specify name for the output file | **–o** *file* |
| Keep temporary files | **–tmp** |
| Enable C and assembler warnings for C++ files | **–wc++** |
| Produce an ELF/DWARF object file | **–elf** |
| Produce an IEEE–695 object file | **–ieee** |
| Produce an Intel Hex object file | **–ihex** |
| Produce an Motorola S–record object file | **–srec** |

*Table 8–1: Overview of control program options*

For a complete list and description of all control program options, see section 4.4, *Control Program Options*, in Chapter *Tool Options* of the *Reference Guide*.

The options in table 8–1 are options that the control program interprets itself. The control program however can also pass an option directly to a tool. Such an option is not interpreted by the control program but by the tool itself. The next example illustrates how an option is passed directly to the linker to link a user defined library:

```
cctc -Wlk-lmylib test.c
```

Use the following options to pass arguments to the various tools:

| Description | Option |
|---|---|
| Pass argument directly to the C++ compiler | **–Wcp***arg* |
| Pass argument directly to the C++ pre–linker | **–Wpl***arg* |
| Pass argument directly to the C compiler | **–Wc***arg* |
| Pass argument directly to the assembler | **–Wa***arg* |
| Pass argument directly to the second assembler | **–Wpcp***arg* |
| Pass argument directly to the linker | **–Wlk***arg* |
| Pass argument directly to the locating phase of the linker | **–Wlc***arg* |

*Table 8–2: Control program options to pass an option directly to a tool*

If you specify an unknown option to the control program, the control program looks if it is an option for a specific tool. If so, it passes the option directly to the tool. However, it is recommended to use the control program options to passing arguments directly to tools.

With the environment variable CCTCOPT you can define options and/or arguments that the control programs always processes *before* the command line arguments.

For example, if you use the control program always with the option **–nomap** (do not produce a linker map file), you can specify "–nomap" to the environment variable CCTCOPT.

See section 1.3.2, *Configuring the Command Line Environment*, in Chapter *Software Installation*.

## 8.3 MAKE UTILITY

If you are working with large quantities of files, or if you need to build several targets, it is rather time–consuming to call the individual tools to compile, assemble, link and locate all your files.

You save already a lot of typing if you use the control program **cctc** and define an options file. You can even create a batch file or script that invokes the control program for each target you want to create. But with these methods *all* files are completely compiled, assembled, linked and located to obtain the target file, even if you changed just one C source. This may demand a lot of (CPU) time on your host.

The make utility **mktc** is a tool to maintain, update, and reconstruct groups of programs. The make utility looks which files are out–of–date and only recreates these files to obtain the updated target.

### *Make process*

In order to build a target, the make utility needs the following input:

- the target it should build, specified as argument on the command line
- the rules to build the target, stored in a file usually called `makefile`

In addition, the make utility also reads the file `mktc.mk` which contains predefined rules and macros. See section 8.3.2, *Writing a Makefile*.

The makefile contains the relationships among your files (called *dependencies*) and the commands that are necessary to create each of the files (called *rules*). Typically, the absolute object file (`.elf`) is updated by locating the `.out` file, which depends on `.o` files and libraries that must be linked together. The `.o` files on their turn depend on `.src` files that must be assembled and finally, `.src` files depend on the C source files (`.c`) that must be compiled. In the makefile `makefile` this looks like:

```
test.src  : test.c                    # dependency
            ctc test.c                # rule

test.o    : test.src
            astc test.src

test.out  : test.o
            ltc test.o –lc –lfp –lrt –otest.out

test.elf  : test.out
            ltc test.out –otest.elf
```

You can use any command that is valid on the command line as a rule in the `makefile`. So, rules are not restricted to invocation of the toolchain.

***Example***

To build the target `test.elf`, call **mktc** with one of the following lines:

```
mktc test.elf

mktc –f mymake.mak test.elf
```

Default the make utility reads `makefile` so you do not need to specify it on the command line. If you want to use another name for the makefile, use the option **–f** *my_makefile*.

If you do not specify a target, **mktc** uses the first target defined in the makefile. In this example it would build `test.src` instead of `test.elf`.

The make utility now tries to build `test.elf` based on the `makefile` and peforms the following steps:

1. From the makefile the make utility reads that `test.elf` depends on `test.out`.
2. If `test.out` does not exist or is out–of–date, the make utility first tries to build this file and reads from the makefile `test.out` depends on `test.o`.
3. If `test.o` does exist, the make utility now creates `test.out` by executing the rule for it: `ltc test.o –lc –lfp –lrt –otest.out`.
4. There are no other files necessary to create `test.elf` so the make utility now can use `test.out` to create `test.elf` by executing the rule `ltc test.out –otest.elf`.

The make utility has now build `test.elf` but it only used the linker to update `test.out` and the linker to create `test.elf`.

If you compare this to the control program:

```
cctc test.c
```

This invocation has the same effect but now *all* files are recompiled (assembled, linked and located).

### 8.3.1 CALLING THE MAKE UTILITY

You can only call the make utility from the command line. The invocation syntax is

```
mktc [ [options] [targets] [macro=def]... ]
```

For example:

```
mktc test.elf
```

*target* — You can specify any target that is defined in the makefile. A target can also be one of the intermediate files specified in the makefile.

*macro=def* — Macro definition. This definition remains fixed for the **mktc** invocation. It overrides any regular definitions for the specified macro within the makefiles and from the environment. It is inherited by subordinate **mktc**'s but act as an environment variable for these. That is, depending on the **–e** setting, it may be overridden by a makefile definition.

