

# **TriCore v1.5**

## **CROSS-ASSEMBLER, LINKER/LOCATOR, UTILITIES USER'S GUIDE**

A publication of  
Altium BV  
Documentation Department  
Copyright © 2002 Altium BV

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

TASKING is a brand name of Altium Limited.

The following trademarks are acknowledged:

FLEXlm is a registered trademark of Globetrotter Software, Inc.

HP and HP-UX are trademarks of Hewlett-Packard Co.

Intel is a trademark of Intel Corporation.

Motorola is a registered trademark of Motorola, Inc.

MS-DOS and Windows are registered trademarks of Microsoft Corporation.

SUN is a trademark of Sun Microsystems, Inc.

UNIX is a registered trademark of X/Open Company, Ltd.

All other trademarks are property of their respective owners.

Data subject to alteration without notice.

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

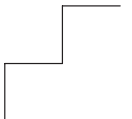
*The information in this document has been carefully reviewed and is believed to be accurate and reliable. However, Altium assumes no liabilities for inaccuracies in this document. Furthermore, the delivery of this information does not convey to the recipient any license to use or copy the software or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

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

# CONTENTS

## TABLE OF CONTENTS

---



---

# CONTENTS

---

## **OVERVIEW** **1-1**

1.1	Introduction . . . . .	1-3
1.2	An Example TriCore Program . . . . .	1-5
1.2.1	Using EDE . . . . .	1-5
1.2.2	Using the Control Program . . . . .	1-12
1.2.3	Using the Makefile . . . . .	1-15
1.3	Environment Variables . . . . .	1-16
1.4	Temporary Files . . . . .	1-17
1.5	Debugging with CrossView Pro . . . . .	1-17
1.6	File Extensions . . . . .	1-18
1.7	Preprocessing . . . . .	1-19
1.8	Assembler Listing . . . . .	1-20
1.9	Errors and Warnings . . . . .	1-20
1.10	Command Line Processing . . . . .	1-20
1.10.1	UNIX Scripts . . . . .	1-20

## **ASSEMBLER** **2-1**

2.1	Description . . . . .	2-3
2.2	Invocation . . . . .	2-3
2.3	astri Invocation . . . . .	2-4
2.4	aspcp Invocation . . . . .	2-6
2.5	Detailed Description of Assembler Options . . . . .	2-8
2.6	Environment Variables used by the Assembler . . . . .	2-44
2.7	List File . . . . .	2-44
2.7.1	Absolute List File Generation . . . . .	2-45
2.7.2	Page Header . . . . .	2-45
2.7.3	Source Listing . . . . .	2-46

## **SOFTWARE CONCEPT** **3-1**

3.1	Introduction . . . . .	3-3
3.2	Modules . . . . .	3-3
3.2.1	Modules and Symbols . . . . .	3-3
3.3	Sections . . . . .	3-4
3.3.1	Section Names . . . . .	3-4
3.3.2	Absolute Sections . . . . .	3-6
3.3.3	Section Examples . . . . .	3-7

## **ASSEMBLY LANGUAGE** **4-1**

4.1	Input Specification . . . . .	4-3
4.2	Assembler Significant Characters . . . . .	4-4

## **OPERANDS AND EXPRESSIONS** **5-1**

5.1	Operands . . . . .	5-3
5.1.1	TriCore Addressing Modes . . . . .	5-3
5.1.2	PCP Addressing Modes . . . . .	5-5
5.2	Expressions . . . . .	5-5
5.2.1	Number . . . . .	5-6
5.2.2	Expression String . . . . .	5-7
5.2.3	Symbol . . . . .	5-7
5.2.4	Expression Type . . . . .	5-8
5.3	Operators . . . . .	5-10
5.3.1	Addition and Subtraction . . . . .	5-11
5.3.2	Sign Operators . . . . .	5-11
5.3.3	Multiplication and Division . . . . .	5-12
5.3.4	Shift Operators . . . . .	5-12
5.3.5	Relational Operators . . . . .	5-13
5.3.6	Bitwise Operators . . . . .	5-13
5.3.7	Logical Operators . . . . .	5-14

5.4	Functions .....	5-15
5.4.1	Mathematical Functions .....	5-15
5.4.2	String Functions .....	5-15
5.4.3	Macro Functions .....	5-16
5.4.4	Assembler Mode Functions .....	5-16
5.4.5	Address Calculation Functions .....	5-16
5.4.6	Fractional Functions .....	5-17
5.4.7	Detailed Description .....	5-17

## **MACRO OPERATIONS**

**6-1**

6.1	Introduction .....	6-3
6.2	Macro Operations .....	6-3
6.3	Macro Definition .....	6-4
6.4	Macro Calls .....	6-5
6.5	Dummy Argument Operators .....	6-7
6.5.1	Dummy Argument Concatenation Operator – \ .....	6-7
6.5.2	Return Value Operator – ? .....	6-8
6.5.3	Return Hex Value Operator – % .....	6-9
6.5.4	Dummy Argument String Operator – ” .....	6-10
6.5.5	Macro Local Label Operator – ^ .....	6-11
6.6	.DUP, .DUPA, .DUPC, .DUPF Directives .....	6-11
6.7	Conditional Assembly .....	6-12

## **ASSEMBLER DIRECTIVES**

**7-1**

7.1	Overview .....	7-3
7.1.1	Debugging .....	7-3
7.1.2	Assembly Control .....	7-3
7.1.3	Symbol Definition .....	7-4
7.1.4	Data Definition/Storage Allocation .....	7-4
7.1.5	Macros and Conditional Assembly .....	7-5
7.2	Directives .....	7-5



## **ASSEMBLER CONTROLS** **8-1**

8.1	Introduction . . . . .	8-3
8.2	Overview Assembler Controls . . . . .	8-4
8.3	Description of Assembler Controls . . . . .	8-6

## **LINKER** **9-1**

9.1	Overview . . . . .	9-3
9.2	Linker Invocation . . . . .	9-4
9.2.1	Detailed Description of Linker Options . . . . .	9-5
9.3	Libraries . . . . .	9-25
9.3.1	Library Search Path . . . . .	9-25
9.3.2	Linking with Libraries . . . . .	9-27
9.3.3	Library Member Search Algorithm . . . . .	9-27
9.4	Linker Output . . . . .	9-28
9.5	Type Checking . . . . .	9-33
9.5.1	Introduction . . . . .	9-33
9.5.2	Recursive Type Checking . . . . .	9-34
9.5.3	Type Checking between Functions . . . . .	9-35
9.5.4	Missing Types . . . . .	9-37
9.6	Linker Messages . . . . .	9-37

## **LOCATOR** **10-1**

10.1	Overview . . . . .	10-3
10.2	Invocation . . . . .	10-4
10.2.1	Detailed Description of Locator Options . . . . .	10-5
10.2.2	Format Suboptions . . . . .	10-24
10.3	Locating Your Application . . . . .	10-25
10.4	Calling the Locator via the Control Program . . . . .	10-27
10.5	Locator Output . . . . .	10-27
10.6	Locator Messages . . . . .	10-27
10.7	Copy Table . . . . .	10-28
10.8	Locator Labels . . . . .	10-29
10.8.1	Locator Labels Reference . . . . .	10-30

## **UTILITIES** **11-1**

11.1	Overview .....	11-3
11.2	artri .....	11-4
11.3	cctri .....	11-7
11.4	mktri .....	11-13
11.5	prtri .....	11-26
11.5.1	Preparing the Demo Files .....	11-29
11.5.2	Displaying Parts of an Object File .....	11-29
11.5.2.1	Option -h, display general file info .....	11-29
11.5.2.2	Option -s, display section info .....	11-30
11.5.2.3	Option -c, display call graphs .....	11-32
11.5.2.4	Option -e, display external part .....	11-34
11.5.2.5	Option -g, display global type information .....	11-36
11.5.2.6	Option -d, display debug information .....	11-39
11.5.2.7	Option -i, display the section images .....	11-43
11.5.3	Viewing an Object at Lower Level .....	11-45
11.5.3.1	Object Layers .....	11-45
11.5.3.2	The Level Option -ln .....	11-46
11.5.3.3	The Verbose Option -vn .....	11-49

## **ASSEMBLER ERROR MESSAGES** **A-1**

1	Introduction .....	A-3
2	Warnings (W) .....	A-4
3	Errors (E) .....	A-11
4	Fatal Errors (F) .....	A-22

## **LINKER ERROR MESSAGES** **B-1**

1	Introduction .....	B-3
2	Warnings (W) .....	B-3
3	Errors (E) .....	B-6
4	Fatal Errors (F) .....	B-9
5	Verbose (V) .....	B-11

**LOCATOR ERROR MESSAGES** **C-1**

1	Introcuotion	C-3
2	Warnings (W)	C-3
3	Errors (E)	C-7
4	Fatal Errors (F)	C-11
5	Verbose (V)	C-13

**ARCHIVER ERROR MESSAGES** **D-1**

1	Introduction	D-3
2	Warnings (W)	D-3
3	Errors (E)	D-4
4	Fatal Errors (F)	D-4

**EMBEDDED ENVIRONMENT ERROR MESSAGES** **E-1**

1	Introduction	E-3
2	Errors (E)	E-3
3	Warnings (W)	E-5

**DESCRIPTIVE LANGUAGE FOR EMBEDDED ENVIRONMENTS** **F-1**

1	Introduction	F-3
2	Getting Started	F-3
2.1	Introduction	F-3
2.2	Basic Structure	F-3
3	CPU Part	F-6
3.1	Introduction	F-6
3.2	Address Translation: map and mem	F-8
3.3	Address Spaces	F-10
3.4	Addressing Modes	F-11
3.5	Busses	F-13
3.6	Chips	F-15
3.7	External Memory	F-16

4	Software Part .....	F-17
4.1	Introduction .....	F-17
4.2	Load Module .....	F-17
4.3	Layout Description .....	F-17
4.4	Space Definition .....	F-19
4.5	Block Definition .....	F-20
4.6	Selecting Sections .....	F-21
4.7	Cluster Definition .....	F-23
4.8	Amode Definition .....	F-24
4.9	Manipulating Sections in Amodes .....	F-25
4.10	Section Placing Algorithm .....	F-26
5	Memory Part .....	F-27
5.1	Introduction .....	F-27
6	Delfee Preprocessing .....	F-28
6.1	Introduction .....	F-28
6.2	User Defined Macros .....	F-28
6.3	File Inclusion .....	F-29
6.4	Conditional Statements .....	F-31
7	Delfee Keyword Reference .....	F-32
7.1	Abbreviation of Delfee Keywords .....	F-73
7.2	Delfee Keywords Summary .....	F-73

## **DELFE SYNTAX**

**G-1**

## **IEEE-695 OBJECT FORMAT**

**H-1**

1	TIOF and IEEE-695 .....	H-3
2	Command Language Concept .....	H-3
3	Notational Conventions .....	H-5
4	Expressions .....	H-5
4.1	Functions without Operands .....	H-8
4.2	Monadic Functions .....	H-8
4.3	Dyadic Functions and Operators .....	H-8
4.4	MUFOM Variables .....	H-9

4.5 @INS and @EXT Operator ..... H-10

4.6 Conditional Expressions ..... H-10

5 MUFOM Commands ..... H-11

5.1 Module Level Commands ..... H-11

5.1.1 MB Command ..... H-11

5.1.2 ME Command ..... H-11

5.1.3 DT Command ..... H-11

5.1.4 AD Command ..... H-12

5.2 Comment and Checksum Command ..... H-12

5.3 Sections ..... H-13

5.3.1 SB Command ..... H-13

5.3.2 ST Command ..... H-13

5.3.3 SA Command ..... H-15

5.4 Symbolic Name Declaration and Type Definition ..... H-15

5.4.1 NI Command ..... H-15

5.4.2 NX Command ..... H-16

5.4.3 NN Command ..... H-16

5.4.4 AT Command ..... H-16

5.4.5 TY Command ..... H-17

5.5 Value Assignment ..... H-18

5.5.1 AS Command ..... H-18

5.6 Loading Commands ..... H-18

5.6.1 LD Command ..... H-18

5.6.2 IR Command ..... H-18

5.6.3 LR Command ..... H-19

5.6.4 RE Command ..... H-20

5.7 Linkage Commands ..... H-20

5.7.1 RI Command ..... H-20

5.7.2 WX Command ..... H-20

5.7.3 LI Command ..... H-21

5.7.4 LX Command ..... H-21

6 MUFOM Functions ..... H-22

<b><u>MOTOROLA S-RECORDS</u></b>	<b><u>I-1</u></b>
----------------------------------	-------------------

<b><u>INTEL HEX RECORDS</u></b>	<b><u>J-1</u></b>
---------------------------------	-------------------

<b><u>INDEX</u></b>	
---------------------	--



CONTENTS

## **MANUAL PURPOSE AND STRUCTURE**

### **PURPOSE**

This manual is aimed at users of the TriCore Cross-Assembler, Linker/Locator and Utilities. It assumes that you are conversant with programming the TriCore.

### **MANUAL STRUCTURE**

Related Publications

Conventions Used In This Manual

1. Overview  
Makes you familiar with the assembler itself, through the use of sample programs.
2. Software Concept  
Describes the basics of modular programming and sections.
3. Assembler  
Describes the actions and invocation of the TASKING TriCore assembler (**astri**) and the PCP assembler (**aspcp**).
4. Assembly Language  
Describes the formats of the possible statements for an assembly program and describes the registers.
5. Operands and Expressions  
Describes the operands and expressions to be used in the assembler instructions and pseudos (directives).
6. Macro Operations  
Describes the use of macros and conditional assembly.
7. Assembler Directives  
Describes the Pseudo instructions to pass information to the assembler program.
8. Assembler Controls  
Describes the syntax and semantics of all assembler controls.



### 9. Linker

Describes the action of, and options/controls applicable, to the linker.

### 10. Locator

Describes the action of, and options/controls applicable, to the locator.

### 11. Utilities

Contains descriptions of the utilities supplied with the package, which may be useful during program development.

## **APPENDICES**

### A. Assembler Error Messages

Gives a list of error messages which can be generated by the assembler.

### B. Linker Error Messages

Gives a list of error messages which can be generated by the linker.

### C. Locator Error Messages

Gives a list of error messages which can be generated by the locator.

### D. Archiver Error Messages

Gives a list of error messages which can be generated by the archiver.

### E. Embedded Environment Error Messages

Gives a list of error messages from the embedded environment which can be generated by the linker/locator.

### F. DDescriptive Language For Embedded Environments

Describes the Delfee description language.

### G. Delfee Syntax

Contains a syntax description of the Delfee language.

### H. IEEE-695 Object Format

Contains a description of the IEEE-695 object format and the TIOF format.

### I. Motorola S-Records

Contains a description of the S0, S3 and S7 Motorola S-records.

### J. Intel Hex Records

Contains a description of the Intel Hex format.

## **INDEX**

## **RELATED PUBLICATIONS**

- TriCore C Cross-Compiler User's Guide [TASKING, MA060-002-00-00]
- TriCore CrossView Pro Debugger User's Guide [TASKING, MA060-043-00-00]
- TriCore Architecture Manual [Siemens]
- TriCore Architecture Manual [1999, Infineon]
- TriCore Architecture v1.3 Manual [2000, Infineon]



MANUAL STRUCTURE

## **CONVENTIONS USED IN THIS MANUAL**

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

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

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

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

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

*filename*

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

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

**screen font** Represents input examples and screen output examples.

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

### ***For example***

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

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

### ***Illustrations***

The following illustrations are used in this manual:



This is a note. It gives you extra information.



This is a warning. Read the information carefully.



This illustration indicates actions you can perform with the mouse.



This illustration indicates keyboard input.



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

# CHAPTER

# 1

## OVERVIEW

---



---

# 1

# CHAPTER

---

## 1.1 INTRODUCTION

TASKING offers a complete toolchain for the Siemens TriCore Family of processors and their derivatives. This manual uses 'TriCore' as a shorthand notation for the Siemens TriCore Family of processors and their derivatives.

The TriCore family cross-assembler produces load files running on the TriCore family. The assembler **astri** accepts programs written according to the assembly language specification for the TriCore. The assembler **aspcp** accepts programs written according to the assembly language specification for the Peripheral Control Processor (PCP).

Both assemblers generate relocatable object files in the IEEE-695 object format. This file format specifies code parts as well as symbol definition and symbolic debug information parts. The locator optionally produces absolute output files in Motorola S-file format or Intel Hex format. You can load these formats into a PROM programmer.

The TriCore toolchain provides an environment for modular program development and debugging. The TriCore toolchain contains the following programs:

<b>cctri</b>	A handy control program which activates the other programs depending on its input files.
<b>astri</b>	The TriCore assembler which produces a relocatable object file from a given TriCore assembly file.
<b>aspcp</b>	The PCP assembler which produces a relocatable object file from a given PCP assembly file.
<b>lktri</b>	A linker combining several objects and object libraries into one target load file.
<b>lctri</b>	A locator that links a number of linker output files to one absolute load file. This program can also produce Motorola S-file format, and Intel Hex format.
<b>artri</b>	An IEEE archiver. This is a librarian facility, which can be used to create and maintain object libraries.
<b>prtri</b>	An IEEE object reader. This utility dumps the contents of IEEE files which have been created by a tool from the TASKING TriCore family toolchain.



The following diagram shows the structure of the package.

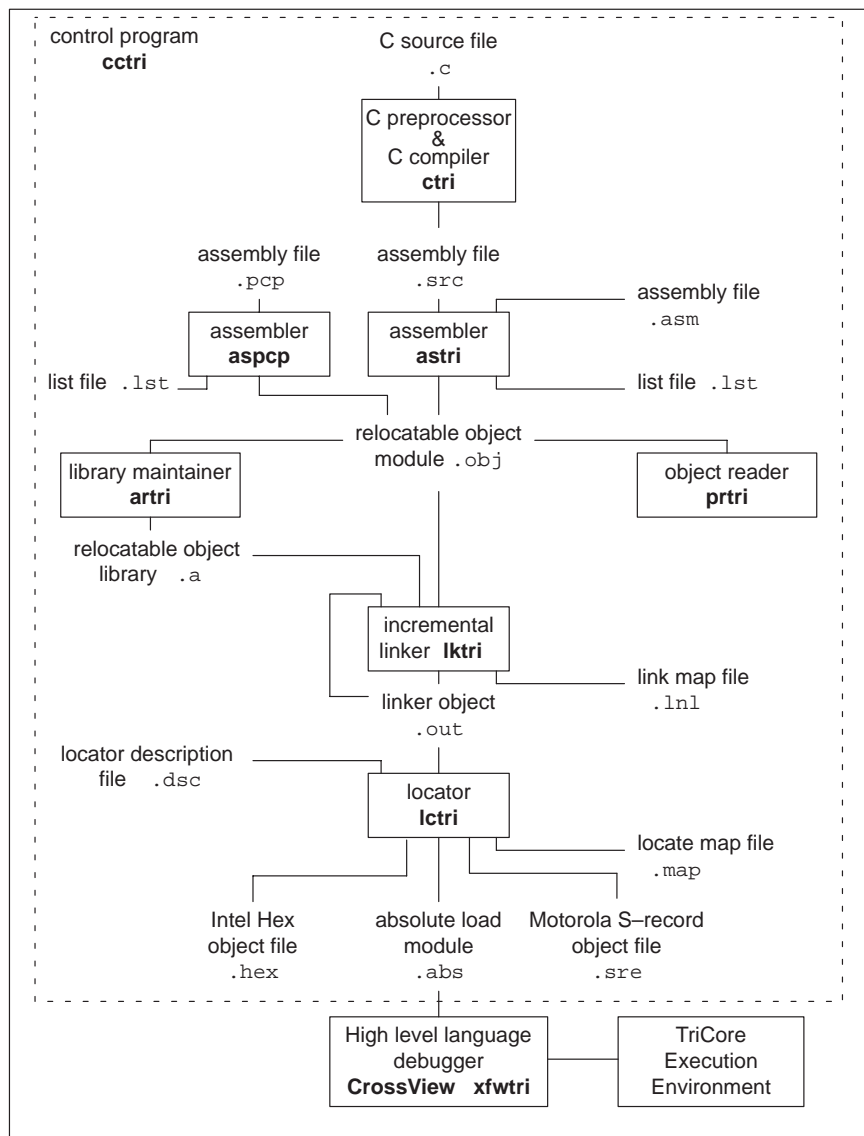


Figure 1-1: TriCore family development flow

## **1.2 AN EXAMPLE TRICORE PROGRAM**

This section illustrates the use of the TriCore toolchain by building an example application. As a part of the installation a directory `examples\xvw` (`examples/xvw` for UNIX) is created depending on the place where you installed the package on your system. This example directory contains, among others, the following source files:

`addone.asm`, `demo.c`



This program has been written for illustrative purposes only.

### **1.2.1 USING EDE**

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

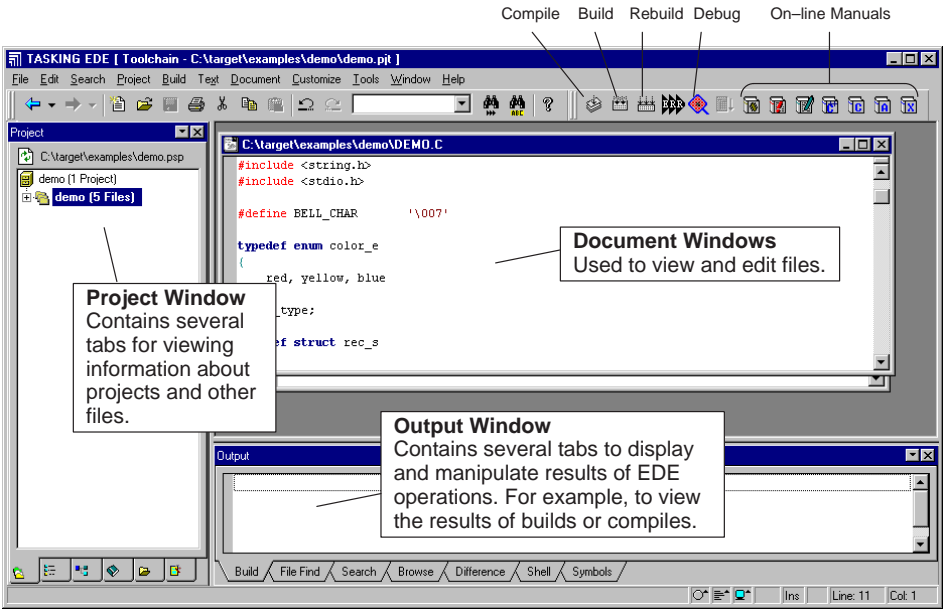
To use EDE on the demo program in the subdirectory `xvw` in the `examples` subdirectory of the TriCore product tree follow the steps below.

#### ***How to Start EDE***

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



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



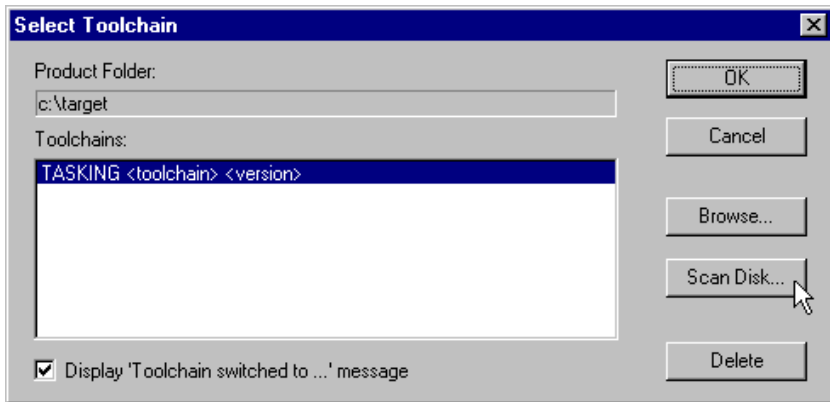
***How to Select a Toolchain***

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

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

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

The Select Toolchain dialog appears.



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

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

### ***How to Open an Existing Project***

Follow these steps to open an existing project:

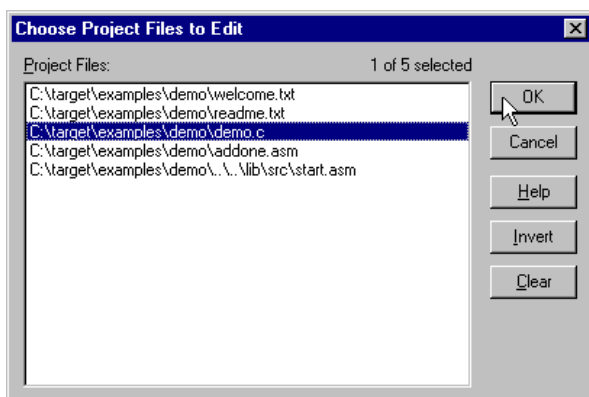
1. From the **Project** menu, select **Set Current ->**.
2. Select the project file to open. For the **xvw** demo program select the file **demo.pjt** in the subdirectory **xvw** in the **examples** subdirectory of the TriCore product tree. If you have used the defaults, the file **demo.pjt** is in the directory **c:\ctri\examples\xvw**.

### ***How to Load/Open Files***

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

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

The **Choose Project Files to Edit** dialog appears.



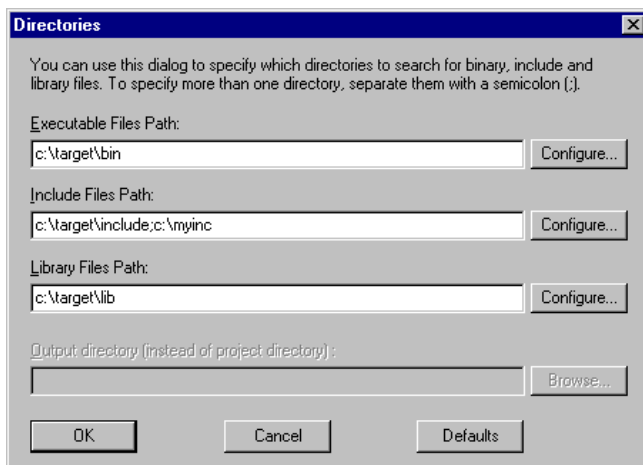
2. Choose the file(s) you want to open by clicking on it. You can select multiple files by pressing the <Ctrl> or <Shift> key while you click on a file. With the <Ctrl> key you can make single selections and with the <Shift> key you can select everything from the first selected file to the file you click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

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

### ***Check the directory paths***

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

The Directories dialog appears.



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

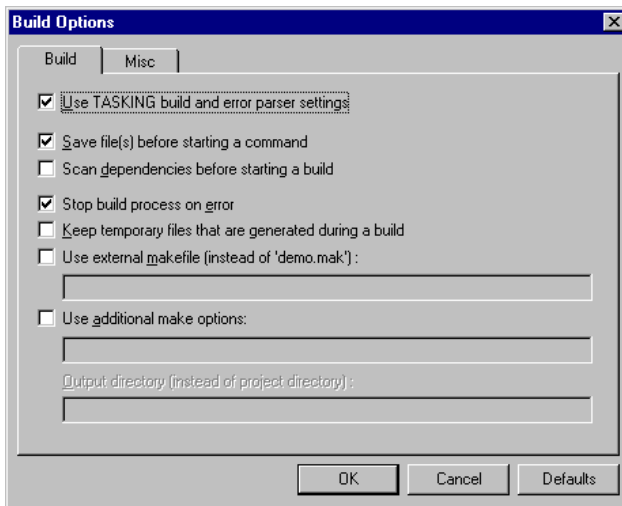
### ***How to Build the Demo Application***

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

Steps 1 and 2 are optional. Follow these steps if you want to specify additional build options such as to stop the build process on errors and to keep temporary files that are generated during a build.

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

The Build Options dialog appears.



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



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

### ***How to View the Results of a Build***

Once the files have been processed you can inspect the generated messages.

1. In the Window menu select the Output menu item.

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

```
TASKING program builder vx.y rz    Build nnn SN 00000000
Assembling "addone.asm"
Compiling "demo.c"
Assembling "demo.src"
Linking and locating to demo.out
Converting demo.out to demo.abs in IEEE-695 format
.
.
.
```

### ***How to Start the CrossView Pro Debugger***

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

To execute CrossView Pro:

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



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

### ***How to Load an Application***

You must tell CrossView Pro which program you want to debug:

1. From the File menu, select Load Symbolic Debug Info...

The Load Symbolic Debug Info dialog box appears.

2. Click Load.

### ***How to View and Execute an Application***

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

1. From the View menu, select `Source->Source lines`.

The source window opens.

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

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

To run your application step-by-step:

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

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

### ***How to Start a New Project***

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

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

The `Create a New Project Space` dialog appears.

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

The `Project Properties` dialog box appears.

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

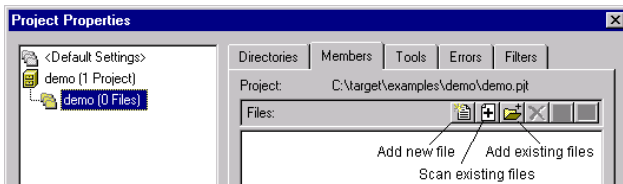
The `Add New Project to Project Space` dialog appears.

4. Give your project a name and then click `OK`.

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

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





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

The new project is now open.

6. From the Project menu, select Load Files... to open the files you want on your EDE desktop.

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

## 1.2.2 USING THE CONTROL PROGRAM

1. Instead of invoking all the individual translation phases by hand, it is possible (and recommended) to use the control program **cctri**:

```
cctri -g demo.c addone.asm -o demo.abs
```

As you can see, you may enter multiple input files on the command line. Also, you may specify options and controls for the assembler, linker and locator together. The control program recognizes the options and controls and places them in the appropriate command when invoking the assembler, linker or locator. The control program is described in detail in Chapter 11, *Utilities*.

The **-g** option enables symbolic debug information.

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

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

```
cctri -g demo.c addone.asm -o demo.abs -v0
```

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

```
demo.c:
+ ctri -e -g -o /tmp/cc2120b.src demo.c
+ astri /tmp/cc2120b.src -e -ghl -o demo.objaddone.asm:
+ astri addone.asm -e -ghl -o addone.obj
+ lktri -e demo.obj addone.obj -lc -lfpn -Ltc1
-o/tmp/cc2120c.out
+ lctri -e -M -dtri.dsc /tmp/cc2120c.out -odemo.abs
```

The **-e** option removes output files after errors occur. The **-gs1** option of the assembler specifies to pass HLL debug information and to generate local symbols debug information. The **-lc** and **-lfpn** option of the linker specify to link the appropriate C library and floating point library respectively. The **-O** option of the linker specifies the basename of the map file. The **-d** option of the locator specifies the name of the locator description file.

3. In step 2, the tools use temporary files for intermediate results. If you want to keep the intermediate files you can use the **-tmp** option. The following command makes this clear.

```
cctri -g demo.c addone.asm -o demo.abs -v0 -tmp
```

This command produces the following output:

```
demo.c:
+ /usr/src/dolphin/dvl/linux/ctri/bin/ctri -e -g -o
demo.src demo.c
+ /usr/src/dolphin/dvl/linux/ctri/bin/astri demo.src -e
-ghl -o demo.obj
addone.asm:
+ /usr/src/dolphin/dvl/linux/ctri/bin/astri addone.asm
-e -ghl -o addone.obj
+ /usr/src/dolphin/dvl/linux/ctri/bin/lktri -e demo.obj
addone.obj -lc -lfpn
-L/usr/src/dolphin/dvl/linux/ctri/lib/tcl -odemo.out
+ /usr/src/dolphin/dvl/linux/ctri/bin/lctri -e -M
-dtri.dsc demo.out -odemo.abs
```

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

Assuming the program assembles successfully, the assembler produces the relocatable output modules, `demo.obj` and `addone.obj`.

Linking and locating the program to absolute addresses is done by two programs: the linker combines objects into a relocatable file with the extension `.out`. The locator binds the program to absolute addresses. The linker takes `a.out` as the default name of the output file. If this name is not suitable, you can specify another filename with the **-O** option.

Besides the output file produced by the linker, the locator can take a so-called description file as input. This file contains a description of the virtual and physical addresses of the program. The chapter *Locator* discusses the exact contents and layout of a description file.

The result of the locator command is the absolute output file `demo.abs`. The file `demo.abs` can be loaded into the CrossView Pro debugger.

### 1.2.3 USING THE MAKEFILE

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

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

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

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

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

```
mktri
```

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

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

```
mktri -n
```

This command produces the following output:

```
TASKING TriCore program builder    vx.yrz Build nnn  
Copyright 1996-year Altium BV      Serial# 00000000  
cctri -g -w303 -c demo.c  
cctri -c -o addone.obj addone.asm  
cctri -o demo.abs demo.obj addone.obj
```

To remove all generated files type:

```
mktri clean
```

## 1.3 ENVIRONMENT VARIABLES

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

Environment Variable	Description
ASPCPIINC	Specifies an alternative path for include files for the assembler <b>aspcp</b> .
ASTRIINC	Specifies an alternative path for include files for the assembler <b>astri</b> .
CTRIINC	Specifies an alternative path for #include files for the C compiler.
CTRILIB	Specifies a path to search for library files used by the linker <b>lktri</b> . See also the section <i>Library Search Path</i> in the chapter <i>Linker</i> .
CCTRIBIN	When this variable is set, the control program, <b>cctri</b> , prepends the directory specified by this variable to the names of the tools invoked.
CCTRIOPT	Specifies extra options and/or arguments to each invocation of <b>cctri</b> . The control program processes the arguments from this variable before the command line arguments.
LM_LICENSE_FILE	Identifies the location of the license data file. Only needed for hosts that need the FLEXlm license manager. Not needed for MS-Windows.
PATH	Specifies the search path for your executables.
TMPDIR	Specifies an alternative directory where programs can create temporary files. Used by <b>cctri</b> , <b>cctri</b> , <b>astri</b> , <b>lktri</b> , <b>lctri</b> , <b>artri</b> . See also next section.

Table 1-1: Environment variables

## 1.4 TEMPORARY FILES

The assemblers, linker, locator and archiver may create temporary files. By default these files will be created in the current directory. If you want the tools to create temporary files in another directory you can enforce this by setting the environment variable `TMPDIR`.

UNIX:

Bourne shell, Korn shell:

```
TMPDIR=/tmp ; export TMPDIR
```

csh:

```
setenv TMPDIR /tmp
```

Note that if you create your temporary files on a directory which is accessible via the network for other users as well, conflicts in the names chosen for the temporary files may arise. It is safer to create temporary files in a directory that is solely accessible to yourself. Of course this does not apply if you run the tools with several users on a multi-user system, such as UNIX. Conflicts may arise if two different computer systems use the same network directory for tools to create their temporary files.

## 1.5 DEBUGGING WITH CROSSVIEW PRO

To facilitate debugging, you can include symbolic debug information in the load file. During compilation of a high-level-language program, symbolic debug information must be retained that serves as input for the symbolic debugger (**-g** option). The compiler passes symbolic debug information to the **astri** assembler by generating SYMB assembler directives in the assembly source file. The **astri** assembler translates the SYMB directives to be included in the symbolic debug part of an IEEE-695 object file.

The CrossView Pro debugger accepts files with the IEEE-695 format. This is the default output format of the locator. So, you can directly load the file generated by the locator into the CrossView Pro debugger.

The simplest way to build this program ready for debugging is:

```
cctri-g demo.c addone.asm
```

The result of this command is (output of **-v0** option):

```
demo.c:
+ ctri -e -g -o /tmp/cc28309b.src demo.c
+ astri -e -gsl -o demo.obj /tmp/cc28309b.src
addone.asm:
+ astri -e -gsl -o addone.obj addone.asm
+ lktri -e demo.obj addone.obj -lc -lfpn -odemo.out
+ lctri -e -M -odemo.abs -dtri.dsc demo.out
```

The control program is described in detail in Chapter 11, *Utilities*.

You can start the text version of the debugger for debugging the absolute file `demo.abs` with:

```
xfwtri demo.abs
```

For more information on the debugger, see the *TriCore CrossView Pro Debugger User's Guide*.



The debugger examples are installed in the subdirectory `xvw` of the `examples` directory.

## 1.6 FILE EXTENSIONS

The extension `.src` or `.asm` is used as input file for the TriCore assembler **astri**. The extension `.pcp` is used as input file for the PCP assembler **aspcp**. Files with the extension `.src` are output files of a C compiler. Actually, the assemblers accept files with any extension (or even no extension), but by adding the extension `.asm` (for **astri**) or `.pcp` (for **aspcp**) to assembler source files, you can distinguish them easily.

If you do not provide a filename extension the **aspcp** assembler will try:

1. the filename itself
2. the filename with `.pcp` extension

If you do not provide a filename extension the **astri** assembler will try:

1. the filename itself
2. the filename with `.asm` extension
3. the filename with `.src` extension

So,

```
astri text
```

only has the same effect as

```
astri text.asm
```

if the file `text` is not present. In this case, both these commands assemble the file `text.asm` and create a relocatable object module `text.obj`.

For compatibility with future TASKING Cross-Software the following extensions are suggested:

<b>.asm</b>	input assembly source file for <b>astri</b>
<b>.src</b>	output from the C compiler <b>ctri</b> / input for <b>astri</b>
<b>.pcp</b>	input assembly source file for <b>aspcp</b>
<b>.c</b>	C source file, input for the C compiler <b>ctri</b>
<b>.cc</b>	C++ source file, input for the C++ compiler <b>cptri</b>
<b>.ic</b>	temporary C source file, generated by the C++ compiler <b>cptri</b>
<b>.obj</b>	relocatable object file generated by the assembler
<b>.a</b>	object library files, output from <b>artri</b>
<b>.out</b>	relocatable output files from <b>lktri</b>
<b>.dsc</b>	description file, input for <b>lctri</b> and CrossView Pro debugger
<b>.abs</b>	absolute locator output, object file from <b>lctri</b>
<b>.hex</b>	absolute Intel Hex file
<b>.sre</b>	absolute Motorola S-record file
<b>.elf</b>	absolute ELF/DWARF object file
<b>.lst</b>	assembler list file
<b>.cal</b>	C functions call tree
<b>.lnl</b>	linker map file
<b>.map</b>	locator map file

## 1.7 PREPROCESSING

The assemblers have a built-in macro preprocessor. For a description of the possibilities offered by the macro preprocessor see the chapter *Macro Operations*.



## 1.8 ASSEMBLER LISTING

The assembler does not generate a listing file by default. You can generate a listing file with the **-l** option. (See also the **-L** option, for the listing options). As a result of the command:

```
astri -l text.src
```

the listing file `text.lst` is created.

## 1.9 ERRORS AND WARNINGS

Any errors detected by the assembler are displayed in the listing file after the actual line containing the error is printed. If no listing file is produced, error messages are still displayed to indicate that the assembly process did not proceed normally.

Three classes of messages can be reported depending on the severity of the error. Warning messages (W) are the least severe messages. Warnings do not terminate assembly. Error messages (E) appear when a user error situation occurs. These errors do not terminate assembly immediate. If one or more of these errors occur, assembly stops at the end of the active pass. Fatal error messages (F) cause the assembler to terminate immediately. Fatal errors are usually due to user errors or memory errors.

## 1.10 COMMAND LINE PROCESSING

This section contains a description of the use of UNIX scripts. The use of Makefiles is explained in the chapter *Utilities*.

### 1.10.1 UNIX SCRIPTS

Scripts are a facility within UNIX whereby one or more commands can be executed from within a file.

Assume that the following sequence of calls is frequently used:

```
cctri -c ifile.asm -o outfile.obj
```

The files *ifile* and *outfile* may vary from one call to the next. To reduce the number of calls you can make a script, for example, `proj`. The file should contain:

```
cctri -c $1.asm -o $2.obj
```

On invocation `$1` and `$2` will be replaced by the first and second parameters after the script file name. Using the name mentioned above (`proj`) and after you have set the execute bits of `proj` (`chmod +x proj`) the call becomes:

```
proj ifile outfile
```



# OVERVIEW

# CHAPTER

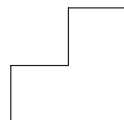
# 2

## ASSEMBLER

---



TASKING



---

# 2

# CHAPTER

---

## 2.1 DESCRIPTION

The TriCore toolchain has two assemblers. The main assembler is **astri**, which assembles sources that are either hand written in the TriCore assembly language or generated by the compiler **cctri**. The other assembler is the PCP assembler **aspcp**, which assembles PCP coprocessor specific assembly files.

The TriCore assemblers **astri** and **aspcp** are optimizing assemblers. During assembly the assembler builds an internal representation of the program. This representation, the *flow graph*, is used to optimize the program. Examples of the optimizations are choosing alternatives for instructions. After optimization the object file and, optionally, the list file are generated.

The following phases can be identified during assembly:

1. Preprocess, check the syntax and create the flow graph
2. Type determination of all expressions
3. Legality check of all instructions
4. Optimization
5. Address calculation, jump optimization
6. Generation of object and (when requested) list file

The assembler generates relocatable object files using the IEEE-695 object format. This file format specifies a code part and a symbol part as well as a symbolic debug information part.

File inclusion and macro facilities are integrated into the assembler.



See the chapter *Macro Operations* for more information.

## 2.2 INVOCATION

The compiler control program, **cctri**, may call the assemblers automatically. **cctri** translates some of its command line options to options of **astri** or **aspcp** depending on the filename extension. Files with extension **.asm** or **.src** are sent to **astri**, files with extension **.pcp** are sent to **aspcp**. However, the assemblers can be invoked as individual programs also.

## 2.3 ASTRI INVOCATION

The **PC** invocation of **astri** is:

```
astri [option]... source-file [map-file]  
astri -V  
astri -?
```

When you use a UNIX shell (**C-shell**, **Bourne shell**), options containing special characters (such as **'( )'**) must be enclosed with **" "**. The invocations for UNIX and PC are the same, except for the **-?** option in the **C-shell**:

```
astri "-?"      or      astri -\?
```

Invocation with **-V** only displays a version header. **-?** shows the invocation syntax.

The *source-file* must be an assembly source file. This file is the input source of the assembler. This file contains assembly code which is either user written or generated by **ctri**. Any name is allowed for this file. If the file is not found and the filename does not have an extension, the extension **.asm** is assumed or, if the file is still not found, the extension **.src** is assumed.

The optional *map-file* is passed to the assembler when producing an absolute list file. The map file is produced by the locator. To produce an absolute list file, see section 2.7.1, *Absolute List File Generation*.

In the default situation, an object file with extension **.obj** is produced. With the **-l** option a list file with extension **.lst** is produced.

Options are preceded by a **'-'** (minus sign). Options can not be combined after a single **'-'**. If all goes well, the assembler generates a relocatable object module which contains the object code, with the default extension **.obj**. You can specify another output filename with the **-o** option. Error messages are written to the terminal, unless they are directed to an error list file with the **-err** assembler option.

The following list describes the assembler options briefly. Section 2.5 gives a more detailed description.

Option	Description
-?	Display invocation syntax
-C <i>cpu</i>	Include SFR definition file <i>regcpu.def</i> before source
-D <i>macro</i> [= <i>def</i> ]	Define preprocessor <i>macro</i>
-FPU	Allow the use of single precision floating point instructions.
-H <i>file</i>	Include <i>file</i> before source
-Idirectory	Look in <i>directory</i> for include files
-L[ <i>flag</i> ...]	Remove specified source lines from list file
-MMU	Allow memory management instructions in the assembly source.
-TC2	Allow the use of TriCore2 instructions
-V	Display version header only
-WAE	Treat warning messages as errors
-c	Switch to case insensitive mode (default case sensitive)
-e	Remove object file on assembly errors
-err	Redirect error messages to error file
-f <i>file</i>	Read options from <i>file</i>
-g[a h s]	Generate assembly level debug information
-hw_only	All mnemonics are directly mapped on the corresponding instruction, no clever substitutions allowed
-i[l g]	Default label style local or global
-l	Generate listing file
-llength	Set number of lines per page in a list file
-lnfile	Specify a name for the list file
-lttitle	Specify a header title for list file pages
-lwwidth	Set the number of characters per line in a list file
-o <i>filename</i>	Specify name of output file
-t[ <i>flag</i> ...]	Display section summary
-v	Verbose mode. Print the filenames and numbers of the passes while they progress



Option	Description
<code>-w[num]</code>	Suppress one or all warning messages
<code>-zdefect</code>	Enable checks for CPU functional problem <i>defect</i>

Table 2-1: Options summary *astri*

2.4 ASPCP INVOCATION

The PC invocation of **aspcp** is:

```
aspcp [option]... source-file [map-file]
aspcp -V
aspcp -?
```

When you use a UNIX shell (**C-shell**, **Bourne shell**), options containing special characters (such as `'( )'`) must be enclosed with `" "`. The invocations for UNIX and PC are the same, except for the `-?` option in the **C-shell**:

```
aspcp "-?" or aspcp -\?
```

Invocation with `-V` only displays a version header. `-?` shows the invocation syntax.

The *source-file* must be an assembly source file, either use the original PCP syntax or the PCP2 syntax. This file is the input source of the assembler. This file contains assembly code which is user written. Any name is allowed for this file. If the file is not found and the filename does not have an extension, the extension `.pcp` is assumed.

The optional *map-file* is passed to the assembler when producing an absolute list file. The map file is produced by the locator. To produce an absolute list file, see section 2.7.1, *Absolute List File Generation*.

In the default situation, an object file with extension `.obj` is produced. With the `-I` option a list file with extension `.lst` is produced.

Options are preceded by a `'-'` (minus sign). Options can not be combined after a single `'-'`. If all goes well, the assembler generates a relocatable object module which contains the object code, with the default extension `.obj`. You can specify another output filename with the `-o` option. Error messages are written to the terminal, unless they are directed to an error list file with the `-err` assembler option.

The following list describes the assembler options briefly. Section 2.5 gives a more detailed description.

Option	Description
<b>-?</b>	Display invocation syntax
<b>-Ccpu</b>	Include sfr definition file <i>regcpu.def</i> before source
<b>-Dmacro[=def]</b>	Define preprocessor <i>macro</i>
<b>-Hfile</b>	Include <i>file</i> before source
<b>-Idirectory</b>	Look in <i>directory</i> for include files
<b>-L[flag...]</b>	Remove specified source lines from list file
<b>-Pprefix</b>	Add <i>prefix</i> to each global and external symbol (default '_PCP_')
<b>-V</b>	Display version header only
<b>-c</b>	Switch to case insensitive mode (default case sensitive)
<b>-e</b>	Remove object file on assembly errors
<b>-err</b>	Redirect error messages to error file
<b>-f file</b>	Read options from <i>file</i>
<b>-g[a l]</b>	Generate assembly level debug information
<b>-i[l g]</b>	Default label style local or global
<b>-l</b>	Generate listing file
<b>-o filename</b>	Specify name of output file
<b>-p [1 2]</b>	Select PCP version (default is 1)
<b>-pTC1775</b>	Generate code for PCP as found on the TC1775
<b>-pRAPTOR</b>	Generate code for PCP as found on the Raptor
<b>-t</b>	Display section summary
<b>-v</b>	Verbose mode. Print the filenames and numbers of the passes while they progress
<b>-w[num]</b>	Suppress one or all warning messages

Table 2-2: Options summary *aspcp*

## 2.5 DETAILED DESCRIPTION OF ASSEMBLER OPTIONS



All options in this section apply to both assembler **astri** and **aspcp**, unless explicitly stated otherwise.



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

**-?**

### Option:

**-?**

### Description:

Display an explanation of options at stdout.

### Example:

**astri -?**

# -C

## Option:



Select the Project | Processor Options... menu item and choose the CPU tab. Select a CPU type in the CPU type field.

If you select User defined in the CPU type field, type the name of your TriCore derivative in the User specified CPU name field.

If you select User defined in the CPU type field and leave the User specified CPU name field empty, the option **-C** is not used.



**-Ccpu**

## Arguments:

The CPU name which identifies your TriCore derivative.

## Description:

Use special function register definitions for *cpu*. The filename looked for is "reg*cpu*.def". The search algorithm for .sfr files is the same as for include files that are enclosed in "" at the beginning of the C source. The file is included before the source.