#### *Options of the make utility*

The following make utility options are available:

| Description | Option |
|---|---|
| Display options | **–?** |
| Display version header | **–V** |
| **Verbose** | |
| Print makefile lines while being read | **–D/–DD** |
| Display time comparisons which indicate a target is out of date | **–d/–dd** |
| Display current date and time | **–time** |
| Verbose option: show commands without executing (dry run) | **–n** |
| Do not show commands before execution | **–s** |
| Do not build, only indicate whether target is up–to–date | **–q** |
| **Input files** | |
| Use *makefile* instead of the standard makefile `makefile` | **–f** *makefile* |
| Change to directory before reading the makefile | **–G** *path* |
| Read options from file | **–m** *file* |
| Do not read the `mktc.mk` file | **–r** |

| Description | Option |
|---|---|
| **Process** | |
| Always rebuild target without checking whether it is out–of–date | **–a** |
| Run as a child process | **–c** |
| Environment variables override macro definitions | **–e** |
| Do not remove temporary files | **–K** |
| On error, only stop rebuilding current target | **–k** |
| Overrule the option **–k** (only stop rebuilding current target) | **–S** |
| Make all target files precious | **–p** |
| Touch the target files instead of rebuilding them | **–t** |
| Treat target as if it has just been reconstructed | **–W** *target* |
| **Error messages** | |
| Redirect error messages and verbose messages to a file | **–err** *file* |
| Ignore error codes returned by commands | **–i** |
| Redirect messages to standard out instead of standard error | **–w** |
| Show extended error messages | **–x** |

*Table 8–3: Overview of control program options*

For a complete list and description of all control program options, see section 4.5, *Make Utility Options*, in Chapter *Tool Options* of the *Reference Guide*.

### 8.3.2 WRITING A MAKEFILE

In addition to the standard makefile `makefile`, the make utility always reads the makefile `mktc.mk` before other inputs. This system makefile contains implicit rules and predefined macros that you can use in the makefile `makefile`.

With the option **`-r`** (Do not read the `mktc.mk` file) you can prevent the make utility from reading `mktc.mk`.

The default name of the makefile is `makefile` in the current directory. If on a UNIX system this file is not found, the file `Makefile` is used as the default. If you want to use other makefiles, use the option **–f** *my_makefile*.

The makefile can contain a mixture of:

- targets and dependencies
- rules
- macro definitions or functions

- comment lines
- include lines
- export lines

To continue a line on the next line, terminate it with a backslash (\):

```
# this comment line is continued\
on the next line
```

If a line must end with a backslash, add an empty macro.

```
# this comment line ends with a backslash \$(EMPTY)
# this is a new line
```

### *Targets and dependencies*

The basis of the makefile is a set of targets, dependencies and rules. A target entry in the makefile has the following format:

```
target ... : [dependency ...] [; rule]
       [rule]
        ...
```

Target lines must always start at the beginning of a line, leading white spaces (tabs or spaces) are not allowed. A target line consists of one or more targets, a semicolon and a set of files which are required to build the target (*dependencies*). The target itself can be one or more filenames or symbolic names.:

```
all:                    demo.elf final.elf

demo.elf final.elf:     test.out demo.out final.out
```

You can now can specify the target you want to build to the make utility. The following three invocations all have the same effect:

```
mktc
mktc all
mktc demo.elf final.elf
```

If you do *not* specify a target, the first target in the makefile (in this example `all`) is build. The target `all` depends on `demo.elf` and `final.elf` so the second and third invocation have also the same effect and the files `demo.elf` and `final.elf` are built.

In MS–DOS you can normally use colons to denote drive letters. The following works as intended: `c:foo.o : a:foo.c`

If a target is defined in more than one target line, the dependencies are added to form the target's complete dependency list:

```
all:  demo.elf   # These two lines are equivalent with:
all:  final.elf  # all: demo.elf final.elf
```

For target lines, macros and functions are expanded at the moment they are read by the make utility. Normally macros are not expanded until the moment they are actually used.

### *Special Targets*

There are a number of special targets. Their names begin with a period.

.DEFAULT: If you call the make utility with a target that has no definition in the make file, this target is build.

.DONE: When the make utility has finished building the specified targets, it continues with the rules following this target.

.IGNORE: Non–zero error codes returned from commands are ignored. Encountering this in a makefile is the same as specifying the option **–i** on the command line.

.INIT: The rules following this target are executed before any other targets are build.

.SILENT: Commands are not echoed before executing them. Encountering this in a makefile is the same as specifying the option **–s** on the command line.

.SUFFIXES: This target specifies a list of file extensions. Instead of building a completely specified target, you now can build a target that has a certain file extension. Implicit rules to build files with a number of extensions are included in the system makefile `mktc.mk`.

If you specify this target with dependencies, these are added to the existing `.SUFFIXES` target in `mktc.mk`. If you specify this target without dependencies, the existing list is cleared.

.PRECIOUS: Dependency files mentioned for this target are never removed. Normally, if a command in a rule returns an error or when the target construction is interrupted, the make utility removes that target file. You can use the **–p** command line option to make all target files precious.