## Example:

To use SFR definitions from the file regtc10gp.def, enter:

```
astri -Ctc10gp test.src
```

## -C

### Option:



Select the Project | Assembler Options | Project Options... menu item. Disable the Assemble case sensitive check box in the Misc tab.



-c

### Default:

Case sensitive

### Description:

Switch to case insensitive mode. By default, the assembler operates in case sensitive mode.

### Example:

To switch to case insensitive mode, enter:

```
astri -c test.src
```

# -D

## Option:



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



**-D***macro*[=*def*]

## Arguments:

The macro you want to define and optionally its definition.

## Description:

Define *macro* as in 'define'. If *def* is not given (= is absent), '1' is assumed. Any number of symbols can be defined.

## Example:

```
astri -DTWO=2 test.src
```

# -e

**Option:**

EDE always removes the output file on errors.



**-e**

**Description:**

Use this option if you do not want an object file when the assembler generates errors. With this option the 'make' utility always does the proper productions.

**Example:**

```
astri -e test.src
```

## -err

### Option:



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



**-err**

### Description:

The assembler redirects error messages to a file with the same basename as the output file and the extension `.ers`. The assembler uses the basename of the output file instead of the input file.

### Example:

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

```
astri -err test.src
```



## -f

### Option:

**-f** *file*

### Arguments:

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

### Description:

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

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

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

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

Example:

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

or

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

or

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

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

Example:

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

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

### Example:

Suppose the file `mycmds` contains the following line:

```
-err  
test.src
```

The command line can now be:

```
astri -f mycmds
```

# -FPU

## Option:



Select the Project | Assembler Options | Project Options... menu item. Enable the Use hardware single precision floating point instructions check box in the Misc tab.



This option is only available (and relevant) when you enable the presence of a floating point unit (FPU) check box on the CPU tab in the Project | Processor Options... menu item.



## -FPU

## Default:

(none) The use of fpu instructions is *not* allowed.

## Description:

The **-FPU** option allows the use of single precision floating point instructions in the assembly code. When you select this option, the define `_FPU` is set to 1 allowing you to use single precision floating point instructions. Default the define `_FPU` is set to 0.



For a more detailed description about the floating point arithmetic see section *Floating Point Arithmetic* in Chapter *Runtime* of the *C Cross Compiler Users Manual*.

## Example:

To allow the use of floating point unit (FPU) instructions in the assembly code, enter:

```
astri -FPU test.src
```

# -g (aspcp)

## Option:



Select the Project | PCP Assembler Options | Project Options... menu item. Choose a Debug information option in the Debug tab.



**-g[a|l]...**

## Default:

**-gAL** (no debug)

## Description:

Specify to generate debug information. If you do not use this option or if you specify **-g** without a flag, the default is **-gAL** (no debug information).

Flags can be switched on with the lower case letter and switched off with the uppercase letter.

An overview of the flags is given below.

- a** – assembler source line information
- l** – local symbols debug information

With **-ga** you enable assembler source line information. With **-gl** you enable the generation of local symbols debug information. You can use this option independent of the setting of option **-ga**.

## Examples:

To generate local symbols debug information, enter:

```
aspcp -gl test.pcp
```

To generate assembler source line information, enter:

```
aspcp -ga test.pcp
```

# -g (astri)

## Option:



Select the Project | Assembler Options | Project Options... menu item. Choose a Debug information option in the Debug tab.



**-g[a|h|l|s]...**

## Default:

**-gAhLS** (only HLL debug)

## Description:

Specify to generate debug information. If you do not use this option or if you specify **-g** without a flag, the default is **-gAhLS**, which only passes the high level language debug information.

Flags can be switched on with the lower case letter and switched off with the uppercase letter.

An overview of the flags is given below.

- a** – assembler source line information
- h** – pass HLL debug information
- l** – local symbols debug information
- s** – always debug; either **"AhL"** or **"aHl"**

With **-ga** you enable assembler source line information. With **-gh** the assembler passes the high level language debug information from the compiler to the object file. These two types of debug information cannot be used both. So, **-gah** is not allowed.

With **-gl** you enable the generation of local symbols debug information. You can use this option independent of the setting of the **-ga** and **-gh** options.

With **-gs** you instruct the assembler to always generate debug information. If HLL debug information is present in the source file, the assembler passes this information (same as **-gAhL**). If no HLL debug information is present, the assembler generates assembler source line information and local symbols debug information (same as **-gaHl**).

**Examples:**

To pass high level symbolic debug information to the output files and generate local symbols debug information, enter:

```
astri -ghl test.src
```

To generate assembler source line information, enter:

```
astri -ga test.src
```

To always generate debug information, depending on the debug information in the source file, enter:

```
astri -gs test.src
```

# -H

## Option:



Select the Project | Assembler Options | Project Options... menu item. Add the name of an include file to the Include file before source field in the Files tab.



**-H***file*

## Arguments:

The name of an include file.

## Description:

Include *file* before assembling the source.

## Example:

To include the file `tri.inc` before any other include file, enter:

```
astri -Htri.inc test.src
```

# -hw\_only

## Option:



**-hw\_only**

## Default:

none                      Substitutions by faster or smaller instructions is allowed.

## Description:

Normally the assembler may replace instructions by other, smaller or faster instructions. For example, the instruction `jeq d0,#0,label1` is replaced by `jz d0,label1`.

With the option **-hw\_only** you instruct the assembler to encode all instruction as they are. The assembler does not substitute instructions with other, faster or smaller instructions. When you use this option, the define `_HW_ONLY` is set to 1 (no substitution of instructions).

## Example:

To prevent the assembler from substituting instructions with other, smaller or faster instructions:

```
astri -hw_only test.src
```





### Option:



Select the Project | Assembler Options | Project Options... menu item. Add one or more directory paths to the Include search path field in the Files tab. You can specify more directories by separating them with commas.



**-I***directory*

### Arguments:

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

### Description:

Change the algorithm for searching include files whose names do not have an absolute pathname to look in *directory*. Thus, include files whose names are enclosed in "" are searched for first in the directory of the file containing the include line, then in the current directory, then in directories named in **-I** options in left-to-right order. If the include file is still not found, the assembler searches in a directory specified with the environment variable ASTRINC for **astri** or ASPCPINC for **aspcp**. This environment variable can contain more than one directory. Separate multiple directories with ';' for PC (':' for UNIX). Finally, the directory ../include relative to the directory where the assembler binary is located is searched.

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

### Example:

```
astri -I/proj/include test.src
```

**Option:**

Select the Project | Assembler Options | Project Options... menu item. Choose one of the Labels are by default Local/Global options in the Misc tab.



**-i**[l|g]

**Default:**

**-il** (local labels)

**Description:**

Select default handling for label identifiers. **-il** specifies that data and code assembly labels are by default treated as LOCAL labels, unless overruled by the **.global** directive. With **-ig** data and code assembly labels are by default treated as GLOBAL labels, unless overruled by the **.local** directive.

**Example:**

To specify that assembly label identifiers are treated as GLOBAL labels by default, enter:

```
astri -ig test.src
```

# -L

## Option:



Select the Project | Assembler Options | Project Options... menu item. Enable or disable one or more Include lines containing check boxes in the Adv. List File tab.



**-L**[*flag...*]

## Arguments:

Optionally one or more flags specifying which source lines are to be removed from the list file.

## Default:

**-LcDElMNPQsWX** (for **astri**)

**-LcDElMNPQWX** (for **aspcp**)

## Description:

Specify which source lines are to be removed from the list file. A list file is generated when the **-I** option is specified. If you do not specify the **-L** option the assembler removes source lines containing #line directives or symbolic debug information, empty source lines and puts wrapped source lines on one line. **-L** without any flags, is equivalent to **-LcdeImnpqswx**, which removes all specified source lines from the list file.

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

- c** Default. Remove source lines containing assembler controls (**\$page**).
- C** Keep source lines containing assembler controls.
- d** Remove source lines containing section directives (the **.sdecl**, **.sect** directives).
- D** Default. Keep source lines containing section directives.
- e** Remove source lines containing one of the symbol definition directives **.extern**, **.global**, **.local** or **.calls**.
- E** Default. Keep source lines containing symbol definition directives.

- l** Default. Remove source lines containing C preprocessor line information (lines with **#line**).
- L** Keep source lines containing C preprocessor line information.
- m** Remove source lines containing **.macro/.dup** directives (lines with **.macro** or **.dup**).
- M** Default. Keep source lines containing **.macro/.dup** directives.
- n** Remove empty source lines (newlines).
- N** Default. Keep empty source lines.
- p** Remove source lines containing conditional assembly (lines with **.if**, **.elif**, **.else**, **.endif**). Only the valid condition is shown.
- P** Default. Keep source lines containing conditional assembly.
- q** Remove source lines containing assembler equates (lines with **.equ**).
- Q** Default. Keep source lines containing assembler equates.
- s** Default. This option is for **astri** only. Remove source lines containing high level language symbolic debug information (lines with **.symb**).
- S** This option is for **astri** only. Keep source lines containing HLL symbolic debug information.
- w** Remove wrapped part of source lines.
- W** Default. Keep wrapped source lines.
- x** Remove source lines containing **.macro/.dup** expansions.
- X** Default. Keep source lines containing **.macro/.dup** expansions.

## Example:

To remove source lines with assembler controls from the resulting list file and to remove wrapped source lines, enter:

```
astri -l -Lcw test.src
```



**-l** (generate list file)



### Option:



Select the Project | Assembler Options | Project Options... menu item. Enable the Generate list file check box in the List File tab.



**-l**

### Description:

Generate listing file. The listing file has the same basename as the output file. The extension is .lst.

### Example:

To generate a list file with the name test.lst, enter:

```
astri -l test.src
```



**-L** (remove source lines from list file)

**-ll** (set number of lines per page in a list file)

**-ln** (specify a name for the list file)

**-lt** (specify a header title for list file pages)

**-lw** (set the number of characters per line in a list file)

# -ll

## Option:



Select the **Project | Assembler Options | Project Options...** menu item. Select the **List File** tab and enter the page length in the **Lines per page** field.



**-llength**

## Description:

Sets the maximum number of lines on one page of the listing file. This number includes the lines used by the page header (4) and the lines with error messages. The default page length is 80. The minimum page length is 10.

## Example:

To set the page length to 50 lines per page:

```
astri -l -ll50 test.src
```



**-l** (generate list file)

# -ln

## Option:



Select the Project | Assembler Options | Project Options... menu item. Select the List File tab and enter a name in the Name of list file field.



**-ln***file*

## Description:

When you use the option **-l** to generate a list file, the default name for the list file is *source.lst*. With this option you can specify another name for the list file.

## Example:

To generate a list file with the name *mylist.lst*:

```
astri -l -lnmylist.lst test.src
```



**-l** (generate list file)

# -lt

## Option:



Select the Project | Assembler Options | Project Options... menu item. Select the List File tab and enter a title in the Title of list file field.



**-lt***title*

## Description:

This control specifies the *title* to be used in the page header of each page of the list file. The page number and date are automatically included in the page header.

## Example:

To specify the title "MYTITLE" for the headers in the list file:

```
astri -l -ltMYTITLE test.src
```



**-l** (generate list file)



## -lw

### Option:



Select the Project | Assembler Options | Project Options... menu item. Select the List File tab and enter the page length in the Characters per line field.



**-lw***width*

### Description:

Sets the maximum number of characters on one line in the list file. Lines that exceed this width are wrapped on the next lines. The default page width is 66. The minimum page width is 40.

### Example:

To set the page width to 50 characters per page:

```
astri -l -lw50 test.src
```



**-l** (generate list file)

# -MMU

## Option:



Select the Project | Assembler Options | Project Options... menu item. Enable the Allow memory management instructions check box in the Misc tab.



This option is only available (and relevant) when you enable the MMU Present check box on the CPU tab in the Project | Processor Options... menu item.



**-MMU**

## Default:

-MMU      The use of memory management instructions is allowed.

## Description:

The **-MMU** option allows the use of memory management instructions in the assembly source. When you select this option, the define `_MMU` is set to 1 allowing you to use memory management instructions.

## Example:

To allow the use of memory management instructions in the assembly source, enter:

```
astri -MMU test.src
```

# -TC2

## Option:



Select the Project | Processor Options... menu item. Select the CPU tab. Select TC2 in the CPU type box.



**-TC2**

## Default:

(none) The use of TriCore2 instructions is *not* allowed.

## Description:

The **-TC2** option allows the use TriCore2 instructions in the assembly code. When you select this option, the define `_TC2` is set to 1 allowing you to use single precision floating point instructions. Default the define `_TC2` is set to 0.

## Example:

To allow the use of TriCore2 instructions in the assembly code, enter:

```
astri -TC2 test.src
```

## -O

### Option:

**-o** *filename*

### Arguments:

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

### Default:

Baseline of assembly file with `.obj` suffix.

### Description:

Use *filename* as output filename of the assembler, instead of the baseline of the assembly file with the `.obj` extension.

### Example:

To create the object file `myfile.obj` instead of `test.obj`, enter:

```
astri test.src -o myfile.obj
```

## -P (aspcp only)

### Option:



Select the Project | PCP Assembler Options | Project Options... menu item. Enter a prefix (or no prefix) in the Prefix for global symbols field in the Misc tab.



**-P**[*prefix*]

### Default:

**-P\_PCP\_**

### Description:

With this option you can specify a prefix to use for global and external symbols. The default prefix is `_PCP_`. The prefix is useful to distinguish these symbols from symbols generated by the **astri** assembler.

### Example:

To use no prefix for global and external symbols (same as **astri** assembler), enter:

```
aspcp -P test.pcp
```

# -p (aspcp only)

## Option:



Select the Project | PCP Assembler Options | Project Options... menu item. Select either PCP or PCP2 in the misc. tab.



**-p[1|2]**

## Default:

**-p1**

## Description:

Choose the syntax for the PCP assembler, either the original syntax (-p1, the default), or the PCP2 syntax (-p2)

## Example:

To use PCP syntax, enter:

```
aspcp less.PCP
aspcp -p less.PCP
aspcp -p1 less.PCP
```

To use PCP2 syntax, enter:

```
aspcp -p2less2.PCP
```

# -pRAPTOR (aspcp only)

## Option:



Select the `Project | Processor Options...` menu item. Select the CPU tab. Select either the Raptor in the CPU type box.



## -pRAPTOR

## Description:

With this option you tell the assembler to generate code for the Raptor.

## Example:

To assemble the file `test.src` for the Raptor:

```
aspcp -pRAPTOR test.src
```

# -pTC1775 (aspcp only)

## Option:



Select the Project | Processor Options... menu item. Select the CPU tab. Select either the TC1775A or the TC1775B in the CPU type box.



**-pTC1775**

## Description:

With this option you tell the assembler to generate code for the TC1775A or TC1775B.

## Example:

To assemble the file `test.src` for the TC1775A or TC1775B:

```
aspcp -pTC1775 test.src
```



# -t

**Option:**



Select the Project | Assembler Options | Project Options... menu item. Select the List File tab. Enable the Section Summary checkbox. (Only available if the Generate List File checkbox is enabled.)



**-t**[flag]

**Description:**

With this option you tell the assembler to display section information. For each section its memory address, size, number of cycles and name is listed on stdout or in the list file.

You can set the following flags. Use a lowercase letter to switch the flag 'on' and an uppercase letter to switch the flag 'off'.

**c/C**    Display section information on stdout

**l/L**    Write section information to the list file

With **-tl**, the assembler writes the section information to the list file. You must specify this option in combination with the option **-l** (generate listing file). If you specify **-t** without flags, both flags are enabled (**-tcl**).

**Example:**

```
astri -tcl test.src
```

The following section information is displayed on stdout:

Section summary:

NR	ADDR	SIZE	CYCLE	NAME
1		0007	5	.text
2	021234	000e	0	.data
3		0001	0	.tiny



**-l**

## -V

### Option:

**-V**

### Description:

With this option you can display the version header of the assembler. This option must be the only argument of the assembler. Other options are ignored. The assembler exits after displaying the version header.

### Example:

**astri -V**

```
TASKING TriCore assembler      vx.yrz Build nnn  
Copyright 1996-year Altium BV  Serial# 00000000
```

**aspcp -V**

```
TASKING PCP assembler          vx.yrz Build nnn  
Copyright 1998-year Altium BV  Serial# 00000000
```

## **-V**

### **Option:**

**-v**

### **Description:**

Verbose mode. With this option specified, the assembler prints the filenames and the assembly passes while they progress. So you can see the current status of the assembler.

### **Example:**

```
astri -v test.src
```

```
Parsing "test.src"  
    30 lines (total now 31)  
Optimizing  
Evaluating absolute ORG addresses  
Parsing symbolic debug information  
Creating object file "test.obj"  
Closing object file
```

## -W

### Option:



Select the Project | Assembler Options | Project Options... menu item. Select the Suppress all warnings check box in the Misc tab.



**-w***[num]*

### Arguments:

Optionally the warning number to suppress.

### Description:

**-w** suppress all warning messages. **-wnum** suppresses warning messages with number *num*. More than one **-wnum** option is allowed.

### Example:

The following example suppresses warnings 113 and 114:

```
astri -w113 -w114 file.src
```

# -WAE

**Option:**

**-WAE**

**Description:**

Treat warning messages as errors. This also affects the return value of the application when only warnings occur. A build process will now stop when warnings occur.

**Example:**

```
astri -WAE test.src
```

## -z

### Option:



Select the **Project | Processor Options...** menu and choose the **Bypasses TC1 v1.2** tab or the **Bypasses TC1 v1.3** tab. Then select the bypasses you want to enable.



**-zdefect**

### Arguments:

The name of a particular CPU functional problem that you want to be checked. The syntax used by Infineon to identify a CPU functional problems is:

*TC<architecture\_nr><version>\_<module\_name><problem\_nr>*

For example: **TC113\_CPU5** (TC1, version 1.3, module “CPU”, problem #5)

### Description:

With the option **-z** you can enable or disable specific CPU functional problem checks. You can specify more than one defect.

To enable the assembler checks for *all* TriCore CPU TC112 problems (respectively TC113 problems) at once, use the command line option **-zTC112\_DEFECTS** (respectively **-zTC113\_DEFECTS**).

### Example:

The following example enables the checks for CPU functional problem *TC112\_COR1*:

```
astri -zTC112_COR1
```



Appendix *CPU Functional Problems* in the C Cross-Compiler User's Guide for a complete overview of all CPU functional problems that are supported by the assembler.

## 2.6 ENVIRONMENT VARIABLES USED BY THE ASSEMBLER

- ASTRIINC** With this environment variable you can specify directories where the **astri** assembler will search for include files. You can overrule this search path with the **-I** command line option. Multiple pathnames can be separated with semicolons.
- ASPCINC** With this environment variable you can specify directories where the **aspcp** assembler will search for include files. You can overrule this search path with the **-I** command line option. Multiple pathnames can be separated with semicolons.
- TMPDIR** With the **TMPDIR** environment symbol you can specify the directory where the assembler can generate temporary files. If the assembler terminates normally, the temporary file will be removed automatically.  
If you do not set **TMPDIR**, the temporary file will be created in the current working directory.

## 2.7 LIST FILE

The list file is the output file of the assembler which contains information about the generated code. The amount and form of information depends on the use of the **-L** option. The name is the basename of the output file with the extension **.lst**. The list file is only generated when the **-l** option is supplied. When **-l** is supplied, a list file is also generated when assembly errors/warnings occur. In this case the error/warning is given just below the source line containing the error/warning.



From EDE you can control the list file generation in the **Listing** tab of the **Project | Assembler Options | Project Options...** menu item.

### 2.7.1 ABSOLUTE LIST FILE GENERATION

After locating the whole application, an absolute list file can be generated for all assembly source input files with the assembler. To generate an absolute list file from an assembly source file the source code needs to be assembled again with use of the locator map file of the application the assembly source belongs to. See section 10.5, *Locator Output*, how to produce a locator map file.

An absolute list file contains absolute addresses whereas a standard list file contains relocatable addresses.

When a map file is specified as input for the assembler, only the absolute list file is generated when list file generation is enabled with the list file option **-l**. The previously generated object file is not overwritten when absolute list file generation is enabled. Absolute list file generation is only enabled when a map file is specified on the input which contains the filename extension `.map`.



When you want to generate an absolute list file, you have to specify the same options as you did when generating the object file. If the options are not the same you might get an incorrect absolute list file.

#### **Example:**

Suppose your first invocation was:

```
astri -ig test.src
```

then when you want to generate an absolute list file you have to specify the same option (**-ig**) and the **-l** option:

```
astri -ig -l test.src test.map
```

With this command the absolute list file "test.lst" is created.

### 2.7.2 PAGE HEADER

The page header consists of four lines.

The first line contains the following information:

- information about assembler name
- version and serial number
- copyright notice





The second line contains a title specified by the TITLE (first page) or STITLE (succeeding pages) control and a page number.

The third line contains the name of the file (first page) or is empty (succeeding pages).

The fourth line contains the header of the source listing as described in the next section.

**Example:**

```
TriCore assembler va.b rc      SNzzzzzzzz-zzz (c) year TASKING, Inc.
Title for demo use only                               page 1
/tmp/hello.asm
ADDR      CODE      CYCLES LINE SOURCELINE
```

**2.7.3 SOURCE LISTING**

The following line appears in the page header:

```
ADDR      CODE      LINE SOURCELINE
```

The different columns are discussed below.

**ADDR** This is the memory address. The address is a (8 digit) hexadecimal number that represents the offset from the beginning of a relocatable section or the absolute address for an absolute section.

In lines that generate object code, the value is at the beginning of the line. For any other line there is no display.

**Example:**

ADDR	CODE	LINE	SOURCELINE
00000000		1	.sdecl "code.add",CODE,ABS24
00000000		2	.sect "code.add"
00000000		3	add:
00000000	85r5rrrr	4	ld.w d5,incr
00000004	0B540020	5	add d2,d4,d5
00000008	0090	6	ret
00000000		8	.sdecl "data.add",DATA,ABS18
00000000		9	.sect "data.add"
00000000	34120000	10	incr: .word 0x1234
00000000		12	.sdecl "bss.add",DATA,ABS18
00000000		13	.sect "bss.add"
00000000		14	buf: .space 4
	RESERVED		
00000003			

**CODE**

This is the object code generated by the assembler for this source line, displayed in hexadecimal format. The displayed code need not be the same as the generated code that is entered in the object module. The code can also be relocatable code or a relocatable part and external part. In this case the letter 'r' is printed for the relocatable code part in the listing.

For lines that allocate space (.SPACE) the code field contains the text "RESERVED".

**Example:**

ADDR	CODE	LINE	SOURCELINE
		.	
		.	
00000000	85r5rrrr	4	ld.w d5,incr
00000004	0B540020	5	add d2,d4,d5
00000008	0090	6	ret
		.	
		.	
00000000	34120000	10	incr: .word 0x1234
00000000		12	.sdecl "bss.add",DATA,ABS18
00000000		13	.sect "bss.add"
00000000		14	buf: .space 4
	RESERVED		
00000003			

In this example the word "RESERVED" marks the space reserved for the **.space** directive.

**LINE** This column contains the line number. This is a decimal number indicating each input line, starting from 1 and incrementing with each source line. If listing of the line is suppressed (i.e. by \$LIST OFF), the number increases by one anyway.

**Example:**

The following source part,

```
        ;Line 12
$LIST OFF
        ;Line 14
$LIST ON
        ;Line 16
```

results in the following list file part:

ADDR	CODE	CYCLES	LINE	SOURCELINE
			.	
			.	
			12	;Line 12
			16	;Line 16

**SOURCELINE** This column contains the source text. This is a copy of the source line from the source module. For ease of reading the list file, tabs are expanded with sufficient numbers of blank spaces.

If the source column in the listing is too narrow to show the whole source line, the source line is continued in the next listing line.

Errors and warnings are included in the list file following the line in which they occurred.

**Example:**

ADDR	CODE	LINE	SOURCELINE
00000000		1	.sdecl "code.error",CODE,ABS24
00000000		2	.sect "code.error"
00000000		3	error:
00000000	C202	4	add d2,16
astri W140:	error.asm	line 4	: expression must be between -8 and 7
00000002	0090	5	ret

# CHAPTER

# 3

## **SOFTWARE CONCEPT**

---



---

# 3

# CHAPTER

---

### **3.1 INTRODUCTION**

Complex software projects often are divided into smaller program units. These subprograms may be written by a team of programmers in parallel, or they may be programs written for a precious development effort that are going to be reused. The TASKING assembler provides directives to subdivide a program into smaller parts, modules. Symbols can be defined local to a module, so that symbol names can be used without regard to the symbols in other modules. Code and data can be organized in separate sections. These sections can be named in such a way that different modules can implement different parts of these sections. These sections can be located in memory by the locator so that concerns about memory placement are postponed until after the assembly process. By using separate modules, a module can be changed without re-assembling the other modules. This speeds up the turnaround time during the development process.

### **3.2 MODULES**

Modules are the separate implementation parts of a project. Each module is defined in a separate file. A module is assembled separately from other modules. By using the `.INCLUDE` directive common definitions and macros can be included in each module. Using the **mktri** utility the module file and include file dependencies can be specified so only the correct modules are re-assembled after changes to one of the files the modules depend upon.

#### **3.2.1 MODULES AND SYMBOLS**

A module can use symbols defined in other modules and in the module itself. Symbols defined in a module can be local (other modules cannot access it) or global (other modules have access to it). Symbols outside of a module can be defined with the `.EXTERN` directive. Local symbols are symbols defined by the `.LOCAL` directive or symbols defined with an `.SET` or `.EQU` directive. Global symbols are either labels, or symbols explicitly defined global with the `.GLOBAL` directive.

### 3.3 SECTIONS

Sections are relocatable blocks of code and data. Sections are defined with the `.SDECL` directive and have a name. A section may have attributes to instruct the locator to place it on a predefined starting address, in short or non-short memory or that it may be overlaid with another section. See the `.SDECL` directive discussion for a complete description of all possible attributes. Sections are defined once and are activated with the `.SECT` directive. The linker will check between different modules and emits an error message if the section attributes do not match. The linker will also concatenate all matching section definitions into one section. So, all `".text"` sections generated by the compiler will be linked into one big `".text"` chunk which will be located in one piece. By using this naming scheme it is possible to collect all pieces of code or data belonging together into one bigger section during the linking phase. A `.SECT` directive referring to an earlier defined section is called a continuation. Only the name can be specified.

#### 3.3.1 SECTION NAMES

The assembler generates object files in relocatable IEEE-695 object format. The assembler groups units of code and data in the object file using sections. All relocatable information is related to the start address of a section. The locator assigns absolute addresses to sections. A section is the smallest unit of code or data that can be moved to a specific address in memory after assembling a source file. The compiler requires that the assembler supports several different sections with appropriate attributes to assign specific characteristics to those sections. (section with read only data, sections with code etc.)

**`.SDECL`** *sect\_name*, *sect\_type* [, *attrib*] ... [ **AT** *address* ]

A section must be declared before it can be used. The `.SDECL` directive declares a section with its attributes. A section name can be any identifier. The '@' character is not allowed in regular section names. The assembler and linker use this character to create overlayable sections. This is explained below.

The section type can be:

*sect\_type* : CODE | DATA

This defines in what memory (CODE or DATA) the section is located.

The section attributes can be:

<i>attrib</i> :	ABS24	direct addressable code memory ( <b>astri</b> only)
	ABS18	direct addressable data memory ( <b>astri</b> only)
	FPI	addressable through FPI bus ( <b>aspcp</b> only)
	CLEAR	clear section during program startup
	NOCLEAR	section is not cleared during startup
	INIT	initialization data copied from ROM to RAM at startup
	MAX	common, overlay with other parts with the same name, is implicit a type of 'noclear'
	ROMDATA	section contains data instead of executable code

The **ABS24** and **ABS18** attributes specify direct addressable CODE or DATA memory for the **astri** assembler. **FPI** is only valid for the **aspcp** assembler; this attribute specifies that a section must be accessed through the FPI bus. This section will not be located in PCP internal memory, but in TriCore memory.

Unless disabled, the startup code in the toolchain has to clear data sections with the **CLEAR** attribute. These sections contain data space allocations for which no initializers have been specified. CLEAR sections are zeroed (cleared) at program startup. Sections can be excluded from this initialization with the **NOCLEAR** attribute. This is also the default situation for all sections.

The **MAX** attribute changes the way the linker determines the section size. Normally the linker determines the section size by accumulating the contents and the sizes of sections with the same name in different object modules. When sections with the same name occur in different object modules with the MAX attribute, the linker generates a section of which the size is the maximum of the sizes in the individual object modules.

It is also possible to control the size of MAX sections on a module basis, with the section activation attribute **RESET**. The assembler starts recounting storage allocation for MAX sections with the same name, when they are re-activated with the RESET section activation attribute. For these sections the assembler generates the maximum size they occupy in the object file. The section activation attribute RESET applies to MAX sections only. Example:



```
.SDECL ".newdat", DATA, MAX      ;section declaration
.SECT  ".newdat"                  ;section activation
...
.SECT  ".newdat", RESET           ;section re-activation
```

Sections become absolute when an address has been specified in the declaration using the **AT** keyword. The assembler generates information in the object file which instructs the locator to put the section contents at the specified address. It is not allowed to make an overlayable section absolute. The assembler reports an error if the AT keyword is used in combination with the OVERLAY section attribute.

After a section has been declared, it can be activated and re-activated with the .SECT directive:

```
.SDECL ".STRING", CODE, ROMDATA
.SECT  ".STRING"
_1001: .ASCII "hello world"
```

All instructions and pseudos which generate data or code must be within an active section. The assembler emits a warning if code or data starts without a section definition and activation.

### **3.3.2 ABSOLUTE SECTIONS**

Absolute sections (i.e. .SDECL directives with a start address) may only be continued in the defining module (continuation). When such a section is defined in the same manner in another module, the locator will try to place the two sections at the same address. This results in a locator error. When an absolute section is defined in more than one module, the section must be defined relocatable and its starting address must be defined in the locator description (.dsc) file. Overlay sections may not be defined absolute.

### 3.3.3 SECTION EXAMPLES

Some examples of the .SDECL and .SECT directives are as follows:

```
.SDECL ".CONST", CODE AT 0x1000  
.SECT ".CONST"
```

Defines and activates a section named .CONST starting on address 0x1000. Other parts of the same section, and in the same module, must be defined with:

```
.SECT ".CONST"
```

```
.SDECL ".text", CODE  
.SECT ".text"
```

Defines and activates a relocatable section in CODE memory. Other parts of this section, with the same name, may be defined in the same module or any other module. Other modules should use the same .SDECL statement. When necessary, it is possible to give the section an absolute starting address with the locator description file.

```
.SDECL ".fardata", DATA, CLEAR  
.SECT ".fardata"
```

Defines a relocatable named section in DATA memory. The CLEAR attribute instructs the locator to clear the memory located to this section. When this section is used in another module it must be defined identically. Continuations of this section in the same module are as follows:

```
.SECT ".fardata"
```



# CHAPTER

# 4

## ASSEMBLY LANGUAGE



---

# 4

# CHAPTER

---

## 4.1 INPUT SPECIFICATION

An assembly program consists of zero or more statements, one statement per line. A statement may optionally be followed by a comment, which is introduced by a semicolon character (;) and terminated by the end of the input line. Any source statement can be extended to one or more lines by including the line continuation character (\) as the last character on the line to be continued. The length of a source statement (first line and any continuation lines) is only limited by the amount of available memory. Upper and lower case letters are considered equivalent for assembler mnemonics and directives, but are considered distinct for labels, symbols, directive arguments, and literal strings.

A *statement* can be defined as:

[*label*:] [*instruction* | *directive* | *macro\_call*] [*comment*]

where,

*label* is an *identifier* or *number*. A label does not have to start on the first position of a line, but a label must always be followed by a colon.

*identifier* can be made up of letters, digits and/or underscore characters (\_). The first character may not be a digit. The size of an identifier is only limited by the amount of available memory.

*number* is a number ranging from 1 to 255. This type of label is called a **numeric label** or **local label**. To refer to a numeric label, you must put an **n** (next) or **p** (previous) immediately after the label. This is required because the same label number may be used repeatedly.

Example:

```
LAB1:           ;This is a label
1: j lp         ;This is an endless loop
                ;using numeric labels
```

*instruction* is any valid TriCore assembly language instruction consisting of a mnemonic and operands. Operands are described in the chapter *Operands and Expressions*.

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*     any one of the assembler directives; described separately in the chapter *Assembler Directives*.

*macro\_call*   a call to a previously defined macro. See the chapter *Macro Operations*.

A statement may be empty.

**4.2 ASSEMBLER SIGNIFICANT CHARACTERS**

There are several one character sequences that are significant to the assembler. Some have multiple meanings depending on the context in which they are used. Special characters associated with expression evaluation are described in Chapter 5, *Operands and Expressions*. Other assembler-significant 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 operator
- ” - Macro string delimiter or  
Quoted string **DEFINE** expansion character
- @ - Function delimiter
- \* - Location counter substitution

Individual descriptions of each of the assembler special characters follow. They include usage guidelines, functional descriptions, and examples.

;

***Comment Delimiter Character***

Any number or characters preceded by a semicolon (;), but not part of a literal string, is considered a comment. Comments are not significant to the assembler, but they can be used to document the source program. Comments will be reproduced in the assembler output listing. Comments are preserved in macro definitions.

Comments can occupy an entire line, or can be placed after the last assembler-significant field in a source statement. The comment is literally reproduced in the listing file.

***Examples:***

```
; This comment begins in column 1 of the source file

Loop: CALL COMPUTE ; This is a trailing comment
                  ; These two comments are preceded
                  ; by a tab in the source file
```



## ***Line Continuation Character or Macro Dummy Argument Concatenation Operator***

### ***Line Continuation***

The backslash character (\), if used as the last character on a line, indicates to the assembler that the source statement is continued on the following line. The continuation line will be concatenated to the previous line of the source statement, and the result will be processed by the assembler as if it were a single line source statement. The maximum source statement length (the first line and any continuation lines) is 512 characters.

### ***Example:***

```
; THIS COMMENT \
EXTENDS OVER \
THREE LINES
```

### ***Macro Argument Concatenation***

The backslash (\) is also used to cause the concatenation of a macro dummy argument with other adjacent alphanumeric characters. For the macro processor to recognize dummy arguments, they must normally be separated from other alphanumeric characters by a non-symbol character. However, sometimes it is desirable to concatenate the argument characters with other characters. If an argument is to be concatenated in front of or behind some other symbol characters, then it must be followed by or preceded by the backslash, respectively.

See also section 6.5.1.

### ***Example:***

Suppose the source input file contained the following macro definition:

```
SWAP_MEM .macro REG1,REG2           ;swap memory contents
    LD.W D0,[A\REG1]                ;using D0 as temp
    LD.W D1,[A\REG2]                ;using D1 as temp
    ST.W [A\REG1],D1
    ST.W [A\REG2],D0
    .endm
```

The concatenation operator (\) indicates to the macro processor that the substitution characters for the dummy arguments are to be concatenated in both cases with the character A. If this macro were called with the following statement,

```
SWAP_MEM 0,1
```

the resulting expansion would be:

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



### ***Return Value of Symbol Character***

The `?symbol` sequence, when used in macro definitions, will be replaced by an ASCII string representing the value of *symbol*. This operator may be used in association with the backslash (\) operator. The value of *symbol* must be an integer.

See also section 6.5.2.

### ***Example:***

Consider the following macro definition:

```
SWAP_MEM .macro REG1,REG2      ;swap memory contents
    LD.W D0,_lab\?REG1        ;using D0 as temp
    LD.W D1,_lab\?REG2        ;using D1 as temp
    ST.W _lab\?REG1,D1
    ST.W _lab\?REG2,D0
    .endm
```

If the source file contained the following .SET statements and macro call,

```
AREG .set 1
BREG .set 2
    SWAP_MEM  AREG,BREG
```

the resulting expansion as it would appear on the source listing would be:

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

***Return Hex Value of Symbol Character***

The `%symbol` sequence, when used in macro definitions, will be replaced by an ASCII string representing the hexadecimal value of *symbol*. This operator may be used in associations with the backslash (`\`) operator. The value of *symbol* must be an integer.

See also section 6.5.3.

***Example:***

Consider the following macro definition:

```
GEN_LAB    .macro  LAB,VAL,STMT
LAB\%VAL:  STMT
    .endm
```

If this macro were called as follows,

```
NUM .set      10
    GEN_LAB  HEX,NUM, 'NOP'
```

The resulting expansion as it would appear in the listing file would be:

```
HEXA: NOP
```



### ***Macro Local Label Character***

The circumflex (^), when used as a unary operator in a macro expansion, will cause name mangling of any associated local label. Normally, the macro preprocessor will leave any local label inside a macro expansion to a normal label in the current module. By using the Local Label character (^), the label is made a unique label. This is done by removing the leading underscore and appending a unique string "\_\_M\_Lxxxxxx" where "xxxxxx" is a unique sequence number. The ^-operator has no effect outside of a macro expansion. The ^-operator is useful for passing label names as macro arguments to be used as local label names in the macro. Note that the circumflex is also used as the binary exclusive or operator.

See also section 6.5.5.

### ***Example:***

Consider the following macro definition:

```
LOAD .macro ADDR
    ADDR:
        LD.W D0, ADDR
    ^ADDR:
        LD.W D0, ^ADDR
.endm
```

If this macro were called as follows,

```
LOAD _LOCAL
```

the resulting expansion as it would appear in the listing file would be:

```
_LOCAL:
    LD.W D0, _LOCAL
_LOCAL__M_L000001:
    LD.W D0, _LOCAL__M_L000001
```

”

***Macro String Delimiter or  
Quoted String .DEFINE Expansion Character******Macro String***

The double quote (”), when used in macro definitions, is transformed by the macro processor into the string delimiter, the single quote ('). The macro processor examines the characters between the double quotes for any macro arguments. This mechanism allows the use of macro arguments as literal strings.

See also section 6.5.4.

***Example:***

Using the following macro definition,

```
CSTR .macro   STRING
      .ascii  "STRING"
      .endm
```

and a macro call,

```
CSTR      ABCD
```

the resulting macro expansion would be:

```
.ascii   'ABCD'
```

***Quoted String DEFINE Expansion***

A sequence of characters which matches a symbol created with a **.DEFINE** directive will not be expanded if the character sequence is contained within a quoted string. Assembler strings generally are enclosed in single quotes ('). If the string is enclosed in double quotes (") then **.DEFINE** symbols will be expanded within the string. In all other respects usage of double quotes is equivalent to that of single quotes.

***Example:***

Consider the source fragment below:

```
.define LONG 'short'
STR_MAC .macro STRING
    .message      'This is a LONG STRING'
    .message      "This is a LONG STRING"
.endm
```

If this macro were invoked as follows,

```
STR_MAC    sentence
```

then the resulting expansion would be:

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

***Function Delimiter***

All assembler built-in functions start with the @ symbol. See section 5.4 for a full discussion of these functions.

***Example:***

```
SVAL .equ @ABS(VAL)      ; Obtain absolute value
```



***Location Counter Substitution***

When used as an operand in an expression, the asterisk represents the current integer value of the run-time location counter.

***Example:***

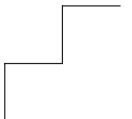
```
.sdecL   ".CODE", CODE   AT 0x100
.sect    ".CODE"
XBASE .equ  *+0x20          ; XBASE = 0x120
```

# CHAPTER

# 5

## **OPERANDS AND EXPRESSIONS**

---



---

# 5

# CHAPTER

---

## 5.1 OPERANDS

An operand is the part of the instruction that follows the instruction opcode. There can be one or two or even no operands in an instruction. An operand of an assembly instruction has one of the following types:

<u>Operands</u>	<u>Description</u>
expr	any valid expression as described in the section 5.2, <i>Expressions</i> .
reg	any valid register.
symbol	a symbolic name as created by an equate. A symbol can be an expression or a register name.
address	a combination of expr, reg and symbol.
label	a label reference as described in the section 4.1, <i>Input Specification</i> .

If an expression can be completely evaluated at assembly time, it is called an absolute expression; if it is not, it is called a relocatable expression.



See the section 5.2, *Expressions*, for more details.

### 5.1.1 TRICORE ADDRESSING MODES

The TriCore assembly language has several addressing modes. These are listed below with a short description. For details see the TriCore CPU Manual.

#### ***Absolute***

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:

$[An]offset10$

$[An]offset16$

**Pre-increment**

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.

Syntax:

$[+An]offset10$

**Post-increment**

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. Post-decrement is obtained by using a negative offset.

Syntax:

$[An+]offset10$

**Circular**

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:

$[An+c]offset10$

### ***Bit-reverse***

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  $B + \text{reverse}(I)$ . The `reverse()` function exchanges bit  $n$  with bit  $(15-n)$  for  $n = 0, \dots, 7$ . 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:

$[An+r]$

## **5.1.2 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.

## **5.2 EXPRESSIONS**

An operand of an assembler instruction or directive is either an assembler symbol, a register name or an expression. An expression is a sequence of symbols that denotes an address in a particular memory space or a number.

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 expressions**. When any operand of an expression is relocatable the entire expression is relocatable. Relocatable expressions are emitted in the object file and evaluated by the linker or the locator. Relocatable expressions may only contain integral functions. An error is emitted when during object creation non-IEEE relocatable expressions are found.

An expression has a type which depends on the type of the identifiers in the expression. See section 5.2.4, *Expression Type*, for details.

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

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

- *number*
- *expression\_string*
- *symbol*
- *expression binary\_operator expression*
- *unary\_operator expression*
- *( expression )*
- *function*

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.

**Examples:**

```
(3+4)*5      ; Result is 35.
               ; 3 + 4 is evaluated first.
3+(4*5)      ; Result is 23.
               ; 4 * 5 is evaluated first.
               ; parentheses are superfluous
               ; here
```

### 5.2.1 NUMBER

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

*number* can be one of the following:

- *0Bbin\_num*
- *dec\_num*
- *0Xbex\_num*

Lowercase equivalences are allowed: 0b, 0x.

*bin\_num* is a binary number formed of '0'-'1' starting with '0B' or '0b'.

**Examples:** 0B1001; 0B1011; 0b01100100;

*dec\_num* is a decimal number formed of '0'-'9'.

**Examples:**    12; 5978;

*bex\_num*    is a hexadecimal number formed of the characters '0'-'9' and 'a'-'f' or 'A'-'F' starting with a '0X' or '0x'.

**Examples:**    0x45; 0xFFD4; 0x9abc

### 5.2.2 EXPRESSION STRING

An *expression\_string* is a *string* with an arbitrary length evaluating to a number. The value of the string is calculated by taking the first 4 characters padded with 0 to the left.

*string*    is a string of ASCII characters, enclosed in single (') or double (") quotes. The starting and closing quote must be the same. To include the enclosing quote in the string, double it. E.g. the string containing both quotes can be denoted as: `"'"""` or `''''`.

See the chapter *Macro Operations* for the differences between single and double quoted strings.

#### Examples:

```
'A'+1            ; a 1-character ASCII string,
                 ; result 0x42
"9C"+1          ; a 2-character ASCII string,
                 ; result 0x3944
```

### 5.2.3 SYMBOL

A *symbol* is an *identifier*. A *symbol* represents the value of an *identifier* which is already defined, or will be defined in the current source module by means of a label declaration or an equate directive.

#### Examples:

```
CON1 .EQU 3                    ; The variable CON1 represents
                               ; the value of 3

LD.W D0,CON1+20               ; Load D0 with contents of
                               ; address 23
```



When you invoke the assembler, one of the following predefined symbol exists:

- `_ASPCP` contains a string with the name of the assembler ("aspcp")
- `_ASTRI` contains a string with the name of the assembler ("astri")

5.2.4 EXPRESSION TYPE

The type of an expression is either a number (integral) or an address. The result type of an expression depends on the operator and its operands. The tables below summarize all available operators.

Please note:

- 1. a label is of type 'address'; an equate symbol has the type of the equate expression; see section 4.1, *Input Specification*, for a description of labels.
- 2. the type of an untyped symbol can be an address or a number, depending on the context; the result of the operation can be determined using the tables;
- 3. the binary logical and relational operators (`|`, `&`, `==`, `!=`, `<`, `<=`, `>`, `>=`) accept any combination of operands, the result is always the integral number 0 or 1;
- 4. the binary shift and bitwise operators `<<`, `>>`, `|`, `&` and `^` only accept integral operands.

The following table shows the result type of expressions with unary operators (a '\*' indicates an illegal combination).

Operator	integer	addr
~	integer	*
!	integer	*
-	integer	*
+	integer	integer

Table 5-1: Expression type, unary operators

The following table shows the result type of expressions with binary numerical operators.

Operator	integer, integer	addr, integer	integer, addr	addr,addr
–	integer	addr	*	integer
+	integer	addr	addr	*
*	integer	*	*	*
/	integer	*	*	*
%	integer	*	*	*

Table 5-2: Expression type, binary numerical operators



a string operand will be converted to an integral number

The following table shows the result type of functions. A '–' in the column Operands means that the function has no operands.

Function	Operands	Result
@ABS()	integer	integer
@ARG()	symbol integer	integer integer
@ASPCP()	–	string
@ASTRI()	–	string
@CAT()	string,string	string
@CNT()	–	integer
@DEF()	symbol	integer
@HI()	integer	integer
@HIS()	integer	integer
@LEN()	string	integer
@LO()	integer	integer
@LOS()	integer	integer
@LST()	–	integer
@MAC()	symbol	integer
@MAX()	integer,integer,...	integer
@MIN()	integer,integer,...	integer
@MXP()	–	integer
@POS()	string,string string,string,integer	integer integer
@SCP()	string,string	integer

Function	Operands	Result
@SGN()	integer	integer
@SUB()	string, integer, integer	string

Table 5-3: Expression type, functions

5.3 OPERATORS

There are two types of operators:

- unary operators
- binary operators

Operators can be arithmetic operators, shift operators, relational operators, bitwise operators, or logical operators. All operators are described in the following sections.

If the grouping of the operators is not specified with parentheses, the operator precedence is used to determine evaluation order. Every operator has a precedence level associated with it. The following table lists the operators and their order of precedence (in descending order).

Operators	Type
+, -, ~, !	unary
*, /, %	binary
+, -	binary
<<, >>	binary
<, <=, >, >=	unary
==, !=	binary
&	binary
^	binary
	binary
&&	binary
	binary

Table 5-4: Operators Precedence List

Except for the unary operators, the assembler evaluates expressions with operators of the same precedence level left-to-right. The unary operators are evaluated right-to-left. So,  $-4 + 3 * 2$  evaluates to  $(-4) + (3 * 2)$ .

### 5.3.1 ADDITION AND SUBTRACTION

#### **Synopsis:**

Addition:    *operand +operand*

Subtraction: *operand - operand*

The + operator adds its two operands and the – operator subtracts them. The operands can be any expression evaluating to an absolute number or a relocatable operand, with the restrictions of Table 5-2.

#### **Examples:**

```
0xA342  + 23           ; addition of absolute numbers
0xFF1A  - AVAR         ; subtraction with the value of
                        ; symbol AVAR
```

### 5.3.2 SIGN OPERATORS

#### **Synopsis:**

Plus:        *+operand*

Minus:      *-operand*

The + operator does not modify its operand. The – operator subtracts its operand from zero. See also the restrictions in Table 5-1.

#### **Example:**

```
5+-3           ; result is 2
```

5.3.3 MULTIPLICATION AND DIVISION

Synopsis:

Multiplication:	<i>operand</i>	*	<i>operand</i>
Division:	<i>operand</i>	/	<i>operand</i>
Modulo:	<i>operand</i>	%	<i>operand</i>

The \* operator multiplies its two operands, the / operator performs an integer division, discarding any remainder. The % operator also performs an integer division, but discards the quotient and returns the remainder. The operands can be any expression evaluating to an absolute number or a relocatable operand, with the restrictions of Table 5-2. Note that the right operands of the / and % operator may not be zero.