### *Rules*

A line with leading white space (tabs or spaces) is considered as a rule and associated with the most recently preceding dependency line. A *rule* is a line with commands that are executed to build the associated target. A target–dependency line can be followed by one or more rules.

```
final.src : final.c            # target and dependency
            mv test.c final.c  # rule1
            ctc final.c        # rule2
```

You can precede a rule with one or more of the following characters:

@ does not echo the command line, except if **–n** is used.

– the make utility ignores the exit code of the command (ERRORLEVEL in MS–DOS). Normally the make utility stops if a non–zero exit code is returned. This is the same as specifying the option **–i** on the command line or specifying the special `.IGNORE` target.

\+ The make utility uses a shell or COMMAND.COM to execute the command. If the '+' is not followed by a shell line, but the command is a DOS command or if redirection is used (<, |, >), the shell line is passed to COMMAND.COM anyway. For UNIX, redirection, backquote (`) parentheses and variables force the use of a shell.

You can force **mktc** to execute multiple command lines in one shell environment. This is accomplished with the token combination ';\'. For example:

```
cd c:\ctc\bin ;\
cctc –V
```

The ';' must always directly be followed by the '\' token. Whitespace is not removed when it is at the end of the previous command line or when it is in front of the next command line. The use of the ';' as an operator for a command (like a semicolon ';' separated list with each item on one line) and the '\' as a layout tool is not supported, unless they are separated with whitespace.

The make utility can generate inline temporary files. If a line contains `<<`*`LABEL`* (no whitespaces!) then all subsequent lines are placed in a temporary file until the line `LABEL` is encountered. Next, `<<`*`LABEL`* is replaced by the name of the temporary file.

Example:

```
ltc –o $@ –f <<EOF
     $(separate "\n" $(match .o $!))
     $(separate "\n" $(match .a $!))
     $(LKFLAGS)
EOF
```

The three lines between `<<EOF` and `EOF` are written to a temporary file (for example `mkce4c0a.tmp`), and the rule is rewritten as `ltc –o $@ –f mkce4c0a.tmp`.

Instead of specifying a specific target, you can also define a general target. A general target specifies the rules to generate a file with extension *`.ex1`* to a file with extension *`.ex2`*. For example:

```
.SUFFIXES:  .c
.c.src   :
            ltc $<
```

Read this as: to build a file with extension `.src` out of a file with extension `.c`, call the compiler with `$<`. `$<` is a predefined macro that is replaced with the basename of the specified file. The special target `.SUFFIXES:` is followed by a list of file extensions of the files that are required to build the target.

### *Implicit Rules*

Implicit rules are stored in the system makefile `mktc.mk` and are intimately tied to the `.SUFFIXES` special target. Each dependency that follows the `.SUFFIXES` target, defines an extension to a filename which must be used to build another file. The implicit rules then define how to actually build one file from another. These files share a common basename, but have different extensions.

If the specified target on the command line is not defined in the makefile or has not rules in the makefile, the make utility looks if there is an implicit rule to build the target.

### Example

This makefile says that `prog.out` depends on two files `prog.o` and `sub.o`, and that they in turn depend on their corresponding source files (`prog.c` and `sub.c`) along with the common file `inc.h`.

```
LIB  =     –lc                                          # macro

prog.out:  prog.o sub.o
     ltc prog.o sub.o $(LIB) –o prog.out

prog.o:  prog.c inc.h
     ctc  prog.c
     astc prog.src

sub.o:  sub.c inc.h
     ctc  sub.c
     astc sub.src
```

The following makefile uses implicit rules (from `mktc.mk`) to perform the same job.

```
LKFLAGS = –lc               # macro used by implicit rules
prog.out: prog.o sub.o      # implicit rule used
prog.o: prog.c inc.h        # implicit rule used
sub.o:  sub.c inc.h         # implicit rule used
```

### Files

| | |
|---|---|
| makefile | Description of dependencies and rules. |
| Makefile | Alternative to makefile, for UNIX. |
| mktc.mk | Default dependencies and rules. |

### Diagnostics

**mktc** returns an exit status of 1 when it halts as a result of an error. Otherwise it returns an exit status of 0.

#### *Macro definitions*

A macros is a symbol names that is replaced with it's definition before the makefile is executed. Although the macro name can consist of lower case or upper case characters, upper case is an accepted convention. The general form of a macro definition is:

```
MACRO = text and more text
```

Spaces around the equal sign are not significant. To use a macro, you must access it's contents:

```
$(MACRO)        # you can read this as
${MACRO}        # the contents of macro MACRO
```

If the macro name is a single character, the parentheses are optional. Note that the expansion is done recursively, so the body of a macro may contain other macros. These macros are expanded when the macro is actually used, not at the point of definition:

```
FOOD = $(EAT) and $(DRINK)
EAT = meat and/or vegetables
DRINK = water
export FOOD
```

The macro `FOOD` is expanded as `meat and/or vegetables and water` at the moment it is used in the export line.

### *Predefined Macros*

MAKE — Holds the value `mktc`. Any line which uses `MAKE`, temporarily overrides the option **–n** (Show commands without executing), just for the duration of the one line. This way you can test nested calls to `MAKE` with the option **–n**.

MAKEFLAGS — Holds the set of options provided to **mktc** (except for the options **–f** and **–d**). If this macro is exported to set the environment variable `MAKEFLAGS`, the set of options is processed before any command line options. You can pass this macro explicitly to nested **mktc**'s, but it is also available to these invocations as an environment variable.