Examples:

```
AVAR*2           ; multiplication
0xFF3C/COUNT     ; division
23%4             ; modulo, result is 3
```

5.3.4 SHIFT OPERATORS

Synopsis:

Shift left:	<i>operand</i>	<<	<i>count</i>
Shift right:	<i>operand</i>	>>	<i>count</i>

These operators shift their left operand (*operand*) either left (<<) or right (>>) by the number of bits (absolute number) specified with the right operand (*count*). The operands can be any expression evaluating to an (integer) number.

Examples:

```
AVAR>>4          ; shift right variable AVAR, 4 times
```

### 5.3.5 RELATIONAL OPERATORS

#### **Synopsis:**

Equal:	<i>operand</i>	<b>==</b>	<i>operand</i>
Not equal:	<i>operand</i>	<b>!=</b>	<i>operand</i>
Less than:	<i>operand</i>	<b>&lt;</b>	<i>operand</i>
Less than or equal:	<i>operand</i>	<b>&lt;=</b>	<i>operand</i>
Greater than:	<i>operand</i>	<b>&gt;</b>	<i>operand</i>
Greater than or equal:	<i>operand</i>	<b>&gt;=</b>	<i>operand</i>

These operators compare their operands and return an absolute number (an integer) of 1 for 'true' and 0 for 'false'. The operands can be any expression evaluating to an absolute number or a relocatable operand.

#### **Examples:**

```
3>=4      ; result is 0 (false)
4==COUNT ; 1 (true), if COUNT is 4.
           ; 0 otherwise.
9<10      ; result is 1 (true)
```

### 5.3.6 BITWISE OPERATORS

#### **Synopsis:**

Bitwise AND:	<i>operand</i>	<b>&amp;</b>	<i>operand</i>
Bitwise OR:	<i>operand</i>	<b> </b>	<i>operand</i>
Bitwise XOR:	<i>operand</i>	<b>^</b>	<i>operand</i>
One's complement		<b>~</b>	<i>operand</i>

The AND, OR and XOR operators take the bitwise AND, OR respectively XOR of the left and right operand. The one's complement (bitwise NOT) operator performs a bitwise complement on its operand. The operands can be any expression evaluating to an (integer) number.

**Examples:**

```

0BH&3    ; result is 3
          0B1011
          0B0011 &
          0B0011

~0xA     ; result is 0xFFFF5
          ~ 0B00000000 00001010
          = 0B11111111 11110101

```

**5.3.7 LOGICAL OPERATORS****Synopsis:**

```

Logical AND:  operand && operand
Logical OR:   operand || operand
Logical NOT:  ! operand

```

The logical AND operator returns an integer 1 if both operands are non-zero; otherwise it returns an integer 0. The logical OR operator returns an integer 1 if either of its operands is non-zero; otherwise it returns an integer 0. The ! operator performs a logical not on its operand. ! returns an integer 1 ('true') if the operand is 0; otherwise, ! returns 0 ('false'). The operands can be any expression evaluating to an integer.

**Examples:**

```

0xB&&3    ; result is 1 (true)

!0xA      ; result is 0 (false)
!(4<3)    ; result is 1 (true)
          ; 4 < 3 result is 0 (false)

```

## 5.4 FUNCTIONS

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

*@function\_name(argument[,argument]...)*

Functions start with the '@' sign and have zero or more arguments, and are always followed by opening and closing parentheses. There must be no intervening spaces between the function name and the opening parenthesis and between the (comma-separated) arguments.

Assembler functions can be grouped into four types:

1. Mathematical functions
2. String functions
3. Macro functions
4. Assembler mode functions
5. Address calculation functions
6. Fractional functions

### 5.4.1 MATHEMATICAL FUNCTIONS

The mathematical functions comprise min/max functions, among others:

<b>ABS</b>	- Absolute value
<b>MAX</b>	- Maximum value
<b>MIN</b>	- Minimum value
<b>SGN</b>	- Return sign

### 5.4.2 STRING FUNCTIONS

String functions compare strings, return the length of a string, and return the position of a substring within a string:

<b>CAT</b>	- Catenate strings
------------	--------------------



- LEN** – Length of string
- POS** – Position of substring in string
- SCP** – Compare strings
- SUB** – Substring from a string

### **5.4.3 MACRO FUNCTIONS**

Macro functions return information about macros:

- ARG** – Macro argument function
- CNT** – Macro argument count
- MAC** – Macro definition function
- MPX** – Macro expansion function

### **5.4.4 ASSEMBLER MODE FUNCTIONS**

Miscellaneous functions having to do with assembler operation:

- ASPCP** – Assembler executable name (**aspcp** only)
- ASTRI** – Assembler executable name (**astri** only)
- DEF** – Symbol definition function
- LST** – LIST control flag value

### **5.4.5 ADDRESS CALCULATION FUNCTIONS**

Functions that are used to calculate the high and low parts of an address:

- LO** – Lower 16 bits of an address
- HI** – Upper 16 bits of an address
- LOS** – Lower 16 bits of an address with adjustment for signed addition
- HIS** – Upper 16 bits of an address with adjustment for signed addition

### 5.4.6 FRACTIONAL FUNCTIONS

Functions that are used to return the fractional representation of an expression:

**FRACT** – Return 32-bit fract

**SFRACT** – Return 16-bit fract

### 5.4.7 DETAILED DESCRIPTION

Individual descriptions of each of the assembler functions follow. They include usage guidelines, functional descriptions, and examples.

#### **@ABS(*expression*)**

Returns the absolute value of *expression* as an integer value.

Example:

```
LD A,#@ABS(VAL) ;load absolute value
```

#### **@ARG(*symbol* | *expression*)**

Returns integer 1 if the macro argument represented by *symbol* or *expression* is present, 0 otherwise. If the argument is a symbol it must be single-quoted and refer to a dummy argument name. If the argument is an expression it refers to the ordinal position of the argument in the macro dummy argument list. A warning will be issued if this function is used when no macro expansion is active.

Example:

```
.IF @ARG('TWIDDLE') ;twiddle factor provided?
```

#### **@ASPCP()**

Returns the name of the assembler executable. This is **aspcp** for the PCP assembler.

Example:

```
ANAME: .byte @ASPCP() ;ANAME = 'aspcp'
```

**@ASTRI()**

Returns the name of the assembler executable. This is **astri** for the TriCore assembler.

Example:

```
ANAME: .byte @ASTRI() ;ANAME = 'astri'
```

**@CAT(*str1*,*str2*)**

Concatenates the two strings into one string. The two strings must be enclosed with single or double quotes.

Example:

```
.DEFINE ID "@CAT('Tri','Core')";ID = 'TriCore'
```

**@CNT()**

Returns the count of the current macro expansion arguments as an integer. A warning will be issued if this function is used when no macro expansion is active.

Example:

```
ARGCNT .SET @CNT() ;squirrel away arg count
```

**@DEF(*symbol*)**

Returns an integer 1 if *symbol* has been defined, 0 otherwise. *symbol* may be any label not associated with a **.MACRO** directive. If *symbol* is quoted it is looked up as a **.DEFINE** symbol; if it is not quoted it is looked up as an ordinary label.

Example:

```
.IF @DEF('ANGLE') ;assemble if ANGLE defined
```

**@FRACT(*expression*)**

This function returns the 32-bit fractional representation (Q31) of the floating point *expression*. The expression must be in the range [-1,+1].

Example:

```
.word @fract(0.1), @fract(-1.0)
```

**@HI(expression)**

Returns the upper 16 bits of a value. **@HI(expr)** is equivalent to **((expr >> 16) & 0xffff)**.

Example:

```
mov.u    d2, #@lo(COUNT)
addih    d2, d2, #@hi(COUNT)    ;upper 16 bits
```

**@HIS(expression)**

Returns the upper 16 bits of a value, adjusted for a signed addition of the lower 16 bits. **@HIS(expr)** is equivalent to **((expr + 0x800) >> 16) & 0xffff)**.

Example:

```
movh.a   a3, #@his(label)
lea      a3, [a3]@los(label)
```

**@LEN(string)**

Returns the length of *string* as an integer.

Example:

```
SLEN     .SET @LEN('string')    ;SLEN = 6
```

**@LO(expression)**

Returns the lower 16 bits of a value. **@LO(expr)** is equivalent to **(expr & 0xffff)**.

Example:

```
mov.u    d2, #@lo(COUNT)    ;lower 16 bits
addih    d2, d2, #@hi(COUNT)
```

**@LOS(expression)**

Returns the lower 16 bits of a value, adjusted for a signed addition. **@LOS(expr)** is equivalent to **((expr + 0x8000) & 0xffff) - 0x8000)**.

Example:

```
movh.a   a3, #@his(label)
lea      a3, [a3]@los(label)
```

**@LST()**

Returns the value of the **LIST** control flag as an integer. Whenever a **LIST ON** control is encountered in the assembler source, the flag is incremented; when a **LIST OFF** control is encountered, the flag is decremented.

Example:

```
.DUP @ABS(@LST()) ;list unconditionally
```

**@MAC(symbol)**

Returns an integer 1 if *symbol* has been defined as a macro name, 0 otherwise.

Example:

```
.IF @MAC(DOMUL) ;expand macro
```

**@MAX(expr1[,exprN]...)**

Returns the greatest of *expr1*,...,*exprN* as an integer.

Example:

```
MAX: .byte @MAX(1,5,-3) ;MAX = 5
```

**@MIN(expr1[,exprN]...)**

Returns the least of *expr1*,...,*exprN* as an integer.

Example:

```
MIN: .byte @MIN(1,5,-3) ;Min = -3
```

**@MXP()**

Returns an integer 1 if the assembler is expanding a macro, 0 otherwise.

Example:

```
.IF @MXP() ;macro expansion active?
```

**@POS(str1,str2[,start])**

Returns the position *str2* in *str1* as an integer, starting at position *start*. If *start* is not given the search begins at the beginning of *str1*. If the *start* argument is specified it must be a positive integer and cannot exceed the length of the source string.

Example:

```
ID .EQU @POS('TriCore','Core') ;ID = 3
```

### **@SCP(*str1, str2*)**

Returns an integer 1 if the two strings compare, 0 otherwise. The two strings must be separated by a comma.

Example:

```
.IF @SCP(STR, 'MAIN') ;does STR equal MAIN?
```

### **@SFRAC(*expression*)**

This function returns the 16-bit fractional representation (Q15) of the floating point *expression*. The expression must be in the range [-1,+1>.

Example:

```
.word @sfract(0.1), @sfract(-1.0)
```

### **@SGN(*expression*)**

Returns the sign of *expression* as an integer: -1 if the argument is negative, 0 if zero, 1 if positive. The *expression* may be relative or absolute.

Example:

```
.IF @SGN(INPUT) == 1 ;is sign positive?
```

### **@SUB(*str, expr1, expr2*)**

Returns the substring from *str* as a string. *expr1* is the starting position within *str* and *expr2* is the length of the desired string. The assembler issues an error if either *expr1* or *expr2* exceeds the length of *str*.

Example:

```
.DEFINE ID "@SUB('TriCore',3,4)" ;ID = 'Core'
```



# OPERANDS & EXPRESSIONS

# CHAPTER

# 6

## **MACRO OPERATIONS**

---





---

# 6

# CHAPTER

---

## **6.1 INTRODUCTION**

This chapter describes the macro operations and conditional assembly.

The macro preprocessor is implemented in the assembler.

## **6.2 MACRO OPERATIONS**

Programming applications frequently involve the coding of a repeated pattern or group of instructions. 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. Having determined the iterated pattern, the programmer can, within the macro, designate selected fields of any statement as variable. Thereafter by invoking a macro the programmer can use the entire pattern as many times as needed, substituting different parameters for the designated variable portions of the statements.

When the pattern is defined it is given a name. This name becomes the mnemonic by which the macro is subsequently invoked (called). If the name of the macro is the same as an existing assembler directive or mnemonic opcode, the macro will replace the directive or mnemonic opcode, and a warning will be issued.

The macro call causes source statements to be generated. The generated statements may contain substitutable arguments. The statements produced by a macro call are relatively unrestricted as to type. They 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 that are applied to statements written by the programmer.

To invoke a macro, the macro name must appear in the operation code field of a source statement. Any arguments are placed in the operand field. By suitably selecting the arguments in relation to their use as indicated by the macro definition, the programmer causes the assembler to produce in-line coding variations of the macro definition.

The effect of a macro call is to produce in-line code to perform a predefined function. The code is inserted in the normal flow of the program so that the generated instructions are executed with the rest of the program each time the macro is called.

An important feature in defining a macro is the use of macro calls within the macro definition. The assembler processes such **nested** macro calls at expansion time only. The nesting of one macro definition within another definition is permitted. However, the nested macro definition will not be processed until the primary macro is expanded. The macro must be defined before its appearance in a source statement operation field.

### 6.3 MACRO DEFINITION

The definition of a macro consists of three parts: the header, which assigns a name to the macro and defines the dummy arguments; the body, which consists of prototype or skeleton source statements; and the terminator. The header is the **.MACRO** directive, its name, and the dummy argument list. The body contains the pattern of standard source statements. The terminator is the **.ENDM** directive.

The header of a macro definition has the form:

*macro\_name* **.MACRO** [*dummy argument list*] [*comment*]

The required name is the symbol by which the macro will be called. The dummy argument list has the form:

[*dumarg* [, *dumarg*] ...]

The dummy arguments are symbolic names that the macro processor will replace with arguments when the macro is expanded (called). Each dummy argument must obey the same rules as global symbol names. Dummy arguments are separated by commas.

When a macro call is executed, the dummy arguments within the macro definition (*reg,value* in the example below) are replaced with the corresponding argument as defined by the macro call.

All local label definitions within a macro which use the local label operator are made unique for this macro call. This is done by appending a unique postfix to every local label, making the scope of the label local to the module. This mechanism allows the programmer to freely use local labels within a macro definition without regard to the number of times that the macro is expanded. Labels without the local label operator are considered to be normal labels and thus cannot occur more than once unless used with the **.SET** directive (see Chapter 7, *Assembler Directives*).

**Example**

The macro:

```
.sdecl  "data", DATA
.sect   "data"

CONSTD  .MACRO  reg,value
mov.u   reg, #@lo(value)
addih   reg, reg, #@hi(value)
.ENDM

CONSTD  d4, 0x12345678
```

expands to:

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

**6.4 MACRO CALLS**

When a macro is invoked the statement causing the action is termed a **macro call**. The syntax of a macro call consists of the following fields:

```
[label:]  macro_name [arguments] [comment]
```

The argument field can have the form:

```
[arg,arg,...]
```

The macro call statement is made up of three besides the comment field: the *label*, if any, will correspond to the value of the location counter at the start of the macro expansion; the operation field which contains the macro name; and the operand field which contains substitutable arguments. Within the operand field each calling argument of a macro call corresponds one-to-one with a dummy argument of the macro definition. For example, the 'load' macro defined earlier could be invoked for expansion (called) by the statement:

```
load    d4, VAL1
```

where the operand field arguments, separated by commas and taken left to right, correspond to the dummy arguments *reg* and *value*, respectively. These arguments are then substituted in their corresponding positions of the definition to produce a sequence of instructions.

Macro arguments consist of sequences of characters separated by commas. Although these can be specified as quoted strings, to simplify coding the assembler does not require single quotes around macro argument strings. However, if an argument has an embedded comma or space, that argument must be surrounded by single quotes ('). An argument can be declared null when calling a macro. However, it must be declared explicitly null. Null arguments can be specified in four ways:

- by writing the delimiting commas in succession with no intervening spaces;
- by terminating the argument list with a comma and omitting the rest of the argument list;
- by declaring the argument as a null string;
- by simply omitting some or all of the arguments.

A null argument will cause no character to be substituted in the generated statements that reference the argument. If more arguments are supplied in the macro call than appear in the macro definition, a warning will be issued by the assembler.

## 6.5 DUMMY ARGUMENT OPERATORS

The assembler macro processor provides for text substitution of arguments during macro expansion. In order to make the argument substitution facility more flexible, the assembler also recognizes certain text operators within macro definitions which allow for transformations of the argument text. These operators can be used for text concatenation, numeric conversion, and string handling.

### 6.5.1 DUMMY ARGUMENT CONCATENATION OPERATOR - \

Dummy arguments that are intended to be concatenated with other characters must be preceded by the concatenation operator, '\ ' to separate them from the rest of the characters. The argument may precede or follow the adjoining text, but there must be no intervening blanks between the concatenation operator and the rest of the characters. To position an argument between two alphanumeric characters, place a backslash both before and after the argument name. For example, consider the following macro definition:

```
SWAP_MEM .macro REG1,REG2      ;swap memory contents
        LD.W  D0,[A\REG1]      ;using D0 as temp
        LD.W  D1,[A\REG2]      ;using D1 as temp
        ST.W  [A\REG1],D1
        ST.W  [A\REG2],D0
        .endm
```

If this macro were called with the following statement,

```
SWAP_MEM 0,1
```

then for the macro expansion, the macro processor would substitute 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 in both cases with the character A. The resulting expansion of this macro call would be:

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

## 6.5.2 RETURN VALUE OPERATOR - ?

Another macro definition operator is the question mark (?) that returns the value of a symbol. When the macro processor encounters this operator, the *?symbol* sequence is converted to a character string representing the decimal value of the *symbol*. For example, consider the following modification of the SWAP\_MEM macro described above:

```
SWAP_MEM .macro REG1,REG2      ;swap memory contents
        LD.W  D0,_lab\?REG1    ;using D0 as temp
        LD.W  D1,_lab\?REG2    ;using D1 as temp
        ST.W  _lab\?REG1,D1
        ST.W  _lab\?REG2,D0
        .endm
```

If the source file contained the following SET statements and macro call,

```
AREG SET 1
BREG SET 2
        SWAP_MEM  AREG,BREG
```

then the sequence of events would be as follows: the macro processor would first substitute the characters AREG for each occurrence of REG1 and BREG for each occurrence of REG2. For discussion purposes (this would never appear on the source listing), the intermediate macro expansion would be:

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

The macro processor would then replace ?AREG with the character X and ?BREG with the character Y, since X is the value of the symbol AREG and Y is the value of BREG. The resulting intermediate expansion would be:

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

Next, the macro processor would apply the concatenation operator (`\`), and the resulting expansion as it would appear on the source listing would be:

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

### 6.5.3 RETURN HEX VALUE OPERATOR - `%`

The percent sign (`%`) is similar to the standard return value operator except that it returns the hexadecimal value of a symbol. When the macro processor encounters this operator, the `%symbol` sequence is converted to a character string representing the hexadecimal value of the *symbol*. Consider the following macro definition:

```
GEN_LABEL    .macro  LAB, VAL, STMT
LAB\%VAL:    STMT
            .endm
```

This macro generates a label consisting of the concatenation of the label prefix argument and a value that is interpreted as hexadecimal. If this macro were called as follows,

```
NUM .SET      10
    GEN_LABEL  HEX, NUM, 'NOP'
```

the macro processor would first substitute the characters `HEX` for `LAB`, then it would replace `%VAL` with the character `A`, since `A` is the hexadecimal representation for the decimal integer `10`. Next, the macro processor would apply the concatenation operator (`\`). Finally, the string `'NOP'` would be substituted for the `STMT` argument. The resulting expansion as it would appear in the listing file would be:

```
HEXA: NOP
```

The percent sign is also the character used to indicate a binary constant. If a binary constant is required inside a macro it may be necessary to enclose the constant in parentheses or escape the constant by following the percent sign by a backslash (`\`).



#### 6.5.4 DUMMY ARGUMENT STRING OPERATOR - "

Another dummy argument operator is the double quote ("). This character is replaced with a single quote by the macro processor, but following characters are still examined for dummy argument names. The effect in the macro call is to transform any enclosed dummy arguments into literal strings. For example, consider the following macro definition:

```
STR_MAC .macro STRING
        .ascii  "STRING"
        .endm
```

If this macro were called with the following macro expansion line,

```
STR_MAC  ABCD
```

then the resulting macro expansion would be:

```
.ascii  'ABCD'
```

Double quotes also make possible **.DEFINE** directive expansion within quoted strings. Because of this overloading of the double quotes, care must be taken to insure against inappropriate expansions in macro definitions. Since **.DEFINE** expansion occurs before macro substitution, any **.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 this macro were invoked as follows,

```
STR_MAC  sentence
```

then the resulting expansion would be:

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

### 6.5.5 MACRO LOCAL LABEL OPERATOR - ^

It may be desirable to pass a name as a macro argument to be used as a local address reference within the macro body. If a circumflex (^) precedes an identifier then the macro preprocessor will perform name mangling on that label so the label is used literally in the resulting macro expansion. Here is an example:

```
LOAD .macro ADDR
    LD.W  D0, ^ADDR
.endm
```

The macro ^-operator performs name mangling on the ADDR argument. Consider the following macro call:

```
_LOCAL: LOAD  _LOCAL
```

With the local label in the macro definition the macro LOAD would expand to the something like this:

```
_LOCAL:
    LD.W  D0, _LOCAL__M_L000001
```

This would result in an assembly error as the label LOCAL\_\_M\_L000001 is nowhere defined. Without the local label operator in the macro definition (as shown above) the macro LOAD would expand, as expected, to this:

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

This will assemble correctly.

### 6.6 .DUP, .DUPA, .DUPC, .DUPF DIRECTIVES

The **.DUP**, **.DUPA**, **.DUPC**, and **.DUPF** directives are specialized macro forms. They can be thought of 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, including (in the case of **.DUPA**, **.DUPC**, and **.DUPF**) the dummy operator characters described previously. For a detailed description of these directives, refer to Chapter 7, *Assembler Directives*.

## 6.7 CONDITIONAL ASSEMBLY

Conditional assembly facilitates the writing of comprehensive source programs that can cover many conditions. Assembly conditions may be specified through the use of arguments in the case of macros, and through definition of symbols via the **.DEFINE**, **.SET**, and **.EQU** directives. Variations of parameters can then cause assembly of only those parts necessary for the given conditions. The built-in functions of the assembler provide a versatile means of testing many conditions of the assembly environment (see section 5.4 for more information on the assembler built-in functions).

Conditional directives can also be used within a macro definition to ensure at expansion time that arguments fall within a range of allowable 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:

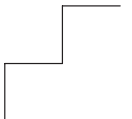
```
.IF expression
.
.
[.ELIF expression]
.      ;(the .ELIF directive is optional and
.      ; can be read as 'else if')
.
[.ELSE] ;(the .ELSE directive is optional)
.
.
.ENDIF
```

A section of a program that is to be conditionally assembled must be bounded by an **IF-ENDIF** directive pair. If the optional **.ELSE** or **.ELIF** directive is not present, then the source statements following the **.IF** directive and up to the next **.ENDIF** directive will be included as part of the source file being assembled only if the *expression* had a nonzero result. If the *expression* has a value of zero, the source file will be assembled as if those statements between the **.IF** and the **.ENDIF** directives were never encountered. If the **.ELSE** directive is present and *expression* has a nonzero result, then the statements between the **.IF** and **.ELSE** directives will be assembled, and the statement between the **.ELSE** and **.ENDIF** directives will be skipped. Alternatively, if *expression* has a value of zero, then the statements between the **.IF** and **.ELSE** directives will be skipped, and the statements between the **.ELSE** and **.ENDIF** directives will be assembled.

# CHAPTER 7

## ASSEMBLER DIRECTIVES

---



---

# 7 | CHAPTER

---

## 7.1 OVERVIEW

Assembler directives, or pseudo instructions, are used to control the assembly process. Rather than being translated into a TriCore or PCP machine instruction, assembler directives are interpreted by the assembler. The directives perform actions such as assembly control, listing control, defining symbols or changing the location counter. Upper and lower case letters are considered equivalent for assembler directives.



All directives valid for the TriCore assembler **astri** also apply to the PCP assembler **aspcp**, unless explicitly stated otherwise.

Assembler directives can be grouped by function into five types:

1. Debugging
2. Assembly control
3. Symbol definition
4. Data definition/storage allocation
5. Macros and conditional assembly

### 7.1.1 DEBUGGING

The compiler generates the following directives to pass high level language symbolic debug information via the assembler into the object file:

- |               |   |
|---------------|---|
| <b>.CALLS</b> | – Pass call information to object file. Used to build a call tree at link time. |
| <b>.SYMB</b>  | – Pass symbolic debug information   |

### 7.1.2 ASSEMBLY CONTROL

The directives used for assembly control are:

- |                 |  |
|-----------------|--|
| <b>.ALIGN</b>   | – Specify alignment  |
| <b>.COMMENT</b> | – Start comment lines. This directive is not permitted in .IF/.ELIF/.ELSE/.ENDIF constructs and .MACRO/.DUP definitions. |

<b>.DEFINE</b>	– Define substitution string
<b>.END</b>	– End of source program
<b>.FAIL</b>	– Programmer generated error message
<b>.INCLUDE</b>	– Include secondary file
<b>.MESSAGE</b>	– Programmer generated message
<b>.NAME</b>	– Identification name for object file
<b>.ORG</b>	– Define a nameless section
<b>.SDECL</b>	– Define section name and attributes
<b>.SECT</b>	– Activate section
<b>.UNDEF</b>	– Undefine <b>.DEFINE</b> symbol
<b>.WARNING</b>	– Programmer generated warning

### **7.1.3 SYMBOL DEFINITION**

The directives used to control symbol definition are:

<b>.EQU</b>	– Equate symbol to a value; accepts forward references
<b>.EXTERN</b>	– External symbol declaration; also permitted in module body
<b>.GLOBAL</b>	– Global symbol declaration; also permitted in module body
<b>.LOCAL</b>	– Local symbol declaration
<b>.SET</b>	– Set symbol to a value; accepts forward references

### **7.1.4 DATA DEFINITION/STORAGE ALLOCATION**

The directives used to control constant data definition and storage allocation are:

<b>.ACCUM</b>	– Define 64-bit constant of 17Q46 format
<b>.ASCII</b>	– Define ASCII string
<b>.ASCIIZ</b>	– Define NULL padded ASCII string

<b>.BYTE</b>	– Define constant byte
<b>.DOUBLE</b>	– Define 64-bit floating-point constant
<b>.FLOAT</b>	– Define 32-bit floating-point constant
<b>.FRACT</b>	– Define 32-bit constant fraction
<b>.HALF</b>	– Define constant half word
<b>.SFRACT</b>	– Define 16-bit constant fraction
<b>.SPACE</b>	– Define storage
<b>.WORD</b>	– Define constant word

### **7.1.5 MACROS AND CONDITIONAL ASSEMBLY**

The directives used for macros and conditional assembly are:

<b>.DUP</b>	– Duplicate sequence of source lines
<b>.DUPA</b>	– Duplicate sequence with arguments
<b>.DUPC</b>	– Duplicate sequence with characters
<b>.DUPF</b>	– Duplicate sequence in loop
<b>.ENDIF</b>	– End of conditional assembly
<b>.ENDM</b>	– End of macro definition
<b>.EXITM</b>	– Exit macro
<b>.IF</b>	– Conditional assembly directive
<b>.MACRO</b>	– Macro definition
<b>.PMACRO</b>	– Purge macro definition

## **7.2 DIRECTIVES**

The rest of this chapter contains an alphabetical list of the assembler directives.



# .ACCUM

## Syntax:

```
[label:] .ACCUM arg[,arg]...
```

## Description:

Define 64-bit constant(s) of 17Q46 format. The **.ACCUM** directive allocates and initializes two words (64-bits) of memory for each *arg* argument. *arg* can be a constant, a symbol, or an expression. The **.ACCUM** directive may have one or more arguments separated by commas. Multiple arguments are stored in successive address locations. If multiple arguments are present, one or more of them can be null (two adjacent commas), in which case the corresponding address location will be filled with zeros. An error will occur if the evaluated argument value is out of the range  $[-2^{17}, 2^{17}]$ .

*label*, if present, will be assigned the value of the run-time location counter at the start of the directive processing.

## Examples:

```
ACC: .accum 0.1,0.2,0.3
```



```
.fract, .sfract
```

# .ALIGN

## Syntax:

**.ALIGN** *expression*

## Description:

Align the location counter. The *expression* must be represented by a value of  $2^k$ . The default alignment is on a multiple of 1 byte. *expression* must be greater than 0. If *expression* is not a value of  $2^k$ , a warning is issued and the alignment will be set to the next  $2^k$  value. Alignment will be performed once at the place where you write the **.align** pseudo. The start of a section is aligned automatically to the largest alignment value occurring in that section.

Depending on the section type the assembler has two cases for this directive.

- Relocatable sections  
The section will be aligned on the calculated alignment boundary. A gap is generated depending on the current relative location counter for this section.
- Absolute sections  
The section location is not changed.  
A gap is generated according to the current absolute address.

## Examples:

```
.align 4           ;align at 4 bytes
lab1: .align 6      ;not a 2k value.
                   ;a warning is issued
                   ;lab1 is aligned on 8 bytes
```

# .ASCII

## Syntax:

[*label*:] **.ASCII** *string* [, *string*]...

## Description:

Define list of ASCII characters. The **.ASCII** directive allocates and initializes an array of memory for each *string* argument. No NULL byte is added to the end of the array. Therefore, the behavior is identical to the **.BYTE** directive with a string argument.

## Examples:

```
HELLO: .ascii "Hello world"  
;Is the same as .byte "Hello world"
```



**.asciiz**, **.byte**

# .ASCIIZ

## Syntax:

[*label*:] **.ASCIIZ** *string* [, *string*]...

## Description:

Define list of ASCII characters. The **.ASCIIZ** directive allocates and initializes an array of memory for each *string* argument. A NULL byte is added to the end of each array.

## Examples:

```
HELLO: .ascii "Hello world"  
      ;Is the same as .byte "Hello world",0
```



**.ascii, .byte**

# .BYTE

## Syntax:

*[label:]* **.BYTE** *arg[,arg]...*

## Description:

Define Constant Byte. The **.BYTE** directive allocates and initializes a byte of memory for each *arg* argument. *arg* can be a numeric constant, a single or multiple character string constant, a symbol, or an expression. The **.BYTE** directive may have one or more arguments separated by commas. Multiple arguments are stored in successive address locations. If multiple arguments are present, one or more of them can be null (two adjacent commas), in which case the corresponding address location will be filled with zeros. An error will occur if the evaluated argument value is too large to represent in a single byte.

*label*, if present, will be assigned the value of the run-time location counter at the start of the directive processing.

Integer arguments are stored as is, but must be byte values (e.g. within the range 0–255). Single and multiple character strings are handled in the following manner:

1. Single character strings are stored in a byte whose bits represent the ASCII value of the character.

Example: 'R' = 0x52

2. Multiple character strings represent bytes composed of the ASCII representation of the characters in the string.

Example:

'ABCD' = 0x41, 0x42, 0x43, 0x44

## Examples:

TABLE: **.byte** 14,253,0x62,'ABCD'

CHARS: **.byte** 'A','B','C','D'



**.space, .half, .word**

# .CALLS

## Syntax:

**.CALLS** '*caller*', '*callee*'

## Description:

Create a flow graph reference between *caller* and *callees*. The linker needs this information to build a flow graph. *caller* and *callee* are names of functions.

## Examples:

```
.calls 'main', 'nfunc'
```



See also the section *Linker Output* in the chapter *Linker*.

# .COMMENT

## Syntax:

```
.COMMENT    delimiter
.
delimiter
```

## Description:

Start Comment Lines. The **.COMMENT** directive is used to define one or more lines as comments. The first non-blank character after the **.COMMENT** directive is the comment delimiter. The two delimiters are used to define the comment text. The line containing the second comment delimiter will be considered the last line of the comment. The comment text can include any printable characters and the comment text will be produced in the source listing as it appears in the source file.

A label is not allowed with this directive.

This directive is not permitted in **.IF/.ELIF/.ELSE/.ENDIF** constructs and **.MACRO/.DUP** definitions.

## Examples:

```
.comment    + This is a one line comment +
.comment    * This is a multiple line
               comment. Any number of lines
               can be placed between the two
               delimiters.
               *
```

# .DEFINE

## Syntax:

**.DEFINE** *symbol* *string*

## Description:

Define Substitution String. The **.DEFINE** directive is used to define substitution strings that will be used on all following source lines. All succeeding lines will be searched for an occurrence of *symbol*, which will be replaced by *string*. This directive is useful for providing better documentation in the source program. *symbol* must adhere to the restrictions for labels. That is, the first character must be alphabetic or the underscore (`_`), and the remainder of which must be either alphanumeric or the underscore (`_`). A warning will result if a new definition of a previously defined symbol is attempted.

Macros represent a special case. **.DEFINE** directive translations will be applied to the macro definition as it is encountered. When the macro is expanded any active **.DEFINE** directive translations will again be applied.

A label is not allowed with this directive.

## Examples:

If the following **.DEFINE** directive occurred in the first part of the source program:

```
.define ARRAYSIZ '10 * SAMPLSIZ'
```

then the source line below:

```
.space ARRAYSIZ
```

would be transformed by the assembler to the following:

```
.space 10 * SAMPLSIZ
```



**.undef**



# .DOUBLE

## Syntax:

[*label*:] **.DOUBLE** *arg*[,*arg*]...

## Description:

Define 64-bit floating-point constant(s) in IEEE-754 double precision format. The **.DOUBLE** directive allocates and initializes a double-word (64-bits) of memory for each *arg* argument. *arg* can be a constant, a symbol, or an expression. The constant can be expressed as a signed whole number with fraction or the "e" format as that in the C language. 12.457 and +0.27E-13 are legal floating-point constants.

The **.DOUBLE** directive may have one or more arguments separated by commas. Multiple arguments are stored in successive address locations. If multiple arguments are present, one or more of them can be null (two adjacent commas), in which case the corresponding address location will be filled with zeros. An error will occur if the evaluated argument value is too large to represent in a single word.

*label*, if present, will be assigned the value of the run-time location counter at the start of the directive processing.

## Examples:

DBL: **.double** 0.1,0.2,0.3



**.float**

# .DUP

## Syntax:

```
[label:].DUP expression
.
.
.ENDM
```

## Description:

Duplicate Sequence of Source Lines. The sequence of source lines between the **.DUP** and **.ENDM** directives will be duplicated by the number specified by the integer *expression*. If the expression evaluates to a number less than or equal to 0, the sequence of lines will not be included in the assembler output. The expression result must be an absolute integer and cannot contain any forward references to address labels (labels that have not already been defined). The **.DUP** directive may be nested to any level.

*label*, if present, will be assigned the value of the run-time location counter at the start of the **.DUP** directive processing.

## Examples:

The sequence of source input statements,

```
COUNT .set 3
      .dup COUNT          ; NOP BY COUNT
      NOP
      .endm
```

would generate the following in the source listing:

```
COUNT .set 3
      .dup COUNT          ; NOP BY COUNT
      NOP
      .endm

      ; NOP
      ; NOP
      ; NOP
```



**.dupa, .dupc, .dupf, .endm, .macro**

# .DUPA

## Syntax:

```
[label:].DUPA  dummy,arg[,arg]...
.
.
.ENDM
```

## Description:

Duplicate Sequence With Arguments. The block of source statements defined by the **.DUPA** and **.ENDM** directives will be repeated for each argument. For each repetition, every occurrence of the dummy parameter within the block is replaced with each succeeding argument string. If the argument string is a null, then the block is repeated with each occurrence of the dummy parameter removed. If an argument includes an embedded blank or other assembler-significant character, it must be enclosed with single quotes.

*label*, if present, will be assigned the value of the run-time location counter at the start of the **.DUPA** directive processing.

## Examples:

If the input source file contained the following statements,

```
.dupa  VALUE,12,32,34
.byte  VALUE
.endm
```

then the assembler source listing would show

```
.dupa  VALUE,12,32,34
.byte  VALUE
.endm

;      .byte  12
;      .byte  32
;      .byte  34
```



**.dup, .dupc, .dupf, .endm, .macro**

# .DUPC

## Syntax:

```
[label:].DUPC    dummy,string  
.  
.  
.ENDM
```

## Description:

Duplicate Sequence With Characters. The block of source statements defined by the **.DUPC** and **.ENDM** directives will be repeated for each character of *string*. For each repetition, every occurrence of the dummy parameter within the block is replaced with each succeeding character in the string. If the string is null, then the block is skipped.

*label*, if present, will be assigned the value of the run-time location counter at the start of the **.DUPC** directive processing.

## Examples:

If the input source file contained the following statements,

```
.dupc VALUE, '123'  
.byte VALUE  
.endm
```

then the assembler source listing would show

```
.dupc VALUE, '123'  
.byte VALUE  
.endm  
  
;      .byte  1  
;      .byte  2  
;      .byte  3
```



**.dup**, **.dupa**, **.dupf**, **.endm**, **.macro**

# .DUPF

## Syntax:

```
[label:].DUPF    dummy,[start],end[,increment]
.
.
.ENDM
```

## Description:

Duplicate Sequence In Loop. The block of source statements defined by the **.DUPF** and **.ENDM** directives will be repeated in general  $(end - start) + 1$  times when *increment* is 1. *start* is the starting value for the loop index; *end* represents the final value. *increment* is the increment for the loop index; it defaults to 1 if omitted (as does the *start* value). The *dummy* parameter holds the loop index value and may be used within the body of instructions.

*label*, if present, will be assigned the value of the run-time location counter at the start of the **.DUPF** directive processing.

## Examples:

If the input source file contained the following statements,

```
.dupf NUM, 0, 3
.WORD NUM
.endm
```

then the assembler source listing would show

```
.dupf NUM, 0, 3
.WORD NUM
.endm

;      .WORD  0
;      .WORD  1
;      .WORD  2
;      .WORD  3
```



**.dup, .dupa, .dupc, .endm, .macro**

# **.END**

## **Syntax:**

**.END**

## **Description:**

End of Source Program. The optional **.END** directive indicates that the logical end of the source program has been encountered. The **.END** directive cannot be used in a macro expansion.

A label is not allowed with this directive.

## **Examples:**

```
.end           ;End of source program
```

# .ENDIF

## Syntax:

**.ENDIF**

## Description:

End Of Conditional Assembly. The **.ENDIF** directive is used to signify the end of the current level of conditional assembly. Conditional assembly directives can be nested to any level, but the **.ENDIF** directive always refers to the most previous **.IF** directive.

A label is not allowed with this directive.

## Examples:

```
.if DEB  
; Report building of the debug version  
.message 'Debug Version'  
.endif
```



**.if**

# .ENDM

## Syntax:

**.ENDM**

## Description:

End of Macro Definition. Every **.MACRO**, **.DUP**, **.DUPA**, and **.DUPC** directive must be terminated by an **.ENDM** directive.

A label is not allowed with this directive.

## Examples:

```
SWAP_MEM .macro REG1,REG2           ;swap memory contents
    LD.W  D0,[A\REG1]                ;using D0 as temp
    LD.W  D1,[A\REG2]                ;using D1 as temp
    ST.W  [A\REG1],D1
    ST.W  [A\REG2],D0
.endm
```



**.dup**, **.dupa**, **.dupc**, **.macro**



# .EQU

## Syntax:

*name* **.EQU** *expression*

## Description:

Equate Symbol to a Value. The **.EQU** directive assigns the value of *expression* to the symbol *name*.

The **.EQU** directive is one of the directives that assigns a value other than the program counter to the name. The symbol name cannot be redefined anywhere else in the program. The *expression* may be relative or absolute, and forward references are allowed.

An **.EQU** symbol can be made global.

## Examples:

```
A_D_PORT .equ 0x4000
```

This would assign the value 0x4000 to the symbol A\_D\_PORT.



**.set**

# **.EXITM**

## **Syntax:**

**.EXITM**

## **Description:**

Exit Macro. The **.EXITM** directive will cause immediate termination of a macro expansion. It is useful when used with the conditional assembly directive **.IF** to terminate macro expansion when error conditions are detected.

A label is not allowed with this directive.

## **Examples:**

```
CALC .macro XVAL,YVAL
    .if XVAL<0
    .message 'Macro parameter value out of range'
    .exitm      ;Exit macro
    .endif
    .
    .
    .
    .endm
```



**.dup, .dupa, .dupc, .macro**

# .EXTERN

## Syntax:

**.EXTERN** [(*attrib* [, *attrib*] ...)] *symbol* [, *symbol*] ...

## Description:

External Symbol Declaration. The **.EXTERN** directive is used to specify that the list of symbols is referenced in the current module, but is not defined within the current module. These symbols must either have been defined outside of any module or declared as globally accessible within another module using the **.GLOBAL** directive.

The optional argument *attrib* can be one of the following symbol attributes:

<b>CODE</b>	symbol is in ROM
<b>DATA</b>	symbol is in RAM
<b>ABS24</b>	symbol is in direct addressable code memory ( <b>astri</b> only)
<b>ABS18</b>	symbol is in direct addressable data memory ( <b>astri</b> only)
<b>FPI</b>	symbol is in TriCore code or data memory ( <b>aspcp</b> only)

If the **.EXTERN** directive is not used to specify that a symbol is defined externally and the symbol is not defined within the current module, a warning is generated, and an **.EXTERN** symbol is inserted.

A label is not allowed with this directive.

## Examples:

```
.extern AA,CC,DD           ;defined elsewhere
.extern (ABS24) EE         ;within direct addr.
                           ;code memory (astri)

.extern (CODE,FPI) FF      ;within TriCore
                           ;code memory (aspcp)
```



**.global**

# .FAIL

## Syntax:

**.FAIL** [*str* | *exp*][*str* | *exp*][...]

## Description:

Programmer Generated Error. The **.FAIL** directive will cause an error message to be output by the assembler. The total error count will be incremented as with any other error. The **.FAIL** directive is normally used in conjunction with conditional assembly directives for exceptional condition checking. The assembly proceeds normally after the error has been printed. An arbitrary number or strings and expressions, in any order but separated by commas, can be specified optionally to describe the nature of the generated error.

A label is not allowed with this directive.

## Examples:

```
.fail 'Parameter out of range'
```



```
.message, .warning
```

# .FLOAT

## Syntax:

```
[label:] .FLOAT arg[,arg]...
```

## Description:

Define 32-bit floating-point constant(s) in IEEE-754 single precision format. The **.FLOAT** directive allocates and initializes a word (32-bits) of memory for each *arg* argument. *arg* can be a constant, a symbol, or an expression. The constant can be expressed as signed whole number with fraction or the "e" format as that in the C language. 12.457 and +0.27E-13 are legal floating-point constants.

The **.FLOAT** directive may have one or more arguments separated by commas. Multiple arguments are stored in successive address locations. If multiple arguments are present, one or more of them can be null (two adjacent commas), in which case the corresponding address location will be filled with zeros. An error will occur if the evaluated argument value is too large to represent in a single word.

*label*, if present, will be assigned the value of the run-time location counter at the start of the directive processing.

## Examples:

```
flt: .float 0.1,0.2,0.3
```



```
.double
```

# .FRACT

## Syntax:

*[label:]*    **.FRACT**   *arg*[,*arg*]...

## Description:

Define 32-bit constant fraction of Q31. The **.FRACT** directive allocates and initializes a word (32-bits) of memory for each *arg* argument. *arg* can be a constant, a symbol, or an expression.

The **.FRACT** directive may have one or more arguments separated by commas. Multiple arguments are stored in successive address locations. If multiple arguments are present, one or more of them can be null (two adjacent commas), in which case the corresponding address location will be filled with zeros. An error will occur if the evaluated argument value is out the range of  $[-1,1>$ .

*label*, if present, will be assigned the value of the run-time location counter at the start of the directive processing.

## Examples:

FRCT:    **.fract**   0.1,0.2,0.3



**.accum, .sfract**

# .GLOBAL

## Syntax:

**.GLOBAL**    *symbol[,symbol]...*

## Description:

Global Section Symbol Declaration. The **.GLOBAL** directive is used to specify that the list of symbols is defined within the current section or module, and that those definitions should be accessible by all modules. If the symbols that appear in the operand field are not defined in the module, an error will be generated. Symbols that are defined "global" are accessible from other modules using the **.EXTERN** directive.

A label is not allowed with this directive.

Only program labels and **.EQU** labels can be made global.

## Examples:

```
.global LOOPA           ;LOOPA will be globally  
                        ;accessible by other modules
```



**.extern, .local**

# .HALF

## Syntax:

[*label*:].**HALF***arg*[,*arg*]...

## Description:

Define Constant Half Word. The **.HALF** directive allocates and initializes a half-word of memory for each *arg* argument. *arg* can be a numeric constant, a single or double character string constant, a symbol, or an expression. The **.HALF** directive may have one or more arguments separated by commas. Multiple arguments are stored in successive address locations. If multiple arguments are present, one or more of them can be null (two adjacent commas), in which case the corresponding address location will be filled with zeros. An error will occur if the evaluated argument value is too large to represent in a single half-word.

*label*, if present, will be assigned the value of the run-time location counter at the start of the directive processing.



Half-word values are stored in memory with the least significant 8 bits on the lowest address (little endian).

Integer arguments are stored as is. Single and multiple character strings are handled in the following manner:

1. Single character strings are stored in a half-word whose lower seven bits represent the ASCII value of the character.

Example: 'R' = 0x52

2. Multiple character strings consisting of more than two characters are not allowed. Two-character strings are stored as if the ASCII value of the first character is the high byte value of the half-word. The second character is used as the low byte.

Example:

'AB' = 0x4142



**Examples:**

```
TABLE:  .half 14,1635,0x2662,'AB'
```

is equal to

```
TABLE:  .byte 14,0,1635%256,6,0x62,0x26,'B','A'
```



**.byte, .space, .word**

# .IF

## Syntax:

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

## Description:

Conditional Assembly Directive. Part of a program that is to be conditionally assembled must be bounded by an **IF-ENDIF** directive pair. If the optional **.ELSE** or **.ELIF** (can be read as 'else if') directive is not present, then the source statements following the **.IF** directive and up to the next **.ENDIF** directive will be included as part of the source file being assembled only if the *expression* has a nonzero result. If the *expression* has a value of zero, the source file will be assembled as if those statements between the **.IF** and the **.ENDIF** directives were never encountered. If the **.ELSE** directive is present and *expression* has a nonzero result, then the statements between the **.IF** and **.ELSE** directives will be assembled, and the statements between the **.ELSE** and **.ENDIF** directives will be skipped. Alternatively, if *expression* has a value of zero, then the statements between the **.IF** and **.ELSE** directives will be skipped, and the statements between the **.ELSE** and **.ENDIF** directives will be assembled.

The *expression* must have an absolute integer result and is considered true if it has a nonzero result. The *expression* is false only if it has a result of 0. Because of the nature of the directive, *expression* must be known on pass one (no forward references allowed). **.IF** directives can be nested to any level. The **.ELSE** directive will always refer to the nearest previous **.IF** directive as will the **.ENDIF** directive.

A label is not allowed with this directive.

**Examples:**

```
.if XVAL<0  
.message  'Please select larger value for XVAL'  
.endif
```



```
.endif
```

# .INCLUDE

## Syntax:

**.INCLUDE** *string* | *<string>*

## Description:

Include Secondary File. This directive is inserted into the source program at any point where a secondary file is to be included in the source input stream. The string specifies the filename of the secondary file. The filename must be compatible with the operating system and can include a directory specification.

The file is searched for first in the current directory, unless the *<string>* syntax is used, or in the directory specified in *string*. If the file is not found, and the **-I** option was used on the command line that invoked the assembler, then the string specified with the **-I** option is prefixed to *string* and that directory is searched. If the *<string>* syntax is given, the file is searched for only in the directories specified with the **-I** option.

A label is not allowed with this directive.

## Examples:

```
.include 'headers/io.asm'
.include <data.asm>           ; Do not look in
                               ; current directory
```

# .LOCAL

## Syntax:

**.LOCAL** *symbol*[*symbol*]...

## Description:

Local Section Symbol Declaration. The **.LOCAL** directive is used to specify that the list of symbols is defined within the current module, and that those definitions are explicitly local to that section or module. It is useful in cases where a symbol may not be exported outside of the module (as labels in a module are defined "global" by default). If the symbols that appear in the operand field are not defined in the module, an error will be generated.

A label is not allowed with this directive.