PRODDIR — Holds the name of the directory where **mktc** is installed. You can use this macro to refer to files belonging to the product, for example a library source file.

```
DOPRINT = $(PRODDIR)/lib/src/_doprint.c
```

When **mktc** is installed in the directory `/ctc/bin` this line expands to:

```
DOPRINT = /ctc/lib/src/_doprint.c
```

SHELLCMD — Holds the default list of commands which are local to the SHELL. If a rule is an invocation of one of these commands, a SHELL is automatically spawned to handle it.

TMP_CCPROG

Holds the name of the control program: `cctc`. If this macro and the TMP_CCOPT macro are set and the command line argument list for the control program exceeds 127 characters, then **mktc** creates a temporary file with the command line arguments. **mktc** calls the control program with the temporary file as command input file.

TMP_CCOPT

Holds **–f**, the control program option that tells it to read options from a file. (This macro is only available for the Windows command prompt version of **mktc**.)

$ This macro translates to a dollar sign. Thus you can use "$$" in the makefile to represent a single "$".

There are several dynamically maintained macros that are useful as abbreviations within rules. It is best not to define them explicitly.

$* The basename of the current target.

$< The name of the current dependency file.

$@ The name of the current target.

$? The names of dependents which are younger than the target.

$! The names of all dependents.

The $< and $* macros are normally used for implicit rules. They may be unreliable when used within explicit target command lines. All macros may be suffixed with F to specify the Filename components (e.g. ${*F}, ${@F}). Likewise, the macros $*, $< and $@ may be suffixed by D to specify the directory component.

The result of the $* macro is always without double quotes ("), regardless of the original target having double quotes (") around it or not.
The result of using the suffix F (Filename component) or D (Directory component) is also always without double quotes ("), regardless of the original contents having double quotes (") around it or not.

### *Functions*

A function not only expands but also performs a certain operation. Functions syntactically look like macros but have embedded spaces in the macro name, e.g. '$(match arg1 arg2 arg3 )'. All functions are built–in and currently there are five of them: `match`, `separate`, `protect`, `exist` and `nexist`.

`match` The `match` function yields all arguments which match a certain suffix:

```
$(match .o prog.o sub.o mylib.a)
```

yields:

```
prog.o sub.o
```

`separate` The `separate` function concatenates its arguments using the first argument as the separator. If the first argument is enclosed in double quotes then '\n' is interpreted as a newline character, '\t' is interpreted as a tab, '*ooo*' is interpreted as an octal value (where, *ooo* is one to three octal digits), and spaces are taken literally. For example:

```
$(separate "\n" prog.o sub.o)
```

results in:

```
prog.o
sub.o
```

Function arguments may be macros or functions themselves. So,

```
$(separate "\n" $(match .o $!))
```

yields all object files the current target depends on, separated by a newline string.

`protect` The `protect` function adds one level of quoting. This function has one argument which can contain white space. If the argument contains any white space, single quotes, double quotes, or backslashes, it is enclosed in double quotes. In addition, any double quote or backslash is escaped with a backslash.

Example:

```
echo $(protect I'll show you the "protect"
function)
```

yields:

```
echo "I'll show you the \"protect\"
function"
```

`exist` The `exist` function expands to its second argument if the first argument is an existing file or directory.

Example:

```
$(exist test.c cctc test.c)
```

When the file `test.c` exists, it yields:

```
cctc test.c
```

When the file `test.c` does not exist nothing is expanded.

`nexist` The `nexist` function is the opposite of the exist function. It expands to its second argument if the first argument is not an existing file or directory.

Example:

```
$(nexist test.src cctc test.c)
```

### *Conditional Processing*

Lines containing `ifdef`, `ifndef`, `else` or `endif` are used for conditional processing of the makefile. They are used in the following way:

**`ifdef`** *macro–name*
*if–lines*
**`else`**
*else–lines*
**`endif`**

The *if–lines* and *else–lines* may contain any number of lines or text of any kind, even other `ifdef`, `ifndef`, `else` and `endif` lines, or no lines at all. The `else` line may be omitted, along with the *else–lines* following it.

First the *macro–name* after the `if` command is checked for definition. If the macro is defined then the *if–lines* are interpreted and the *else–lines* are discarded (if present). Otherwise the *if–lines* are discarded; and if there is an `else` line, the *else–lines* are interpreted; but if there is no `else` line, then no lines are interpreted.

When using the `ifndef` line instead of `ifdef`, the macro is tested for not being defined. These conditional lines can be nested up to 6 levels deep.

See also *Defining Macros* in section 4.5, *Make Utility Options*, in Chapter *Tools Options* of the *Reference Guide*.

### *Comment lines*

Anything after a "#" is considered as a comment, and is ignored. If the "#" is inside a quoted string, it is not treated as a comment. Completely blank lines are ignored.

```
test.src  : test.c      # this is comment and is
            ctc test.c  # ignored by the make utility
```

### *Include lines*

An *include* line is used to include the text of another makefile (like including a .h file in a C source). Macros in the name of the included file are expanded before the file is included. Include files may be nested.

```
include makefile2
```

### *Export lines*

An *export* line is used to export a macro definition to the environment of any command executed by the make utility.

```
GREETING = Hello

export GREETING
```

This example creates the environment variable `GREETING` with the value `Hello`. The macros is exported at the moment the export line is read so the macro definition has to proceed the export line.

## 8.4 ARCHIVER

The archiver **artc** is a program to build and maintain your own library files. A library file is a file with extension `.a` and contains one or more object files (`.o`) that may be used by the linker.

The archiver has five main functionalities:

- Deleting an object module from the library
- Moving an object module to another position in the library file
- Replacing an object module in the library or add a new object module
- Showing a table of contents of the library file
- Extracting an object module from the library

The archiver takes the following files for input and output:

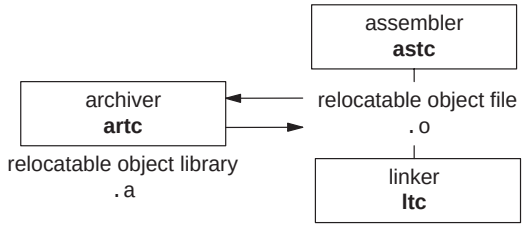



*Figure 8–1: **artc** ELF/DWARF archiver and library maintainer*

The linker optionally includes object modules from a library if that module resolves an external symbol definition in one of the modules that are read before.

### 8.4.1 CALLING THE ARCHIVER

You can only call the archiver from the command line. The invocation syntax is:

```
artc key_option [sub_option...] library [object_file]
```

*key_option* With a key option you specify the main task which the archiver should perform. You must *always* specify a key option.

*sub_option* Sub–options specify into more detail how the archiver should perform the task that is specified with the key option. It is not obligatory to specify sub–options.

*library* The name of the library file on which the archiver performs the specified action. You must always specify a library name, except for the option **–?** and **–V**. When the library is not in the current directory, specify the complete path (either absolute or relative) to the library.

*object_file* The name of an object file. You must always specify an object file name when you add, extract, replace or remove an object file from the library.

### *Options of the archiver utility*

The following archiver options are available:

| Description | Option | Sub–option |
|---|---|---|
| Display options | **–?** | |
| Display version header | **–V** | |
| Print object module to standard output | **–p** | |
| **Main functions** | | |
| Delete object module from library | **–d** | **–v** |
| Move object module to another position | **–m** | **–a –b –v** |
| Replace or add an object module | **–r** | **–a –b –c –u –v** |
| Print a table of contents of the library | **–t** | **–s0 –s1** |
| Extract an object module from the library | **–x** | **–v** |

*Table 8–4: Overview of archiver options and sub–options*

For a complete list and description of all archiver options, see section 4.6, *Archiver Options*, in Chapter *Tool Options* of the *Reference Guide*.

### 8.4.2 EXAMPLES

***Create a new library***

If you add modules to a library that does not yet exist, the library is created. To create a new library with the name `mylib.a` and add the object modules `cstart.o` and `calc.o` to it:

```
artc –r mylib.a cstart.o calc.o
```

***Add a new module to an existing library***

If you add a new module to an existing library, the module is added at the end of the module. (If the module already exists in the library, it is replaced.)

```
artc –r mylib.a mod3.o
```

***Print a list of object modules in the library***

To inspect the contents of the library:

```
artc –t mylib.a
```

The library has the following contents:

```
cstart.o
calc.o
mod3.o
```

***Move an object module to another position***

To move `mod3.o` to the beginning of the library, position it just before `cstart.o`:

```
artc –mb cstart.o mylib.a mod3.o
```

***Delete an object module from the library***

To delete the object module `cstart.eln` from the library `mylib.a`:

```
artc –d mylib.a cstart.o
```

***Extract all modules from the library***

Extract all modules from the library `mylib.a`:

```
artc –x mylib.a
```

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

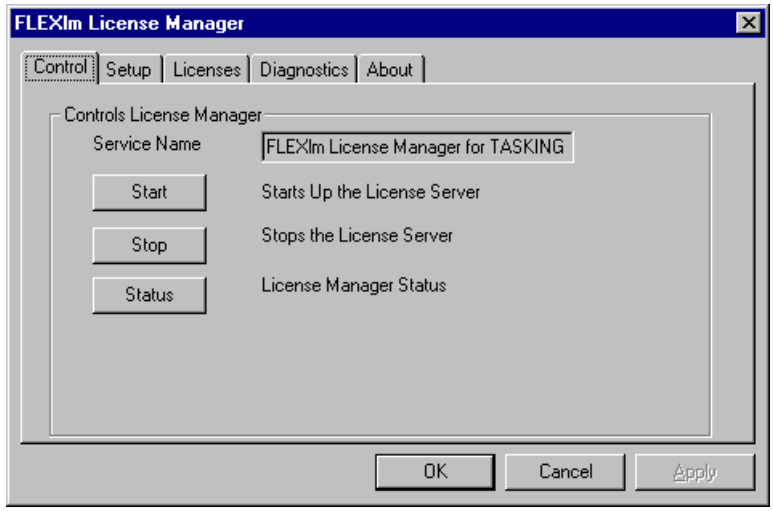
### 3.11.1 LMTOOLS FOR WINDOWS

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.