## Examples:

```
.local LOOPA      ;LOOPA local to this module
```



```
.global
```

# .MACRO

## Syntax:

```
name .MACRO [dummy argument list]  
.  
  macro definition statements  
.  
.  
.endm
```

## Description:

Macro Definition. The dummy argument list has the form:

*[dumarg[,dumarg]...]*

The required name is the symbol by which the macro will be called.

The definition of a macro consists of three parts: the header, which assigns a name to the macro and defines the dummy arguments; the body, which consists of prototype or skeleton source statements; and the terminator.

The header is the **.MACRO** directive, its name, and the dummy argument list. The body contains the pattern of standard source statements. The terminator is the **.ENDM** directive.

The dummy arguments are symbolic names that the macro processor will replace with arguments when the macro is expanded (called). Each dummy argument must obey the same rules as symbol names. Within each of the three dummy argument field, the dummy arguments are separated by commas. The dummy argument fields are separated by one or more blanks.

Macro definitions may be nested but the nested macro will not be defined until the primary macro is expanded.

Chapter 6, *Macro Operations*, contains a complete description of macros.

**Examples:**

```

SWAP_MEM .macro REG1,REG2           ;swap memory contents
    LD.W  D0,[A\REG1]               ;using D0 as temp
    LD.W  D1,[A\REG2]               ;using D1 as temp
    ST.W  [A\REG1],D1
    ST.W  [A\REG2],D0
.endm

```



```
.dup, .dupa, .dupc, .dupf, .endm
```

# .MESSAGE

## Syntax:

**.MESSAGE**      [{*str* | *exp*},{*str* | *exp*}...]

## Description:

Programmer Generated Message. The **.MESSAGE** directive will cause a message to be output by the assembler. The error and warning counts will not be affected. The **.MESSAGE** directive is normally used in conjunction with conditional assembly directives for informational purposes. The assembly proceeds normally after the message has been printed. An arbitrary number of strings and expressions, in any order but separated by commas, can be specified optionally to describe the nature of the message.

A label is not allowed with this directive.

## Examples:

**.message**    'Generating tables'



**.fail, .warning**



# .NAME

## Syntax:

```
.NAME "str"
```

## Description:

The **.NAME** directive is used by the assembler to give an identification to the produced object file. The linker and locator can then use this information to identify the source within the map files. Also a debugger may display the value as a 'module' name.

When this directive is omitted, the assembler will use the module's source name as an identification. When using the control program, this name might become a 'random' name.

## Examples:

```
.name "strcat"           ;object is identified by the  
                           ;name "strcat"
```

# .ORG

## Syntax:

**.ORG** [*abs-loc*][,*sect\_type*][,*attrib*]...

## Description:

The **.ORG** directive is used to define a nameless section. This is the same as a **.SDECL/.SECT** without a section name. The absolute location of the section in memory can be set using this directive.

*abs-loc* Initial value to assign to the runtime location counter. *abs-loc* must be an absolute expression.

*sect\_type* An optional section type:

<b>code</b>	a code section
<b>data</b>	a data section

*attrib* An optional section attribute:

<b>abs24</b>	direct addressable code ( <b>astri</b> only)
<b>abs18</b>	direct addressable data ( <b>astri</b> only)
<b>fpi</b>	addressable through FPI bus ( <b>aspcp</b> only)
<b>clear</b>	clear section during program startup
<b>noclear</b>	section is not cleared during startup
<b>init</b>	initialization data copied from ROM to RAM at startup
<b>max</b>	common, overlay with other parts with the same name, is implicit a type of 'noclear'
<b>romdata</b>	section contains data instead of executable code

**Examples:**

```
; define a section on location 100 decimal
.org    100

; define a relocatable nameless section
.org

; define a relocatable data section
.org    ,data

; define a data section on 0x8000
.org    0x8000,data
```



```
.sdecl
```

# .PMACRO

## Syntax:

**.PMACRO** *symbol[,symbol]...*

## Description:

Purge Macro Definition. The specified macro definition will be purged from the macro table, allowing the macro table space to be reclaimed.

A label is not allowed with this directive.

## Examples:

**.pmacro** MAC1,MAC2

This statement would cause the macros named MAC1 and MAC2 to be purged.



**.macro**

# .SDECL

## Syntax:

**.SDECL** *section*, *type* [, *attr*]... [**AT** *address*]

## Description:

Use this directive to define section names and declaration attributes. Before any code or data can be placed in a section, you must use the **.SECT** directive to activate the section. The definition can have declaration attributes and must have a section type (*type*).

The section type can be:

*type*:           **DATA** | **CODE**

The section declaration attribute can be:

*attr*:

Group1:	<b>ABS24</b>   <b>ABS18</b>	( <b>astri</b> only)
	<b>FPI</b>	( <b>aspcp</b> only)
Group2:	<b>ROMDATA</b>   <b>NOCLEAR</b>   <b>CLEAR</b>	
	<b>INIT</b>   <b>MAX</b>	

For each group one attribute can be specified at the most. The **ABS24** and **ABS18** attributes specify direct addressable CODE or DATA memory for the **astri** assembler. **FPI** is only valid for the **aspcp** assembler; this attribute specifies that a section must be accessed through the FPI bus. This section will not be located in PCP internal memory, but in TriCore memory.

**CLEAR** sections are zeroed at startup. This attribute can only be used on a DATA type section.

Sections with the **NOCLEAR** attribute are not zeroed at startup. This is a default attribute for DATA sections. The attribute can only be used for DATA sections.

The **INIT** attribute defines that the DATA section contains initialization data, which is copied from ROM to RAM at program startup.

**ROMDATA** sections (allowed on DATA and CODE sections) contain data to be placed in ROM. This ROM area is not executable.

When DATA sections with the same name occur in different object modules with the **MAX** attribute, the linker generates a section of which the size is the maximum of the sizes in the individual object modules. The MAX attribute only applies to DATA sections.

**Examples:**

```
.sdecl ".text", DATA      ;declare section
    ;.text
.sect ".text"
    ;switch to section
    ;.text
```

**.sect, .org**

See the paragraph *Section Names* in the chapter *Software Concept* for detailed information about sections, section types and section attributes.

# .SECT

## Syntax:

```
.SECT "str" [, RESET]
```

## Description:

The **.SECT** directive flags the assembler that another section, with name *str*, becomes active. Before a section can be activated for the first time, it must be defined first, by the **.SDECL** directive. Subsequent activations can be done by the **.SECT** directive only.

You can use the section attribute **RESET** to reset counting storage allocation in **DATA** sections with section attribute **MAX**.

## Examples:

```
.sdecl ".text", DATA      ;declare section
    ;.text
.sect    ".text"
    ;switch to section
    ;.text
```



### **.sdecl**

See the paragraph *Section Names* in the chapter *Software Concept* for detailed information about sections.

# .SET

## Syntax:

*name* **.SET** *expression*

## Description:

Set Symbol to a Value. The **.SET** directive is used to assign the value of the expression in the operand field to the symbol name. The **.SET** directive functions somewhat like the **.EQU** directive. However, symbols defined via the **.SET** directive can have their values redefined in another part of the program (but only through the use of another **.SET** directive). The **.SET** directive is useful in establishing temporary or reusable counters within macros. The expression in the operand field of a **.SET** may have forward references.

**.SET** symbols cannot be made global.

## Examples:

```
COUNT .set 0      ; Initialize COUNT
```





# .SFRACT

## Syntax:

```
[label:] .SFRACT arg[,arg]...
```

## Description:

Define 16-bit constant fraction of Q15 format. The **.SFRACT** directive allocates and initializes a half word (16-bits) of memory for each *arg* argument. *arg* can be a constant, a symbol, or an expression.

The **.SFRACT** directive may have one or more arguments separated by commas. Multiple arguments are stored in successive address locations. If multiple arguments are present, one or more of them can be null (two adjacent commas), in which case the corresponding address location will be filled with zeros. An error will occur if the evaluated argument value is out the range of  $[-1,1]$ .

*label*, if present, will be assigned the value of the run-time location counter at the start of the directive processing.

## Examples:

```
SFRCT: .sfract 0.1,0.2,0.3
```



```
.accum, .fract
```

# .SPACE

## Syntax:

*[label:]*    **.SPACE**    *expression*

## Description:

Define Storage. The **.SPACE** directive reserves a block of memory the length of which in bytes is equal to the value of *expression*. This directive causes the run-time location counter to be advanced by the value of the absolute integer expression in the operand field. The block of memory reserved is not initialized to any value. The expression must be an integer greater than zero and cannot contain any forward references to address labels (labels that have not yet been defined).

*label*, if present, will be assigned the value of the run-time location counter at the start of the directive processing.

## Examples:

```
S_BUF:    .space 12                    ; Sample buffer
```



**.byte, .half, .word**

# .SYMB

## Syntax:

**.SYMB** *string*, *expression* [, *abs\_expr*] [, *abs\_expr*]

## Description:

The **.SYMB** directive is used for passing high-level language symbolic debug information via the assembler (and linker/locator) to the debugger. *expression* can be any expression. *abs\_expr* can be any expression resulting in an absolute value.

The .SYMB directive is not meant for 'hand coded' assembly files. It is documented for completeness only and is supposed to be 'internal' to the tool chain.

# .UNDEF

## Syntax:

**.UNDEF** *symbol*

## Description:

Undefine .DEFINE Symbol. The **.UNDEF** directive causes the substitution string associated with *symbol* to be released, and *symbol* will no longer represent a valid **.DEFINE** substitution. See the **.DEFINE** directive for more information.

A label is not allowed with this directive.

## Examples:

```
.undef DEBUG           ; Undefines the DEBUG  
                        ; substitution string
```



**.define**

# .WARNING

## Syntax:

**.WARNING**      [{*str* | *exp*},{*str* | *exp*}...]

## Description:

Programmer Generated Warning. The **.WARNING** directive will cause a warning message to be output by the assembler. The total warning count will be incremented as with any other warning. The **.WARNING** directive is normally used in conjunction with conditional assembly directives for exceptional condition checking. The assembly proceeds normally after the warning has been printed. An arbitrary number of strings and expressions, in any order but separated by commas, can be specified optionally to describe the nature of the generated warning.

A label is not allowed with this directive.

## Examples:

**.warning**    'parameter too large'



**.fail, .message**

# .WORD

## Syntax:

[*label*:] **.WORD** *arg*[,*arg*]...

## Description:

Define Constant Word. The **.WORD** directive allocates and initializes a word of memory for each *arg* argument. *arg* can be a numeric constant, a single or double character string constant, a symbol, or an expression. The **.WORD** directive may have one or more arguments separated by commas. Multiple arguments are stored in successive address locations. If multiple arguments are present, one or more of them can be null (two adjacent commas), in which case the corresponding address location will be filled with zeros. An error will occur if the evaluated argument value is too large to represent in a single word.

*label*, if present, will be assigned the value of the run-time location counter at the start of the directive processing.



Word values are stored in memory with the least significant 8 bits on the lowest address (little endian).

Integer arguments are stored as is. Single and multiple character strings are handled in the following manner:

1. Single character strings are stored in a word whose lower seven bits represent the ASCII value of the character.

Example: 'R' = 0x52

2. Multiple character strings consisting of more than four characters are not allowed. Four-character strings are stored as if the ASCII value of the first character is the high byte value of the word. The last character is used as the low byte.

Example:

'ABCD' = 0x41424344

**Examples:**

```
TABLE:  .word 14,1635,0x34266243,'ABCD'
```

is equal to

```
TABLE:  .byte 14,0,0,0,1635%256,6,0,0,  
          0x43,0x62,0x26,0x34,'D','C','B','A'
```



**.byte, .half, .space**

# CHAPTER

# 8

## ASSEMBLER CONTROLS

---





---

# 8

# CHAPTER

---

## 8.1 INTRODUCTION

Assembler controls are provided to alter the default behavior of the assembler. They can be specified on 'control lines', embedded in the source file. A control line is a line starting with a dollar sign (\$). Such a line is not processed like a normal assembly source line, but as an assembler control line. One control per source line is allowed. An assembler control line may contain comments. Upper and lower case letters are considered equivalent for assembler directives.

The controls are classified as: primary or general.

### ***Primary controls***

Primary controls effect the overall behavior of the assembler and remain in effect throughout the assembly. For this reason, you must use primary controls only at the beginning of a source file, before the assembly starts. If you use a primary control after the beginning of the source file, a warning is given and the control is ignored. If you specify a primary control more than once, a warning is given and the last definition is used. When you use a corresponding command line option to specify a control, this overrides the control specified in the source file.

### ***General controls***

General controls are used to control the assembler *during* assembly. General controls may appear anywhere in a source file. When you use a corresponding command line option to specify a general control, the general controls in the source file are ignored.

In the following sections the available assembler controls are listed in alphabetic order. Some controls are set by default, and some controls have a default value.

## 8.2 OVERVIEW ASSEMBLER CONTROLS

Control	Type	Def.	Description
\$CASE ON \$CASE OFF	pri	ON	All user names are case sensitive. User names are not case sensitive.
\$DEBUG ON \$DEBUG OFF \$DEBUG "flags"	pri	AhLS	Produce local symbolic debug info. Do not produce local symbols Produce symbolic debug info.
\$HW_ONLY	pri		Map mnemonics literally on the instructions.
\$FPU	pri		Allow single precision FPU instructions in the assembly source.
\$IDENT LOCAL \$IDENT GLOBAL	pri	LOCAL	Default local labels. Default global labels.
\$LIST ON \$LIST OFF	gen		Resume listing. Stop listing.
\$LIST "flags"	pri	cDEIMnPQ-sWXy	Define what to include in/exclude from the list file.
\$MMU	pri		Allow memory management – instructions in the assembly source
\$OBJECT "file" \$OBJECT OFF	pri pri	src.obj	Alternative name for object file. Do not produce an object file.
\$PAGE	gen		Generate formfeed in list file.
\$PAGE width[,length, top, bottom, left]	gen	80,66,0,0,0	Change list file page settings.
\$PAGELENGTH (length)	pri	66	Sets page length of list file
\$PAGEWIDTH (width)	pri	80	Sets width of list file
\$PAGING \$NOPAGING	pri	PAGING	Format print file into pages Do not format print file into pages
\$PRCTL expl"string"	gen		Output <i>string</i> to the list file.
\$PRINT(print-file) \$NOPRINT	pri	src.lst	Define print file name Do not create print file
\$TCdefect ON \$TCdefect OFF	gen	OFF	Enable checks for CPU functional problem <i>defect</i> Enable checks for CPU functional problem <i>defect</i>
Type: Def.:	Type of control: pri for primary controls, gen for general controls. Default.		

Control	Type	Def.	Description
\$STITLE "title"	gen		Set list page header title for next pages.
\$TITLE "title"	pri	spaces	Set list page header title for first page.
\$WARNING OFF \$WARNING OFF num	pri		Suppress all warnings. Suppress one warning.
Type: Def.:	Type of control: pri for primary controls, gen for general controls. Default.		

Table 8-1: Assembler controls

### **8.3 DESCRIPTION OF ASSEMBLER CONTROLS**

## **CASE**

**Control:**

\$CASE ON  
\$CASE OFF

**Related option:**

-c Set case sensitivity off; overrides the control.

**Class:**

Primary

**Default:**

\$CASE ON

**Description:**

Selects whether the assembler operates in case sensitive mode or not. In case insensitive mode the assembler maps characters on input to uppercase. (literal strings excluded).

**Example:**

```
;Begin of source  
$case off      ;assembler in case insensitive mode
```

# DEBUG

## Control:

```
$DEBUG ON
$DEBUG OFF
$DEBUG "flags"
```

## Related option:

```
-g[a|h|l|s]...    Produce assembly debug information
```

## Class:

Primary

## Default:

```
$DEBUG "AhLS"    (only HLL debug)
```

## Description:

Controls the generation of debugging information in the object file. \$DEBUG ON enables the generation of local (assembler) debugging information (same as **-gl**) and \$DEBUG OFF disables it.

With \$DEBUG "flags" you have more control over the generation of debugging information. Flags can be switched on with the lower case letter and switched off with the uppercase letter. The following flags are available:

- a** - assembler source line information
- h** - pass HLL debug information
- l** - local symbols debug information
- s** - always debug; either "AhL" or "aHl"

Flags **a** and **h** cannot be combined.



The debugging information generated by the C compiler is always passed on to the object file.

## Example:

```
;Begin of source
$debug on      ; generate local assembly
               ; debug information
```

# FPU

## Control:

\$FPU

## Related option:

-FPU

## Class:

Primary

## Default:

(none) No fpu instructions allowed.

## Description:

When you use the \$FPU control you allow the assembler to accept single precision FPU instructions in the assembly source file and encode them properly. When you use this control, the define \_FPU is set to 1 allowing you to use single precision floating point instructions. Default the define \_FPU is set to 0 which tells the assembler not to accept single precision floating point instructions.



For a more detailed description about the floating point arithmetic see section *Floating Point Arithmetic* in Chapter *Runtime* of the *C Cross Compiler Users Manual*.

## Example:

```
;Begin of source
$FPU      ; the use of single precision FPU instructions
          ; in this source are allowed.
```

# HW\_ONLY

**Control:**

`$HW_ONLY`

**Related option:**

`-hw_only`

**Class:**

Primary

**Default:**

none                      Substitutions by faster or smaller instructions is allowed.

**Description:**

Normally the assembler may replace instructions by other, smaller or faster instructions. For example, the instruction `jeq d0,#0,label1` is replaced by `jz d0,label1`.

With the `$HW_ONLY` control you instruct the assembler to encode all instruction as they are. The assembler does not substitute instructions with other, faster or smaller instructions. When you use this control, the define `_HW_ONLY` is set to 1 (no substitution of instructions).

**Example:**

To prevent the assembler from substituting instructions with other, smaller or faster instructions:

```
;Begin of source
$HW_ONLY      ; the assembler does not substitute
               ; instructions with other, smaller or
               ; faster instructions.
```



# IDENT

## Control:

```
$IDENT LOCAL
$IDENT GLOBAL
```

## Related option:

**-i[l|g]** Default labels are local or global.

## Class:

Primary

## Default:

```
$IDENT LOCAL
```

## Description:

With the \$IDENT control you specify how a label is to be treated by the assembler. This is for code and data labels only. \$IDENT LOCAL specifies that labels are local by default, with \$IDENT GLOBAL labels are global by default.

.SET identifiers are always treated as local symbols.

You can always overrule the default settings with the .LOCAL or .GLOBAL directives for a specific label.

## Example:

```
;Begin of source
$ident global      ; assembly labels are global
                    ; by default
```

# LIST ON/OFF

**Control:**

```
$LIST ON  
$LIST OFF
```

**Related option:**

**-l** Produce an assembler list file

**Class:**

General

**Default:**

```
$LIST ON
```

**Description:**

Switch the listing generation on or off. These controls take effect starting at the next line. Actual list file generation is selected on the command line. Without the command line option **-l**, no list file is produced.

**Example:**

```
$list off      ; Turn listing off. These lines are  
               ; not present in the list file  
.  
.  
$list on      ; Turn listing back on. These lines  
               ; are present in the list file  
.  
.
```

# LIST

## Control:

`$LIST "flags"`

## Related option:

`-L[flag...]` Remove specified source lines from list file

## Class:

Primary

## Default:

`$LIST "cDElMnPQsWXy"`

## Description:

Specify which source lines are to be removed from the list file. The flags defined within the string are the same as for the `-L` command line option.

## Example:

```
;Begin of source
$list "cw"          ; Remove source lines with assembler
                   ; controls from the resulting list
                   ; file and remove wrapped source lines
.
.
```



See the `-L` option for an explanation of each flag available.

# MMU

**Control:**

`$MMU`

**Related option:**

`-MMU`

**Class:**

Primary

**Default:**

`$MMU`      The use of memory management instructions is allowed.

**Description:**

The `$MMU` control allows the use of memory management instructions in the assembly source. When you use this control, the define `_MMU` is set to 1 allowing you to use memory management instructions.

**Example:**

To allow the use of memory management instructions in the assembly source, enter:

```
;Begin of source
$MMU      ; Allow the use of memory management
          ; instructions in the assembly source.
```

# OBJECT

**Control:**

`$OBJECT "file"`  
`$OBJECT OFF`

**Related option:**

`-o file` Specify name of output file

**Class:**

Primary

**Default:**

`$OBJECT "sourcefile.obj"`

**Description:**

The `$OBJECT "file"` control specifies an alternative name for the object file.  
The `$OBJECT OFF` control causes no object file to be generated.

**Examples:**

```
;Begin of source  
$object "x1.obj"           ; generate object file x1.obj
```

# PAGE

**Control:**

\$PAGE

**Class:**

General

**Default:**

New page started when page length is reached.

**Description:**

The current page is terminated with a formfeed after the current (control) line, the page number is incremented and a new page is started. Ignored if LIST OFF is in effect.

**Example:**

```
.           ; assembler source lines
.
$page      ; generate a formfeed
.
.           ; more source lines
$page      ; generate a formfeed
.
.
```

# PAGE Settings

## Control:

`$PAGE exp1[,exp2,...,exp5]`

## Related option:

`-l` Produce an assembler list file

## Class:

General

## Default:

`$PAGE 80,66,0,0,0`

## Description:

Change page settings. The `$PAGE` control with arguments can be used to specify the printed format of the output listing. Arguments may be any positive absolute integer expression. The arguments in the operand field (as explained below) are separated by commas. Any argument can be left as the default or last set value by omitting the argument and using two adjacent commas. The `$PAGE` control with arguments will not cause a page eject and will be printed in the source listing.

The arguments in order are:

`PAGE_WIDTH exp1`

Page width in terms of number of output columns per line (default 80, min 1, max 255).

`PAGE_LENGTH exp2`

Page length in terms of total number of lines per page (default 66, min 10, max 255). As a special case a page length of 0 (zero) turns off all headers, titles, subtitles, and page breaks.

`BLANK_TOP exp3`

Blank lines at top of page. (default 0, min 0, max see below).

BLANK\_BOTTOM *exp4*

Blank lines at bottom of page. (default 0, min 0, max see below).

BLANK\_LEFT *exp5*

Blank left margin. Number of blank columns at the left of the page.  
(default 0, min 0, max see below).

The following relationship must be maintained:

$$\text{BLANK\_TOP} + \text{BLANK\_BOTTOM} \leq \text{PAGE\_LENGTH} - 10$$
$$\text{BLANK\_LEFT} < \text{PAGE\_WIDTH}$$

### Examples:

```
$PAGE 132,,3,3      ;Set width to 132,  
                    ;default 66 lines  
                    ;3 line top/bottom margins
```



# PAGELength

**Control:**

`$PAGELength(length)`

**Related option:**

`-ll` Set listing page length

**Class:**

Primary

**Default:**

`$PAGELength(66)`

**Description:**

Sets the maximum number of lines on one page of the listing file. This number includes the lines used by the page header (4) and the lines with error messages. The default page length is 80. The minimum page length is 10.

**Example:**

`$pagelength(50)` ; set page length to 50 lines per page

# PAGEWIDTH

**Control:**

`$PAGEWIDTH(width)`

**Related option:**

`-lw` Set listing page width

**Class:**

Primary

**Default:**

`$PAGEWIDTH(80)`

**Description:**

Sets the maximum number of characters on one line in the list file. Lines that exceed this width are wrapped on the next lines. The default page width is 66. The minimum page width is 40.

**Example:**

```
$pagelength(50) ; set page width to 50 chars per line
```

# PAGING

**Control:**

`$PAGING`  
`$NOPAGING`

**Related option:**

`-nopi` Disable format listing file into pages

**Class:**

Primary

**Default:**

Paging is turned on.

**Description:**

Turn the generation of formfeeds and page headers on or off.

**Example:**

`$nopaging ; turn paging off`

# PRCTL

**Control:**

`$PRCTL exp [string [, exp | string] ...`

**Related option:**

**-l** Produce an assembler list file

**Class:**

General

**Default:**

—

**Description:**

Send Control String to Printer. `$PRCTL` simply concatenates its arguments and ships them to the listing file (the control line itself is not printed unless there is an error). *exp* is a byte expression and *string* is an assembler string. A byte expression would be used to encode non-printing control characters, such as ESC. The string may be of arbitrary length, up to the maximum assembler-defined limits.

`$PRCTL` may appear anywhere in the source file and the control string will be output at the corresponding place in the listing file. However, if a `$PRCTL` directive is the last line in the last input file to be processed, the assembler insures that all error summaries, symbol tables, and cross-references have been printed before sending out the control string. This is so a `$PRCTL` control can be used to restore a printer to a previous mode after printing is done. Similarly, if the `$PRCTL` control appears as the first line in the first input file, the control string will be output before page headings or titles.

The `$PRCTL` control only works if the **-l** command line option is given; otherwise it is ignored.

**Examples:**

```
$PRCTL $1B,'E'           ;Reset HP LaserJet printer
```

# PRINT

**Control:**

`$PRINT(file)`  
`$NOPRINT`

**Related option:**

- l**      Produce an assembler list file
- ln**     Define an assembler list file name

**Class:**

Primary

**Default:**

`$PRINT(sourcefile.lst)`

**Description:**

The `$PRINT` control specifies an alternative name for the listing file.  
The `$NOPRINT` control causes no listing file to be generated.

**Examples:**

```
$PRINT(mylist.lst) ; generate an assembler list file  
                    with the name 'mylist.lst'.
```

# STITLE

**Control:**

`$STITLE "title"`

**Related option:**

`-l` Produce an assembler list file

**Class:**

General

**Default:**

`$STITLE ""`

**Description:**

Initialize Program Sub-Title. The `$STITLE` control initializes the program subtitle to the *title* in the operand field. The subtitle will be printed on the top of all succeeding pages until another `$STITLE` control is encountered. The subtitle is initially blank. The `$STITLE` control will not be printed in the source listing. An `$STITLE` control with no string argument will cause the current subtitle to be blank.

If the page width is too small for the title to fit in the header, it will be truncated.

**Example:**

```
$stitle "Demo title"
```

```
; title in page header on succeeding pages  
; is Demo title
```



**TITLE**

# TC

## Control:

`$TCdefect` ON  
`$TCdefect` OFF

## Related option:

`-zdefect` Set checking for CPU functional problem *defect* on

## Class:

General

## Default:

`$TCdefect` OFF

## Description:

With this control you can enable or disable specific CPU functional problem checks.

To enable the assembler checks for *all* TriCore CPU TC112 problems (respectively TC113 problems) at once, use the control **\$TC112\_DEFECTS** (**\$TC113\_DEFECTS**).

## Example:

```
$TC112_COR1  ON      ; enables assembler checks for CPU
                    ; functional problem TC112_COR1
```



Appendix *CPU Functional Problems* in the C Cross-Compiler User's Guide for a complete overview of all CPU functional problems that are supported by the assembler.

# TITLE

**Control:**

`$TITLE "title"`

**Related option:**

`-It` Set listing page header title

**Class:**

Primary

**Default:**

spaces

**Description:**

This control specifies the *title* to be used in the page heading of the first page of the list file.

If the page width is too small for the title to fit in the header, it will be truncated.

**Example:**

```
;Begin of source
$title "NEWTITLE"

; title in page header on first page is NEWTITLE
```



**\$TITLE**



# WARNING

## Control:

```
$WARNING OFF
$WARNING OFF num
```

## Related option:

**-w[*num*]** Suppress one or all warning messages

## Class:

Primary

## Default:

- (All warnings enabled)

## Description:

**\$WARNING** suppresses all warnings. This is the same as **-w**. **\$WARNING OFF *num*** suppresses one warning message, where *num* is the warning message number (same as the **-w*num*** option).

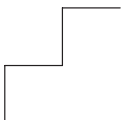
## Example:

```
;Begin of source
$warning off      ; switch all warnings off
```

# CHAPTER 9

**LINKER**

---



---

# 9

# CHAPTER

---

## 9.1 OVERVIEW

This section gives a global overview of the process of linking programs for the TriCore and its derivatives. The linker executable name for the TriCore is **lktri**.

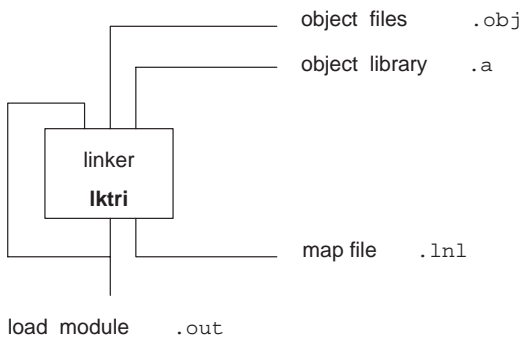
The linker combines relocatable object files, generated by the assembler, into one new relocatable object file (preferred extension `.out`). This file may be used as input in subsequent linker calls: the linkage process may be incremental. Normally the linker complains about unresolved external references. With incremental linking it is normal to have unresolved references in the output file. Incremental linking must be selected separately.

The linker can read normal object files and libraries of object modules. Modules in a library are included only when they are referenced. At the end of the linkage process the generated object, without unresolved references, will be called: a load module.

The TriCore linker is an overlaying linker. The compiler generates overlayable sections. An overlayable section contains space reservations for variables which, at C level, are local to a function. If functions do not call each other, their local variables can be overlayed in memory. It is a task of the linker to combine function call information into a call graph and to determine upon the structure of this call graph how sections can be overlayed, using the smallest amount of RAM.

Incremental linkage disables overlaying, so the last link phase should not be incremental, even if the incremental phase resolves all externals.

The following diagram shows the input files and output files of the linker:



*Figure 9-1: TriCore Linker*

9.2 LINKER INVOCATION

The invocation of the TriCore linker is:

```
lktri [ option ]... file ...
```

When you use a UNIX shell (**C-shell**, **Bourne shell**), options containing special characters (such as ' ( ) ' ) must be enclosed with " ". The invocations for UNIX and PC are the same, except for the -? option in the **C-shell**:

```
lktri "-?"          or      lktri -\?
```

Options may appear in any order. Options start with a '-'. Only the -lx option is position dependent. Option may be combined: -rM is equal to -r -M. Options that require a filename or a string may be separated by a space or not: -oname is equal to -o name.

file can be any object file (.obj), or object library (.a) or incrementally linker (.out) files. The files are linked in the same order as they appear on the command line.


The linker recognizes the following options:

Option	Description
-C	Link case insensitive (default case sensitive)
-H or -?	Display invocation syntax
-L directory	Additional search path for system libraries
-L	Skip system library search
-M	Produce a link map (.lnl)
-N	Turn off overlaying
-O name	Specify basename of the resulting map files
-V	Display version header only
-WAE	Treat warning messages as errors
-c	Produce a separate call graph file (.cal)
-d file	Read description file information from file, '-' means stdin
-e	Clean up if erroneous result
-err	Redirect error messages to error file (.elk)

Option	Description
-f <i>file</i>	Read command line information from <i>file</i> , '-' means <code>stdin</code>
-l <i>x</i>	Search also in system library <code>libx.a</code>
-o <i>filename</i>	Specify name of output file
-r	Suppress undefined symbol diagnostics
-u <i>symbol</i>	Enter <i>symbol</i> as undefined in the symbol table
-v or -t	Verbose option. Print name of each file as it is processed
-w <i>n</i>	Suppress messages above warning level <i>n</i> .

Table 9-1: Options summary

9.2.1 DETAILED DESCRIPTION OF LINKER OPTIONS

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

-?/-H

Option:

-?  
-H

Description:

Display an explanation of the invocation syntax at `stdout`.

Example:

`lktri -H`

## -C

### Option:



Select the | Linker/Locator Options... menu item. Disable the Link case sensitive check box in the Linker tab.



-C

### Default:

Case sensitive

### Description:

With this option the linker links case insensitive. The default is case sensitive linking.

### Example:

To switch to case insensitive mode, enter:

```
lktri -C test.obj
```

Using the control program:

```
cctri -Wlk-C test.obj
```

## -C

### Option:



Select the | Linker/Locator Options... menu item. Enable the Make a separate function call graph file (.cal) check box in the Linker tab.



**-c**

### Description:

Generate separate call graph file (.cal).

### Example:

To create a call graph file (test.cal), enter:

```
lktri -c test.obj
```

Using the control program:

```
cctri -Wlk-c test.obj
```



Section *Linker Output*.



## -d

### Option:



Select the | Linker/Locator Options... menu item. Add the option to the Additional options field in the Linker tab.



**-d** *file*

### Arguments:

A filename to read description file information from. If *file* is a '-', the information is read from standard input.

### Description:

Read description file information from *file* instead of a .dsc file.

### Example:

To read description file information from file `tri.dsc`, enter:

```
lktri -dtri.dsc test.obj
```

# -e

**Option:**

EDE always removes the output files when errors occur.



**-e**

**Description:**

Remove all link products such as temporary files, the resulting output file and the map file, in case an error occurred.

**Example:**

```
lktri -e test.obj
```

## -err

### Option:



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



**-err**

### Description:

The linker redirects error messages to a file with the same basename as the output file and the extension `.elk`. The default filename is `a.elk`.

### Example:

To write errors to the file `a.elk` instead of `stderr`, enter:

```
lktri -err test.obj
```

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

```
lktri -err test.obj -otest.out
```

## -f

### Option:

**-f** *file*

### Arguments:

A filename for command line processing. If *file* is a '-', the information is read from standard input. You need to provide the EOF code to close stdin (usually Ctrl-Z or Ctrl-D on UNIX).

### Description:

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

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

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

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

Example:

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

or

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

or

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

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

Example:

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

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

### Example:

Suppose the file `mycmds` contains the following line:

```
-err  
test.obj
```

The command line can now be:

```
lktri -f mycmds
```

**Option:**

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



**-L** [*directory*]

**Arguments:**

The name of the directory to search for system libraries.

**Description:**

Add *directory* to the list of directories that are searched for system libraries. Directories specified with **-L** are searched before the standard directories specified by the environment variable CTRILIB. If you specify **-L** without a directory, the environment variable CTRILIB is not searched for system libraries. You can use the **-L** option more than once to add several directories to the search path for system libraries. The search path is created in the same order as in which the directories are specified on the command line.

**Example:**

```
lktri -Lc:\ctri\lib test.obj
```



### Option:



Select the | Linker/Locator Options... menu item. Enable the Link default C libraries check box in the Linker tab. Optionally, select a floating point library.



**-l***x*

### Arguments:

A string to form the name of the system library `libx.a`.

### Description:

Search also in system library `libx.a`, where *x* is a string. The linker first searches for system libraries in any directories specified with **-L***directory*, then in the standard directories specified with the environment variable `CTRILIB`, unless the **-L** option is used without a directory specified.



This option is position dependent (see section *Linking with Libraries*).

### Example:

To search in the system library `libc.a` after the user object and library are linked, enter:

```
lktri myobj.obj mylib.a -lc
```

# -M

## Option:



Select the | Linker/Locator Options... menu item. Enable the Generate a linker listing file (.lnl) check box in the Linker tab.



**-M**

## Description:

Produce a link map (.lnl). If no output filename is specified the default name is a.lnl.

## Example:

To create the map file a.lnl, enter:

```
lktri -M test.obj
```



Section *Linker Output*,  
**-O.**



## -N

### Option:



Select the | Linker/Locator Options... menu item. Add the option to the Additional options field in the Linker tab.



-N

### Description:

Turn off overlaying. This can be useful for debugging.

## -O

### Option:



Select the | Linker/Locator Options... menu item. Add the option to the Additional options field in the Linker tab.



**-O** *name*

### Arguments:

The basename to be used for map files.

### Description:

Use *name* as the default basename for the resulting map files.

### Example:

To create the map file `test.lnl` using the linker, enter:

```
lktri -M -Otest test.obj
```

Using the control program:

```
cctri -Wlk-M -Wlk-Otest test.obj
```



Section *Linker Output*,  
-M.

## -O

**Option:**

**-o** *filename*

**Arguments:**

An output filename.

**Default:**

a.out

**Description:**

Use *filename* as output filename of the linker. If this option is omitted, the default filename is a.out.

**Example:**

To create the output file test.out instead of a.out, enter:

```
lktri test.obj -otest.out
```

# -r

## Option:



Select the | Linker/Locator Options... menu item. Add the option to the Additional options field in the Linker tab.



**-r**

## Description:

Specify incremental linking. No report is made for unresolved symbols, and the function overlaying is disabled.



Section *Linker Output*.

## -u

### Option:



Select the | Linker/Locator Options... menu item. Add the option to the Additional options field in the Linker tab.



**-u** *symbol*

### Arguments:

The name of a symbol to undefine.

### Description:

Enter *symbol* as undefined in the symbol table. This is useful for linking from a library.

### Example:

To force symbol `main` as undefined, enter:

```
lktri -u main mylib.a
```



Section *Linking with Libraries*.

## **-V**

### **Option:**

**-V**

### **Description:**

With this option you can display the version header of the linker. This option must be the only argument of **lktri**. Other options are ignored. The linker exits after displaying the version header.

### **Example:**

```
lktri -V
```

```
TASKING TriCore object linker    vx.yrz Build nnn  
Copyright 1996-year Altium BV    Serial# 00000000
```

## -v

### Option:



Select the | Linker/Locator Options... menu item. Enable the Print the name of each file as it is processed check box in the Linker tab.



**-v**  
**-t**

### Description:

Verbose option. Print the name of each file as it is processed.

### Example:

```
lktri -v test.obj
```

```
lktri V008 (1): Embedded environment \ctri\etc\tri.dsc
               read, relaxed addressing mode check enabled
lktri V003 (1): Starting pass 1
lktri V002 (1): File currently in progress:
               test.obj
lktri E208 (0): Found unresolved external(s):
               _printf - (test.obj)
               __START - (test.obj)
lktri V003 (1): Starting pass 2
lktri V002 (1): File currently in progress:
               test.obj
lktri V005 (1): Removing file .\CD5668a.tld
```

Using the control program:

```
cctri -Wlk-v test.obj
```

## -W

### Option:



Select the | Linker/Locator Options... menu item. Select a warning level from the Suppress warning messages above list box in the Linker tab.



**-w** *level*

### Arguments:

A warning level between 0 and 9 (inclusive).

### Default:

**-w8**

### Description:

Give a warning level between 0 and 9 (inclusive). All warnings with a level above *level* are suppressed. The level of a message is printed between parentheses after the warning number. If you do not use the **-w** option, the default warning level is 8.

### Example:

To suppress warnings above level 5, enter:

```
lktri -w5 test.obj
```

Using the control program:

```
cctri -Wlk-w5 test.obj
```



Section *Type Checking*.



# -WAE

## Option:



Select the | Linker/Locator Options... menu item. Enable the Treat warnings as if they were errors check box in the Linker tab.



**-WAE**

## Description:

Treat warning messages as errors. This also affects the return value of the application when only errors occur. A build process will now stop when warnings occur.

## Example:

```
lktri -WAE test.obj
```

## 9.3 LIBRARIES

There are two kinds of libraries. One of them is the user library. If you make your own library of object modules, this library must be specified as an ordinary filename. The linker will not use any search path to find such a library. The file must have the extension `.a`. Example:

```
lktri start.obj -fobj.lnk mylib.a
```

or, if the library resides in a sub directory:

```
lktri start.obj -fobj.lnk libs\mylib.a      (PC)
lktri start.obj -fobj.lnk libs/mylib.a      (UNIX)
```

The other kind of library is the system library. You must define system libraries with the `-l` option. With the option `-lc` you specify the system library `libc.a`.



In EDE you can specify the system libraries in the Linker tab of the | Linker/Locator Options... menu item.

### 9.3.1 LIBRARY SEARCH PATH

The linker searches for system library files according to the following algorithm:

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

PC:

```
lktri -L..\lib -L\usr\local\lib start.obj -fobj.lnk
-lc
```

UNIX:

```
lktri -L../lib -L/usr/local/lib start.obj -fobj.lnk
-lc
```

2. If the `-L` option is not specified without a directory, check if the environment variable `CTRILIB` exists. If it does, use the contents as a directory specifier for library files. It is possible to specify more than one directory in the environment variable `CTRILIB` by separating the directories with a directory separator. Valid directory separators are:

PC: ;

UNIX: : ;

Instead of using **-L** as in the example above, the same directory can be specified using **CTRILIB**:

PC:

```
set CTRILIB=..\lib\usr\local\lib
lktri start.obj -fobj.lnk -lc
```

UNIX:

if using the Bourne shell (sh), or korn shell (ksh)

```
CTRILIB=../lib:/usr/local/lib
export CTRILIB
lktri start.obj -fobj.lnk -lc
```

or if using the C-shell (csh)

```
setenv CTRILIB ../lib:/usr/local/lib
lktri start.obj -fobj.lnk -lc
```

3. Search in the **lib/tcl** directory relative to the installation directory of **lktri** for library files.

PC:

**lktri.exe** is installed in the directory **C:\CTRI\BIN**  
The directory searched for the library file is **C:\CTRI\LIB\TCL**

UNIX:

**lktri** is installed in the directory **/usr/local/ctri/bin**  
The directory searched for the library file is  
**/usr/local/ctri/lib/tcl**

The linker determines run-time which directory the binary is executed from to find this **lib** directory.

A directory name specified with the **-Ldirectory** option or in **CTRILIB** may or may not be terminated with a directory separator, because **lktri** inserts this separator, if omitted.

### 9.3.2 LINKING WITH LIBRARIES

If you are linking from libraries, only those objects you need are extracted from the library. This implies that if you invoke the linker like:

```
lktri 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 undefined with the option **-u**:

```
lktri -u main mylib.a
```

 (space between `-u` and `main` is optional)

In this case the symbol `main` will be searched for in the library and (if found) the object containing `main` will be extracted. If this module contains new unresolved symbols, the linker looks again in `mylib.a`. This process repeats until no new unresolved symbols are found. See also the library member search algorithm in the next section.

The position of the library is important, if you specify:

```
lktri -lc myobj.obj mylib.a
```

the linker starts with searching the system library `libc.a` without unresolved symbols, thus no module will be extracted. After that, the user object and library are linked. When finished, all symbols from the C library remain unresolved. So, the correct invocation is:

```
lktri myobj.obj mylib.a -lc
```

All symbols which remain unresolved after linking `myobj.obj` and `mylib.a` will be searched for in the system library `libc.a`. Note that the link order for objects, user libraries and system libraries is the order in which they appear at the command line. Objects are always linked, object modules in libraries are only linked if they are needed.

### 9.3.3 LIBRARY MEMBER SEARCH ALGORITHM

A library built with **artri** 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 will be scanned again.

Using the **-v** option, you can follow the linker actions in respect to the libraries.

## 9.4 LINKER OUTPUT

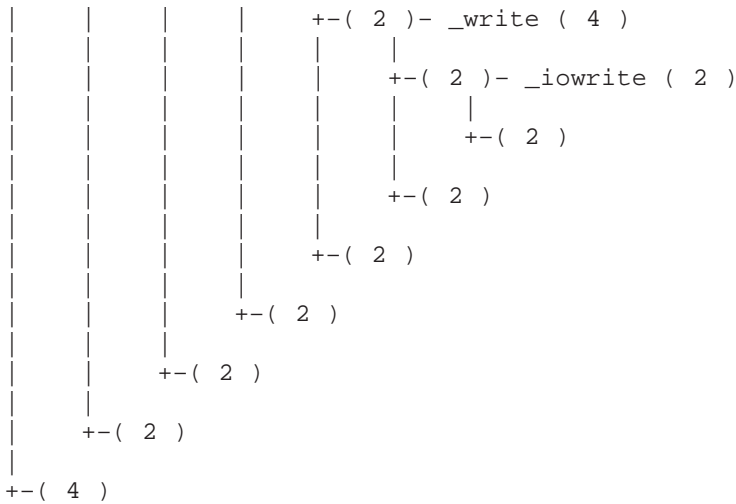
The linker produces an IEEE-695 object output file and, if requested, a map file, and/or a call graph file.

The linker output object is still relocatable. It is the task of the locator to determine the absolute addresses of the sections. The linker combines sections with the same name to one (bigger) output section.

The linker produces a map file if the option **-M** is specified. The name of the map file is the same as the name of the output file. The extension is `.lnl`. If no output filename is specified the default name is `a.lnl`. The map file is organized per linked object. Each object is divided in sections and symbols per section. The map file shows the relative position of each linked object from the start of the section.

The generated call graph will also be printed in the map file. The call graph contains an overview of which function calls are present. The call graph also contains information about the stack usage of the call graph. When a function is called, the stack usage before entering the function is written in front of the function name. The total stack usage of the function (including its calls) is written behind the function. The maximum stack usage of a function itself is written below the function. The number indicates the size of the stack usage (in bytes for the TriCore). See also the example.





Maximum stack usage: 14

Pool offsets  
=====

Pool #1: zp\_ovln (Total of 39 bytes)

Pool: zp_ovln		
	off	siz
puts()	0	6
fputc()	6	7
_flsbuf()	13	12
_write()	25	10
_iowrite()	35	4

Object: cstart.obj  
=====

Section:abs\_65534 ( Start = 0x0 )

Section:.text ( Start = 0x0 )  
0x0000001c E \_\_exit  
0x00000000 E \_\_START

```
Object: hello.obj
=====

Section:.text ( Start = 0x1f )
0x0000001f E _main


Section:.string ( Start = 0x0 )


Object: _puts.obj
=====

Section:.text ( Start = 0x28 )
0x00000028 E _puts


Object: _fputc.obj
=====

Section:.text ( Start = 0x78 )
0x00000078 E _fputc


Object: _iob.obj
=====

Section:.near_data ( Start = 0x0 )
0x00000000 E __iob


Section:.near_bss ( Start = 0x0 )
0x00000000 E __ungetc


Object: _flsbuf.obj
=====

Section:.text ( Start = 0x0102 )
0x00000102 E __flsbuf


Object: _iowrite.obj
=====

Section:.text ( Start = 0x0314 )
0x00000314 E __iowrite
```



```
Object: _write.obj
=====

Section:.text ( Start = 0x0318 )
0x00000318 E __write
```

The addresses in the map file are offsets relative to the start of the section in the output file. For instance, section `.text` of the object module `hello.obj` starts at offset `0x1f` from the output `.text` section. Function `main` also starts at offset `0x1f` from the start of the resulting `.text` section. The `E` after the address indicates the label is external.

When we take the following part of the call graph,

```
+----- _write ( 4 )
|
+-( 2 )- _iowrite ( 2 )
|      |
|      +-( 2 )
|
+-( 2 )
```

we can see from the indentation in the structure of the tree that function `_write` calls function `_iowrite`. The total stack usage of function `_write` (including its calls) is given behind the function name:

```
_write ( 4 )
```

To determine the total stack usage we take the maximum of the following:

1. local usage before calling a function (the first value), added to the total usage of that function (the last value):

```
+-( 2 )- _iowrite ( 2 )
```

2. the usage of the function itself:

```
|
+-( 2 )
```

## 9.5 TYPE CHECKING

### 9.5.1 INTRODUCTION

By default the compiler and the assembler generate high-level type information. Unless you disable generation of type information (**-gn**), each object contains type information of high-level types. The linker compares this type information and warns you if there are conflicts. The linker distinguishes four types of conflicts:

1. Type not completely specified (W109). Occurs if you do not specify the depth of an array, or if you do not specify arguments in one of the function prototypes. The linker does not report this type of conflicts unless you specify a warning level 9 (**-w9**), default is warning level 8.
2. Compatible types, different definitions (W110). Occurs if for instance you link a long with an int. The TriCore takes both as 32 bits, so there will not be a problem. However, the code is not portable. Also structures or types with different names produce this warning. The warning level for this message is 8, so you can switch off this kind of message by specifying warning level 7 or less (**-w7**).
3. Signed/unsigned conflict (W111). If you link a signed int with an unsigned int, you get this message. In many cases there will be no problem, but the unsigned version can hold a bigger integer. The warning level of this warning is 6 and can be suppressed by specifying a warning level of 5 or less (**-w5**).
4. All other type conflicts (W112). If you get warning 112, there is probably a more serious type conflict. This can be a conflict in a function return type, a conflict in length between two built in types (short/long) or a completely different type. This warning has a level of 4, and can be switched off with warning level 3 or less (**-w3**).

## 9.5.2 RECURSIVE TYPE CHECKING

The linker compares type recursively. For instance, the type of `foo`:

```
struct s1 {
    struct s2 *s2_ptr;
};

struct s2 {
    int count;
} sample;

struct s1 foo = { &sample };
```

If you compile this source and link it with another compiled source with only `struct s2` different:

```
struct s1 {
    struct s2 *s2_ptr;
};

struct s2 {
    short count;
};

extern struct s1 foo;
```

message W112 (type conflict) will be generated. Although `struct s1` is the same in both cases, this is a real type conflict: For instance, the code `"foo.s2_ptr->count++"` produces different code in both objects.

If you have several conflicts in one symbol, the linker reports only the one with the lowest warning level. (The most serious one.)

### 9.5.3 TYPE CHECKING BETWEEN FUNCTIONS

If you use K&R style functions and disabled type checking for these functions with the **-K** option of **ctri**, it is not possible to check the type of the arguments and the number of arguments. Return types are 'int' if not specified. Prototypes are only needed if a function has a non-integer return type:

```
test2( par )
int par;
{
    test1( par );
    return test3( 1, 2 );
}
```

In this case, `test1` (defined in another source) has a return type `void`, and `test3` has a return type `int`, which is the default. At the default warning level, the linker does not report any conflict. If you should specify warning level 9 (**-w9**), the linker reports a 'not completely specified' type, because the linker is not able to check the arguments. Conflicts in return types cause real type conflicts at warning level 4.

If the source is ANSI style (which is recommended), the linker checks the types of all parameters, and the number of parameters. In this case the source of the example above looks like:

```
void test1( int );    /* ANSI style prototypes */
int test3( int, int );

test2( int par ) /* ANSI style function definition*/
{
    test1( par );
    return test3( 1, 2 );
}
```

Another source, containing the definition of `test1` and `test3` may look like:

```
void test1( int one )
{
    /*
    **  code for function test1
    */
    .
    .
    .
}

int test3( int one, int two )
{
    /*
    **  Code for function test3
    */
    .
    .
    .
}
```

Prototypes are only needed for functions which are referenced before they are defined within one source. However, it is a good practice to include a prototype file with prototypes of all the functions in a file. If you do so, type checking for functions is done by the compiler. Nevertheless, if you do not compile all sources after you have changed the prototype file, the linker will report the type conflict.

It is possible to add ANSI style prototypes to K&R style C-code. In this case full type checking for functions becomes available. To accomplish this, make a new header file with all prototypes for all functions in your application. Include this file in each source, or tell the compiler to include it for you by means of the option **-H**:

```
cctri -c -Hproto.h *.c
```

### 9.5.4 MISSING TYPES

In C you are allowed to define pointers to unspecified objects. The linker is not able to check such types. For instance:

```
struct s1 {  
    struct s2 *s2_ptr;  
};  
  
struct s1 foo;
```

The structure `s2` is not specified. Because the linker is not able to check whether `struct s2` is the same in all sources, a warning at level 9 will be generated:

```
lkttri W102 (9) <name>: Incomplete type specification, type index = T101
```

It is possible that the `struct s2` is known in an other source. If this source uses variable `foo`, a second message is generated, reporting a level 9 type conflict:

```
lkttri W109 (9) <f1>: Type not completely specified for symbol <foo> in <f2>
```

Because the type definition is not complete, the first warning reports that the linker cannot check the type, although this is allowed in C. This message is given once for each object for each incomplete type. The second warning reports a difference in types, an incomplete type versus a complete type. Note that all these warnings are only generated if you specify warning level 9 (**-w9**).

## 9.6 LINKER MESSAGES

There are four kinds of messages: fatal messages, error messages, warning messages and verbose messages. Fatal messages are generated if the linker is not able to perform its task due to the severity of the error. In those situations, the exit code will be 2. Error messages will be reported if an error occurred which is not fatal for the linker. However, the output of the linker is not usable. The exit code in case of one or more error messages will be 1. Warning messages are generated if the linker detects potential errors, but the linker is unable to judge those errors. The exit code will be 0 in this case, indicating a usable `.out` file. Of course, if the linker reports no messages at all, the exit code is 0 also.

Each linker message has a built-in warning level. With option **-wx** it is possible to suppress messages with a warning level above *x*.

Verbose messages are generated only if the verbose option (**-v**) is on. They report the progress of the link process.

Linker messages have the following layout:

```
TASKING TriCore object linker   vx.yrz Build nnn
Copyright 1996-year Altium BV   Serial# 00000000
lktri W112 (4) a.obj: Type conflict for symbol <f> in b.obj
```

The first line shows the banner of the TriCore linker. The second line reports a type conflict in the file *a.obj*. Apparently there is a conflicting type definition of the function *f* in module *b.obj*. The number between parentheses after the warning number, '(4)', shows the warning level.

There are four message groups:

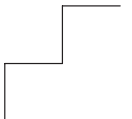
1. **Fatal** (always level 0):
  - Write error
  - Out of memory
  - Illegal input object
2. **Error** (always level 0):
  - Unresolved symbols (and no incremental linking)
  - Can't open input file
  - Illegal recursive use of an non reentrant function
3. **Warning** (levels from 1 to 9):
  - Type conflict between two symbols
  - Illegal option (Ignored)
  - No system library search path, and system library requested
4. **Verbose** (level not relevant, only given with option **-v**):
  - Extracting files from a library
  - Current file/library name
  - Pass one or pass two
  - Rescanning library for new unresolved symbols
  - Cleaning up temp files
  - warning level

# CHAPTER

# 10

**LOCATOR**

---





---

# 10

# CHAPTER

---

## 10.1 OVERVIEW

This chapter describes the TriCore locator.

The task of the locator is to locate a `.out` file, made by **lktri**, to absolute addresses. In an embedded environment an accurate description of available memory and information about controlling the behavior of the locator is crucial for a successful application. For example, it may be necessary to port applications to processors with different memory configurations, or it may be necessary to tune the location of sections to take full advantage of fast memory chips. To perform its task the locator needs a description of the derivative of the TriCore used. The locator uses a special language for this description: DELFEE, which stands for DEscriptive Language For Embedded Environments. This steering language is used in a special file, which is called the description file. See Appendix F *DEscriptive Language For Embedded Environments* for detailed information.

The description file is an optional parameter in the locator invocation. Without a description file name on the command line, or without the **-d** option, the locator searches the file `tri.dsc` in the current directory or in directory `etc` in the TriCore product tree.

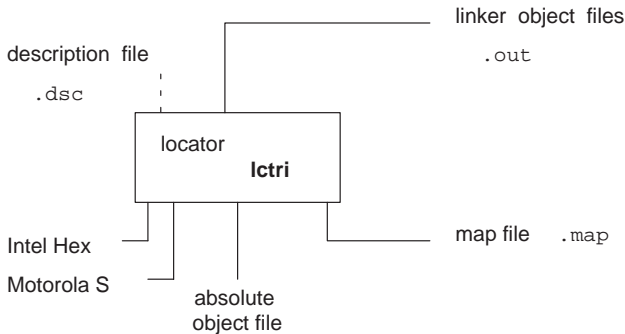


Figure 10-1: Locator

10.2 INVOCATION

The invocation of the locator is:

```
lctri [ option ]... [file ]...
```

When you use a UNIX shell (**C-shell**, **Bourne shell**), options containing special characters (such as '( )' ) must be enclosed with " ". The invocations for UNIX and DOS are the same, except for the **-?** option in the **C-shell**:

```
lctri "-?"      or      lctri -\?
```

Options may appear in any order. Options start with a '-'. They may be combined: **-eM** is equal to **-e -M**. Options that require a filename or a string may be separated by a space or not: **-oname** is equal to **-o name**. *file* may be any file with a .out or .dsc extension.

The locator recognizes the following options:

Option	Description
<b>-H</b> or <b>-?</b>	Display invocation syntax
<b>-M</b> [ <i>n</i> ]	Produce a locate map file (.map) with maximum width <i>n</i> ( <i>n</i> > 132)
<b>-N</b>	Generate external part
<b>-S</b> <i>space</i>	Generate specific <i>space</i>
<b>-V</b>	Display version header only
<b>-WAE</b>	Treat warning messages as errors
<b>-c</b>	Don't generate ROM copy for re-initializing data memory
<b>-d</b> <i>file</i>	Read description file information from <i>file</i> , '-' means stdin
<b>-e</b>	Clean up if erroneous result
<b>-em</b> <i>macro</i> [= <i>def</i> ]	Define preprocessor <i>macro</i>
<b>-err</b>	Redirect error messages (.elc)
<b>-f</b> <i>file</i>	Read command line information from <i>file</i> , '-' means stdin
<b>-f</b> <i>format</i>	Specify output format
<b>-o</b> <i>filename</i>	Specify name of output file
<b>-p</b>	Make a proposal for a software part on stdout
<b>-s</b>	Strip debug info from the input

Option	Description
<b>-v</b>	Verbose option. Print name of each file as it is processed
<b>-w <i>n</i></b>	Suppress messages above warning level <i>n</i> .

*Table 10-1: Options summary*

### **10.2.1 DETAILED DESCRIPTION OF LOCATOR OPTIONS**



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

## **-?/-H**

### **Option:**

**-?**  
**-H**

### **Description:**

Display an explanation of options at `stdout`.

### **Example:**

```
lctri  -?
```

## -C

### Option:



Select the | Configure Selected CPU... menu item. Disable the Initialized and cleared C variables check box in the Startup tab.



**-c**

### Description:

Do not generate ROM-copy for re-initializing data memory.

Normally, when running an application from an EPROM, the startup code will take care of clearing and initializing RAM data memory by keeping a copy of the initialized data variables in ROM. During program startup this ROM-copy is copied from program memory to data memory, thus assuring the program will have the same initial values on every program restart.

In some special situations you can save target memory usage by not generating a ROM copy of the initialized data variables in program memory. When you use the **-c** option, the program will be initialized with all required data values directly (this also includes the clearing and filling of data sections). The program initialization is done during downloading of the program. So, when loading the program, no additional copy action is required.

The draw back of using this option is that when you want to restart the program with the same initial values you have to download the whole program again, because a ROM-copy with the initial values is not available. Also, when the program runs from an EPROM, on program restart no attempt is made to clear or fill data sections, which was normally done at startup. So, a program depending on cleared or initialized RAM variables will fail then.

The **-c** option is of great use when loading the program from a host processor, because the program code and data can be loaded into a target system with the exact required memory space to run correctly (no extra space for copies in ROM is required).

## -d

### Option:



Select the | Linker/Locator Options... menu item. Enter the basename of the description file in the Locator control file (.dsc) field in the Locator tab.



**-d** *file*

### Arguments:

A filename to read description file information from. If *file* is a '-', the information is read from standard input.

### Description:

Read description file information from *file* instead of a .dsc file.

### Example:

To read description file information from file `tri.dsc`, enter:

```
lctri -dtri.dsc test.out
```

# -e

**Option:**

EDE always removes the output files on errors.



**-e**

**Description:**

Remove all locate products such as temporary files, the resulting output file and the map file, in case an error occurred.

**Example:**

```
lctri -e test.out
```

## -em

### Option:

**-em***macro*[=*def*]

### Arguments:

The macro you want to define and optionally its definition.

### Description:

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

### Example:

```
lctri myproject.out -o myproject.abs -emEDE="\myproject.i\" -M
```



## -err

### Option:



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



**-err**

### Description:

The locator redirects error messages to a file with the same basename as the output file and the extension `.elc`. The default filename is `a.elc`.

### Example:

To write errors to the file `a.elc` instead of `stderr`, enter:

```
lctri -err test.out
```

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

```
lctri -err test.out -otest.abs
```

## -f

### Option:

**-f** *file*

### Arguments:

A filename for command line processing. If *file* is a '-', the information is read from standard input. You need to provide the EOF code to close stdin (usually Ctrl-Z or Ctrl-D on UNIX).



*file* must not be a number in the range 0-4, because these numbers are used to specify an output format.

### Description:

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

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

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

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

Example:

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

or

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

or

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

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

Example:

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

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

### Example:

Suppose the file `mycmds` contains the following line:

```
-err  
test.out
```

The command line can now be:

```
lctri -f mycmds
```

# **-f format**

## **Option:**



Select the | Linker/Locator Options... menu item. Choose an Output Format option from the Format tab.



**-f format**

## **Arguments:**

*format* can be one of the following output formats:

- 0** = TIOF 695
- 1** = IEEE Std. 695 (Default)
- 2** = Motorola S records
- 3** = Intel Hex
- 4** = ELF/DWARF1.1

## **Description:**

Specify an output format. The default output format is IEEE Std. 695 (**-f1**), which can directly be used by the CrossView Pro debugger. The other output formats can be used for loading into a PROM-programmer.



Section *Format Suboptions.*,  
Appendix H, *IEEE-695 Object Format*,  
Appendix I, *Motorola S-Records*,  
Appendix J, *Intel Hex Records*.

# -M

## Option:



Select the | Linker/Locator Options... menu item. Enable the Produce a memory map file (.map) check box in the Locator tab.



**-M**[*n*]

## Arguments:

Optionally the maximum line width ( $n > 132$ ). If you omit the width, the default is 132 characters.

## Description:

Produce a locate map (.map). If no output filename is specified the default name is a.map. The map file shows the absolute position of each section. External symbols are listed with their absolute address, both sorted on address and sorted on symbol.

## Example:

To create the map file a.map, enter:

```
lctri -M test.out
```

# -N

**Option:**

Select the | Linker/Locator Options... menu item. Add the option to the Additional options field in the Locator tab.



-N

**Description:**

Generate external part. The external part contains all global symbol references.

## -O

### Option:



Select the | Linker/Locator Options... menu item. Add the option to the Additional options field in the Locator tab.



**-o** *filename*

### Arguments:

An output filename.

### Default:

The default filename depends on the output format specified:

Format	Default output name
<b>0</b>	a.abs
<b>1</b>	a.abs
<b>2</b>	a.sre
<b>3</b>	a.hex
<b>4</b>	a.elf

### Description:

Use *filename* as output filename of the locator.

### Example:

To create the output file `test.abs` instead of `a.abs`, enter:

```
lctri test.out -otest.abs
```

# -p

**Option:**

Select the | Linker/Locator Options... menu item. Add the option to the Additional options field in the Locator tab.



**-p**

**Description:**

Make a proposal for a software part in a description file on standard output.



## -S

### Option:



Select the | Linker/Locator Options... menu item. Add the option to the Additional options field in the Locator tab.



**-S** *space*

### Arguments:

The name of a space from a .dsc file.

### Description:

With this option you can generate a specific output file for a specified *space* instead of generating an output file containing all spaces.

# -S

## Option:



Select the | Linker/Locator Options... menu item. Disable the Include symbolic debug information check box in the Locator tab.



-s

## Description:

Strip debug information from the input file.

## -V

### Option:

-V

### Description:

With this option you can display the version header of the locator. This option must be the only argument of **lctri**. Other options are ignored. The locator exits after displaying the version header.

### Example:

```
lctri -V
```

```
TASKING TriCore locator      vx.yrz Build nnn  
Copyright 1996-year Altium BV  Serial# 00000000
```

## **-v**

### **Option:**



Select the | Linker/Locator Options... menu item. Add the option to the Additional options field in the Locator tab.



**-v**

### **Description:**

Verbose option. Print the name of each file as it is processed.

### **Example:**

```
lctri -v test.out
```

```
lctri V001 (1): Output format: IEEE 695
lctri V004 (1): Warning level 8
lctri V007 (1): Found file <tri.dsc> via path
                \ctri\etc\tri.dsc
lctri V002 (1): Starting pass 1
lctri V000 (1): File currently in progress:
                test.out
```

## -W

### Option:



Select the | Linker/Locator Options... menu item. Select a warning level from the Suppress warning messages above list box in the Locator tab.



**-w** *level*

### Arguments:

A warning level between 0 and 9 (inclusive).

### Default:

**-w8**

### Description:

Give a warning level between 0 and 9 (inclusive). All warnings with a level above *level* are suppressed. The level of a message is printed between parentheses after the warning number. If you do not use the **-w** option, the default warning level is 8.

### Example:

To suppress warnings above level 5, enter:

```
lctri -w5 test.out
```

Using the control program:

```
cctri -Wlc-w5 test.out
```

# -WAE

## Option:



Select the | Linker/Locator Options... menu item. Enable the Treat warnings as if they were errors check box in the Locator tab.



**-WAE**

## Description:

Treat warning messages as errors. This also affects the return value of the application when only errors occur. A build process will now stop when warnings occur.

## Example:

```
lctri -WAE test.out
```

10.2.2 FORMAT SUBOPTIONS

The layout of the **-fformat** switch as shown in the previous section has some extra capabilities. The general form of *format* is:

*format\_number*[*format\_option*]...

The first character is a single digit known as the format specifier. This format specifier can be followed by one or more *format\_options*. These *format\_options* are in principle output format dependent. Currently the following *format\_options* are known:

Suboption	Description	Valid Formats
<b>a</b>	Sort addresses in ascending order	2
<b>bsize</b>	Specify the output buffer size	1, 2, 3
<b>c</b>	Generate file for each chip	3
<b>s</b>	Emit start address record	3
<b>S1, S2, S3</b>	Force Motorola S1, S2 or S3 records	2
<b>u</b>	Unsorted addresses (default)	2

Table 10-2: Format suboptions

For format 2 and 3 the buffer size sets the length of an output record exclusive record code, address and checksum. The following Intel Hex record has a buffer size of 32 bytes:

```
:20004000                                ( Record code and address )
000028A101002925FBFF6015FFFF6B25DDFF001000000000000020A101002925
                                           ( 32 data bytes )
8F                                           ( Checksum )
```

For format 1, *size* is the maximum number of bytes in one LD command.

Examples:

- Format 2 with buffer size 64: **-f2b64**
- Format 1 with buffer size 128: **-f1b128**
- Format 2 with ascending addresses: **-f2a**
- Format 2 with unsorted addresses: **-f2u**
- Format 3 with separate hex files for each chip: **-f3c**
- Format 3 including start address record: **-f3s**
- Format 2 with S2 records: **-f2S2**

## 10.3 LOCATING YOUR APPLICATION

To ease the locating process a set of locator controls has been developed. Instead of manually adapting the DELFEE files (`tri.cpu`, `tri.def` and `tri.dsc`), you can now write a simple and quite straightforward locator control file. The locator control file is passed to the locator on the command line.



When using EDE, the locating process is even simpler, since EDE generates a locator control file for you. This control file, *project.i*, is the result of your (graphical) selections in the | Linker/Locator options... tabs.

The locator still uses the DELFEE files. However, the contents of the DELFEE files are now controlled by the new locator control file (`tri.i`). You can, of course, still inspect (and even edit) the files `tri.cpu`, `tri.def` and `tri.dsc`, but it is recommended not to touch them. You can specify the locating behavior and your memory configuration (such as ROM and RAM areas) using EDE.

The locator supports the following controls (which are implemented as C language macros):

Locator control	Default	Description
RESET	0x30000000	Reset start address
USTACK	12K	User stack size
ISTACK	4K	Interrupt stack size
HEAP	32K	Heap size
CSA	256	Number of context blocks
XVWBUF	32	CrossView Pro buffer size
RAM( <i>name</i> , <i>start</i> , <i>end</i> )	—	Define RAM memory area
ROM( <i>name</i> , <i>start</i> , <i>end</i> )	—	Define ROM memory area
RESERVED( <i>start</i> , <i>end</i> )	—	Exclude memory range from section allocation
ROM_LINEAR( <i>name</i> [ <i>addr</i> = <i>address</i> ] )	—	Specify section with <i>name</i> must be located at address <i>addr</i> or before/after another ROM_LINEAR control.
ROM_ABS24( <i>name</i> [ <i>addr</i> = <i>address</i> ] )	—	Specify section with <i>name</i> must be located at address <i>addr</i> or before/after another ROM_ABS24 control.



Locator control	Default	Description
ROM_ABS18( <i>name</i> [addr= <i>address</i> ] )	—	Specify section with <i>name</i> must be located at address <i>addr</i> or before/after another ROM_ABS18 control.
RAM_LINEAR( <i>name</i> [addr= <i>address</i> ] )	—	Specify section with <i>name</i> must be located at address <i>addr</i> or before/after another RAM_LINEAR control.
RAM_ABS18( <i>name</i> [addr= <i>address</i> ] )	—	Specify section with <i>name</i> must be located at address <i>addr</i> or before/after another RAM_ABS18 control.

Table 10-3: Locator controls

Below is an example locator control file.

```
#define RESET      0x30000000    /* reset start address */
#define USTACK     12k          /* user stack size */
#define ISTACK     4k           /* interrupt stack size */
#define HEAP       32k          /* heap size */
#define CSA        256          /* number of context blocks */
#define XVWBUF     32           /* size of CrossView Pro buffer */

RAM( sdata , 0x00000000 , 0x000007ff ) // 2k
RAM( ddata , 0x10000000 , 0x1000ffff ) // 64k
ROM( dcode , 0x20000000 , 0x2001ffff ) // 128k
ROM( scode , 0x28000000 , 0x280003ff ) // 1k
ROM( boot , 0xd0000000 , 0xdfffffff )
ROM( fpi1 , 0x2ffeff00 , 0x7fffffff )
RAM( fpi2 , 0x80000000 , 0xcfffffff )

// RESERVED( 0x28000000, 0x280003ff )

// ROM_ABS24( code.cstart )
```

EDE generates the locator control file and the proper invocation for you.

If you do not use EDE, you must create a locator control file yourself (for example, `myproject.i`) and specify it on the command line. The default name is `tri.i`. For example:

```
lctri myproject.out -o myproject.abs -emeDE="myproject.i" -M
```

## **10.4 CALLING THE LOCATOR VIA THE CONTROL PROGRAM**

It is recommended to call the locator via the control program **cctri**. The control program translates certain options for the locator (e.g., **-srec** to **-f2**). Other options (such as **-M**) are passed directly to the locator. Typical, you can use the control program to get an **.abs** file directly from **.c**, **.src**, **.asm** or **.obj** files. The invocation:

```
cctri -M -g addone.src demo.c -o demo.abs tri.dsc
```

builds an absolute demo file called **demo.abs** ready for running via the CrossView Pro debugger.

## **10.5 LOCATOR OUTPUT**

The locator produces an absolute file and, if requested, a map file and/or an error file. The output file is absolute and in Intel Hex format, Motorola S-record format or in IEEE-695 format, depending on the usage of the **-f** option. The default output name is **a.hex**, **a.sre** or **a.abs**, respectively.

The map file (**-M** option) always has the same basename as the output object file, with an extension **.map**. The map file shows the absolute position of each section. External symbols are listed with their absolute address, both sorted on address and sorted on symbol.

The error output file (**-err** option) has the same name as the object output file, but with extension **.elc**. Errors occurred before the **-err** option is evaluated are printed on stderr.

## **10.6 LOCATOR MESSAGES**

There are four kinds of messages: fatal messages, error messages, warning messages and verbose messages. Fatal messages are generated if the locator is not able to continue with its task due to the severity of the error. In those situations, the exit code will be 2. Error messages will be reported if an error occurred, not fatal for the locator. However, the output of the locator is not usable. The exit code in case of one or more error messages will be 1. Warning messages are generated if the locator detects potential errors, but the locator is unable to judge those errors. The exit code will be 0 in this case, indicating a usable **.abs** file. Of course, if the locator reports no messages, the exit code is also 0.

Each locator message has a built-in warning level. With option **-wx** it is possible to suppress messages with a warning level above *x*.

Verbose messages are generated only if the verbose option (**-v**) is on. They report the progress of the locate process.

Locator messages have the following layout:

```
TASKING TriCore locator          vx.yrz Build nnn
Copyright 1996-year Altium BV    Serial# 00000000
lctri W112 (3) calc.out: Copy table not referenced, initial data is
not copied
```

The first line shows the locator banner. (Suppressed if the locator invocation is done by the control program.) The second line shows the warning. The number after the warning number shows the warning level.

## 10.7 COPY TABLE

One of the actions with the process initialization is copy data from ROM to RAM, and initialize memory with the CLEAR attribute. The locator generates a copy table for each process. The copy table can be referenced by label `_lc_cp`. One entry in the copy table has the following layout:

```
typedef struct cp_entry {
    int          cp_actions;      /* 4 bytes */
    unsigned char *cp_destin;     /* 4 byte address */
    unsigned char *cp_source;     /* 4 byte address */
    int          cp_length;       /* 4 byte length */
} cp_entry_t;
```

The first member, `cp_actions`, defines what action you must perform with the current entry. Actions are organized as a bit per action:

value 0	Reached end of the table.
CP_COPY (value 1)	Copy from <code>cp_source</code> to <code>cp_destin</code> over <code>cp_length</code> bytes.
CP_BSS (value 2)	Clear memory from <code>cp_destin</code> over <code>cp_length</code> bytes.

Table entries are generated as follows:

- one entry for each section with the CLEAR attribute
- one entry for each section with the INIT attribute
- one 'zero' entry to indicate the end-of-table.

If there is nothing to do (no sections to clear and no data to copy) the copy table has only one action entry with value zero.

At C level, the copy table can be declared as:

```
cpt_t _lc_cp;
```

And accessing a member of entry *x* becomes:

```
_lc_cp[ x ].cp_actions;
```

If label `_lc_cp` is not used, the table is not generated.

## 10.8 LOCATOR LABELS

The locator assigns addresses to the following labels when they are referenced:

<code>_lc_cp :</code>	Start of copy table The copy table gives the source and destination addresses of sections to be copied. This table will be generated by the locator only if this label is used.
<code>_lc_bs :</code>	Begin of stack space (using keyword <code>stack</code> ).
<code>_lc_es :</code>	End of stack space.
<code>_lc_b_name :</code>	Begin of section <i>name</i> .
<code>_lc_e_name :</code>	End of section <i>name</i> .
<code>_lc_u_name :</code>	User defined label. The label must be defined in the description file. For example:  <code>label mylab;</code>
<code>_lc_ub_name :</code>	Begin of user defined label. The label must be defined in the description file. For example:  <code>label mybuffer length=100;</code>

`_lc_ue_name :` End of user defined label.

### **10.8.1 LOCATOR LABELS REFERENCE**

This section contains a description of all locator labels. Locator labels are labels starting with **\_lc\_**. They are ignored by the linker and resolved at locate time. 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 locator generated data. The data is only generated if the label is used.

Because labels that start with **\_lc\_** are treated differently in both the linker and the locator, you can only use this type of labels as references, not as definitions.

# ***\_lc\_b\_section,*** ***\_lc\_e\_section***

## **Syntax:**

```
extern unsigned char  _lc_b_section[ ];  
extern unsigned char  _lc_e_section[ ];
```

## **Description:**

You can use the general locator labels ***\_lc\_b\_section*** and ***\_lc\_e\_section*** to obtain the addresses of section *section* in a program. The **b** version points to the start of the section, while the **e** version points to its end.

You can replace the dot before a section name by an underscore ( **\_** ), making it possible to access these labels from 'C'. This convention introduces a possible name conflict. If, for example, both sections *.text* and *\_text* exist, the general label ***\_lc\_b\_\_text*** is set to the start of *\_text*. The label for section *.text* is only usable at assembly level with its real name. Of course, you should avoid such a conflict by not using section names with a leading underscore.

## **Example:**

```
printf( "Text size is 0x%x\n",  
        _lc_e__text - _lc_b__text );
```

## **`_lc_bh, _lc_eh`**

### **Syntax:**

```
extern unsigned char _lc_bh[];
extern unsigned char _lc_eh[];
```

### **Description:**

All locator **h** labels are related to the heap. You can allocate a heap by defining it in a cluster description. See also the Delfee keyword **heap**.

**`_lc_bh`** is a label at the begin of the heap. At 'C' level **`_lc_bh`** represent the heap. The label is defined as a char array, but an array of any basic type will do. **`_lc_eh`** represents the end of the heap.

### **Example:**

Heap definition:

```
block total_range {
    .
    .
    cluster ram {
        amode data {
            heap length = 200;
        }
    }
    .
}
```

Sbrk code:

```
extern unsigned char _lc_bh[];
extern unsigned char _lc_eh[];

static char *
sbrk( long length ) {
    .
    .

    if ( (lastmem + length) > _lc_eh ) {
        return (char *) -1; /* overflow */
    }
```

# **\_lc\_bs, \_lc\_es**

## **Syntax:**

```
extern unsigned char    _lc_bs[ ];
extern unsigned char    _lc_es[ ];
```

## **Description:**

All locator **s** labels are related to the stack. You can allocate a stack by defining it in a cluster description. See also the Delfee keyword **stack**.

**\_lc\_bs** is a label at the begin of the stack. At 'C' level **\_lc\_bs** represent the stack. The label is defined as a char array, but an array of any basic type will do. **\_lc\_es** represents the end of the stack. Because **\_lc\_es** is on a higher address than **\_lc\_bs** and because the stack for the TriCore grows to lower addresses, the stack actually starts at the label **\_lc\_es** and ends at **\_lc\_bs**.

## **Example:**

Stack definition:

```
block total_range {
    cluster ram {
        amode data {
            stack length = 100;
        }
    }
}
```

Stack initialization:

```
_START:
    LD SP, #_lc_es        ; set stack pointer to
                        ; begin of stack space
```



## **\_lc\_cp**

### **Syntax:**

```
extern char *_lc_cp;
```

### **Description:**

The copy table is generated per process. Each entry in this table represents a copy or clearing action. Entries for the table are automatically generated by the locator for:

- All sections with attribute b, which must be cleared at startup time : a clearing action.
- All sections with attribute i, which must be copied from rom to ram at program startup: a copy action

The layout of the copy table is described in 10.7, *Copy Table*. Type `cpt_t` is defined in `locate.h`.

# **`_lc_u_identifier`**

## **Syntax:**

```
extern int _lc_u_identifier[];
```

## **Description:**

This locator label can be defined by the user by means of the Delfee keyword **label**. This label must be defined in the Delfee file without the prefix **`_lc_u_`**. From assembly the label can be referenced with the prefix **`_lc_u_`**.

## **Example:**

In description file:

```
block total_range {  
    cluster ram {  
        amode data {  
            label bstart;  
            section text;  
            label bend;  
        }  
    }  
    .  
    .  
    .  
}
```

From C:

```
#include <stdio.h>  
extern int _lc_u_bstart[];  
extern int _lc_u_bend[];  
int main()  
{  
    printf( "Size of cluster ram is %d\n",  
        (long)_lc_u_bend -  
        (long)_lc_u_bstart );  
}
```

# **`_lc_ub_identifier,` `_lc_ue_identifier`**

## **Syntax:**

```
extern int _lc_ub_identifier[ ];
extern int _lc_ue_identifier[ ];
```

## **Description:**

These locator labels can be defined by the user by means of the Delfee keywords **`reserved label=`**. The locator labels specify the begin and end of a reserved area. The *identifier* is the name for the reserved area and must be defined in the Delfee file without the prefix **`_lc_ub_`** or **`_lc_ue_`**. From assembly the labels can be referenced with the prefix **`_lc_ub_`** and **`_lc_ue_`**.

## **Example:**

In description file:

```
block total_range {
  cluster ram {
    attribute w;
    amode data {
      section selection=w;
      reserved label=xvwbuffer length=0x10;
      // Start address of reserved area is
      // label _lc_ub_xvwbuffer
      // End address of reserved area is
      // label _lc_ue_xvwbuffer
    }
  }
}
```

From C:

```
#include <stdio.h>
extern int _lc_ub_xvwbuffer[];
extern int _lc_ue_xvwbuffer[];
int main()
{
    printf( "Size of reserved area xvwbuffer is %d\n",
            (long)_lc_ue_xvwbuffer -
            (long)_lc_ub_xvwbuffer );
}
```

# LOCATOR

# CHAPTER

# 11

## UTILITIES

---



**TASKING**



---

# 11

# CHAPTER

---

## 11.1 OVERVIEW

The following utilities are supplied with the Cross-Assembler for the TriCore processor family which can be useful at various stages during program development.

- |              |  |
|--------------|--|
| <b>artri</b> | An IEEE archiver. This is a librarian facility, which can be used to create and maintain object libraries.                   |
| <b>cctri</b> | A control program for the TriCore tool chain.  |
| <b>mktri</b> | A utility program to maintain, update, and reconstruct groups of programs.   |
| <b>prtri</b> | An IEEE object reader that views the contents of files which have been created by a tool from the TASKING TriCore toolchain. |



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

The utilities are explained on the following pages.



## 11.2 ARTRI

### Name

**artri** IEEE archiver and library maintainer

### Synopsis

**artri** *key\_option* [*option*]... *library* [*object\_file*]...

**artri** **-V**

**artri** **-?** ( UNIX C-shell : **"-?"** or **-\\?** )

### Description

With **artri** you can combine separate object modules in a library file. The linker optionally includes modules from a library when a specific module resolves an external symbol definition in one of the modules that has been read before. The library maintainer **artri** is a program to build library files and it offers the possibility to replace, extract or remove modules from an existing library.

*key\_option* one of the main options indicating the action **artri** has to take. Key options may appear in any order, at any place.

*option* optional sub-options as explained on the next pages.

*library* is the library file.

*object\_file* is an object module to be added, extracted, replaced or removed from the library.

### Options

You may specify options with or without a leading '-'. Options may occur in random order. You may also combine options. So **-xv** is allowed. **-V** and **-?** however, must be the only option on the command line.

### Key options:

- d** Delete the named object modules from the library.
- m** Move the named object modules to the end of the library, or to another position as specified by one of the positioning options.
- p** Print the named object modules in the library on standard output.



The object is in binary format. The **-p** option is normally used with a redirection:

```
artri -p lib.a object.obj > t.obj
```

- r** Replace the named object modules in the library if they exist. If they are not in the library, add them. If no names are given, only those object modules are replaced for which a file with the same name is found in the current directory. New modules are placed at the end.
- t** Print a table of contents of the library. If no names are given, all object modules in the library are printed. If names are given, only those object modules are tabled.
- x** Extract the named object modules from the library. If no names are given, all modules are extracted from the library. In neither case does **x** alter the library.

#### Other options:

- ?** Display an explanation of options at `stdout`.
- V** Display version information at `stderr`.
- a *posname***  
Append or move new object modules after existing module *posname*. This option can only be used in combination with the **m** or **r** option.
- b *posname***  
Insert or move new object modules before existing module *posname*. This option can only be used in combination with the **m** or **r** option.
- c** Create the library file without notification if the library does not exist.
- f *file*** Read options from file *file*. '-' means `stdin`. You need to provide the EOF code to close `stdin` (usually Ctrl-Z or Ctrl-D on UNIX).
- o** Reset the last-modified date to the date recorded in the library. It can only be used in combination with the **x** option.
- s** Print a list of symbols. This option must be combined with **-t**.

- s1**            Print a list of symbols. Each symbol is preceded by the library name and the name of the object file. This option must be combined with **-t**.
- u**            Replace only those object modules with the last-modified date later than the library file. It can only be used in combination with the **r** option.
- v**            Verbose. Under the verbose option, **artri** gives a module-by-module description of the making of a new library file from the old library and the constituent modules. It can only be used in combination with the **d**, **m**, **r**, or **x** option.
- wn**           Set warning level *n*.

## Examples

1. Create library **clib.a** consisting of the modules **cstart.obj**, and **demo.obj** :

```
artri cr clib.a cstart.obj demo.obj
```

2. Extract all modules form library **clib.a** :

```
artri x clib.a
```

3. Print a list of symbols from library **clib.a** :

```
artri ts clib.a

cstart.obj
  symbols:
    _start
    _copytable
demo.obj
  symbols:
    _entry
```

4. Print a list of symbols from library **clib.a** in a different form:

```
artri ts1 clib.a

clib.a:cstart.obj:_start
clib.a:cstart.obj:_copytable
clib.a:demo.obj:_entry
```

5. Delete module **demo.obj** from library **clib.lib** :

```
artri d clib.a demo.obj
```

## 11.3 CCTRI

### Name

**cctri**                      control program for the TriCore tool chain

### Synopsis

**cctri** [ *[option]*... *[control]* ... *[file]*... ]...

**cctri** -V

**cctri** -? ( UNIX C-shell : "-?" or -\? )

### Description

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

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

The control program recognizes the following argument types:

- Arguments starting with a '-' character are options. Some options are interpreted by the control program itself; the remaining options are passed to those programs in the tool chain that accept the option.
- Arguments with a .c suffix are interpreted as C source programs and are passed to the compiler.
- Arguments with a .asm or .src suffix are interpreted as assembly source files. They are directly passed to the assembler.
- Arguments with a .a suffix are interpreted as library files and are passed to the linker.
- Arguments with a .obj suffix are interpreted as object files and are passed to the linker.
- Arguments with a .out suffix are interpreted as linked object files and are passed to the locator. The locator accepts only one .out file in the invocation.
- Arguments with a .dsc suffix are treated as locator command files. If there is a file with extension .dsc on the command line, the control program assumes a locate phase has to be added. If there is no file with extension .dsc, the control program stops after linking (unless it has been directed to stop in an earlier phase)

- If other arguments are found, an error message is given.

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

## Options

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

**-cc**  
**-cs**  
**-c**  
**-cl**  
**-cm**  
**-cp**

Normally the control program invokes all stages to build an absolute file from the given input files. With these options it is possible to stop after one of the stages or to skip the linker stage.

With the **-cc** option the control program stops after compilation of the C++ files and retains the resulting .c files.

With the **-cs** option the control program stops after the C compiler or macro preprocessor, with as output file the assembly source file (.src).

With **-c** option the control program stops after the assembler, with as output an object file (.obj).

With the **-cl** option the control program stops after the link stage, with as output a linker object file (.lno).

With the **-cm** option the control program always also invokes the C++ muncher.

With the **-cp** option the control program always also invokes the C++ pre-linker.

**-f file**

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

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

1. It is possible to have multiple arguments on the same line in the command file.
2. To include whitespace in the argument, surround the argument with either single or double quotes.
3. If single or double quotes are to be used inside a quoted argument, we have to go by the following rules:

- a. If the embedded quotes are only single or double quotes, use the opposite quote around the argument. Thus, if a argument should contain a double quote, surround the argument with single quotes.
- b. If both types of quotes are used, we have to split the argument in such a way that each embedded quote is surrounded by the opposite type of quote.

Example:

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

or

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

or

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

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

Example:

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

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

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

**-elf**  
**-ieee**  
**-ihex**  
**-srec**  
**-tiof**

With these options you can specify the locator output format of the absolute file. The output file can be an ELF/DWARF1.1 (`.elf`), IEEE-695 file (`.abs`), Intel Hex file (`.hex`), Motorola S-record file (`.sre`) or TIOF-695 file (`.abs`). The default output is IEEE-695 (`.abs`).

**-fptrap**

Use floating point library with trap handling (`libfpt.a`). Without this option a floating point library is selected which uses no trapping.

**-nolib**

With this option the control program does not supply the standard libraries to the linker. Normally the control program supplies the default C and run-time libraries to the linker. Which libraries are needed is derived from the compiler options.

**-o *file***

Normally, this option is passed to the locator to specify the output file name. When you use the **-cl** option to suppress the locating phase, the **-o** option is passed to the linker. When you use the **-c** option to suppress the linking phase, the **-o** option is passed to the assembler, provided that only one source file is specified. When you use the **-cs** option to suppress the assembly phase, the **-o** option is passed to the compiler. The argument may be either directly appended to the option, or follow the option as a separate argument of the control program.

**-tmp**

With this option the control program creates intermediate files in the current directory. They are not removed automatically. Normally, the control program generates temporary files for intermediate translation results, such as compiler generated assembly files, object files and the linker output file. If the next phase in the translation process completes successfully, these intermediate files will be removed.

**-v**

When you use the **-v** option, the invocations of the individual programs are displayed on standard output, preceded by a '+' character.



- v0** This option has the same effect as the **-v** option, with the exception that only the invocations are displayed, but the programs are not started.
- wc++** Enable C and assembler warnings for C++ files. The assembler and C compiler may generate warnings on C output of the C++ compiler. By default these warnings are suppressed.

### ***Environment Variables used by cctri***

The control program uses the following environment variables:

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

## 11.4 MKTRI

### Name

**mktri** maintain, update, and reconstruct groups of programs

### Syntax

**mktri** [*option*]... [*target*]... [*macro=value*]...

**mktri -V**

**mktri -?** ( UNIX C-shell: **"-?"** or **-\\?** )

### Description

**mktri** takes a file of dependencies (a 'makefile') and decides what commands have to be executed to bring the files up-to-date. These commands are either executed directly from **mktri** or written to the standard output without executing them.

If no target is specified on the command line, **mktri** uses the first target defined in the first makefile.



Long filenames are supported when they are surrounded by double quotes ("). It is also allowed to use spaces in directory names and file names.

### Options

**-?** Show invocation syntax.

**-D** Display the text of the makefiles as read in.

**-DD** Display the text of the makefiles and 'mktri.mk'.

**-G *dirname*** Change to the directory specified with *dirname* before reading a makefile. This makes it possible to build an application in another directory than the current working directory.

**-K** Do not remove temporary files.

**-S** Undo the effect of the **-k** option. Stop processing when a non-zero exit status is returned by a command.

**-V** Display version information at stderr.

**-W *target*** Execute as if this target has a modification time of "right now". This is the "What IF" option.

- d** Display the reasons why **mktri** chooses to rebuild a target. All dependencies which are newer are displayed.
- dd** Display the dependency checks in more detail. Dependencies which are older are displayed as well as newer.
- e** Let environment variables override macro definitions from makefiles. Normally, makefile macros override environment variables. Command line macro definitions always override both environment variables and makefile macros definitions.
- f *file*** Use the specified file instead of 'makefile'. A - as the makefile argument denotes the standard input.
- i** Ignore error codes returned by commands. This is equivalent to the special target .IGNORE:.
- k** When a nonzero error status is returned by a command, abandon work on the current target, but continue with other branches that do not depend on this target.
- m *file*** Read command line information from *file*. If *file* is a '-', the information is read from standard input.
- n** Perform a dry run. Print commands, but do not execute them. Even lines beginning with an @ are printed. However, if a command line is an invocation of **mktri**, that line is always executed.
- q** Question mode. **mktri** returns a zero or non-zero status code, depending on whether or not the target file is up to date.
- r** Do not read in the default file 'mktri.mk'.
- s** Silent mode. Do not print command lines before executing them. This is equivalent to the special target .SILENT:.
- t** Touch the target files, bringing them up to date, rather than performing the rules to reconstruct them.
- w** Redirect warnings and errors to standard output. Without, **mktri** and the commands it executes use standard error for this purpose.

*macro=value*

Macro definition. This definition remains fixed for the **mktri** invocation. It overrides any regular definitions for the specified macro within the makefiles and from the environment. It is inherited by subordinate **mktri**'s but act as an environment variable for these. That is, depending on the **-e** setting, it may be overridden by a makefile definition.

## Usage

### *Makefiles*

The first makefile read is 'mktri.mk', which is looked for at the following places (in this order):

- in the current working directory
- in the directory pointed to by the HOME environment variable
- in the etc directory relative to the directory where **mktri** is located

Example (PC):

when **mktri** is installed in \CTRI\BIN the directory \CTRI\ETC is searched for makefiles.

Example (UNIX):

when **mktri** is installed in /usr/local/ctri/bin the directory /usr/local/ctri/etc is searched for makefiles.

It typically contains predefined macros and implicit rules.

The default name of the makefile is 'makefile' in the current directory. If this file is not found on a UNIX system, the file 'Makefile' is then used as the default. Alternate makefiles can be specified using one or more **-f** options on the command line. Multiple **-f** options act as if all the makefiles were concatenated in a left-to-right order.

The makefile(s) may contain a mixture of comment lines, macro definitions, include lines, and target lines. Lines may be continued across input lines by escaping the NEWLINE with a backslash (\). If a line must end with a backslash then an empty macro should be appended. Anything after a '#' is considered to be a comment, and is stripped from the line, including spaces immediately before the '#'. If the '#' is inside a quoted string, it is not treated as a comment. Completely blank lines are ignored.

An *include* line is used to include the text of another makefile. It consists of the word "include" left justified, followed by spaces, and followed by the name of the file that is to be included at this line. Macros in the name of the included file are expanded before the file is included. Include files may be nested.

An *export* line is used for exporting a macro definition to the environment of any command executed by **mktri**. Such a line starts with the word "export", followed by one or more spaces and the name of the macro to be exported. Macros are exported at the moment an export line is read. This implies that references to forward macro definitions are equivalent to undefined macros.

### **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 macroname
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 *macroname* 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.

## Macros

Macros have the form ‘WORD = text and more text’. The WORD need not be uppercase, but this is an accepted standard. Spaces around the equal sign are not significant. Later lines which contain `$(WORD)` or `${WORD}` will have this replaced by ‘text and more text’. 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 macro invocations. The right side of a macro definition is expanded when the macro is actually used, not at the point of definition.

Example:

```
FOOD = $(EAT) and $(DRINK)
EAT = meat and/or vegetables
DRINK = water
export FOOD
```

‘\$(FOOD)’ becomes ‘meat and/or vegetables and water’ and the environment variable FOOD is set accordingly by the export line. However, when a macro definition contains a direct reference to the macro being defined then those instances are expanded at the point of definition. This is the only case when the right side of a macro definition is (partially) expanded. For example, the line

```
DRINK = $(DRINK) or beer
```

after the export line affects ‘\$(FOOD)’ just as the line

```
DRINK = water or beer
```

would do. However, the environment variable FOOD will only be updated when it is exported again.



You are advised not to use the double quotes (") for long filename support in macros, otherwise this might result in a concatenation of two macros with double quotes (") in between.

## Special Macros

**MAKE** This normally has the value **mktri**. Any line which invokes MAKE temporarily overrides the **-n** option, just for the duration of the one line. This allows nested invocations of MAKE to be tested with the **-n** option.

**MAKEFLAGS**

This macro has the set of options provided to **mktri** as its value. If this is set as an environment variable, the set of options is processed before any command line options. This macro may be explicitly passed to nested **mktri**'s, but it is also available to these invocations as an environment variable. The **-f** and **-d** flags are not recorded in this macro.

**PRODDIR** This macro expands the name of the directory where **mktri** is installed without the last path component. The resulting directory name will be the root directory of the installed TriCore package, unless **mktri** is installed somewhere else. This macro can be used to refer to files belonging to the product, for example a library source file.

Example:

```
DOPRINT = $(PRODDIR)/lib/src/_doprint.c
```

When **mktri** is installed in the directory `/ctri/bin` this line expands to:

```
DOPRINT = /ctri/lib/src/_doprint.c
```

**SHELLCMD**

This contains 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\_CCPRG**

This macro contains the name of the control program. If this macro and the **TMP\_CCOPT** macro are set and the command line argument list for the control program exceeds 127 characters then **mktri** will create a temporary file filled with the command line arguments. **mktri** will call the control program with the temporary file as command input file. This macro is only known by the PC version of **mktri**.

**TMP\_CCOPT**

This macro contains the option for the control program which tells the control program to read a file as command arguments. This macro is only known by the PC version of **mktri**.

Example:

```
TMP_CCPROG = cctri
TMP_CCOPT  = -f
```

\$ 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.

The match function yields all arguments which match a certain suffix:

```
$(match .obj prog.obj sub.obj mylib.a)
```



will yield

```
prog.obj sub.obj
```

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.obj sub.obj)
```

will result in

```
prog.obj
sub.obj
```

Function arguments may be macros or functions themselves. So,

```
$(separate "\n" $(match .obj $!))
```

will yield all object files the current target depends on, separated by a newline string.

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)
```

will yield

```
echo "I'll show you the \"protect\" function"
```

The `exist` function expands to its second argument if the first argument is an existing file or directory.

Example:

```
$(exist test.c cctri test.c)
```

When the file `test.c` exists it will yield:

```
cctri test.c
```

When the file `test.c` does not exist nothing is expanded.

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 cctri test.c)
```

## ***Targets***

A target entry in the makefile has the following format:

```
target ... : [dependency ...] [; rule]
          [rule]
          ...
```

Any line which does not have leading white space (other than macro definitions) is a 'target' line. Target lines consist of one or more filenames (or macros which expand into same) called targets, followed by a colon (:). The ':' is followed by a list of dependent files. The dependency list may be terminated with a semicolon (;) which may be followed by a rule or shell command.