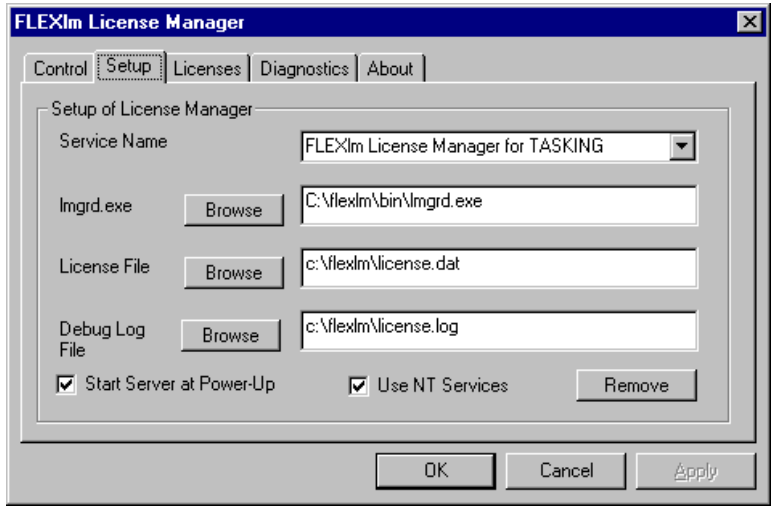
The `HOSTID` button displays the hostid's for the computer on which the program is running. The `TIME` button prints out the system's internal time settings, intended to diagnose any time zone problems. The `TCP Settings` button is intended to fix a bug in the Microsoft TCP protocol stack which has a 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 HP–UX or SunOS 5.x (Solaris 2.x):

```
ps -ef | grep lmgrd | grep -v grep
```

If the command does not produce any output then the license server daemon is not running. See below for an example how to start **lmgrd**.

Make sure that both license server daemon (**lmgrd**) and the program are using the same license data. All TASKING products use the license file `/usr/local/flexlm/licenses/license.dat` unless overruled by the environment variable `LM_LICENSE_FILE`. However, not all existing **lmgrd** daemons may use the same default. In case of doubt, specify the license file pathname with the **-c** option when starting the license server daemon. For example:

```
lmgrd -c /usr/local/flexlm/licenses/license.dat \
      -l /usr/local/flexlm/licenses/license.log &
```

and set the `LM_LICENSE_FILE` environment variable to the `license.dat` pathname mentioned with the **-c** option of **lmgrd** before running any license based program (including **lmreread**, **lmstat**, **lmdown**). If **lmgrd** and the program run on different hosts, transparent access to the license file is assumed in the situation described above (e.g. NFS). If this is not the case, make a local copy of the license file (not recommended) or set `LM_LICENSE_FILE` to the form *number@host*, as described earlier.

If none of the above seems to apply (i.e. **lmgrd** was already running and `LM_LICENSE_FILE` has been set correctly) then it is very likely that there is a TCP port mismatch. The fourth field of a SERVER line in the license file specifies a TCP port number. That number can be changed without affecting any license. However, it must never be changed while the license server daemon is running. If it has been changed, change it back to the original value. If you do not know the original number anymore, restart the license server daemon after typing the following command on the license server host:

```
kill PID
```

where `PID` is the process id of **lmgrd**.

# 6 FREQUENTLY ASKED QUESTIONS (FAQS)

## 6.1 LICENSE FILE QUESTIONS

***I've received FLEXlm license files from 2 different companies. Do I have to combine them?***

You don't have to combine license files. Each license file that has any 'counted' lines (the 'number of licenses' field is >0) requires a server. It's perfectly OK to have any number of separate license files, with different **lmgrd** server processes supporting each file. Moreover, since **lmgrd** is a lightweight process, for sites without system administrators, this is often the simplest (and therefore recommended) way to proceed. With v6+ **lmgrd**/**lmdown**/**lmreread**, you can stop/reread/restart a single vendor daemon (of any FLEXlm version). This makes combining licenses more attractive than previously. Also, if the application is v6+, using 'dir/*.lic' for license file management behaves like combining licenses without physically combining them.

***When is it recommended to combine license files?***

Many system administrators, especially for larger sites, prefer to combine license files to ease administration of FLEXlm licenses. It's purely a matter of preference.

***Does FLEXlm handle dates in the year 2000 and beyond?***

Yes. The FLEXlm date format uses a 4–digit year. Dates in the 20th century (19xx) can be abbreviated to the last 2 digits of the year (xx), and use of this feature is quite widespread. Dates in the year 2000 and beyond must specify all 4 year digits.

## 6.2 FLEXLM VERSION

***Which FLEXlm versions does TASKING deliver?***

For Windows we deliver FLEXlm v6.1 and for UNIX we deliver v2.4.

***I have products from several companies at various FLEXlm version levels. Do I have to worry about how these versions work together?***

If you're not combining license files from different vendors, the simplest thing to do is make sure you use the tools (especially **lmgrd**) that are shipped by each vendor.

**lmgrd** will always correctly support older versions of vendor daemons and applications, so it's **always** safe to use the latest version of **lmgrd** and the other FLEXlm utilities. If you've combined license files from 2 vendors, you **must** use the latest version of lmgrd.

If you've received 2 versions of a product from the same vendor, you must use the latest vendor daemon they sent you. An older vendor daemon with a newer client will cause communication errors.

Please ignore letters appended to FLEXlm versions, i.e., v2.4d. The appended letter indicates a patch, and does NOT indicate any compatibility differences. In particular, some elements of FLEXlm didn't require certain patches, so a 2.4 **lmgrd** will work successfully with a 2.4b vendor daemon.

***I've received a new copy of a product from a vendor, and it uses a new version of FLEXlm. Is my old license file still valid?***

Yes. Older FLEXlm license files are always valid with newer versions of FLEXlm.

## 6.3 WINDOWS QUESTIONS

***What Windows Host Platforms can be used as a server for Floating Licenses?***

The system being used as the server (where the FLEXlm License Manager is running) for Floating licenses, must be Windows NT. The FLEXlm License Manager does not run properly with Windows 95/98.