Special allowance is made on MS-DOS for the colons which are needed to specify files on other drives, so for example, the following will work as intended:

```
c:foo.obj : a:foo.c
```

If a target is named in more than one target line, the dependencies are added to form the target's complete dependency list.

The dependents are the ones from which a target is constructed. They in turn may be targets of other dependents. In general, for a particular target file, each of its dependent files is 'made', to make sure that each is up to date with respect to its dependents.

The modification time of the target is compared to the modification times of each dependent file. If the target is older, one or more of the dependents have changed, so the target must be constructed. Of course, this checking is done recursively, so that all dependents of dependents of dependents of ... are up-to-date.

To reconstruct a target, **mktri** expands macros and functions, strips off initial white space, and either executes the rules directly, or passes each to a shell or COMMAND.COM for execution.

For target lines, macros and functions are expanded on input. All other lines have expansion delayed until absolutely required (i.e. macros and functions in rules are dynamic).

### ***Special Targets***

- .DEFAULT:      The rule for this target is used to process a target when there is no other entry for it, and no implicit rule for building it. **mktri** ignores all dependencies for this target.
- .DONE:        This target and its dependencies are processed after all other targets are built.
- .IGNORE:      Non-zero error codes returned from commands are ignored. Encountering this in a makefile is the same as specifying **-i** on the command line.
- .INIT:        This target and its dependencies are processed before any other targets are processed.
- .SILENT:      Commands are not echoed before executing them. Encountering this in a makefile is the same as specifying **-s** on the command line.
- .SUFFIXES:    The suffixes list for selecting implicit rules. Specifying this target with dependents adds these to the end of the suffixes list. Specifying it with no dependents clears the list.
- .PRECIOUS:    Dependency files mentioned for this target are not removed. Normally, **mktri** removes a target file if a command in its construction rule returned an error or when target construction is interrupted.

### ***Rules***

A line in a makefile that starts with a TAB or SPACE is a shell line or rule. This line is associated with the most recently preceding dependency line. A sequence of these may be associated with a single dependency line. When a target is out of date with respect to a dependent, the sequence of commands is executed. Shell lines may have any combination of the following characters to the left of the command:

@ will not echo the command line, except if **-n** is used.

- **mktri** will ignore the exit code of the command, i.e. the `ERRORLEVEL` of MS-DOS. Without this, **mktri** terminates when a non-zero exit code is returned.

- + **mktri** will use a shell or `COMMAND.COM` to execute the command.

If the '+' is not attached to 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 **mktri** to execute multiple command lines in one shell environment. This is accomplished with the token combination ';'.

Example:

```
cd c:\ctri\bin ;\
ctri -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.

**mktri** can generate inline temporary files. If a line contains '<<WORD' then all subsequent lines up to a line starting with `WORD`, are placed in a temporary file. Next, '<<WORD' is replaced with the name of the temporary file.



No whitespace is allowed between '<<' and 'WORD'.

Example:

```
lktri -o $@ -f <<EOF
$(separate "\n" $(match .obj $!))
$(separate "\n" $(match .a $!))
$(LKFLAGS)
EOF
```

The three lines between the tags (EOF) are written to a temporary file (e.g. `"\tmp\mk2"`), and the command line is rewritten as `"lktri -o $@ -f \tmp\mk2"`.

## ***Implicit Rules***

Implicit rules are intimately tied to the `.SUFFIXES:` special target. Each entry in the `.SUFFIXES:` list defines an extension to a filename which may be used to build another file. The implicit rules then define how to actually build one file from another. These files are related, in that they must share a common basename, but have different extensions.

If a file that is being made does not have an explicit target line, an implicit rule is looked for. Each entry in the `.SUFFIXES:` list is combined with the extension of the target, to get the name of an implicit target. If this target exists, it gives the rules used to transform a file with the dependent extension to the target file. Any dependents of the implicit target are ignored.

If a file that is being made has an explicit target, but no rules, a similar search is made for implicit rules. Each entry in the `.SUFFIXES:` list is combined with the extension of the target, to get the name of an implicit target. If such a target exists, then the list of dependents is searched for a file with the correct extension, and the implicit rules are invoked to create the target.

## **Examples**

This makefile says that `prog.out` depends on two files `prog.obj` and `sub.obj`, and that they in turn depend on their corresponding source files (`prog.c` and `sub.c`) along with the common file `inc.h`.

```
LIB = -ls

prog.out: prog.obj sub.obj
    lktri prog.obj sub.obj $(LIB) -o prog.out

prog.obj: prog.c inc.h
    ctri prog.c
    astri prog.src

sub.obj: sub.c inc.h
    ctri sub.c
    astri sub.src
```

The following makefile uses implicit rules (from `mktri.mk`) to perform the same job.

```
LDFLAGS = -ls
prog.out: prog.obj sub.obj
prog.obj: prog.c inc.h
sub.obj:  sub.c inc.h
```

## Files

makefile Description of dependencies and rules.

Makefile Alternative to makefile, for UNIX.

mktri.mk Default dependencies and rules.

## Diagnostics

**mktri** returns an exit status of 1 when it halts as a result of an error. Otherwise it returns an exit status of 0.

## 11.5 PRTRI

### Name

**prtri** IEEE object reader  
Displays the contents of a relocatable object file or an absolute file

### Synopsis

**prtri** [*option*]... *file*  
**prtri** -V  
**prtri** -? ( UNIX C-shell: "-?" or -\? )

### Description

**prtri** gives you a high level view of an object file which has been created by a tool from the TASKING TriCore toolchain. Note that **prtri** is not a disassembler.

### Options

Options start with a '-' sign and can be combined after a single '-'. There are options to print a specific part of an object file. For example, with option **-h** you can display the header part, the environment part and the AD/extension part as a whole. These parts are small, and you cannot display these parts separately. If you do not specify a part, the default is **-hscegd0i0** (all parts, the debug part and the image part displayed as a table of contents).

Furthermore, there are some additional options by which you can control the output.

### Input Control Option

**-f file** Read command line information from *file*. If *file* is a '-', the information is read from standard input.

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

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

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

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

Example:

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

or

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

or

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

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



Example:

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

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

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

### Output Control Options

- H** or **-?** Display an explanation of options at `stdout`.
- V** Display version information at `stderr`.
- Wn** Set output width to *n* columns. Default 128, minimum 78.
- ln** Level control, see paragraph 11.5.3.
- ofile** Name of the output file, default `stdout`.
- v** Print the selected parts in a verbose form.
- vn** Print level *n* verbose, see paragraph 11.5.3.
- wn** Suppress messages above warning level *n*.

### Display Options

- c** Print call graphs.
- d** Print all debug info except for the global types.
- d0** Print table of contents for the debug part.
- dn** Print debug info from file number *n*.
- e** Print variables with external scope.
- e1** Print variables with external scope and precede symbol name with name of the object file.
- g** Print global types.
- h** Print general file info.

- i** Print all section images.
- i0** Print table of contents for the image part.
- in** Print image of section *n*.
- s** Print section info.

### **11.5.1 PREPARING THE DEMO FILES**

There are three files which are used in this chapter to show how you can use **prtri**. These files are:

```
calc.obj  
calc.out  
calc.abs
```

If you want to try the examples yourself, prepare these files by copying the calc example files to a working directory. Be sure that the TriCore tools can be found via a search path. Make the files with the following command:

```
cctri -M -Ms -Wa-gl -tiof -nolib startup.asm _copytbl.asm  
calc.asm -o calc.abs tri.dsc -tmp
```

### **11.5.2 DISPLAYING PARTS OF AN OBJECT FILE**

#### **11.5.2.1 OPTION -H, DISPLAY GENERAL FILE INFO**

The **-h** option gives you general information of the file. The invocation:

```
prtri -h calc.out
```

Gives the following information:

```
File name      = calc.out:  
Format         = Relocatable  
Produced by    = TriCore object linker  
Date           = mar 23, 1998 16:35:40h
```

This output speaks for itself. You may combine the **-h** switch with the verbose option:

```
prtri -hv calc.out
```

The output is extended with more general information of less importance:

```
File name      = calc.out:
Format         = Relocatable
Produced by    = TriCore object linker
Date          = mar 23, 1998 16:35:40h
Obj version    = 1.1
Processor      = TriCore
Address size   = 24 bits
Byte order     = Least significant byte at lowest address
Host           = Sun
```

Part	File offset	Length
Header part	0x00000000	0x00000055
AD Extension part	0x00000055	0x00000033
Environment part	0x00000088	0x0000002b
Section part	0x000000b3	0x0000009b
External part	0x0000014e	0x00000098
Debug/type part	0x000001e6	0x000002b8
Data part	0x0000049e	0x000002b8
Module end	0x00000756	

The table gives you the file offsets and the length of the main object parts.

**11.5.2.2 OPTION -S, DISPLAY SECTION INFO**

With the **-s** option, you can obtain the section information from an object module. The section **contents** can be obtained with the **-i** option, see 11.5.2.7.

```
prtri -s calc.out
```

Section	Size
-----	-----
.startup_vector	0x000002
.startup	0x000063
.watchdog_vector	0x000002
.watchdog	0x000001
.text	0x00002d
.data	0x000003
.zdata	0x000001

Note that the section information is not available any more in a located file. Once located, the separate *sections* are combined to new *clusters*. For an absolute file **prtri -s** will give the *cluster* information:

**prtri -s calc.abs**

Section	Size
-----	-----
rom	0x0000b9
ram	0x00f800

The locate map shows you which section is located in which cluster. Of course, you can also use the verbose option to see all section information available:

**prtri -sv calc.out**

Section	Size	Address	Align	PgSize	Mau	Attributes
-----	-----	-----	-----	-----	-----	-----
.startup_vector	0x000002	0x000000	0x001	-	-	ReadOnly Execute ZeroPage Space 1 Abs Separate
.startup	0x000063	-	0x001	-	-	ReadOnly Execute ZeroPage Space 1 Cumulate
.watchdog_vector	0x000002	0x000004	0x001	-	-	ReadOnly Execute ZeroPage Space 1 Abs Separate
.watchdog	0x000001	-	0x001	-	-	ReadOnly Execute ZeroPage Space 1 Cumulate
.text	0x00002d	-	0x001	-	-	ReadOnly Execute ZeroPage Space 1 Cumulate
.data	0x000003	-	0x001	-	-	Write Space 2 Initialized Cumulate
.zdata	0x000001	-	0x001	-	-	Write Space 2 Cleared Cumulate

The first two columns give you the section name and the section size. The column 'Address' gives you the section address, or a '-' if the section is still relocatable. The section alignment is always 1 for the TriCore. The page size is valid only for the short sections. MAU is the minimum addressable unit of an address space (in bits). There are two main groups of section attributes, the allocation attributes, used by the locator and the overlap attributes, used by the linker:

Allocation attributes	
Write	Must be located in ram
ReadOnly	May be located in rom
Execute	May be located in rom
Space <i>num</i>	Must be located in addressing mode <i>num</i>
Abs	Already located by the assembler
Cleared	Section must be initialized to '0'
Initialized	Section must be copied from ram to rom
Scratch	Section is not filled or cleared

Table 11-1: Allocation attributes

Overlap attributes	
MaxSize	Use largest length encountered
Unique	Only one section with this name allowed
Cumulate	Concatenate sections with the same name to one bigger section
Overlay	Sections with the name <i>name@func</i> must be combined to one section <i>name</i> , according to the rules for <i>func</i> obtained from the call graph.
Separate	Sections are not linked.

Table 11-2: Overlap attributes

### 11.5.2.3 OPTION -C, DISPLAY CALL GRAPHS

The call graph is used by the linker overlaying algorithm. Once a file is linked and overlaying is done, the call graph information is removed from the object file. If you try to see the call graph in `calc.out` you will get the message 'No call graph found'.

The file `calc.obj` is not yet linked. You can use this file to see what a call graph looks like:

```
prtri -c calc.obj
```

Because the `calc` example does not contain any sections which need to be overlaid you will again get the message 'No call graph found'. The following is just an example of what a call graph could look like:

```
Call graph(s)
=====

Call graph 0:

main()
  ->See call graph 1
  ->See call graph 4
  ->See call graph 2
  _exit()
  print_str()
  clear_screen()

Call graph 1:

queens?find_legal_row()
  ->See call graph 1
  ->See call graph 2
  abs()
  ->See call graph 3
```

Each call graph consists of a function (`main` in graph 0), followed by a list of functions and/or other graphs, which are called by the first function. The functions and call graphs called by this function are indented by two spaces. If a function calls other functions, those functions are listed again with another indentation of two spaces.

As you can see, there are references from one call graph to another. Call graph 1 even calls itself!! This means that function `find_legal_row()` is a recursive function. If you use the verbose switch the output is somewhat nicer:

```

main()
|
+--->See call graph 1
|
+--->See call graph 4
|
+--->See call graph 2
|
+--exit()
|
+--print_str()
|
+--clear_screen()

```

The function `find_legal_row` from call graph 1 is a static function. In order to avoid name conflicts, the source name is added to this function name.

If you want a call graph with resolved call graph references, you can use the linker to generate one:

```
lktri -o call.out -Mcr calc.obj
```

Option **-M** tells the linker to generate a `.lnl` file. This file contains the call graph in the verbose layout. Option **-c** causes the linker to generate a `.cal` file. This file contains also the (same) call graph, but in the compact (non verbose) layout. Option **-r** tells the linker that this is an incremental link.

#### **11.5.2.4 OPTION -E, DISPLAY EXTERNAL PART**

In the external part of an object file, you can find all symbols used at link time. These symbols have an external scope. With the **-e** option (or **-e0**) **prtri** displays the external symbols:

```
prtri -e calc.out
```

Variable	S	Address/Size
-----	-----	-----
_start_cpt	I	.startup + 0x00
_START	I	.startup + 0x00
_exit	I	.startup + 0x20
_copytable	I	.startup + 0x22
_main	I	.text + 0x20
_lc_es	X	-
_lc_cp	X	-

With option **-e1** also the name of the output object file is displayed.

```
prtri -e1 calc.out
```

Variable	S	Address/Size
-----	-----	-----
calc.out:_start_cpt	I	.startup + 0x00
calc.out:_START	I	.startup + 0x00
calc.out:_exit	I	.startup + 0x20
calc.out:_copytable	I	.startup + 0x22
calc.out:_main	I	.text + 0x20
calc.out:_lc_es	X	-
calc.out:_lc_cp	X	-

The first column contains the name of the symbol. In general, this symbol is a high level symbol with an 'F' added at the front. The next column gives you the symbol status. This can be **I** for a defined symbol, and **X** for a symbol which is referred to, but which is not yet defined. In the last column you can find the symbols address. If this address is still relocatable, the section offsets are printed in the form '*section* + offset'. If a symbol has already received an absolute address, this address is printed. Symbols that are not yet defined (marked with a **X**) have a dash printed as address, indicating *unknown*.

You can add the verbose option as usual. With verbose on more information is printed:

```
prtri -ev calc.out
```



Variable	S	Type	Attrib	MAU	Amod	Address/Size
__start_cpt	I	-	-	8	1	.startup + 0x00
__START	I	-	-	8	1	.startup + 0x00
__exit	I	-	-	8	1	.startup + 0x20
__copytable	I	-	-	8	1	.startup + 0x22
__main	I	-	-	8	1	.text + 0x20
__lc_es	X	-	-	8	2	-
__lc_cp	X	-	-	8	2	-

Four additional columns appear. The Type column gives you the symbol type, if available. You can find the meaning of the types in the global type part, section 11.5.2.5. The global types are used to type check the symbols during linking. The Attribute column specifies the attribute of the symbol, if available. For example, the attribute value 0x0020 indicates that the symbol is generated by the assembler. The MAU column indicates the minimum addressable unit in bits. So, MAU 8 means the symbol is 8-bit addressable. The Amod column lists the addressing mode of the symbol.

**11.5.2.5 OPTION -G, DISPLAY GLOBAL TYPE INFORMATION**

The linker uses the global type information to check on type mismatches of the symbols in the external part. This information is always available, unless you explicitly suppress the generation of these types with option **-gn** at compile time. Of course, type checking can only be done if the types are available. The global types in `calc.out`:

```
prtri -g calc.out
```

In this example you will get the message 'No global types available'. The following is just an example of what the global type information could look like:

```

Tp#  Mnem  Name  Entry
-----
101  X      -      0, T10, 0, 0
102  X      -      0, T1, 0, 0
103  X      -      0, T1, 0, 1, T104
104  P      -      T105
105  n      -      T2, 1
106  X      -      0, T1, 0, 1, T10
107  X      -      0, T10, 0, 1, T10
108  X      -      0, T1, 0, 2, T109, T109
109  T      Byte    T3
10a  X      -      0, T1, 0, 1, T109
...
10f  X      -      0, T1, 0, 3, T12, T110, T12
110  O      -      T111
111  n      -      T2, 0
112  Z      -      T2, 13
113  Z      -      T2, 7

```

In the first column you find the type index. This is the number by which the type is referred to. This number is always a hexadecimal number. Numbering starts at 0x101, because the indices less than 0x100 are reserved for, so-called, 'basic types'. The second column contains the type mnemonic. This mnemonic defines the new 'high level' type. In the Name column you will find the name for the type, if any.

The last column contains type parameters. They tell you which (basic) types a high level type is based on and give other parameters such as modes and sizes. Types are preceded by a **T**. So, in the example above, type 105 is based upon type 2 (T2 in the parameter list) and type 103 is based upon type 1 and type 104.

In the next table you can find an overview of the basic types:

Type index	Type	Meaning
1	void	–
2	char	8 bits signed
3	unsigned char	8 bits unsigned
4	short	16 bits signed
5	unsigned short	16 bits unsigned
6	long	32 bits signed
7	unsigned long	32 bits unsigned

Type index	Type	Meaning
10	float	32 bit floating point
11	double	64 bit floating point
16	int	16 bits signed
17	unsigned int	16 bits unsigned

Table 11–3: Basic types

The type mnemonics define the class of the newly created type. The next table shows the type mnemonics with a short description:

Mnemonic	Description	Parameters
G	generalized structure	size, [member, <i>Tindex</i> , offset, size ]...
N	enumerated type	[name, value ]...
n	pointer qualifier	<i>Tindex</i> , memspace
O	small pointer	<i>Tindex</i>
P	large pointer	<i>Tindex</i>
Q	type qualifier	q–bits, <i>Tindex</i>
S	structure	size, [member, <i>Tindex</i> , offset ]...
T	typedef	<i>Tindex</i>
t	compiler generated type	<i>Tindex</i>
U	union	size, [member, <i>Tindex</i> , offset ]...
X	function	x–bits, <i>Tindex</i> , 0, nbr–arg, [ <i>Tindex</i> ]...
Z	array	<i>Tindex</i> , upper–bound
g	bit type	sign, nbr–of–bits

Table 11–4: Type mnemonics

The *Tindex* for mnemonic n, O, P, Q, T, t and Z are the types upon which the new type is built. The *Tindex* for the union and the structures are the type indices for the members. For the function type, the first *Tindex* is the return type of the function. The second *Tindex* is repeated for each parameter, and gives the type of each parameter. The value -1 (0xffffffff) always means 'unknown'. This can occur with a function type if the number of parameters is unknown, or with an array if the upper bound is unknown. The sizes and offset for the generalized structure are in bits. The first size is the size of the structure, the second size is the size for the member.

The type information obtained with the **-g** switch has no verbose equivalent.

### **11.5.2.6 OPTION -D, DISPLAY DEBUG INFORMATION**

The **-d** switch has two variants. With **-d0** you get a table of contents:

```
prtri -d0 calc.out
```

Choose option -d with the number of the file:

- 1 - startup
- 2 - \_copytbl
- 3 - calc

Now, you can use **-dn** to examine a single (linked) file. For instance, **-d3** shows you only the debug info of `calc.obj`. It is also possible to see all debug info, by using option **-d** without a value.

The **-d** switch without the verbose option **-v** shows you only local variables and procedure information. If you combine the **-d** switch with the verbose switch **-v**, also local type info, line numbers, stack update information and more procedure information is displayed.

In the example you are using the verbose switch. Where required, the remark 'Only with verbose on' will be given.

```
prtri -d3v calc.out
```

The object reader starts with a header, followed by the local type information:

```
*****
*   O b j e c t   c a l c   *
*****
```

```
Module   info
=====
```

```
Type info calc:
=====
```

No local types available

This type info is only printed if you use the verbose option **-v**. The information found in this table is exactly the same as the information explained for the global type information, see 11.5.2.5.

After the local types, you will find the local symbols.

```
Symbols calc:
=====
```

Variable	S	Type	Attrib	MAU	Amod	Address/Size
-----						
_factorial	N	-	0x0020	8	1	-
_compute	N	-	0x0020	8	1	-
_val	N	-	0x0020	8	2	-
_zero	N	-	0x0020	8	2	-
_c11	N	-	0x0020	8	2	-

The value for the symbol status in the external part was an **I** or an **X**. Here, you can see a new letter. The **N** stands for a local symbol. Other possible entires can have the letter **G** or **S**. They are no symbols, but procedures. These procedures are printed at this place in order to define their relative position. The actual procedure information is given in the next block of information. Here you can find the additional procedure information. The procedure block is printed only if you use the verbose switch:

```
Procedures calc:
=====
```

No procedures

The following is an example of some procedures:

Name	S	Additional information
main	G	0x00, 0x00, T101, QUEENS_PR + 0x00, ( QUEENS_PR + 0x49 ) - 0x01
find_legal_row	S	0x00, 0x00, T120, QUEENS_PR + 0x49, ( QUEENS_PR + 0x156 ) - 0x01
display_board	S	0x00, 0x00, T10a, QUEENS_PR + 0x156, ( QUEENS_PR + 0x2a4 ) - 0x01
display_field	S	0x00, 0x00, T121, QUEENS_PR + 0x2a4, ( QUEENS_PR + 0x302 ) - 0x01
display_status	S	0x00, 0x00, T103, QUEENS_PR + 0x302, ( QUEENS_PR + 0x31d ) - 0x01

The first two columns are the same as those in the local variable table. The **G** stands for an external (global) function, the **S** for a static (local) function.

Each function has 5 parameters with the following meaning:

- param #1    Frame type, not used
- param #2    Frame size, the distance from the stack pointer before the function call to the stack position just after the local variables.
- param #3    The type of the function
- param #4    The start address of the function. In a relocatable object the syntax '*section* + offset' is used.
- param #5    The last function address. See also param #4.

Next in the debug info is the line number information and the stack information. Both items are only printed if you had turned the verbose switch on:

```
Lines include/stdarg.h:
=====
No line info available

Lines include/stdio.h:
=====
No line info available
```

```
Lines queens.c:
=====

Address          | Line   Address          | Line   Address          ...
-----
QUEENS_PR + 0x000000 | 52     QUEENS_PR + 0x0000c2 | 90     QUEENS_PR + ...
QUEENS_PR + 0x000000 | 53     QUEENS_PR + 0x0000d9 | 101    QUEENS_PR + ...
QUEENS_PR + 0x000006 | 55     QUEENS_PR + 0x0000d9 | 103    QUEENS_PR + ...
.
.
.
QUEENS_PR + 0x0000bd | 98     QUEENS_PR + 0x00018e | 133    QUEENS_PR + ...
QUEENS_PR + 0x0000c0 | 99     QUEENS_PR + 0x000190 | 136    QUEENS_PR + ...
QUEENS_PR + 0x0000c2 | 100    QUEENS_PR + 0x00019f | 137

Stack info include/stdarg.h:
=====
No stack info available

Stack info include/stdio.h:
=====
No stack info available

Stack info queens.c:
=====
No stack info available
```

The stack info gives the actual stack position for each executable address. This value is measured from the start position, just after the functions local variables to the actual stack position. If you push one byte on stack, the delta will be increased by one.

The debug info per module ends with a block for each function. Within this block the local variables per function are displayed:

```
Procedure info
=====

Procedure find_legal_row:
=====

Symbols find_legal_row:
=====

Variable S  Type      Attrib  Mau  Amod  Address/Size
-----
accepted  N   0x0109  0x0004  0     0     QUEENS_DA +
0x09
row       N   0x0109  0x0805  0     0     0x02
col       N   0x0109  0x0805  0     0     0x03
chk_row   N   0x0109  0x0005  0     0     0x01
chk_col   N   0x0109  0x0005  0     0     0x00
```

```

E n d   o f   p r o c e d u r e   i n f o
=====

```

### 11.5.2.7 OPTION -I, DISPLAY THE SECTION IMAGES

As with the **-d** option, you can ask a table with available section images by specifying option **-i0**:

```
prtri -i0 calc.out
```

Choose option **-i** with the number of the section:

```

1 - .startup_vector
2 - .startup
3 - .watchdog_vector
4 - .watchdog
5 - .text
6 - .data
7 - .zdata

```

You can select the image to display by specifying the image number:

```
prtri -i5 calc.out
```

```

Section .text:
=====

```

```

02 32 05 e3 ce 00 01 c4 f8 b0 cf 88 f3 f0 50 b4
cf d8 ce a1 51 d8 ce e1 cf 00 b1 cc a9 01 cf f8
rr rr rr rr rr rr rr rr rr rr rr rr rr rr rr

```

It is also possible to get the section offsets or absolute addresses by specifying the verbose flag:

```
prtri -i5v calc.out
```

```

Section .text:
=====

```

```

000000 02 32 05 e3 ce 00 01 c4 f8 b0 cf 88 f3 f0 50 b4      .2.....P.
000010 cf d8 ce a1 51 d8 ce e1 cf 00 b1 cc a9 01 cf f8      ....Q.....
000020 rr rr rr rr rr rr rr rr rr rr rr rr rr rr rr      .....

```



The dump always shows the hexadecimal byte value per address. Sometimes however, this is not possible. First of all, it is possible that a certain byte cannot be determined because it is not yet relocated. In this case the byte is represented as **rr**.

Secondly, it is possible that there is no section image allowed. This is for instance the case for sections that are cleared during startup. After the invocation (verbose on) the reader prints:

```
prtri -i7v calc.out

Section .zdata:
=====

No image allowed, cleared during startup
```

It is possible that you read an absolute file. In the absolute file it is possible to combine different sections to new clusters. These clusters do not have the same attributes as the sections and the reader does no longer know where the overlay area is positioned:

```
prtri -v -il calc.abs

Section rom:
=====

000000 00 53 f9 ss 00 02 02 00 f0 00 00 00 00 00 00 00 .S.....
000010 01 01 00 f0 01 00 00 b6 00 00 00 03 00 ss ss ss .....2.....
000020 ss ss ss ss ss ss 02 32 05 e3 ce 00 01 c4 f8 b0 ....P....Q....
000030 cf 88 f3 f0 50 b4 cf d8 ce a1 51 d8 ce e1 cf 00 .....
....
```

As you see, the reader only prints bytes that it actually can read from the object file. The **ss** in the dump means *scratch* memory. It may or may not be initialized by the start-up code. This information is not available anymore to the reader. The start-up code can use a locator generated table to get the information. See the *Locator* chapter.

### **11.5.3 VIEWING AN OBJECT AT LOWER LEVEL**

#### **11.5.3.1 OBJECT LAYERS**

As with the well known OSI layer model for communication, you can also distinguish layers in an object file. The object file is a medium for the compiler which lets the compiler communicate with the debugger or the target board. The lowest level can be classified as mass storage, mostly the disc. The lowest viewable level for the readers concern are the raw bytes.

**prtri** knows this layer as *level 0*.

Of course, the bytes in level 0 have a meaning. Because the object format is a format according to IEEE 695, the object file is a collection of MUFOM commands. The general idea is, that an object producing tool sends commands to a object consuming tool. These commands are described in detail by the official IEEE standard<sup>1</sup>. The raw bytes from level 0 appear to be encoded MUFOM commands. The MUFOM commands are interpreted in a layer just above the raw bytes layer.

**prtri** knows this layer as *level 1*.

The next layer is the MUFOM environment, the type and section tables are built, values are assigned, attributes are set just by performing the MUFOM commands. The IEEE document describes also some predefined meanings about scope, section attributes naming conventions for MUFOM variables. This knowledge is available in the highest MUFOM layer.

**prtri** knows this layer as *level 2*.

With these first layers, the compiler and debugger/target board have a perfect communication channel. The next layers (not supported by the reader at this moment) define a protocol between compiler and debugger about target and language specific information.

In the next sections you can find some examples about the use of the reader at lower levels. Until now, you used the default level of the reader, level 2.

<sup>1</sup> IEEE Trial Use Standard for Microprocessor Universal Format for Object Modules (IEEE std. 695), IEEE Technical Committee on Microcomputers and Microprocessors of the IEEE Computer Society, 1990.

### 11.5.3.2 THE LEVEL OPTION -LN

#### Level 1

Switching to another level is simple. You can use the **-l** option with the level you want to see. As an example, the section part of `calc.out` at level 1:

```
prtri -ll -s calc.out

ST:  1, RXAZS, .startup_vector
AS:  L1, 0x0
AS:  S1, 0x2
ST:  2, RXZC, .startup
AS:  S2, 0x63
ST:  3, RXAZS, .watchdog_vector
AS:  L3, 0x4
AS:  S3, 0x2
ST:  4, RXZC, .watchdog
AS:  S4, 0x1
ST:  5, RXZC, .text
AS:  S5, 0x2d
ST:  6, WIY2C, .data
AS:  S6, 0x3
ST:  7, WBY2C, .zdata
AS:  S7, 0x1
```

If you are not familiar with the MUFOM commands, you can use the verbose switch. The abbreviated commands such as AS, SA or ST are expanded to *Assignment*, *Section alignment* and *Section type*:

```
prtri -v -ll -s calc.out

ST:  Section type:
      Nbr = 1, type = RXAZS, name = .startup_vector
AS:  Assignment:
      Variable = L1, expression = 0x0
AS:  Assignment:
      Variable = S1, expression = 0x2
.
.
ST:  Section type:
      Nbr = 7, type = WBY2C, name = .zdata
AS:  Assignment:
      Variable = S7, expression = 0x1
```

The  $L_n$  and  $S_n$  MUFOM variables are defined as the address and the size of section  $n$ . At level 2 you saw (refer to section 11.5.2.2) that the level 2 view did not mention the L and S variables, because at level 2 the meaning of the L and S variables are known!

### ***Level 0***

Switching to level 0 is accomplished by using `-l0` (as you expected):

```
prtri -l0s calc.out
```

```
e6 01 d2 d8 c1 da d3 0f 2e 73 74 61 72 74 75 70 5f
76 65 63 74 6f 72
e2 cc 01 81 00
e2 d3 01 02
...
e6 07 d7 c2 d9 02 c3 06 2e 7a 64 61 74 61
e2 d3 07 01
```

The bytes are printed in the MUFOM command structure. It should be easy to find the encoding for the used MUFOM commands. You can use the verbose switch if you want to see file offsets:

```
prtri -l0vs calc.out
```

```
0000b3 e6 01 d2 d8 c1 da d3 0f 2e 73 74 61 72 74 75 70 5f .....startup_
      76 65 63 74 6f 72                      vector
0000ca e2 cc 01 81 00                      ....
0000cf e2 d3 01 02                      ....
....
00013c e6 07 d7 c2 d9 02 c3 06 2e 7a 64 61 74 61 .....zdata
00014a e2 d3 07 01                      ....
```

### ***Viewing Mixed Levels***

You can also mix the levels. It is for instance possible to see level 0 and 1 together by specifying option `-l01` (equivalent to `-l10` or `-l0 -l1`):

```
prtri -sl01 calc.out
```



```
ST:  1, RXAZS, .startup_vector
      e6 01 d2 d8 c1 da d3 0f 2e 73 74 61 72 74 75 70 5f

      76 65 63 74 6f 72
AS:  L1, 0x0
      e2 cc 01 81 00
AS:  S1, 0x2
      e2 d3 01 02
.
.
.
ST:  7, WBY2C, .zdata
      e6 07 d7 c2 d9 02 c3 06 2e 7a 64 61 74 61
AS:  S7, 0x1
      e2 d3 07 01
```

And of course, you can turn on the verbose switch. The switch between level 0 and level 1 is done per MUFOM command. This is because a MUFOM command is the smallest unit at level 1.

If you should display level 1 and 2, the switch is made per object part, because the object parts are the smallest units at level 2. It is not possible to show the results of all section related commands before all these commands are executed:

```
prtri -s -l1 -l2 calc.out

ST:  1, RXAZS, .startup_vector
AS:  L1, 0x0
AS:  S1, 0x2
.
.
.
ST:  7, WBY2C, .zdata
AS:  S7, 0x1
```

Section	Size
-----	-----
.startup_vector	0x000002
.startup	0x000063
.watchdog_vector	0x000002
.watchdog	0x000001
.text	0x00002d
.data	0x000003
.zdata	0x000001

### 11.5.3.3 THE VERBOSE OPTION -VN

As you have read in section 11.5.3.2, you can switch to a lower level with the level switch **-ln**. If you want a verbose printout, you can use the **-v** option.

It is also possible to specify **-v0** to see a verbose output of level 0, option **-vn** is a shorthand for options **-v -ln** (or **-vln**). The new notation has the advantage that if you want a mixed level output, you are able to choose the verbose option **per level**. You may specify **-l0 -v1**, and you get a non verbose level 0 and a verbose level 1:

```
prtri -sl0v1 calc.out
```

```
ST:  Section type:
      Nbr = 1, type = RXAZS, name = .startup_vector
      e6 01 d2 d8 c1 da d3 0f 2e 73 74 61 72 74 75 70 5f
      76 65 63 74 6f 72
AS:  Assignment:
      Variable = L1, expression = 0x0
      e2 cc 01 81 00
AS:  Assignment:
      Variable = S1, expression = 0x2
      e2 d3 01 02
.
.
.
ST:  Section type:
      Nbr = 7, type = WBY2C, name = .zdata
      e6 07 d7 c2 d9 02 c3 06 2e 7a 64 61 74 61
AS:  Assignment:
      Variable = S7, expression = 0x1
      e2 d3 07 01
```

The general verbose switch **-v** (without a number) makes all selected levels verbose. The verbose switch **-vn** selects level *n* and makes only level *n* verbose.



# APPENDIX

## ASSEMBLER ERROR MESSAGES

---

# A





---

**A**

**APPENDIX**

---

## **1 INTRODUCTION**

The assembler produces error messages on standard error output. If the list option of the assembler is effective, error messages will be included in the list file as well, when the assembler has started list file generation. Error messages have the following layout:

[E|F|W] *error\_number: filename line number : error\_message*

Example:

astri E214: /tmp/tst.src line 17 : illegal addressing mode

The example reports the error, starting with the severity (E: error, F: fatal error, W: warning) and the error number followed by the source filename and the line number. The last part of the line shows the error message text.

All warnings (W), errors (E), and fatal errors (F) of **astri** are described below.

## 2 WARNINGS (W)

The assembler may generate the following warnings:

W 101: use *option* at the start of the source; ignored

Primary options must be used at the start of the source.

W 102: duplicate attribute "*attribute*" found

An attribute of an .EXTERN directive is used twice or more. Remove one of the duplicate attributes.

W 103: section offset can cause overlap of code or data

W 104: expected an attribute but got *attribute*; ignored

W 105: section activation expected, use *name* directive

Use the .SECT directive to activate a section.

W 106: conflicting attributes specified "*attributes*"

You used two conflicting attributes in an .EXTERN statement directive. For example CODE and DATA or CLEAR and NOCLEAR. Choose which one you want to use and remove the other.

W 107: memory conflict on object "*name*"

A label or other object is explicit or implicit defined using incompatible memory types. Check all usages and definitions of the object *name* to remove this conflict.

W 108: object attributes redefinition "*attributes*"

A label or other object is explicit or implicit defined using incompatible attributes. For example CODE and DATA. Check all usages and definitions of the object to remove the conflict.

W 109: label "*label*" not used

The label *label* is defined with the .GLOBAL directive and neither defined nor referred, or the label is defined with the .LOCAL directive and not referenced. You can remove this label and its definitions (in the case of a .LOCAL label).

W 110: extern label "*label*" defined in module, made global

The label *label* is defined with an .EXTERN directive and defined as a label in the source. The label will be handled as a global label. Change the .EXTERN definition into .GLOBAL or one of the identifiers.

W 111: unknown *name* control flag "*flag*"

You supplied an unknown *flag* to the \$LIST or \$DEBUG control. See the description of the \$LIST or \$DEBUG control for the possible arguments.

W 112: text found after END; ignored

An .END directive designates the end of the source file. All text after the .END directive will be ignored. Remove the text.

W 115: use ON or OFF after control name

The control you specified must have either ON or OFF after the control name. See the description of the control for details.

W 116: unknown parameter "*parameter*" for *control-name* control

See the description of the control for the allowed parameters.

W 118: inserted "extern *name*"

The symbol *name* is used inside an expression, but not defined with an .EXTERN directive. The assembler inserts an .EXTERN definition of the offending symbol. Add an .EXTERN definition.

W 119: "*name*" section has not the MAX attribute; ignoring RESET

W 120: assembler debug information: cannot emit non-tiof expression for *label*

The .SYMB record contains an expression with operations that are not supported by the IEEE-695 object format. When the .SYMB record is generated by the TASKING C compiler, please fill out the error report and send it to TASKING.

W 121: changed alignment size to *size*

W 122: fractional has been saturated because it is outside the fractional domain

A fract must be in the range [-1,1>.

W 123: expression: *type-error*

The expression performs an illegal operation on an address or combines incompatible memory spaces. Check the expression, and change it.

W 124: cannot purge macro during its own definition

W 125: "*symbol*" is not a defined symbol

You tried to `.UNDEF` a symbol that was not previously `.DEFINED` or was already undefined. Check all `.DEFINE/.UNDEF` combinations of the offending symbol.

W 126: redefinition of "*define-symbol*"

The symbol is already `.DEFINED` in the current scope. The symbol is redefined according to this `.DEFINE`. `.UNDEF` any symbol before redefining it.

W 127: redefinition of macro "*macro*"

The macro is already defined. The macro is redefined according to this macro definition. Purge any macro using `.PMACRO` before redefining it.

W 128: number of macro arguments is less than definition

You supplied less arguments to the macro than when defining it. Check your macro definition with this macro call. The undefined macro arguments are left empty (as in `.DEFINE def ' ')`.

W 129: number of macro arguments is greater than definition

You supplied more arguments to the macro than when defining it. Check your macro definition with this macro call. The superfluous macro arguments are ignored.

W 130: `DUPA` needs at least one value argument

The `.DUPA` directive needs at least two arguments, the dummy parameter and a value parameter. Add one or more value-parameters.

W 131: `DUPF` increment value gives empty macro

The step value supplied with the `.DUPF` macro will skip the `.DUPF` macro body. Check the step value.

W 132: `.IF` started in previous file "*file*", line *line*

The `.ENDIF` or `.ELSE` pre-processor directive matches with an `.IF` directive in another file. Check on any missing `.ENDIF` or `.ELSE` directives in that file.

W 133: currently no macro expansion active

The `@CNT()` and `@ARG()` functions can only be used inside a macro expansion. Check your macro definitions or expression.

W 134: *"directive"* is not supported, skipped

The supplied directive is not supported by the TASKING assembler.  
Remove all uses of this directive.

W 135: define symbol of *"define-symbol"* is not an identifier; skipped definition

You supplied an illegal identifier with the **-D** option on the command line. An identifier should start with a letter, followed by any number of letters, digits or underscores.

W 137: label *"label"* defined *attribute* and *attribute*

The label is defined with an .EXTERN and a .GLOBAL directive. The .EXTERN directive is removed, leaving the label global.

W 138: warning: *.WARNING-directive-arguments*

Output from the .WARNING directive.

W 139: expression must be between *hex-value* and *hex-value*

W 140: expression must be between *value* and *value*

W 141: *global/local* label *"name"* not defined in this module; made extern

The label is declared and used but not defined in the source file. Check the current scope of the label and its usage, change the declaration to EXTRN or add a label definition.

W 142: redefinition of *'name'* macro

The macro was defined before. The new macro definition is used.

W 170: duplicate controle used: *control\_name*

A primary assembly control was defined twice.

W 171: suspicious instruction concerning CPU functional defect  
TC112\_COR1

The preceding instruction of a LOOP, LOOP16 or LOOPU instruction is not an ISYNC instruction.

See section *TC112\_COR1* in Appendix *CPU Functional Problems* of the *Cross-Compiler Users Guide* for more details.

- W 172: suspicious instruction concerning CPU functional defect  
TC112\_COR4

An instruction uses an An register for either an effective address calculation or as the target of an indirect branch that is located directly after a (target) label.

See section *TC112\_COR4* in Appendix *CPU Functional Problems* of the *Cross-Compiler Users Guide* for more details.

- W 173: suspicious instruction concerning CPU functional defect  
TC112\_COR10

A store operation is encountered that uses a circular addressing mode with an offset not equal to zero.

See section *TC112\_COR10* in Appendix *CPU Functional Problems* of the *Cross-Compiler Users Guide* for more details.

- W 174: suspicious instruction concerning CPU functional defect  
TC112\_COR13

A loop was encountered that contains a single integer instruction which is a DVSTEP or a DVSTEP.U.

See section *TC112\_COR13* in Appendix *CPU Functional Problems* of the *Cross-Compiler Users Guide* for more details.

- W 175: suspicious instruction concerning CPU functional defect  
TC112\_COR17

A DSYNC is not followed by a NOP instruction.

See section *TC112\_COR17* in Appendix *CPU Functional Problems* of the *Cross-Compiler Users Guide* for more details.

- W 176: suspicious instruction concerning CPU functional defect  
TC113\_CPU9

A DSYNC is not followed by two NOP instructions.

See section *TC113\_CPU9* in Appendix *CPU Functional Problems* of the *Cross-Compiler Users Guide* for more details.

W 177: suspicious instruction concerning CPU functional defect  
TC113\_CPU11

- An LDA, LDDA, or LD16A instruction is directly followed by a JI instruction.
- There is no or just one instruction (not a NOP instruction) between label and RET or RET16.

See section *TC113\_CPU11* in Appendix *CPU Functional Problems* of the *Cross-Compiler Users Guide* for more details.

W 178: suspicious instruction concerning CPU functional defect  
TC113\_CPU14

The first label in a code section is not followed by a DSYNC instruction.

See section *TC113\_CPU14* in Appendix *CPU Functional Problems* of the *Cross-Compiler Users Guide* for more details.

W 179: suspicious instruction concerning CPU functional defect  
TC113\_CPU15

An ST.T, SWAP or LDMST instruction was encountered.

See section *TC113\_CPU15* in Appendix *CPU Functional Problems* of the *Cross-Compiler Users Guide* for more details.

W 180: suspicious instruction concerning CPU functional defect  
TC113\_CPU16

An LDA, LDDA or LD16A instruction is directly followed by a JI or CALLI instruction with the same address register as parameter.

See section *TC113\_CPU16* in Appendix *CPU Functional Problems* of the *Cross-Compiler Users Guide* for more details.

W 181: suspicious instruction concerning CPU functional defect  
TC113\_DMU1

A SWAP, LDMST or ST.T instruction was encountered.

See section *TC113\_DMU1* in Appendix *CPU Functional Problems* of the *Cross-Compiler Users Guide* for more details.



W 182: suspicious instruction concerning CPU functional defect  
TC113\_LF12

A SWAP, LDMST or ST.T instruction was encountered.

See section *TC113\_LF12* in Appendix *CPU Functional Problems* of the *Cross-Compiler Users Guide* for more details.

W 183: suspicious instruction concerning CPU functional defect  
TC113\_LF13

A SWAP, LDMST or ST.T instruction was encountered.

See section *TC113\_LF13* in Appendix *CPU Functional Problems* of the *Cross-Compiler Users Guide* for more details.

W 184: error in \$PAGELENGTH control: *message*

The page length must be larger than 10 lines. Now the default of 66 lines is assumed.

W 185: error in \$PAGEWIDTH control: *message*

The page width must be larger than 40 columns. Now the default of 80 columns is assumed.

### **3 ERRORS (E)**

The assembler generates the following error messages when a user error situation occurs. These errors do not terminate assembly immediate. If one or more of these errors occur, assembly stops at the end of the active pass.

E 200:     *message*; halting assembly

The assembler stops the further processing of your source file. This is only an informative message. Remove all errors reported earlier and try again.

E 201:     unexpected newline or line delimiter

The syntax checker found a newline or line delimiter that does not confirm to the assembler grammar. Check the line for syntax errors or remove the offending newline or line delimiter.

E 202:     unexpected character: '*character*'

The syntax checker found a character that does not confirm to the assembler grammar. Check the line for syntax errors or remove the offending character.

E 203:     illegal escape character in string constant

The syntax checker found an illegal escape character in the string constant that does not confirm to the assembler grammar. Check the line for syntax errors or remove the offending escape character.

E 204:     I/O error: open intermediate file failed (*file* )

The assembler opens an intermediate file to optimize the lexical scanning phase. The assembler cannot open this file. The assembler checks if the environment symbol TMPDIR is set. If so, this directory is used for opening the file. Otherwise the file is opened in the current directory.

E 205:     syntax error: expected *token* instead of *token*

The syntax checker expected to find a token but found another token. The expected token is inserted instead of the found token. Check the line for syntax errors.

E 206:     syntax error: *token* unexpected

The syntax checker found an unexpected token. The offending token is removed from the input and assembling continues. Check the line for syntax errors.

E 207: syntax error: missing `;`

The syntax checker found a label definition or memory space modifier but missed the appended semi-colon. Check the line for syntax errors, for example misspelled mnemonics.

E 208: syntax error: missing `)`

The syntax checker expected to find a closing parentheses. Check the expression syntax for missing operators and nesting of parentheses.

E 209: invalid radix value, should be 2, 8, 10 or 16

The `.RADIX` directive accepts only 2, 8, 10 or 16.

E 210: syntax error

The syntax checker found an error. Check the line for syntax errors.

E 211: unknown model

Substitute the correct model.

E 212: syntax error: expected *token*

The syntax checker expected to find a token but found nothing. The expected token is inserted. Check the line for syntax errors.

E 213: label "*label*" defined *attribute* and *attribute*

The label is defined with a `.LOCAL` and a `.GLOBAL` or `.EXTERN` directive. Check your label scoping or change the label declarations.

E 214: illegal addressing mode

The mnemonic used an illegal addressing mode. Check the register usage of address constructs.

E 215: not enough operands

The mnemonic needs more operands. Check the source line and change the instruction.

E 216: too many operands

The mnemonic needs less operands. Check the source line and change the instruction.

E 217: *description*

There was an error found during assembly of the mnemonic. Check the instruction.

E 218: unknown mnemonic: "*name*"

The assembler found an unknown mnemonic. Check the instruction.

E 220: not within CODE section; instruction is removed

Instructions are not allowed in a DATA section.

E 223: unknown section "*name*"

The section name specified with a .SECT directive has not (yet) been defined with a .SDECL directive. Check the .SECT name and the corresponding .SDECL name.

E 224: unknown label "*name*"

A label was used which was not defined. Check that the label and its definition have the same name.

E 225: invalid memory type

You supplied an invalid memory modifier.

E 226: unknown symbol attribute: *sect\_type*

You specified an invalid section type.

E 227: invalid memory attribute

The assembler found an unknown location counter or memory mapping attribute.

E 228: *name* attribute needs a number

The attribute *attr* needs an extra parameter. For example, the FIT attribute.

E 229: only one of the *name* attributes may be specified

E 230: invalid section attribute: *name*

The assembler found an unknown section attribute.

E 231: absolute section, expected "AT" expression

An absolute section must be specified using an 'AT *address*' expression.

E 232: MAX/OVERLAY sections need to be named sections

Sections with the MAX or OVERLAY attribute must have a name, otherwise the locator cannot overlay the sections.

- E 233: *type* section cannot have *attribute* attribute  
Code sections may not have the CLEAR or OVERLAY attribute.
- E 234: section attributes do not match earlier declaration  
In an previous definition of the same section other attributes were used. Check all section definitions with the same name.
- E 235: redefinition of section  
An absolute section of the same name can only be located once.
- E 236: cannot evaluate expression of *descriptor*  
Some functions and directives must evaluate their arguments during assembly. Change the expression so that it can be evaluated.
- E 237: *descriptor* directive must have positive value  
Some directives need to have a positive argument. Check the expression so that it evaluates to a positive number.
- E 238: Floating point numbers not allowed with .BYTE directive  
The .BYTE directive does not accept floating point numbers. Convert the expressions or use the .HALF directive instead.
- E 239: byte constant out of range  
The .BYTE directive stores expressions in bytes. A byte can only hold numbers between 0 and 255.
- E 240: word constant out of range  
The .HALF directive stores expressions in words. A word can hold 16 bit numbers. Check the range of the expression.
- E 241: Cannot emit non ti of functions, replaced with integral value '0'  
Floating point expressions and some functions can not be represented in the IEEE-695 object format. When an expression contains unknown symbols it cannot be evaluated and not emitted to the object file. Change these expressions to integral expressions, or make sure they can be evaluated during assembly.
- E 242: the *name* attribute must be specified  
A section must have the CODE or DATA attribute.

E 243: use \$OBJECT OFF or \$OBJECT "*object-file*"

E 244: unknown control "*name*"

The specified control does not exist. See chapter 8 for a description of all available controls.

E 245: error in expression: *expression*

Check the expression. The expression might not be a constant value.

E 246: .ENDM within .IF/.ENDIF

The assembler found an .ENDM directive within an .IF/.ENDIF pair. Check the macro and dup definitions or remove this directive.

E 247: illegal condition code

The assembler encountered an illegal condition code within an instruction. Check your input line.

E 248: cannot evaluate origin expression of org "*name: address*"

All origins of absolute sections must be evaluated before creation of the object file. Check the address expression on the usage of undefined or location dependant symbols.

E 249: incorrect argument types for function "*function*"

The supplied argument(s) evaluated to a different type than expected. Change the argument expressions to the correct type.

E 250: tiof function not yet implemented: "*function*"

The supplied tiof function is not yet implemented.

E 251: @POS(, *start*) start argument past end of string

The *start* argument is larger than the length of the string in the first parameter. Change *start* to the correct range.

E 252: second definition of label "*label*"

The label is defined twice in the same scope. Check the label definitions and rename or remove duplicate definitions.

E 253: recursive definition of symbol "*symbol*"

The evaluation of the symbol depends on its own value. Change the symbol value exclude this cyclic definition.

- E 254: missing closing '>' in include directive  
The syntax checker missed the closing '>' bracket in the include directive. Add a closing '>'.
- E 255: could not open include file *include-file*  
The assembler could not open the given include-file. Check the current search path for the presence of the include file and if it may be read.
- E 256: integral divide by zero  
The expression contains an divide by zero. This is not defined. Change the expression to exclude a division by zero.
- E 257: unterminated string  
All strings must end on the same line as they are started. Check for a missing ending quot.
- E 258: unexpected characters after macro parameters, possible illegal white space  
Spaces are not permitted between macro parameters. Check the syntax of the macro call.
- E 259: .COMMENT directive not permitted within a macro definition and conditional assembly  
The TASKING assembler does not permit the usage of the .COMMENT directive within .MACRO/.DUP definitions or .IF/.ELSE/.ENDIF constructs. Replace the offending .COMMENTS with comments starting with a semicolon.
- E 260: definition of "macro" unterminated, missing "endm"  
The macro definition is not terminated with an .ENDM directive. Check the macro definition.
- E 261: macro argument name may not start with an '\_'  
.MACRO and .DUP arguments may not start with an underscore.  
Replace the offending parameter names with non-underscore names.
- E 262: cannot find "symbol"  
Could not find a definition of the argument of a '%' or '?' operator within a macro expansion. Check for a definition of the offending symbol.

- E 263:     cannot evaluate: "*symbol*", value is unknown at this point  
The symbol used with a '%' or '?' operator within a macro expansion has not been defined. Insert a definition of the offending identifier.
- E 264:     cannot evaluate: "*symbol*", value depends on an unknown symbol  
Could not evaluate the argument of a '%' or '?' operator within a macro expansion. Check the definition of the offending symbol.
- E 265:     cannot evaluate argument of *dup* (unknown or location dependant symbols)  
The arguments of the .DUP directive could not be evaluated. Check the argument expressions on forward references or unknown symbols.
- E 266:     *dup* argument must be integral  
The argument of the .DUP directive must be integral. Change the expression so that it evaluates to an integral number.
- E 267:     *dup* needs a parameter  
Check the syntax of the .DUP directive.
- E 268:     ENDM without a corresponding .MACRO or D.UP definition  
The assembler found an .ENDM directive without an corresponding .MACRO or .DUP definition. Check the macro and dup definitions or remove this directive.
- E 269:     .ELSE without a corresponding .IF  
The assembler found an .ELSE directive without an corresponding .IF directive. Check the .IF/.ELSE/.ENDIF nesting or remove this directive.
- E 270:     .ENDIF without a corresponding .IF  
The assembler found an ENDIF directive without an corresponding .IF directive. Check the .IF/.ELSE/.ENDIF nesting or remove this directive.
- E 271:     missing corresponding .ENDIF  
The assembler found an .IF or .ELSE directive without an corresponding .ENDIF directive. Check the .IF/.ELSE/.ENDIF nesting or remove this directive.
- E 272:     label not permitted with this directive  
Some directives do not accept labels. Move the label to a line before or after this line.



E 273: wrong number of arguments for *function*

The function needs more or less arguments. Check the function definition and add or remove arguments.

E 274: illegal argument for *function*

An argument has the wrong type. Check the function definition and change the arguments accordingly.

E 275: expression not properly aligned

E 276: immediate value must be between *value* and *value*

The immediate operand of the instruction does only accept values in the given range. Use the '&' operator to force a value within the needed range or use '#>' to force a long immediate operand.

E 277: address must be between *\$address* and *\$address*

The address operand is not in the range mentioned. Change the address expression.

E 278: operand must be an address

The operand must be an address but has no address attributes. Use an address modifier or change the address expression.

E 279: address must be short

E 280: address must be short

The operand must be an address in the short range. The expression evaluated to a long address or an address in an unknown range.

E 281: illegal option "*option*"

The assembler found an unknown or misspelled command line option. The option will be ignored. Use the **-?** option to see a list of all possible options.

E 282: "Symbols:" part not found in map file "*name*"

The map file may be incomplete. Check if it is correctly produced by the locator.

E 283: "Sections:" part not found in map file "*name*"

The map file may be incomplete. Check if it is correctly produced by the locator.

E 284:    module "*name*" not found in map file "*name*"

The map file may be incomplete. Check if it is correctly produced by the locator.

E 285:    *file-kind* file will overwrite *file-kind* file

The assembler warns when one of its output files will overwrite the source file you gave on the command line or another output file.

Change the name of the source file, use the **-o** option to change the name of the output file or remove the **-err** option to suppress the generation of the error file.

E 286:    \$CASE options must be given before any symbol definition

The \$CASE options may only be given before any symbol is defined. Move the options to the start of the first source file.

E 287:    symbolic debug error: *message*

The assembler found an error in a symbolic debug (.SYMB) instruction. When the SYMB instruction is generated by the TASKING C compiler, please fill out the error report form and send it to TASKING. As a work around you could disable the symbolic debug information of this module (remove the **-g** option).

E 288:    error in PAGE directive: *message*

The arguments supplied to the PAGE directive do not conform to the restrictions. Check the PAGE directive restrictions in the manual and change the arguments accordingly.

E 290:    fail: *message*

Output of the .FAIL directive. This is an user generated error. Check the source code to see why this .FAIL directive is executed.

E 291:    generated check: *message*

Integrity check for the coupling between the TASKING C compiler and TASKING assembler. You should not see this error message, unless there are error in user inserted assembly (using the "#pragma asm" construct).

E 293:    expression out of range

An instruction operand must be in a specified address range. Check the address expression, change it.

- E 294: expression must be between *hexvalue* and *hexvalue*
- E 295: expression must be between *value* and *value*
- E 296: optimizer error: *message*  
The optimizer found an error. Try to change the instruction or turn off the the optimizer.
- E 297: jump address must be a code address  
Jumps and jump-subroutines must have a target address in code memory. Check the address expression or use a memory modifier to force the expression into code memory.
- E 298: size depends on location, cannot evaluate  
The size of some constructions (notably the align directives) depend on the memory address. Change the offending construction.
- E 299: absolute expression expected for section offset  
The expression must result in an absolute expression, to be used as a section offset.
- E 301: #error: *line*  
A C preprocessor error occurred.
- E 302: illegal C preprocessor '#define'-name  
A '#define' name must be a legal identifier.
- E 303: error in parameter list of '#define' symbol definition  
Another parameter or the ')' character was expected.
- E 304: missing '#define'-name for 'defined(...)' function
- E 305: '*name*' is an unknown or non-constant symbol
- E 306: duplicate macro parameter '*name*'  
Each macro parameter must have a unique name.
- E 307: wrong number of arguments for '*name*'  
Check the definition of the macro for the correct number of arguments.

E 308:    *message*

E 350:    SFR include file *regcpu.def* not found

The assembler tries to open the file "regcpu.def" to add the predefined special function registers to the symbol table. Check if you have entered the correct cpu type and/or check if the file is present in the include directory of the installed product.

E 351:    error in control: *message*

The argument of the control may need enclosing parentheses or there is an invalid character in the control name.

## 4 FATAL ERRORS (F)

The following errors cause the assembler to terminate immediately. Fatal errors are usually due to user errors.

F 401: memory allocation error

A request for free memory is denied by the system. All memory has been used. You may have to break your program down into smaller pieces.

F 402: duplicate input filename "*file*" and "*file*"

The assembler requires one input filename on the command line. Two or more filenames is erroneous.

F 403: error opening *file-kind* file : "*file-name*"

The assembler could not open the given file. When this is a source file, check if the file you specified at the command line exists and if it is readable. When the file is a temporary file, check if the environment symbol TMPDIR has been set correctly.

F 404: protection error : *message*

No protection key or not a IBM compatible PC.

F 405: I/O error

The assembler cannot write its output to a file. Check if you have enough free disk space.

F 406: parser stack overflow

F 407: symbolic debug output error

The symbolic debug information is incorrectly written in the object file. Please fill out the error report form and send it to TASKING.

F 408: illegal operator precedence

The operator priority table is corrupt. Please fill out the error report form and send it to TASKING.

F 409: Assembler internal error

The assembler encountered internal inconsistencies. Please fill out the error report form and send it to TASKING.

- F 410: Assembler internal error: duplicate mufom "*symbol*" during rename

The assembler renames all symbols local to a scope to unique symbols. In this case the assembler did not succeed into making an unique name. Please fill out the error report form and send it to TASKING.

- F 411: symbolic debug error: "*message*"

An error occurred during the parsing of the .SYMB directive. When this SYMB directive is generated by the TASKING C compiler, please fill out the error report form and send it to TASKING.

- F 412: macro calls nested too deep (possible endless recursive call)

There is a limit to the number of nested macro expansions. Currently this limit is set to 1000. Check for recursive definitions or try to simplify your source when you encounter this restriction.

- F 413: cannot evaluate "*function*"

A function call is encountered although it should have been processed. As a work-around, try to locate the offending function call and remove it from your source. Please fill out the error report form and send it to TASKING.

- F 414: cannot recover from previous errors, stopped

Due to earlier errors the assembler internal state got corrupted and stops assembling your program. Remove the errors reported earlier and retry.

- F 415: error opening temporary file

The assembler uses temporary files for the debug information and list file generation. It could not open or create one of those temporary files. Check if the environment symbol TMPDIR has been set correctly.

- F 416: internal error in optimizer

The optimizer found a deadlock situation. Try to assemble without any optimization options. Please fill out the error report form and send it to TASKING.



ASTRI ERRORS

# APPENDIX

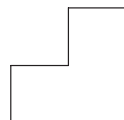
## B

### **LINKER ERROR MESSAGES**

---



**TASKING**





---

**B**

**APPENDIX**

---

## **1 INTRODUCTION**

Error and warning messages of the linker start with a letter followed by a number and an informational text. The error letter indicates the error type:

- W warning
- E error
- F fatal error
- V verbose message

## **2 WARNINGS (W)**

- W 100: Cannot create map file *filename*, turned off -M option  
The given file could not be created.
- W 101: Illegal filename (*filename*) detected  
A filename with an illegal extension was detected.
- W 102: Incomplete type specification, type index = *Texnumber*  
An unknown type reference. Arises if a pointer to an unspecified structure is defined.
- W 103: Object name (*name*) differs from filename  
Internal name of object file not the same as the filename. The file was probably renamed.
- W 104: '-o *filename*' option overwrites previous '-o *filename*'  
Second -o option encountered, previous name is lost.
- W 105: No object files found  
No files where specified at the invocation.
- W 106: No search path for system libraries. Use -L or env "*variable*"  
System library files (those given with the -l option) must have a search path, either supplied by means of the environment, or by means of the option -L.
- W 108: Illegal option: *option* (-H or -\? for help)  
An illegal option was detected.

- W 109: Type not completely specified for symbol *<symbol>* in *file*  
 Not a complete type specification in either the current file or the mentioned file. This could be an array with unknown depth, or a function with unknown parameters.
- W 110: Compatible types, different definitions for symbol *<symbol>* in *file*  
 Name conflict between compatible types. This could be a member name, tag name for a struct, or a different type name for equal sized basic types (int, long). Note that a basic type conflict is a non portable construct.
- W 111: Signed/unsigned conflict for symbol *<symbol>* in *file*  
 Size of both types is correct, but one of the types contains an unsigned where the other uses a signed type.
- W 112: Type conflict for symbol *<symbol>* in *file*  
 A real type conflict.
- W 113: Table of contents of *file* out of date, not searched. (Use **ar ts** *<name>*)  
 The **ar** library has a symbol table which is not up to date. Generate a new one with '**ar ts**'.
- W 114: No table of contents in *file*, not searched. (Use **ar ts** *<name>*)  
 The **ar** library has no symbol table. Generate one with '**ar ts**'.
- W 115: Library *library* contains ucode which is not supported  
 Ucode is not supported by the linker.
- W 116: Not all modules are translated with the same threshold (-G value)  
 The library file has an unknown format, or is corrupted.
- W 117: No type found for *<symbol>*. No type check performed  
 No type has been generated for the symbol
- W 118: Variable *<name>*, has incompatible external addressing modes with file *<filename>*  
 A variable is not yet allocated but two external references are made by non overlapping addressing modes. This is always an error.

- W 119: error from the Embedded Environment: *message*, switched off relaxed addressing mode check

If the embedded environment is readable for the linker, the addressing mode check is relaxed. For instance, a variable defined as data may be accessed as huge. For an overview of the embedded environment error messages, see appendix E, *Embedded Environment Error Messages*.

- W 120: Cannot find target description file *name*, relaxed addressing mode check disabled

The linker cannot find the description file (*.dsc*), this means that the linker cannot verify if addressing modes are compatible. For instance, the linker will now generate an error when far data is accessed as huge.

### 3 ERRORS (E)

- E 200: Illegal object, assignment of non existing var *var*  
The MUFOM variable did not exist. Corrupted object file.
- E 201: Bad magic number  
The magic number of a supplied library file was not ok.
- E 202: Section *name* does not have the same attributes as already linked files  
  
Named section with different attributes encountered. Use **-t flag** to see which files are already linked. It is possible that a previously linked file started a .out section with wrong attributes.
- E 203: Cannot open *filename*  
A given file was not found.
- E 204: Illegal reference in address of *name*  
Illegal MUFOM variable used in value expression of a variable.  
Corrupted object file.
- E 205: Symbol '*name*' already defined in *<name>*  
A symbol was defined twice. The message gives the files involved.
- E 206: Illegal object, multi assignment on *var*  
The MUFOM variable was assigned more than once probably due to a previous error 'already defined', E205.
- E 207: Object for different processor characteristics  
Bits per MAU, MAU per address or endian for this object differs with the first linked object.
- E 208: Found unresolved external(s):  
There were some symbols not found. If **-r** is not set, this is an error.
- E 209: Object format in *file* not supported  
The object file has an unknown format, or is corrupted.
- E 210: Library format in *file* not supported  
The library file has an unknown format, or is corrupted.

- E 211: Function *<function>* cannot be added to the already built overlay pool *<name>*

The overlay pool has already been built in a previous linker action. Use option **-r** to prevent this.

- E 212: Duplicate absolute section name *<name>*

Absolute sections begin on a fixed address. They cannot be linked.

- E 213: Section *<name>* does not have the same size as the already linked one

A section with the EQUAL attribute does not have the same size as other, already linked, sections.

- E 214: Missing section address for absolute section *<name>*

Each absolute section must have a section address command in the object. Corrupted object file.

- E 215: Section *<name>* has a different address from the already linked one

Two absolute sections may be linked (overlaid) on some conditions. They must have the same address.

- E 216: Variable *<name>*, *name <name>* has incompatible external addressing modes

A variable is allocated outside a referencing addressing space. For instance, the variable was not allocated in the zero page and this variable was referenced with the zero page addressing mode. This is always an error.

- E 217: Variable *<name>*, has incompatible external addressing modes with file *<filename>*

A variable is not yet allocated but two external references are made by non overlapping addressing modes. This is always an error.

- E 218: Variable *<name>*, also referenced in *<name>* has an incompatible address format

Addresses are often expressed in bytes. In some special cases, the address is expressed in bits. This is necessary for bit variables. An attempt was made to link different address formats between the current file and the mentioned file.

- E 219: Not supported/illegal *feature* in object format *format*  
An option/feature is not supported or illegal in given object format.
- E 220: page size (*0xbexvalue*) overflow for section *<name>* with size *0xbexvalue*  
Section is too big to fit into the page.
- E 221: *message*  
Error generated by the object. These errors are in fact generated by the assembler. It has been caused by a jump instruction which is out of range.
- E 222: Address of *<name>* not defined  
No address was assigned to the variable. Corrupted object file.
- E 223: Illegal object, empty name assignment on variable *name*  
An empty name assignment of a MUFOM variable (type N, X or I).

## **4 FATAL ERRORS (F)**

- F 400:     Cannot create file *filename*  
          The given file could not be created.
- F 401:     Illegal object: Unknown command at offset *offset*  
          An unknown command was detected in the object file. Corrupted object file.
- F 402:     Illegal object: Corrupted hex number at offset *offset*  
          Wrong byte count in hex number. Corrupted object file.
- F 403:     Illegal section index  
          A section index out of range was detected. Corrupted object file.
- F 404:     Illegal object: Unknown hex value at offset *offset*  
          An unknown variable was detected in the object file. Corrupted object file.
- F 405:     Internal error *number*  
          Internal fatal error. Passed number will give more information!
- F 406:     *message*  
          No key no IBM compatible PC
- F 407:     Missing section size for section *<name>*  
          Each section must have a section size command in the object. Corrupted object file.
- F 408:     Out of memory.  
          An attempt to allocate more memory failed.
- F 409:     Illegal object, offset *offset*  
          Inconsistency found in the object module.
- F 410:     Illegal object  
          Inconsistency found in the object module at unknown offset.
- F 413:     Only *name* object can be linked  
          It is not possible to link object for other processors.



- F 414: Input file *file* same as output file  
Input file and output file cannot be the same.
- F 415: Demonstration package limits exceeded  
One of the limits in this demo version was exceeded.
- F 416: Only one description file allowed  
The linker accepts only one description file.

## **5 VERBOSE (V)**

V 000:    Abort !

          The program was aborted by the user.

V 001:    Extracting files

          Verbose message extracting file from library.

V 002:    File currently in progress:

          Verbose message file currently processed.

V 003:    Starting pass *number*

          Verbose message, start of given pass.

V 004:    Rescanning....

          Verbose message rescanning library. Rescanning is done if there were new unsatisfied externals during the last scan.

V 005:    Removing file *file*

          Verbose message cleaning up. Temp files are always removed, map file and .out file are removed if switch -e is on and the exit code is unequal to zero.

V 006:    Object file *file* format *format*

          Named object file does not have the standard toolchain object format TIOF-695.

V 007:    Library *file* format *format*

          Named library file does not have the standard toolchain archiver format

V 8:       Embedded environment *name* read, relaxed addressing mode  
          check enabled

          Embedded environment successfully read.



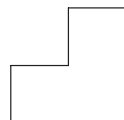
LKTRI ERRORS

# APPENDIX

## C

### **LOCATOR ERROR MESSAGES**

---



---

# C

# APPENDIX

---

## **1 INTRODUCTION**

Error and warning messages of the locator start with a letter followed by a number and an informational text. The error letter indicates the error type:

- W warning
- E error
- F fatal error
- V verbose message

## **2 WARNINGS (W)**

- W 100: Maximum buffer size for *name* is *size* (Adjusted)  
For the given format, a maximum buffer size is defined.
- W 101: Cannot create map file *filename*, turned off -M option  
The given file could not be created.
- W 102: Only one -g switch allowed, ignored -g before *name*  
Only one .out file can be debugged.
- W 104: Found a negative length for section *name*, made it positive  
Only stack sections can have a negative length.
- W 107: Inserted '*name*' keyword at line *line*  
A missing keyword in the description file was inserted.
- W 108: Object name (*name*) differs from filename  
Internal name of object file not the same as the filename. Maybe renamed?
- W 110: Redefinition of system start point  
Usually only one load module will access the system table (\_lc\_pm).
- W 111: Two -o options, output name will be *name*  
Second -o option, the message gives the effective name.
- W 112: Copy table not referenced, initial data is not copied  
If you use a copy statement in the layout part, the initial data is located in rom. Your start-up code should copy this data to their ram location.

- W 113: No .out files found to locate  
No files where specified at the invocation.
- W 114: Cannot find start label *label*  
No start point found.
- W 116: Redefinition of *name* at line *line*  
Identifier was defined twice.
- W 119: File *filename* not found in the argument list  
All files to be located must be given as an argument.
- W 120: unrecognized *name* option *<name>* at line *line* (inserted '*name*')  
Wrong option assignment. Check the manual for possibilities.
- W 121: Ignored illegal sub-option '*name*' for *name*  
An illegal format sub option was detected. See the format description for this format in the manual.
- W 122: Illegal option: *option* (-H or -\? for help)  
An illegal option was detected.
- W 123: Inserted *character* at line *line*  
The given character was missing in the description file.
- W 124: Attribute *attribute* at line *line* unknown  
An unknown attribute was specified in the description file.
- W 125: Copy table not referenced, blank sections are not cleared  
Sections with attribute blank are detected, but the copy table is not referenced. The locator generates info for the startup module in the copy table for clearing blank sections at startup. See *\_lc\_cp* in the manual.
- W 127: Layout *name* not found  
The used layout in the named file must be defined in the layout part.
- W 130: Physical block *name* assigned for the second time to a layout  
It is not possible to assign a block more than once to a layout block.

W 136: Removed *character* at line *line*

The character is not needed here.

W 137: Cluster *name* declared twice (layout part)

The named cluster is declared twice. Duplicate cluster names are allowed in the layout part under conditions, because the clusters are referred only. In the layout part the cluster is declared, which may be done only once.

W 138: Absolute section *name* at non-existing memory address  
*0xbexnumber*

Absolute section with an address outside physical memory. Either the address is not correct, or the memory description for your target is not consistent.

W 139: *message*

Warning message from the embedded environment. For an overview of the embedded environment error messages, see appendix E, *Embedded Environment Error Messages*.

W 140: File *filename* not found as a parameter

All processes defined in the locator description file (software part) must be specified on the invocation line.

W 141: Unknown space *<name>* in -S option

An unknown space name was specified with a -S option.

W 142: No room for *name* in read-only memory, trying writable memory ...

A section with attribute read-only could not be placed in read-only memory, the section will be placed in writable memory.

W 143: Section *name* has different page size than previous group members

Section has a different page size then other sections in the same group.

W 144: Filename *name* is too long, truncated to *name*

Filename is too long and is truncated.



W 145: Conflicting output options c (chip level) and s (start record), s ignored

Output sub-options 's' and 'c' are conflicting sub-options. The s option is ignored.

W 146: Address width in output format (*number* bytes) is too small for address *address*(hex). Only first occurrence reported.

The width of the address format is too small to contain the complete address.

### **3 ERRORS (E)**

- E 200: Absolute address *0xbexnumber* occupied  
An absolute address was requested, but the address was already occupied by another section.
- E 201: No physical memory available for section *name*  
An absolute address was requested, but there is no physical memory at this address.
- E 202: Section *name* with mau size *size* cannot be located in an addressing mode with mau size *size*  
A bit section cannot be located in a byte oriented addressing mode.
- E 203: Illegal object, assignment of non existing var *var*  
The MUFOM variable did not exist. For some variables this is an error.
- E 204: Cannot duplicate section '*name*' due to hardware limitations  
The process must be located more than once, but the section is mapped to a virtual space without memory management possibilities.
- E 205: Cannot find section for *name*  
Found a variable without a section, should not be possible.
- E 206: Size limit for the section group containing section *name* exceeded by *0xbexnumber* bytes  
Small sections do not fit in a page any more.
- E 207: Cannot open *filename*  
A given file was not found.
- E 208: Cannot find a cluster for section *name*  
No writable memory available, or unknown addressing mode. Often this error occurs due to an error in the description file.
- E 210: Unrecognized keyword *<name>* at line *line*  
An unknown keyword was used in the description file.
- E 211: Cannot find *0xbexnumber* bytes for section *name* (fixed mapping)  
One of virtual or physical memory was occupied, or there was no physical memory at all!

- E 213: The physical memory of *name* cannot be addressed in space *name*  
A mapping failed. There was no virtual address space left.
- E 214: Cannot map section *name*, virtual memory address occupied  
An absolute mapping failed. The memory on the virtual target address was already occupied.
- E 215: Available space within *name* exceeded by *number* bytes for *name*  
The available addressing space for an addressing mode has been exceeded.
- E 217: No room for *name* in cluster *name*  
The size of the cluster as defined in the .dsc file is too small.
- E 218: Missing *identifier* at line *line*  
This identifier must be specified.
- E 219: Missing ')' at line *line*  
Matching bracket missing.
- E 220: Symbol '*symbol*' already defined in <*name*>  
A symbol was defined twice.
- E 221: Illegal object, multi assignment on *var*  
The MUFOM variable was assigned more than once, probably due to an error of the object producer.
- E 223: No software description found  
Each input file must be described in the software description in the .dsc file.
- E 224: Missing <length> keyword in block '*name*' at line *line*  
No length definition found in hardware description.
- E 225: Missing <*keyword*> keyword in space '*name*' at line *line*  
For the given mapping, the keyword must be specified.
- E 227: Missing <start> keyword in block '*name*' at line *line*  
No start definition found in hardware description.

- E 230:    Cannot locate section *name*, requested address occupied  
          An absolute address was requested, but the address was already  
          occupied by another process or section.
- E 232:    Found file *filename* not defined in the description file  
          All files to be located need a definition record in the description file.
- E 233:    Environment variable too long in line *line*  
          Found environment variable in the dsc file contains too many  
          characters.
- E 235:    Unknown section size for section *name*  
          No section size found in this .out file. In fact a corrupted .out file.
- E 236:    Unrecoverable specification at line *line*  
          An unrecoverable error was made in the description file.
- E 238:    Found unresolved external(s):  
          At locate time all externals should be satisfied.
- E 239:    Absolute address *addr.addr* not found  
          In the given space the absolute address was not found.
- E 240:    Virtual memory space *name* not found  
          In the description files software part for the given file, a non existing  
          memory space was mentioned.
- E 241:    Object for different processor characteristics  
          Bits per MAU, MAU per address or endian for this object differs with  
          the first linked object.
- E 242:    *message*  
          Error generated by the object. These errors are in fact generated by the  
          assembler. It has been caused by a jump instruction which is out of  
          range.
- E 244:    Missing *name* part  
          The given part was not found in the description file, possibly due to a  
          previous error.

- E 245:    Illegal *name*value at line *line*  
           A non valid value was found in the description file
- E 246:    Identifier cannot be a number at line *line*  
           A non valid identifier was found in the description file
- E 247:    Incomplete type specification, type index = *Thexnumber*  
           An unknown type was referenced by the given file. Corrupted object file.
- E 250:    Address conflict between block *block1* and *block2* (memory part)  
           Overlapping addresses in the memory part of the description file.
- E 251:    Cannot find 0x*hexnumber* bytes for section *section* in block *block*  
           No room in the physical block in which the section must be located.
- E 255:    Section '*name*' defined more than once at line *line*  
           Sections cannot be declared more than once in one layout/loadmod part.
- E 258:    Cannot allocate reserved space for process *number*  
           The memory for a reserved piece of space was occupied.
- E 261:    User assert: *message*  
           User-programmed assertion failed. These assertions can be programmed in the layout part of the description file.
- E 262:    Label '*name*' defined more than once in the software part  
           Labels defined in the description file must be unique.
- E 264:    *message*  
           Error from the embedded environment. For an overview of the embedded environment error messages, see appendix E, *Embedded Environment Error Messages*.
- E 265:    Unknown section address for absolute section *name*  
           No section address found in this .out file. In fact a corrupted .out file.

- E 266:    *functionality* not (yet) supported  
          The requested functionallity is not (yet) supported in this release.

## **4 FATAL ERRORS (F)**

- F 400:    Cannot create file *filename*  
          The given file could not be created.
- F 401:    Cannot open *filename*  
          A given file was not found.
- F 402:    Illegal object: Unknown command at offset *offset*  
          An unknown command was detected in the object file. Corrupted object file.
- F 403:    Illegal filename (*name*) detected  
          A filename with an illegal extension was detected on the command line.
- F 404:    Illegal object: Corrupted hex number at offset *offset*  
          Wrong byte count in hex number. Corrupted object file.
- F 405:    Illegal section index  
          A section index out of range was detected. This could be a corrupted object file, but also a previous error like E231 (Missing section) is responsible for this message.
- F 406:    Illegal object: Unknown hex value at offset *offset*  
          An unknown variable was detected in the object file. Corrupted object file.
- F 407:    No description file found  
          The locator must have a description file with the description of the hardware and the software of your system.
- F 408:    *message*  
          No protection key or not an IBM compatible PC.
- F 410:    Only one description file allowed  
          The locator accepts only one description file.

- F 411: Out of memory.  
An attempt to allocate more memory failed.
- F 412: Illegal object, offset *offset*  
Inconsistency found in the object module.
- F 413: Illegal object  
Inconsistency found in the object module at unknown offset.
- F 415: Only *name* .out files can be located  
It is not possible to locate object for other processors.
- F 416: Unrecoverable error at line *line*, *name*  
An unrecoverable error was made in the description file in the given part.
- F 417: Overlaying not yet done  
Overlaying is not yet done for this .out file, link it first without **-r** flag!
- F 418: No layout found, or layout not consistent  
If there are syntax errors in the layout, it may occur that the layout is not usable for the locator. Syntax errors in the description file must be resolved!
- F 419: *message*  
Fatal from the embedded environment. For an overview of the embedded environment error messages, see appendix E, *Embedded Environment Error Messages*.
- F 420: Demonstration package limits exceeded  
One of the limits in this demo version was exceeded.
- F 421: Error writing file *name*  
An error occurred when writing to the file.
- F 422: Input file *name* same as output file  
Input file and output file cannot be the same.

## **5 VERBOSE (V)**

V 000: File currently in progress:

Verbose message. On the next lines single filenames are printed as they are processed.

V 001: Output format: *name*

Verbose message for the generated output format.

V 002: Starting pass *number*

Verbose message, start of given pass.

V 003: Abort !

The program was aborted by the user.

V 004: Warning level *number*

Verbose message, report the used warning level.

V 005: Removing file *file*

Verbose message cleaning up. Temporary files are always removed, map file and .out file are removed if switch **-e** is on and the exit code is unequal zero.

V 006: Found file *<filename>* via path *pathname*

The description (include) file was not found in the standard directory. The locator searches also in the install directory etc, in which the file was found.

V 007: *message*

Verbose message from the embedded environment. For an overview of the embedded environment error messages, see appendix E, *Embedded Environment Error Messages*.



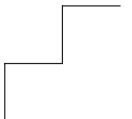


LCTRI ERRORS

# APPENDIX D

---

## **ARCHIVER ERROR MESSAGES**



---

# D | APPENDIX

---

## **1 INTRODUCTION**

This appendix contains all warnings (W), errors (E) and fatal errors (F) of the archiver **artri**.

## **2 WARNINGS (W)**

W 100: Illegal warning level: *level*

Warning level is a single digit.

W 101: Member *name* not found

Library member not found, warning only.

W 102: Can't modify modification time for *name*

The archiver cannot access the file *name* to change the modification time.

W 103: creating archive *name*

The **q** option was used while archive file did not exist (**r** option would be more appropriate).

W 104: Option **-a** or **-b** only allowed with key option 'r' or 'm'. Ignored!

Option **a** or **b**, which specifies a position in the archive can only be applied with replace or move actions.

W 105: Only one position specification allowed, ignored '**-a** or **-b**  
*file\_offset*'

It is not possible to specify more than one position in the archive. The options **-a** and **-b** are both used to specify a position.

W 106: Option **-o** only allowed with key option 'x'. Ignored!

Library date can only be preserved with extraction of a library member.

W 107: Option **-u** only allowed with key option 'r'. Ignored!

Objects newer than the archive are only replaced with key option r.

W 108: Option **-z** only allowed with key option 'r'. Ignored!

Only objects which are moved to the archive can be checked.

W 109: Option **-v** has no meaning with key option 'p' or 't'. Ignored!

For options p and t the verbose switch is meaningless.

- W 110: Option `-s` may be combined with `-t` only.
- W 111: Illegal symbol level: *level*  
Symbol level is a single digit.
- W 112: Name *name* is too long, truncated to *name*  
The name exceeded the limit, and is truncated.

### **3 ERRORS (E)**

- E 200: filename too long  
The filename was too long to fit into the internal buffer.
- E 201: Member *name* not found  
Library member not found.
- E 204: Can't obtain file-status information *filename*  
Cannot access *filename* to obtain file status information.
- E 207: illegal option: *option*  
An illegal option was detected.
- E 209: Can't rename file: *name* to: *name*  
Renaming the library file to a tempfile failed.

### **4 FATAL ERRORS (F)**

- F 300: user abort  
The library manager is aborted by the user.
- F 301: too much errors  
The maximum number of errors is exceeded.
- F 302: protection error: *error*  
error message received from `ky_init`.
- F 303: can't create "*filename*"  
Cannot create the file with the mentioned name.

- F 304:    can't open "*filename*"  
           Cannot open the file with the mentioned name.
- F 305:    can't reopen '*filename*'  
           The file *filename* could not be reopened.
- F 306:    read error while reading "*filename*"  
           A read error occurred while reading named file.
- F 307:    write error  
           A write error occurred while writing to the output file.
- F 308:    out of memory  
           An attempt to allocate memory failed.
- F 309:    illegal character  
           A character which is not allowed was found.
- F 310:    *filename* not in archive format  
           the archive file given is not in the proper format.
- F 311:    specification of more than one key {*rxdmpt*} is not permitted  
           More than one key was given.
- F 312:    no one of the keys {*rxdmpt*} was specified  
           No key was given.
- F 313:    error in the invocation. Use option *-?* or *-H* to get help.  
           Show usage. For more help, use option *-?*.
- F 314:    *name* does not exist  
           Library will only be created in case the *r* key-option is specified.
- F 315:    IEEE violation for object module *name* at address *address*  
           IEEE violation detected (*z* option enabled).
- F 316:    corrupted object module *name*  
           The object module name does not conform to the IEEE object specification.

F 317:    *name*: illegal byte count in hex number, offset = *offset*

          Illegal byte count in hex number (IEEE violation).

F 318:    evaluation date expired !!

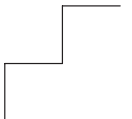
F 319:    *message*

          No protection key or not an IBM compatible PC.

# APPENDIX E

---

## **EMBEDDED ENVIRONMENT ERROR MESSAGES**





---

**■**

# APPENDIX

---

## **1 INTRODUCTION**

Error and warning messages from the embedded environment are part of the linker and/or locator error messages. The error numbers mentioned below are not part of the message.

E error  
W warning

## **2 ERRORS (E)**

- E 1:       Conflicting attributes *attributes* at line *number*  
          Conflicting attributes.
- E 2:       Unknown attribute '*character*' at line *number*  
          Unknown attribute.
- E 3:       Unknown keyword '*name*' at line *number*  
          Unknown keyword.
- E 4:       Illegal character '*character*' at line *number*  
          Illegal character.
- E 5:       Page size only allowed in a space definition at line *number*  
          Page size only allowed in space definition.
- E 6:       Page size must be a power of 2 at line *number*  
          Page size must be a power of 2.
- E 7:       Mau size must be a power of 2 at line *name*  
          Mau size must be a power of 2.
- E 8:       Cannot synchronize any more line *number*  
          Cannot synchronize any more.
- E 9:       Illegal value '*value*' at line *number*  
          Illegal value.
- E 10:      Illegal hex value '*value*' at line *number*  
          Illegal hex value.

- E 11: Illegal octal value '*value*' at line *number*  
Illegal octal value.
- E 12: Missing value at line *number*  
Missing value.
- E 13: Illegal identifier at line *number*  
Illegal identifier.
- E 14: Wrong attribute '*attribute*' at line *number*  
Attribute not allowed.
- E 15: Unknown identifier '*name*' at line *number*  
Unknown identifier.
- E 16: Inserted '*character*' at line *number*  
Inserted character.
- E 17: Cannot find bus/space '*name*' in definition for space '*name*'  
Error in the destination of mapping from space.
- E 18: Cannot find space/amode '*name*' in definition for amode '*name*'  
Map error.
- E 19: Cannot find chip '*name*' in definition for bus '*name*'  
Map error.
- E 20: Cannot find space/amode '*name*' in layout definition for cluster '*name*'  
Map error.
- E 21: Cannot find bus '*name*' in definition for mapping '*name*'  
Map error.

### **3 WARNINGS (W)**

W 100: Cannot find mapping '*name*' in cluster definition for space '*name*'

Warning in cluster mapping.

W 101: Section '*name*' should be defined in amode '*name*', not amode '*name*'

The section was specified in the wrong addressing mode



EEL ERRORS

# APPENDIX

## F

### **DESCRIPTIVE LANGUAGE FOR EMBEDDED ENVIRONMENTS**

---



---

**F**

**APPENDIX**

---

## **1 INTRODUCTION**

In an embedded environment an accurate description of available memory and control over the behavior of the locator is crucial for a successful application. For example, it may be necessary to port applications to processors with different memory configurations, or it may be necessary to tune the location of sections to take full advantage of fast memory chips.

For this purpose the DELFEE language, which stands for DEscriptive Language For Embded Environments, was designed.

## **2 GETTING STARTED**

### **2.1 INTRODUCTION**

This section gives a general introduction about the DELFEE description language. The goal is to give you an overview and some basic knowledge what the DELFEE description language is about, and how a basic description file looks. A more detailed description and examples are given in the following sections.

### **2.2 BASIC STRUCTURE**

The DELFEE language describes where code or data sections should be placed on the actual memory chips. This language has to define the interface between a virtual world (the software) and a physical world (the hardware configuration).

On the one side, in the virtual world, there are the code and data sections which are described by the assembly language. Sections can have names, attributes like writable or read-only and can have an address in the addressing space or an addressing mode describing the range of the address space in which they may be located.



On the other side, the physical world, the actual processor is present which reads instructions from memory chips and interprets these instructions. With the DELFEE language you can instruct the locator to place the code and data sections at the correct addresses, taking into account things like the type of memory chip (rom/ram, fast/slow), availability of memory, etc. The DELFEE language gives the possibility to tune the same application for different hardware configurations.

In the DELFEE language the interface between virtual and physical world is described in three parts:

1. **software part** (\* .dsc)

The software part belongs to the virtual world and describes the ordering of the data and code sections. The software part may vary for different applications and can even be empty.

2. **cpu part** (\* .cpu)

The cpu part is the interface between the virtual world and the real world. It contains the application independent part of the virtual world (the address translation of addressing modes to the addressing space), and the configuration independent part of the physical world (on-chip memory, address busses). The cpu part is independent of application and configuration.

3. **memory part** (\* .mem)

The memory belongs to the physical world. It contains the description of the external memory. The memory part may vary for different configurations and can even be empty (if there is no external memory).



The software part and the memory part can be empty, but that the cpu part must always be defined.

The DELFEE language is used in a special file, which is called the description file. In the DELFEE description language the different parts are defined with the following syntax:

```
software {  
    layout {  
        // ordering of sections  
    }  
}  
  
cpu {  
    // mapping of addressing modes to address space  
    // defining address space  
    // mapping of address space to actual busses  
    // defining on-chip memory  
}  
  
memory {  
    // description of external memory  
}
```

For convenience the cpu part and the memory part can be placed in different files, which makes it possible to have different layout parts for different applications and different memory parts for different configurations. The files can be included using the syntax:

```
cpu filename      // include cpu part defined in file filename  
mem filename      // include memory part defined in file filename
```

### 3 CPU PART

#### 3.1 INTRODUCTION

The cpu part contains the application and configuration independent part of the description file. This part defines the translations of the addresses from the assembler language (virtual addresses) all the way down to the chips (physical addresses). To describe the translations, DELFEE recognizes four main levels:

1. addressing mode(s) definitions. Addressing modes are subsets of an address space. They define address ranges within an address space.
2. address space(s) definitions. The address space is the total range of addresses available.
3. bus(es) definitions.
4. (on-chip) memory chips definitions.

The address translation is defined from addressing mode via space and bus to the chip. The addressing modes and the busses can be nested, the space and the chip cannot.

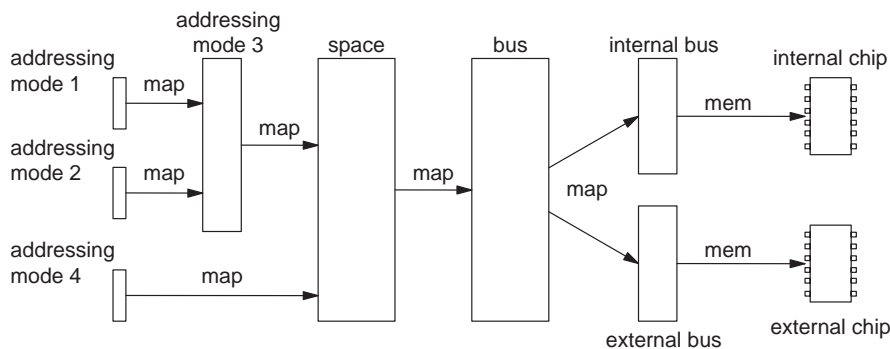


Figure F-1: Address translation

The addressing modes and addressing spaces belong to the virtual part, the busses and chips belong to the physical part. The following sections describe the address space and the addressing modes which are subsets of the address space. Then a description of the physical side (hardware configuration) follows, describing the busses and chips that are available.

The following example illustrates how a cpu part could look like. It is a fictitious example, mainly used to illustrate the definitions. You should be able to recognize the addressing mode definitions, address space definition, bus definitions and on-chip memory definition. Each definition is explained in the following sub-sections.