***Why do I need to include NWlink IPX/SPX on NT?***

This is necessary for either obtaining the Ethernet card address, or to provide connectivity with a Netware License server.

## 6.4 TASKING QUESTIONS

***How will the TASKING licensing/pricing model change with License Management (FLEXlm)?***

TASKING will now offer the following types of licenses so you can purchase licenses based upon usage:

| License | Description | Pricing |
|---|---|---|
| Node Locked | This license can only be used on a specific system. It cannot be moved to another system. | The pricing for this license will be the current product pricing. |
| Floating | This license requires a network (license server and a TCP/IP (or IPX/SPX) connection between clients and server) and can be used on any host system **(using the same operating system)** in the network. | The pricing for this license will be 50% higher than the node locked license. |

***How does FLEXlm affect future product ordering?***

For all licenses, node locked or floating, you must provide information that is used to create a license key. For node locked licenses we must have the HOST ID. Floating licenses require the HOST ID and HOST NAME. The HOST ID is a unique identification of the machine, which is based upon different hardware depending upon host platform. The HOST NAME is the network name of the machine.

TASKING Logistics CANNOT ship ANY orders that do not include the HOST ID and/or HOST NAME information.

***What if I do not know the information needed for the license key?***

We have a software utility (**tkhostid.exe**) which will obtain and display the HOST ID so a customer can easily obtain this information. This utility is available from our web site, placed on all product CDs (which support FLEXlm), and from technical support. If you have already installed FLEXlm, you can also use **lmhostid**.

- In the case of a *Node locked license*, it is important that the customer runs this utility on the exact machine he intends to run the TASKING tools on.

- In the case of a *Floating License*, the **tkhostid.exe** (or **lmhostid**) utility should be run on the machine on which the FLEXlm license manager will be installed, e.g. the server. The HOST NAME information can be obtained from within the Windows Control Panel. Select "Network", click on "Identification", look for "Computer name".

***How will the "locking" mechanism work?***

- For node locked licenses, FLEXlm will first search for an ethernet card. If one exists, it will lock onto the number of the ethernet card. If an ethernet card does not exist, FLEXlm will lock onto the hard disk serial number.
- For floating licenses, the ethernet card number will be used.

***What happens if I try to move my node locked license to another system?***

The software will not run.

***What does linger–time for floating licenses mean?***

When the TASKING product starts to run, it will try to obtain a license from the license server. The license server keeps track of the number of licenses already issued, and grants or denies the request. When the software has finished running, the license is kept by the license server for a period of time known as the "linger–time". If the same user requests the TASKING product again within the linger–time, he is granted the license again. If another user requests a license during the linger–time, his request is denied until the linger–time has finished

***What is the length of the linger–time for floating licenses?***

The length of the linger–time for both the PC and UNIX floating licenses is 5 minutes.

***Can the linger–time be changed?***

Yes. A customer can change the linger–time to be larger (but not shorter) than the time specified by TASKING.

***What happens if my system crashes or I upgrade to a new system?***

You will need to contact Technical Support for temporary license keys due to a system crash or to move from one system to another system. You will then need to work with your local sales representative to obtain a permanent new license key.

## 6.5 USING FLEXLM FOR FLOATING LICENSES

***Does FLEXlm work across the internet?***

Yes. A server on the internet will serve licenses to anyone else on the internet. This can be limited with the 'INTERNET=' attribute on the FEATURE line, which limits access to a range of internet addresses. You can also use the INCLUDE and EXCLUDE options in the daemon option file to allow (or deny) access to clients running on a range of internet addresses.

***Does FLEXlm work with Internet firewalls?***

Many firewalls require that port numbers be specified to the firewall. FLEXlm v5 **lmgrd** supports this.

***If my client dies, does the server free the license?***

Yes, unless the client's whole system crashes. Assuming communications is TCP, the license is automatically freed immediately. If communications are UDP, then the license is freed after the UDP timeout, which is set by each vendor, but defaults to 45 minutes. UDP communications is normally only set by the end–user, so TCP should be assumed. If the whole system crashes, then the license is not freed, and you should use '**lmremove**' to free the license.

***What happens when the license server dies?***

FLEXlm applications send periodic heartbeats to the server to discover if it has died. What happens when the server dies is then up to the application. Some will simply continue periodically attempting to re–checkout the license when the server comes back up. Some will attempt to re–checkout a license a few times, and then, presumably with some warning, exit. Some GUI applications will present pop–ups to the user periodically letting them know the server is down and needs to be re–started.

***How do you tell if a port is already in use?***

99.44% of the time, if it's in use, it's because **lmgrd** is already running on the port – or was recently killed, and the port isn't freed yet. Assuming this is not the case, then use '**telnet** *host port*' – if it says *"can't connect"*, it's a free port.

***Does FLEXlm require root permissions?***

No. There is no part of FLEXlm, **lmgrd**, vendor daemon or application, that requires root permissions. In fact, it is strongly recommended that you do not run the license server (**lmgrd**) as root, since root processes can introduce security risks.
If **lmgrd** must be started from the root user (for example, in a system boot script), we recommend that you use the '**su**' command to run **lmgrd** as a non–privileged user:

```
su username –c"/path/lmgrd –c /path/license.dat \
            –l /path/log"
```

where *username* is a non–privileged user, and *path* is the correct paths to **lmgrd**, `license.dat` and debug log file. You will have to ensure that the vendor daemons listed in */path–to–license*/license.dat have execute permissions for username. The paths to all the vendor daemons in the license file are listed on each DAEMON line.

***Is it ok to run lmgrd as 'root' (UNIX only)?***

It is not prudent to run any command, particularly a daemon, as root on UNIX, as it may pose a security risk to the Operating System. Therefore, we recommend that **lmgrd** be run as a non–privileged user (not 'root'). If you are starting **lmgrd** from a boot script, we recommend that you use

```
su username –c"umask 022; /path/lmgrd \
            –c /path/license.dat –l /path/log"
```

to run **lmgrd** as a non–privileged user.

***Does FLEXlm licensing impose a heavy load on the network?***

No, but partly this depends on the application, and end–user's use. A typical checkout request requires 5 messages and responses between client and server, and each message is < 150 bytes.

When a server is not receiving requests, it requires virtually no CPU time. When an application, or **lmstat**, requests the list of current users, this can significantly increase the amount of networking FLEXlm uses, depending on the number of current users. Also, prior to FLEXlm v5, use of 'port@host' can increase network load, since the license file is down–loaded from the server to the client. 'port@host' should be, if possible, limited to small license files (say < 50 features). In v5, 'port@host' actually improves performance.

***Does FLEXlm work with NFS?***

Yes. FLEXlm has no direct interaction with NFS. FLEXlm uses an NFS–mounted file like any other application.

***Does FLEXlm work with ATM, ISDN, Token–Ring, etc.?***

In general, these have no impact on FLEXlm. FLEXlm requires TCP/IP or SPX (Novell Netware). So long as TCP/IP works, FLEXlm will work.

***Does FLEXlm work with subnets, fully–qualified names, multiple domains, etc.?***

Yes, although this behavior was improved in v3.0, and v6.0. When a license server and a client are located in different domains, fully–qualified host names have to be used. A fully–qualified hostname is of the form:

*node.domain*

where *node* is the local hostname (usually returned by the '**hostname**' command or '**uname –n**') *domain* is the internet domain name, e.g. 'globes.com'.

To ensure success with FLEXlm across domains, do the following:

1. Make the sure the fully–qualified hostname is the name on the SERVER line of the license file.

2. Make sure ALL client nodes, as well as the server node, are able to 'telnet' to that fully–qualified hostname. For example, if the host is locally called 'speedy', and the domain name is 'corp.com', local systems will be able to logon to speedy via 'telnet speedy'. But very often, 'telnet speedy.corp.com' will fail, locally.
   Note that this telnet command will always succeed on hosts in other domains (assuming everything is configured correctly), since the network will resolve speedy.corp.com automatically.

3. Finally, there must be an 'alias' for speedy so it's also known locally as speedy.corp.com. This alias is added to the `/etc/hosts` file, or if NIS/Yellow Pages are being used, then it will have to be added to the NIS database. This requirement goes away in version 3.0 of FLEXlm.

   If all components (application, **lmgrd** and vendor daemon) are v6.0 or higher, no aliases are required; the only requirement is that the fully–qualified domain name, or IP–address, is used as a hostname on the SERVER, or as a hostname in LM_LICENSE_FILE port@host, or @host.

***Does FLEXlm work with NIS and DNS?***

Yes. However, some sites have broken NIS or DNS, which will cause FLEXlm to fail. In v5 of FLEXlm, NIS and DNS can be avoided to solve this problem. In particular, sometimes DNS is configured for a server that's not current available (e.g., a dial–up connection from a PC). Again, if DNS is configured, but the server is not available, FLEXlm will fail.

In addition, some systems, particularly Sun, SGI, HP, require that applications be linked dynamically to support NIS or DNS. If a vendor links statically, this can cause the application to fail at a site that uses NIS or DNS. In these situations, the vendor will have to relink, or recompile with v5 FLEXlm. Vendors are strongly encouraged to use dynamic libraries for libc and networking libraries, since this tends to improve quality in general, as well as making NIS/DNS work.

On PCs, if a checkout seems to take 3 minutes and then fails, this is usually because the system is configured for a dial–up DNS server which is not currently available. The solution here is to turn off DNS.

Finally, hostnames must NOT have periods in the name. These are not legal hostnames, although PCs will allow you to enter them, and they will not work with DNS.

***We're using FLEXlm over a wide-area network. What can we do to improve performance?***

FLEXlm network traffic should be minimized. With the most common uses of FLEXlm, traffic is negligible. In particular, checkout, checkin and heartbeats use very little networking traffic. There are two items, however, which can send considerably more data and should be avoided or used sparingly:

- '**lmstat –a**' should be used sparingly. '**lmstat –a**' should not be used more than, say, once every 15 minutes, and should be particularly avoided when there's a lot of features, or concurrent users, and therefore a lot of data to transmit; say, more than 20 concurrent users or features.
- Prior to FLEXlm v5, the 'port@host' mode of the LM_LICENSE_FILE environment variable should be avoided, especially when the license file has many features, or there are a lot of license files included in LM_LICENSE_FILE. The license file information is sent via the network, and can place a heavy load. Failures due to 'port@host' will generate the error LM_SERVNOREADLIC (–61).

INDEX

# INDEX

TASKING

# INDEX

# Symbols



# A



# B



# C

## D



## E



## F

## H



## I



## J



## L

# M

# N



# O



# P

## R



## S



## T



## U



## V