```
cpu {
  //
  // addressing mode definitions
  //
  amode near_code {
    attribute Y1;
    mau 8;
    map src=0 size=1k dst=0 amode = far_code;
  }
  amode far_code {
    attribute Y2;
    mau 8;
    map src=0 size=32k dst=0 space = address_space;
  }
  amode near_data {
    attribute Y3;
    mau 8;
    map src=0 size=1k dst=0 amode = far_data;
  }
  amode far_data {
    attribute Y4;
    mau 8;
    map src=0 size=32k dst=32k space = address_space;
  }
}

//
// space definitions
//
space address_space {
  mau 8;
  map src=0 size=32k dst=0 bus = address_bus label = rom;
  map src=32k size=32k dst=32k bus = address_bus label = ram;
}

//
// bus definitions
//
bus address_bus {
  mau 8;
  mem addr=0 chips=rom_chip;
  map src=0x100 size=0x7f00 dst=0x100 bus = external_rom_bus;
  mem addr=32k chips=ram_chip;
  map src=0x8100 size=0x7f00 dst=0x100 bus = external_ram_bus;
}

//
// internal memory definitions
//
```

```

chips rom_chip attr=r mau=8 size=0x100; // internal rom
chips ram_chip attr=w mau=8 size=0x100; // internal ram
}

```

### 3.2 ADDRESS TRANSLATION: MAP AND MEM

In DELFEE there are two ways to describe a memory translation between two levels (the source level and the destination level):

1. **map** keyword. This is for address translations between amodes, spaces, busses (not chips).
2. **mem** keyword. This describes the address translation between bus and chip. **mem** is a simplified case of **map**.

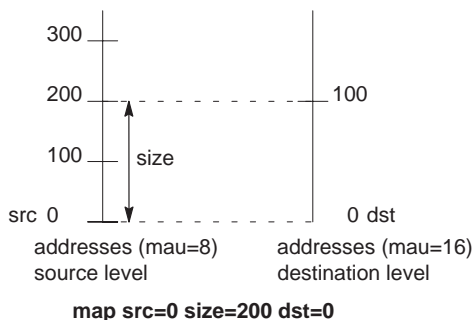


Figure F-2: Map address translation

The generalized syntax for the map definition is (see figure F-2):

```

map src=number size=number dst=number
    destination_type=destination_name optional_specifiers;

```

where,

- |             |   |
|-------------|---|
| <b>src</b>  | start address of the source level. In case of an address translation between amodes and spaces, the source level is the amode and the destination level is the space. |
| <b>size</b> | length of the source level.   |
| <b>dst</b>  | start address at the destination level.   |

*destination\_type* the destination type depends on the context the mapping is used in and can have three different types:

1. **amode** allowed in context: amode.
2. **space** allowed in context: amode.
3. **bus** allowed in context: space, bus.

*optional\_specifiers* The optional identifiers are also dependent of the context they are used in:

1. **label** Only allowed in space context and needed as a reference for the block definition in the software part (see section 4.5).  
**label = name ;**
2. **align** This indicates that every section will be aligned at the specified value.

**align = number ;**

3. **page** This indicates that every section should be within a given page size.

**page = number ;**

Both the source level and the destination level have an address range that is expressed in a number of Minimum Addressable Units (MAU, the minimal amount of storage, in bits, that is accessed using an address). The mapping only describes the range and the destination of the address mapping, the actual transformation also depends on the memory unit that an address can access. If a source level with a minimum addressable unit of 8 bits (mau=8) maps to a destination level with a minimum addressable unit of 16 bits (mau=16), the size of the destination level, expressed in address range, is half the original size. So, according to figure F-2, the size of the destination level is 100.

If a map is present from *level1* down to *level2*, the map definition works as follows:

*end\_address of level2* = *dst* + ( *size* \* *mau of level1* / *mau of level2* )

The **mem** description is actually a simplified case of the **map** description. The length of the address translation is taken from the chip size, the destination address is always zero. It is used to map a bus to a chip.

The syntax is:

```
mem addr=number chips=name;
```

where,

**addr**            start address location of a chip.

**chips**          the name of the chip that is located at address *number*.

### **3.3 ADDRESS SPACES**

The link between the virtual and the physical world is the description of the address space and the way it maps onto the internal address busses.

The address space is defined by the complete range of addresses that the instruction set can access. Some instruction sets support multiple address spaces (for example a data space and a code space).

An address space is described by the syntax:

```
space name {
    mau number;
    map src=number size=number dst=number bus=bus_name label=name;
    // :
    // more maps
}
```

where,

**space**          defines the name by which the space can be referenced in the description file.

**mau**            the Minimum Addressable Unit, meaning the minimum amount of storage (in bits) that is accessed using an address.

**map** this specifies the mapping of a range of addresses in the address space to a bus defined by *bus\_name*. The range of addresses is defined by **src** and **length**, the offset on the bus is defined by **dst**. (The bus you map the address space on, may have a different MAU, which will lead to another length of the range of the bus). An address space can only map onto a bus.

Usually an address in the address space corresponds to the same address on the bus. In that case **src** and **dst** have the same value.

In the previous example there is one space definition:

```
space address_space {
    mau 8;
    map src=0    size=32k dst=0    bus = address_bus label = rom;
    map src=32k size=32k dst=32k bus = address_bus label = ram;
}
```

In this example the space is named *address\_space*. Note that the **amod** definitions use this name as destination for their mappings. The minimum addressable unit (MAU) is set to 8 bits. The labels *rom* and *ram* are used by **block** definitions in the software part which are discussed in section 4.5.

### 3.4 ADDRESSING MODES

Addressing modes define address ranges in the addressing space. Addressing modes usually have a special characteristic, like bitaddressable part of memory, parts especially for code sections, zero pages, etc. The addressing modes are defined by the instruction set. The syntax of defining an addressing mode in the DELFEE language is:

An address space is described by the syntax:

```
amode name {
    mau number;
    attr Ynumber;
    map src=number size=number dst=number amode | space=name;
}
```



where,

- amode**      The name by which the addressing mode can be referenced. In the object file the addressing mode of a section is encoded with an **Ynumber**. This means that the *name* given to the addressing mode has only meaning within the description file, not to the sections!
- mau**        the Minimum Addressable Unit, meaning the minimum amount of storage (in bits) that is accessed using an address.
- attr Y**      the addressing mode number. Code or data sections (generated by the assembler) all have a number specifying the addressing mode they belong to. In the DELFEE description file this number is used to identify the addressing mode. This number must never be changed, because the interpretation of the sections will get mixed up.
- map**        defines the mapping of the addressing mode to another addressing mode (**amode**) or an address space (**space**).

Below is an example of two addressing mode definitions:

```
amode near_data {
    attribute Y3;
    mau 8;
    map src=0 size=1k dst=0 amode = far_data;
}
amode far_data {
    attribute Y4;
    mau 8;
    map src=0 size=32k dst=32k space = address_space;
}
```

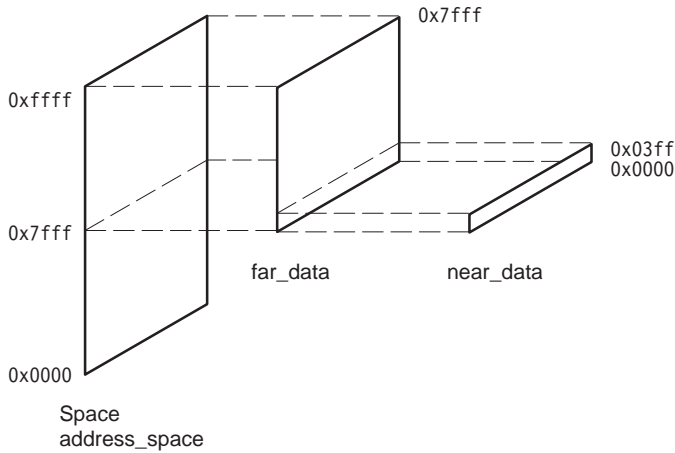


Figure F-3: Addressing mode mapping

In this example the addressing modes are named `near_data` and `far_data`. They are identified by the addressing mode numbers `Y3` and `Y4` respectively. The minimum addressable unit (MAU) is set to 8 bits. Addressing mode `near_data` maps on addressing mode `far_data`, and `far_data`, in its turn, maps on address space `address_space`. `address_space` is the space as discussed in the previous section.

### 3.5 BUSSES

The **bus** keyword describes the bus configuration of a cpu. In essence it describes the address translation from the address space to the chip. The syntax is:

```
bus name {
    mau number;
    map src=number size=number dst=number bus=name;
    mem addr=number chips=name;
}
```

where,

**bus** the name by which the bus can be referenced.

**mau** the Minimum Addressable Unit, meaning the minimum amount of storage (in bits) that is accessed using an address.

**map** mapping to another bus.

**mem** mapping to a memory chip.

Below is an example of a bus definition:

```
bus address_bus {
    mau 8;
    mem addr=0 chips=rom_chip;
    map src=0x100 size=0x7f00 dst=0x100 bus = external_rom_bus;
    mem addr=32k chips=ram_chip;
    map src=0x8100 size=0x7f00 dst=0x100 bus = external_ram_bus;
}
```

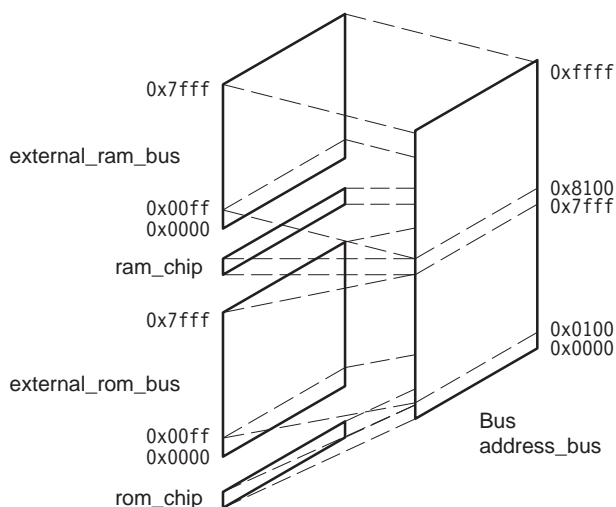


Figure F-4: Bus mapping

In this example the address bus is named `address_bus`. The minimum addressable unit (MAU) is set to 8 bits. The internal memory chip `rom_chip` is located at address 0 of the bus, and the chip `ram_chip` is located at address 32k.

Two address mappings to other busses are present: one to `external_rom_bus` and one to `external_ram_bus`.

The first mapping translates addresses `0x100`–`0x7ff` of `address_bus` (`src=0x100 size=0x7f00`) onto addresses of `external_rom_bus` starting at address `0x100` (`dst=0x100`).

The second mapping translates addresses 0x8100-0xffff of `address_bus` (`src=0x8100 size=0x7f00`) onto addresses of `external_ram_bus` starting at address 0x100 (`dst=0x100`).



The second mapping maps to RAM, not ROM. That is why both destination addresses are the same.

### 3.6 CHIPS

The **chips** keyword describes the memory chip. The syntax is:

**chips** *name* **attr**=*letter\_code* **mau**=*number* **size**=*number*;

where,

**chips** the name by which the chip can be referenced.

**attr** defines the attributes of the chip with a letter code

*letter\_code* one of the following attributes:

**r** read-only memory.

**w** writable memory.

**s** special memory (it must not be located).

**mau** the Minimum Addressable Unit, meaning the minimum amount of storage (in bits) that is accessed using an address.

**size** the size of the chip (address range from 0-*size*).

Below is an example of two chip definitions:

```
chips rom_chip attr=r mau=8 size=0x100; // internal rom
chips ram_chip attr=w mau=8 size=0x100; // internal ram
```

In this example the chips are named `rom_chip` and `ram_chip`. The minimum addressable unit (MAU) is set to 8 bits. The size of both chips is 0x100 MAUs (= 256 bytes). Chip `rom_chip` is read-only and chip `ram_chip` writable, as you would expect with ROM and RAM.

### 3.7 EXTERNAL MEMORY

With the syntax described in the previous sections it would be possible to define mappings from an address space to external memory chips (DELFE does not actually know, or care, if memory is on-chip). However, this is not advisory. For maintenance and flexibility reasons it is better to keep the internal (static) memory part apart from the external (variable) memory part. The chapter *Memory Part* describes how to deal with external memory.

In the cpu part you only have to define a mapping to an external bus, which can later be defined in the memory part. The following example contains references to two external busses: `external_ram_bus` and `external_rom_bus`.

```
bus address_bus {
    mau 8;
    mem addr=0 chips=rom_chip;
    map src=0x100 size=0x7f00 dst=0x100 bus = external_rom_bus;
    mem addr=32k chips=ram_chip;
    map src=0x8100 size=0x7f00 dst=0x100 bus = external_ram_bus;
}
```

## **4 SOFTWARE PART**

### **4.1 INTRODUCTION**

The software part has two main parts:

1. **load\_mod**
2. **layout description**

```
software {  
    load_mod start = start_label;  
  
    layout {  
        // ordering of sections  
    }  
}
```

### **4.2 LOAD MODULE**

The keyword **load\_mod** defines the program start label. The program start label is the start of the code and the reset vector should point to this label. The locator generates a warning if this label is not referenced.

```
load_mod start = start_label;
```

### **4.3 LAYOUT DESCRIPTION**

First of all, the layout definition can be omitted. If you omit the layout definition, the locator will generate a layout definition based on the DELFEE description of the amodes (addressing modes) in the cpu part (See section 3). However this does not allow you to control the order in which sections (like stack and heap) are located. If you define the layout part, the locator uses this description.

The layout part is probably the most difficult part of the DELFEE language. It is designed to give the locate algorithm the information it needs to locate the sections correctly. Through some examples you will be shown how to influence the locate algorithm using the DELFEE language.

To give you an idea of where all this will lead to, an example of a layout part is given:

```
layout {
    space address_space {
        block rom {
            cluster first_code_clstr {
                attribute i;
                amode near_code;
                amode far_code;
            }
            cluster code_clstr {
                attribute r;
                amode near_code {
                    section selection=x;
                    section selection=r;
                }
                amode far_code {
                    table;
                    section selection=x;
                    section selection=r;
                    copy;          // locate rom copies here
                }
            }
        }
        block ram {
            cluster data_clstr {
                attribute w;
                amode near_data {
                    section selection=w;
                }
                amode far_data {
                    section selection=w;
                    heap;
                    stack;
                }
            }
        }
    }
}
```

The layout definition is defined with the syntax:

```
layout {
    // space definitions
}
```

The first thing to notice is the different levels inside the layout definition:

**space**      This level can only occur inside a layout level. There are as much space levels as there are space definitions in the cpu part.

<b>block</b>	This level can only occur inside a space level. There are as much block levels as there are mappings defined in the space definition in the cpu part.
<b>cluster</b>	This level can only occur inside a block level. There can be multiple clusters inside a block. Their main purpose is to group (code/data) sections. The locator locates each cluster in the specified order.
<b>amode</b>	This level can only occur inside a cluster level. An <b>amode</b> corresponds to an <b>amode</b> definition in the cpu part. Within an <b>amode</b> you can specify the order in which data/code sections are located.

The four levels can roughly be divided in two groups. The **space** and **block** definition correspond to address ranges and the **cluster** and **amode** definition correspond to (groups of) sections.

The following paragraphs first introduce the **space** and **block** definition. Then separate paragraphs show how to select certain groups of sections and how this is used in the **cluster** and **amode** definition.

## 4.4 SPACE DEFINITION

Section 3.3 already defined the address translation of a space in the cpu part. In the example in that section, the following space was defined:

```
space address_space {
    mau 8;
    map src=0   size=32k dst=0   bus = address_bus label = rom;
    map src=32k size=32k dst=32k bus = address_bus label = ram;
}
```

For every space defined in the cpu part you have to provide a description in the layout definition.

The space level should be inside the layout definition and can only contain one or more block levels.

The name of the space must correspond to a space definition in the cpu part.



The syntax is:

```
space name {
    // block definitions
}
```

Below is an example of a space definition from the software part:

```
space address_space {
    block rom {
        ....
    }
    block ram {
        ...
    }
}
```

In this example space `address_space` defines two blocks: block `rom` and block `ram`.

## 4.5 BLOCK DEFINITION

With the **block** description you can set boundaries to the sections based on chip sizes.

A block references a physical area of memory. Selected sections are only allowed within the range of the block description. In effect a block limits the range in which a section can be located.

The physical address range of a block is actually defined in the `cpu` part by a labeled mapping:

```
space address_space {
    mau 8;
    map src=0 size=32k dst=0 bus = address_bus label = rom; //<--
    // --> block name: rom
    map src=32k size=32k dst=32k bus = address_bus label = ram; //<--
    // --> block name: ram
}
```

The name of the **block** description must correspond to a label in the **map** definition of a **space** definition in the `cpu` part. The **block** definition must be inside the **space** definition and can only contain one or more cluster levels.

The syntax is:

```
block name {
    // cluster definitions
}
```

Below is an example of a bus definition from the software part:

```
block rom {
    cluster first_code_clstr {
        ...
    }
    cluster code_clstr {
        ...
    }
}
```

In this example block `rom` defines two clusters: cluster `first_code_clstr` and cluster `code_clstr`.

## 4.6 SELECTING SECTIONS

The previous paragraphs explained how the address ranges are defined by block definitions, now it is time to select the sections that should be placed in these blocks. In DELFEE there are two levels in which you can define the order of locating:

1. **cluster**
2. **amode**

To define the locating order you need to have some kind of handle to specify a section or a group of sections. DELFEE recognizes the following characteristics of a section:

name of the section This is unique to a specific section.

attribute(s) of a section

The attributes of a section are specified by the assembler or compiler. Possible attributes are defined in table F-1. By selecting an attribute you select a group of sections. The attributes can be grouped to an attribute string, for example: **by1w**.

addressing mode    All sections have an addressing mode (as defined in the cpu part).

<i>attr</i>	Meaning	Description
W	Writable	Must be located in ram
R	Read only	Can be located in rom
X	Execute only	Can be located in rom
Z	Zero page	Must be located in the zero page
<i>Ynum</i>	Addressing mode	Must be located in addressing mode <i>num</i>
A	Absolute	Already located by the assembler
B	Blank	Section must be initialized to '0' (cleared)
F	Not filled	Section is not filled or cleared (scratch)
I	Initialize	Section must be initialized in rom
N	Now	Section is located before normal sections (without N or P)
P	Postponed	Section is located after normal sections (without N or P)

Table F-1: Section Attributes

To specify a (group) of sections, DELFEE has the following syntax:

- 1. select a group on section attribute:

**section selection = *attr*;**

- 2. select a section by name:

**section *name*;**

- 3. select a special section:

**heap;**            //locate heap here  
**stack;**          //locate stack here  
**table;**          //locate copy table here  
**copy;**           //locate all initial data here  
**copy *name*;** //locate initial data of the named section here

- 4. create a section:

**reserved label=*name* length=*number*;**

Instead of selecting a section by an attribute, DELFEE also allows excluding a section by its attribute.

Excluding an attribute is done by placing a '-' (minus sign) in front of *attr*.

So, the example:

```
section selection=attr1-attr2
```

selects a group of sections with attribute *attr1* and without attribute *attr2*.

## 4.7 CLUSTER DEFINITION

Clusters are used to place specified sections in a group. The locator will handle the clusters in the order that they are specified. This gives you the possibility to create a group of selected sections and give it a higher locate priority.

There are several possibilities to specify that a section is part of a cluster. The exact rules and their priorities are given in the paragraph *Section Placing Algorithm*. The three main possibilities are:

1. attribute
2. section selection=
3. amode definition

Examine the following example:

```
layout {
    space address_space {
        block rom {
            cluster first_code_clstr {
                attribute i;
                amode near_code;
                amode far_code;
            }
            cluster code_clstr {
                attribute r;
                amode near_code {
                    section selection=x;
                    section selection=r;
                }
                amode far_code {
                    table;
                    section selection=x;
                    section selection=r;
                }
            }
        }
    }
}
```



For example, suppose you want to locate all writable sections first, then the heap, followed by the stack. In the DELFEE language this is specified by:

```

:
section selection = w;    // 'w' means writable sections
heap;
stack;
:

```

## 4.9 MANIPULATING SECTIONS IN AMODES

The previous paragraphs explained how to set the order of the sections within an **amode** definition. DELFEE recognizes an extra set of keywords to further tune the locating of code and data sections.

An **amode** definition can contain the following keywords:

Keyword	Description
section	Selects a section, or group of sections
selection	Specifies attributes for grouping sections
attribute	Assigns attributes (are past to the cluster
copy	Selects a rom copy of a section by name, or all rom copies in general
fixed	Forces a section to be located around a fixed address
gap	Creates a gap in the address range where sections will not be located
reserved	Reserves a memory area, which can be referenced using locator labels
heap	Defines the place and attributes of the heap
stack	Defines the place and attributes of the stack
table	Defines the place and attributes of the copy table
assert	A user defined assertion
length	Specifies the length of stack, heap, physical block or reserved space

Table F-2: *amode* keywords

All keywords are described in section 7, *Delfee Keyword Reference*.

## **4.10 SECTION PLACING ALGORITHM**

There are different ways to reference a section. Sections can be referenced as a group based on a certain attribute, or they can be referenced very specific by name. To find out where sections are placed in the layout part, DELFEE uses the following algorithm:

1. First, try to find a selection by section name.
2. If not found, search for a 'section selection=' within a matching amode block.
3. If not found, search for a 'section selection=' not within an amode block.
4. If not found, search for a cluster with a correct 'amode= ...,..., ;' and correct attributes.
5. If not found, search for a cluster with correct attributes.
6. If not found, relax attribute checking, and start over again.

Relax attributes using the following rules:

1. If stack, heap or reserved, switch indication off and try again.
2. If attribute 'f' (not filled), switch 'f' off and try again.
3. If attribute 'b' (clear), switch 'b' off and try again.
4. If attribute 'i' (initialize), switch 'i' off and try again.
5. If attribute 'x' (executable code), switch 'x' off and 'r' (read-only) on and try again. (Try to place executable sections in read-only memory).
6. If attribute 'r' (read-only), switch 'r' off 'w' (writable) on and try again. (Try to place read-only sections in writable memory).

## **5 MEMORY PART**

### **5.1 INTRODUCTION**

The memory part defines the variable part of the memory configuration. It can be placed in a different file, which allows to easily switch between different memory configurations. The syntax used for the mappings is the same as used in the cpu part.

As you have seen in the example of the cpu part in section 3, there were two references to external busses:

```
bus address_bus {
    mau 8;
    mem addr=0 chips=rom_chip;
    map src=0x100 size=0x7f00 dst=0x100 bus = external_rom_bus;
    mem addr=32k chips=ram_chip;
    map src=0x8100 size=0x7f00 dst=0x100 bus = external_ram_bus;
}
```

In the memory part you have to define the description for the busses `external_rom_bus` and `external_ram_bus`. Using the description in sections 3.5 and 3.6 for specifying busses and chips, the memory part could look like:

```
memory {
    bus external_rom_bus {
        mau 8;
        mem addr=0 chips=xrom;
    }

    chips xrom attr=r mau =8 size=0x8000;

    bus external_ram_bus {
        mau 8;
        mem addr=0 chips=xram;
    }

    chips xram attr=w mau=8 size=0x8000;
}
```



## **6 DELFEE PREPROCESSING**

### **6.1 INTRODUCTION**

You can preprocess a DELFEE description file using exactly the same syntax as used by the C preprocessor. This means that all preprocessor directives start with a '#'-sign.

The preprocessor scans the input (description) file looking for macro calls. A macro-call is a request to the preprocessor to replace the call pattern of a built-in or user-defined macro with its definition.

There are two types of macro definitions: 'plain' macros and 'function-like' macros. A plain macro is expanded to a fixed string of characters. A function-like macro looks like a function call. The macro is expanded to its definition, in which the macro parameters are replaced by their corresponding macro arguments.

### **6.2 USER DEFINED MACROS**

You can create macros with the **#define** preprocessor directive.

#### ***Syntax:***

**#define** *macro-name*[(*formal-parameter-list*)] *macro-body*

When you create a parameterless macro, there are two parts to a **#define** call: the *macro-name* and the *macro-body*. The *macro-name* defines the name used when the macro is called; the *macro-body* defines the return value of the call.

The *macro-body* is usually the return value of the macro call. However, the macro-body may contain calls to other macros. If so, the return value is actually the fully expanded macro-body, including the return values of the call to other macros.

#### ***Example:***

```
#define ASIZE 10
```

Every occurrence of **ASIZE** is expanded to '10'.

If the only function of the macro processor was to perform simple string replacement, then it would not be very useful for the most programming tasks. Each time you want to change even the simplest part of the macro's return value you would have to redefine the macro. Parameters in macro calls allow more general-purpose macros. Parameters leave holes in a macro-body that are filled in when you call the macro. This permits you to design a single macro that produces code for typical operations. The term 'parameters' refers to both the formal parameters that are specified when the macro is defined (the holes), and the actual parameters or argument that are specified when the macro is called (the fill-ins). To define macros with parameters you have to add a *formal-parameter-list*. The *formal-parameter-list* is a list of macro identifiers separated by ','. These identifiers comprise the formal parameters used in the macro. The macro identifier for each parameter in the list must be unique.

**Example:**

After

```
#define ADD(a, b) a + b
```

the call `ADD(4, 5)` is expanded to `4 + 5`.

You can undefine a preprocessor macro with the **#undef** preprocessor directive:

```
#undef macro-name
```

### **6.3 FILE INCLUSION**

With the **#include** preprocessor directive:

```
#include <include-file>
```

you can include text from *include-file* within the input text of the description file. At the occurrence of an **#include** control line, the preprocessor reads the text from *include-file* until end-of-file is reached. **#include** files may be nested. *include-file* is any file that contains description file information. *include-file* is searched for in the directory etc directory relative to the installation path of your product.

The ANSI standard defines the following terms for the include directive:

```
#include "include-file"
#include <include-file>
#include token-sequence
```

The preprocessor uses the following search rules for include files between " ":

1. search in the directory of the description file
2. search in the directory etc relative to the installation path of your product

Note that if you nest include files, the preprocessor applies the first rule for each level.

**Example:**

```
product.dsc:
    #include "../inc/product.cpu"

product.cpu:
    #include "prod2.cpu"
```

According to rule 1 the preprocessor searches `prod2.cpu` in the same directory as `product.cpu` since `prod2.cpu` is included by `product.cpu` and not by `product.dsc`.

The preprocessor searches for include files between `< >` in the same way as for include files between " ". The difference is that rule 1 does not apply (the directory of the source description file is not searched).

The third form of include directives:

```
#include token-sequence
```

means that the included file name may be a token sequence that has been defined before. After expansion by the preprocessor this should produce a valid include directive as described by the first two forms:

**Example:**

```
#ifdef STD
#define cpu_incl <product.cpu>
#else
#define cpu_incl "my_cpu.cpu"
#endif

#include cpu_incl
```

## 6.4 **CONDITIONAL STATEMENTS**

Some preprocessor directives expect logical *expressions* in their arguments. Logical *expressions* follow the same rules as numeric *expressions*. The difference is in how preprocessor interprets the value that the *expression* represents. Once the *expression* has been evaluated to a value, the preprocessor uses the '=' 0' comparison to determine whether the expression is TRUE or FALSE (if the value is equal 0 the expression is FALSE else TRUE).

The **#if** and **#elif** preprocessor directives evaluate a logical *expression*, and based on that *expression*, expand or withhold their *statements*. The **#ifdef** and **#ifndef** preprocessor directives evaluates the existence of a user-defined macro, and based on the result, expand or withhold their *statements*.

**Syntax:**

```
if-line
    statements
[#elif expression
    statements]...
[#else
    statements]
#endif
```

where *if-line* is one of:

```
#if expression
#ifdef macro-name
#ifndef macro-name
```

The *expression* in the **#if** directive and subsequent **#elif** directives are evaluated in order until a TRUE value is encountered. If the value is TRUE, then the preprocessor expands the succeeding *statements*; if the value is FALSE and the optional **#else** directive is included in the call, then the *statements* succeeding **#else** are expanded. If the *expression* results to FALSE and the **#else** is not included, the **#if** call returns the null string.

The **#ifdef** tests if the *macro-name* is a previously defined macro. The **#ifndef** evaluates its *statements* if the *macro-name* is not currently defined.

Each **#if**, **#ifdef** and **#ifndef** directive must have a corresponding **#endif**.

**Example:**

```
#define _STCK 100
#if _STCK == 100
    stack length=100;
#else
    stack length=200;
#endif
```

This example always expands to: `stack length=100;`. In this case the **#if** control line could also be written as:

```
#ifdef _STCK
```

## **7 DELFEE KEYWORD REFERENCE**

This section contains an alphabetical description of all keywords that can be used in a description file. Some keywords can be abbreviated to a minimum of four characters.

# **.addr**

## **Syntax:**

**.addr**

(Software part)

## **Description:**

The predefined label **.addr** contains the current address.

## **Example:**

```
block ram {
  cluster data_clstr {
    attribute w;
    amode near_data {
      section selection=w;
      assert ( .addr < 256, "page overflow");
      // if the condition is false,
      // the locator generates an error with
      // the text as message
    }
    ...
  }
}
```

# address

## Syntax:

**address** = *address*

(all parts)

**addr** = *address*

(abbreviated form)

## Description:

Specify an absolute address in memory.

## Example:

Cpu or memory part:

```
bus address_bus {
    mau 8;
    mem addr=0    chips=rom_chip;
    ...
    mem addr=32k chips=ram_chip;
    ...
}
```

Software part:

```
block rom {
    ...
    cluster code_clstr {
        attribute r;
        amode near_code {
            section selection=x;
            section selection=r;
            section .string address = 0x0100;
        }
        ...
    }
}
```



The locate order in the **amode** definition in the example above is fixed. Sections with attribute selection 'x' and/or 'r' are forced to be located before section `.string`. If this fixed order is not desired, the absolute address specification can be done in a separate **amode** definition.

**Example:**

```
amode near_code {  
    section .string address = 0x0100;  
}  
  
amode near_code {  
    section selection=x;  
    section selection=r;  
}
```





From software part:

```
block ram {
  cluster data_clstr {
    attribute w;
    amode near_data {
      // Sections with addressing mode
      // near_data are located here
      section selection=w;
    }
    amode far_data {
      // Sections with addressing mode
      // far_data and the stack and heap
      // are located here
      section selection=w;
      heap;
      stack;
    }
  }
}
```

# assert

## Syntax:

**assert** ( *condition* , *text* ) ; (Software part)  
**asse** ( *condition* , *text* ) ; (abbreviated form)

## Description:

Test condition of virtual address in memory. Generate an error if the assertion fails and give a message with '*text*'. *condition* is specified as one of:

*expr1* > *expr2*  
*expr1* < *expr2*  
*expr1* == *expr2*  
*expr1* != *expr2*

*expr1* and *expr2* can be any expression or label. The predefined label **.addr** contains the current address.

## Example:

```
block ram {
  cluster data_clstr {
    attribute w;
    amode near_data {
      section selection=w;
      assert ( .addr < 256, "page overflow");
      // if the condition is false,
      // the locator generates an error with
      // the text as message
    }
    ...
  }
}
```

# attribute

## Syntax:

<b>attribute</b> <i>attribute_string</i> ;	(Software part)
<b>attr</b> <i>attribute_string</i> ;	(abbreviated form)
<b>attribute</b> = <i>attribute_string</i>	(Software part)
<b>attr</b> = <i>attribute_string</i>	(abbreviated form)

## Description:

With **attribute** you can assign attributes to sections, clusters or memory blocks. See also the keyword **selection**.

For sections these attributes are pure supplementary to the standard section attributes. The standard section attributes such as zero page (Y1), blank (B) and executable (X) are set by the compiler (or by the assembler in the case of an assembler program).

With an action attribute after a section (**attr=**), you can set section attributes or you can disable section attributes with the - (minus) sign.

The attributes have the following meaning:

<i>num</i>	(Section only) Align the section at $2^{num}$ MAUs.
<b>Ynum</b>	(amode and sections only) Identify addressing mode. Indicate that sections with this attribute should be allocated in this cluster.
<b>r</b>	(Memory and clusters) Indicate this is a read-only cluster or read-only memory.
<b>w</b>	(Memory and clusters) Indicate this is a writable cluster or writable memory.
<b>s</b>	(Memory only) Indicate this is special memory, it must not be located.
<b>x</b>	(Clusters/sections only) Indicate that the cluster/section is executable.
<b>g</b>	(Clusters/sections only) Indicate that the cluster/section is global (known in a multi-module environment).

- b** (Clusters/sections only) Indicate that clusters/sections should be cleared before locating.
- i** (Sections only) Indicate that clusters/sections should be copied from ROM to RAM.
- f** (Clusters/sections only) Indicate that clusters/sections should not be filled and not cleared. This is called a scratch cluster/section.

Default attributes if the attribute keyword is omitted:

- sections: The attributes as generated from the assembler/compiler.
- clusters: The attributes as indicated by the underlying memory, thus **r** for rom and **w** for ram.
- memory: If no attributes defined, the default is writable (**w**).

### Example:

From software part:

```
layout {
    space address_space {
        block rom {
            cluster first_code_clstr {
                attribute i; // set cluster attribute
                amode near_code;
                amode far_code;
            }
        }
    }
}
```

```

block ram
  cluster ram {
    amode near_data {
      // Default attribute of cluster
      // data is 'w', because the
      // memory is RAM.

      section selection=w;
      section selection=b attr=-b;
      // Sections with attribute b are
      // are located here, and
      // attribute 'b' is switched off
    }
    .
  }
  .
}

```

From cpu part:

```

amode near_data {
  attribute Y3; //identify code with Y3
  mau 8;
  map src=0 size=1k dst=0 amode = far_data;
}
...

chips rom_chip attr=r mau=8 size=0x100;
chips ram_chip attr=w mau=8 size=0x100;
...
// memory attributes

```

# block

## Syntax:

**block** *identifier* { *block\_description* } (Software part)

## Description:

With **block** you define the contents of a physical area of memory. You can make a **block** description for each chip you use. Each block has a symbolic name as previously defined by the keyword **chips**. It is allowed to combine two or more memory chips in one block as long as their total address range is linear, without gaps. The identifier indicates that a memory block starts at the specified chip, no matter how many chips are combined.

## Example:

```
layout {
  space address_space {
    block ram
      // Memory block starting at chip ram_chip
      cluster ram {
        ...
      }
  }
}
```

# bus

## Syntax:

**bus** *identifier* [, *identifier*] ... { *bus\_description* } (Cpu or memory part)  
**bus** = *identifier* (def)  
 (ref)

## Description:

With **bus** you define the physical memory addresses for the chips that are located on the cpu (definition). When you specify **bus=**, you map a specific address range on a previously defined address bus (reference). The only keywords allowed in an **bus** description are **mem**, **map** and **mau**.

## Example:

```
cpu {
  space address_space {
    // Specify space 'address_space' for the address_bus
    // address bus.
    mau 8;
    map src=0    size=32k dst=0    bus = address_bus label = rom;
    map src=32k  size=32k dst=32k  bus = address_bus label = ram;
                                     // ref
  }

  bus address_bus { // definition
    mau 8;
    mem addr=0    chips=rom_chip;
    map src=0x100 size=0x7f00 dst=0x100 bus = external_rom_bus;
    mem addr=32k  chips=ram_chip;
    map src=0x8100 size=0x7f00 dst=0x100 bus = external_ram_bus;
  }
  ...
}
```



# chips

## Syntax:

`chips identifier[, identifier]... chips_description` (Cpu or memory part)  
`chips = identifier[| identifier]... [, identifier[| identifier]...]...` (def)  
(ref)

## Description:

With **chips** you describe the chips on the cpu or on your target board (definition). For each chip its **size** and minimum addressable unit (**mau**) is specified. With the keyword **attr** you can define if the memory is read-only. The only three attributes allowed are **r** for read-only, **w** for writable, or **s** for special. If omitted, w is default.

You can use **chips=** after the keyword **mem** to specify where a chip is located (reference). You can create chip pairs by separating each chip with a vertical bar '|'.  
 ' | '.

## Example:

```

cpu {
    bus address_bus {
        mau 8;
        mem addr=0 chips=rom_chip; // ref
        ...
    }
    chips rom_chip attr=r mau=8 size=0x100; // def
    chips ram_chip attr=w mau=8 size=0x100;
    ...
}

```

# cluster

## Syntax:

(Software part)

```
cluster cluster_name { cluster_description }
cluster cluster_name [, cluster_name] ... ;
```

## Description:

In the software layout part you can define the cluster name and cluster location order. The attributes as valid for clusters (see **attribute**) can be specified in the first syntax. If you do not specify any attribute, the default attribute **r** or **w** is automatically set.

In a cluster description you can not only determine the locate order of sections within the named cluster, but you can also specify stack and heap size, extra process memory, define labels for the process, etc.

## Example:

```
space address_space {
    block rom {
        cluster first_code_clstr {
            // The default attribute 'r' of cluster
            // text is overruled to 'i'. All
            // sections with attribute 'i' are
            // located here by default.
            attribute i;
            amode near_code;
            amode far_code;
            // Sections with addressing mode
            // near_code or far_cdoe are
            // located here
        }

        block ram {
            cluster data_clstr {
                // default attribute 'w' because the
                // memory is RAM. All writable
                // sections are located here by default.
                attribute w; // can be omitted
                amode near_data {
                    section selection=w;
                }
            }
        }
    }
}
```

# copy

## Syntax:

```
copy section_name [ attr = attribute ] ;           (Software part)
copy selection = attribute [ attr = attribute ] ;
copy ;
```

## Description:

The ROM copy of data sections with the attribute **i** will be copied from ROM to RAM at program startup. With **copy** you define the placement in memory of these ROM copies. You can specify a specific section by giving the section's name, or select sections with a specific attribute. If you do not specify an argument, the locator locates all ROM copies at the specified location. With **attr=** you can change the section attributes.

If you do not specify the keyword **copy** at all, the locator finds a suitable place for ROM copies.

See also the keywords **attribute** and **selection**.

## Example:

```
space address_space {
    block rom {
        ...
        cluster code_clstr {
            attribute r; //cluster attribute
            amode far_code {
                table;
                section selection=x;
                section selection=r;
                copy; // all ROM copies are located here
            }
        }
    }
}
```

# cpu

## Syntax:

```
cpu { cpu_description }           (Cpu part)
cpu filename
```

## Description:

The keyword **cpu** appears together with **software** and **memory** at the highest level in a description file. The actual cpu description starts between the curly braces {}. Normally you do not need to change the cpu part because it is delivered with the product and describes the derivative completely.

The second syntax is the so-called include syntax. The locator opens the file *filename* and reads the actual cpu description from this file. You must start the included file with **cpu** again. The *filename* can contain a complete path including a drive letter (Windows). Parts of *filename*, or the complete *filename* can be put in a environment variable. The file is first searched for in the current directory, and secondly in the etc directory relative to the installation directory.

## Example:

Contents of the description file:

```
software {
    ...
}

cpu target.cpu      //cpu part in separate file
memory target.mem
```

See section 3 for a sample contents of a .cpu file.

# dst

## Syntax:

**dst** = *address* (Cpu or memory part)

## Description:

Specify destination address as part of the keyword **map** in an **amode**, **space** or **bus** description. For *address* you can use any decimal, hexadecimal or octal number. You can also use the (standard) Delfee suffix **k**, for kilo ( $2^{10}$ ) or **M**, for mega ( $2^{20}$ ). The unit of measure depends on the MAU (minimum addressable unit) of the destination memory space.

## Example:

```
cpu {
  ...
  amode near_code {
    attribute Y1;
    mau 8; // 8-bit addressable
    map src=0 size=1k dst=0 amode=far_code;
  }
}
```

# fixed

## Syntax:

**fixed address** = *address* ;

(Software part)

**fixed addr** = *address* ;

(abbreviated form)

## Description:

Define a fixed point in the memory map. The locator allocates the section/cluster preceding the fixed definition and the section/cluster following it as close as possible to the fixed point.

## Example:

```
block ram {
    cluster near_data_clstr {
        amode near_data {
            section selection=w;
            fixed addr = 0x2000;
        }
    }
    cluster far_data_clstr;
}
```

Cluster `far_data_clstr` will be located with its upper bound at address `0x2000` and cluster `near_data_clstr` starts at this address. The same can be applied to sections.

# gap

## Syntax:

**gap;** (Software part)  
**gap length = *value* ;**

## Description

Reserve a gap with a dynamic size. The locator tries to make the memory space as big as possible. You can use this keyword in a block description to create a gap between clusters, or in a cluster description to create a gap between sections. You can also use the **gap** keyword in combination with the **fixed** keyword.

With the second form you can specify a gap of a fixed length. This form can only occur in a block description.

## Example:

```
space address_space {
    block ram {
        cluster data_clstr {
            attr w;
            amode near_data;
        } // low side mapping

        gap; // balloon
        cluster stck; // high side mapping
    }
}
```

# heap

## Syntax:

**heap** *heap\_description* ; (Software part)  
**heap** ;

## Description:

Like **table** and **stack**, **heap** is another special section. The section is not created from the .out file, but generated at locate time. To control the size of this special section the keyword **length** is allowed within the heap description. You can use **heap** to include dynamic memory for a process.

Heap can only be used if a malloc() function has been implemented.

Two locator labels are used to mark begin and end of the heap, `_lc_bh` for the begin of heap, and `_lc_eh` for the end of heap.

Note that if the **heap** keyword is specified in the description file this does not automatically mean that a heap will always be generated. A heap will only be allocated when its section labels (`_lc_bh` for begin of heap and `_lc_eh` for end of heap) are used in the program.

The heap description can be a length specification and/or an attribute specification. See the example.

## Example:

```
layout {
    space address_space {
        block ram {
            cluster data_clstr {
                amode far_data {
                    section selection=w;
                    heap length=100;
                    // Heap of 100 MAUs
                }
            }
        }
    }
}
```



# label

## Syntax:

**label** *identifier* ; (Software part)  
**label** = *identifier* ; (All parts)

## Description:

The first form can be used stand-alone to specify a virtual address in memory by means of a label. The virtual address is label **\_lc\_u\_identifier**. Note that at C level, all locator labels start with one underscore (the compiler adds another underscore '\_').

The second form can only be used as part of another keyword. As part of the keyword **reserved** you can assign a label to an address range. The start of the address range is identified by label **\_lc\_ub\_identifier**. The end of the address range is identified by label **\_lc\_ue\_identifier**. The keyword **label** is also allowed as part of the **map** keyword to assign a name to a block of memory in a space definition.

## Example:

From the software part:

```
block ram {
    cluster data_clstr {
        attribute w;
        amode far_data {
            section selection=w;
            heap;
            stack;
            reserved label=xvwbuffer length=0x10;

            // Start address of reserved area is
            // label _lc_ub_xvwbuffer
            // End address of reserved area is
            // label _lc_ue_xvwbuffer

        }
    }
}
```

From the cpu part:

```
space address_space {  
    mau 8;  
    map src=0    size=32k dst=0    bus = address_bus label=rom;  
    map src=32k  size=32k dst=32k  bus = address_bus label=ram;  
}
```

# layout

## Syntax:

**layout** { *layout\_description* } (Software part)  
**layout** *filename*

## Description:

The **layout** part describes the layout of sections in memory. The **layout** part groups sections into clusters and you can define the name, number and the order of clusters. The **layout** part describes how these clusters must be allocated into physical RAM and ROM block. The space and block names used in the **layout** part must be present in the memory part or the cpu part. The cluster definitions can contain fixed addresses as well as definitions of gaps between sections.

## Example:

```
software {
  layout {
    space address_space {
      block rom {
        cluster first_code_clstr {
          attribute i;
          amode near_code;
        }
      }
    }
  }
  ....
}
```

# length

## Syntax:

**length** = *length*

(Cpu, memory and software part)

**leng** = *length*

(abbreviated form)

## Description:

You can use the keyword **length** to define the length in MAUs (minimum addressable units) of a certain memory area. *length* must be a numeric value and can be given either in hex, octal or decimal. As usual, hex numbers must start with '0x' and octal numbers must start with '0'. You can use the suffix **k** which stands for kilo or **M** which stands for mega.

You can use **length** to specify the length of the reserved memory or to specify the stack, heap or gap length. For details see the keywords **reserved**, **stack**, **heap** and **gap**.

## Example:

```
space address_space {
  block ram {
    cluster data_clstr {
      amode far_data {
        stack leng = 2k;
      }
    }
  }
}
```

# load\_mod

## Syntax:

**load\_mod** *identifier* **start** = *label*; (Software part)  
**load\_mod** **start** = *label*;

## Description:

With **load\_mod** you are introducing a load module description. This keyword is followed by an optional identifier, representing a load module name with or without the .out extension. The load module itself must be supplied to the locator as a parameter in the invocation. If the identifier is omitted, the load module is taken from the command line.

## Example:

```
software {  
    load_mod start = __START;  
}
```

or:

```
software {  
    load_mod hello start = __USER_start;  
}
```

# map

## Syntax:

**map** *map\_description*

(Cpu or memory part)

## Description:

Map a memory part, specified as a source address and a size, to a destination address of an **amode**, **space** or **bus**. The unit of measure depends on the MAU of the memory space.

## Example:

```
cpu {
  .
  amode far_data {
    attribute Y4;
    mau 8;
    map src=0 size=32k dst=32k space=address_space;
  }
  space address_space {
    mau 8;
    map src=0 size=32k dst=0 bus = address_bus label=rom;
    map src=32k size=32k dst=32k bus = address_bus label=ram;
  }
  bus address_bus {
    mau 8;
    mem addr=0 chips=rom_chip;
    map src=0x100 size=0x7f00 dst=0x100 bus=external_rom_bus;
    mem addr=32k chips=ram_chip;
    map src=0x8100 size=0x7f00 dst=0x100 bus=external_ram_bus;
  }
  .
}
```

# mau

## Syntax:

**mau** *number* ; (Cpu or memory part)  
**mau** = *number*

## Description:

You can use the keyword **mau** to specify the minimum addressable unit in bits of a certain memory area. The first form can only be used in an **amode**, **space** or **bus** description. The second form can be used to specify the minimum addressable unit of a chip. Note that **mau** affects the unit of measure for other keywords. If no **mau** is specified, the default number is 8 (byte addressable).

## Example:

```
cpu {  
    amode near_code {  
        attribute Y1;  
        mau 8; // byte addressable  
        map src=0 size=1k dst=0 amode=far_code;  
        // src is at address 0,  
        // size is 1k byte units  
        // dst is at address 0  
    }  
}
```

# mem

## Syntax:

**mem** *mem\_description* ; (Cpu or memory part)

## Description:

Define the start address of a chip in memory. The only keywords allowed in a mem description are **address** and **chips**.

## Example:

```
cpu {
  ...
  bus internal_bus {
    mau 8;
    mem addr=0    chips=rom_chip;
    // chip 'rom_chip' is located at memory
    // address 0
    ...
    mem addr=32k  chips=ram_chip;
    // chip 'ram_chip' is located at memory
    // address 0x8000
    ...
  }
  chips rom_chip attr=r mau=8 size=0x100;
  chips ram_chip  attr=w mau=8 size=0x100;
}
```



# memory

## Syntax:

```
memory { memory_description }           (Memory part)
memory filename
```

## Description:

Together with **software** and **cpu**, **memory** introduces a main part of the description file. You can specify the actual memory part between the curly braces {}.

You can use the memory part to describe any additional memory or addresses of peripherals not integrated on the cpu.

The second syntax is the include syntax. In this case, the memory part is defined in a separate file. This included file must start again with **memory**. The *filename* can contain a complete path, including a drive letter (Windows). You can put parts of *filename*, or the complete *filename* in an environment variable. The file is first searched for in the current directory, and secondly in the `etc` directory relative to the installation directory.

## Example:

```
software {
    ...
}

cpu target.cpu
memory target.mem           //mem part in separate file
```



See section 5 for a sample contents of a `.mem` file.

# regsfr

## Syntax:

**regsfr** *filename* (Cpu or memory part)

## Description:

Specify a register file generated by the register manager for use by the CrossView debugger.

## Example:

```
cpu {  
    .  
    .  
    .  
    regsfr regfile.dat  
    /*  
    * Use file regfile.dat generated by  
    * register manager for CrossView  
    */  
}
```

# reserved

## Syntax:

**reserved** *reserved\_description* ; (Software part)  
**reserved**;

## Description:

Reserve a fixed amount of memory space or reserve as much memory as possible in the memory space. If no length is specified the size of the memory allocation depends on the size of the memory space or the size is limited by a fixed point definition following the **reserved** allocation.

You can only use the keywords **address**, **attribute**, **label** and **length** in the reserved description. You can use the keyword **reserved** in an amode description.

## Example:

```
space address_space {
    block rom {
        cluster code_clstr {
            amode near_code {
                // system reserved
                // (exception vector)
                reserved length=0x2 addr=0x24;
            }
        }
    }
}
```

# section

## Syntax:

(Software part)

```
section identifier [addr = address ] [attr = attribute ] ;
section selection = attribute [addr = address] [attr = attribute];
```

## Description:

**section** can be used in the layout part to specify the location order within a cluster. See also **layout**.

The *identifier* is the name of a section.

With **addr=** you can make a section absolute.

With **attr=** you can assign new attributes to a section or disable attributes.

See also the keywords **address**, **attribute** and **selection**.

## Example:

```
space address_space {
    block ram {
        cluster data_clstr {
            amode near_data {
                // locate section .data here and set
                // attribute 'w'
                section .data attr=w;
                section selection=b attr=-b;
            }
        }
    }
}
```

# selection

## Syntax:

**selection** = *attribute*

## Description:

You can use **selection** after the keywords **section** or **copy** to select all sections with (a) specified attribute(s).

If more attributes are specified, only sections with all attributes are selected. If a minus sign '-' precedes the attribute, only sections **not** having the attribute are selected.

See also the keywords **attribute**, **copy** and **section**.

## Example:

```
space address_space {
    block ram {
        cluster data_clstr {
            amode near_data {
                // select sections with w on and not i.
                // (select all writable sections which
                // are not copied from ROM)
                section selection=-iw;
            }
        }
    }
}
...
```

# size

## Syntax:

**size** = *size*

(Cpu or memory part)

## Description:

You can use the keyword **size** to define the size in minimum addressable units (MAU) of a certain memory area. *size* must be a numeric value and can be given either in hex, octal or decimal. As usual, hex numbers must start with '0x' and octal numbers must start with '0'. You can use the suffix **k** which stands for kilo or **M** which stands for mega.

You can use **size** to specify the size of a part of memory that must be mapped on another part of memory or to specify the size of a chip. For details see the keywords **map** and **chips**.

## Example:

```
cpu {
  amode near_code {
    attribute Y1; //identify near_code with Y1
    map src=0 size=1k dst=0 amode=far_code;
  }
  space address_space {
    mau 8;
    map src=0 size=32k dst=0 bus=address_bus label=rom;
    map src=32k size=32k dst=32k bus=address_bus label=ram;
  }
  chips rom_chip attr=r mau=8 size=0x100;
  chips ram_chip attr=w mau=8 size=0x100;
  // size of chips
}
```

# software

## Syntax:

**software** { *software\_description* } (Software part)  
**software** *filename*

## Description:

The keyword **software** appears at the highest level in a description file. The actual software description starts between the curly braces {}.

The second syntax is the so called include syntax. The locator will open file *filename* and read the actual software description from this file. The first keyword in *filename* must be **software** again. The *filename* can contain a complete path including a drive letter (Windows). You can put parts of *filename*, or the complete *filename* in an environment variable. The file is first searched for in the current directory, and secondly in the etc directory relative to the installation directory.

## Example:

Contents of the description file:

```
software $(MY_OWN_DESCRIPTION)

cpu target.cpu
memory target.mem
```

Environment variable MY\_OWN\_DESCRIPTION contains the name of a file with contents like:

```
software {
    load_mod start = __START;
    layout {
        .
        .
        .
    }
}
```

# space

## Syntax:

**space** *identifier* { *space\_description* } (Software part)  
 (Cpu or memory part)  
**space** *identifier* [, *identifier*] ... { *space\_description* }  
**space** = *identifier*

## Description:

The keyword **space** can be used in the cpu part, memory part and software part. In the cpu or memory part you can use **space** to describe a physical memory address space. The only keywords allowed in a space description in the cpu or memory part are **mau** and **map**.

In the software part you can use **space** to describe one or more memory blocks. Each space has a symbolic name as previously defined by the keyword **space** in the cpu or memory part.

## Example:

From the cpu part:

```
cpu {
  amode far_data {
    attribute Y4;
    mau 8;
    map src=0 size=32k dst=32k space=address_space;
  }
  ...

  space address_space {
    // Specify space 'address_space' for the
    // address_bus address bus.
    mau 8;
    map src=0 size=32k dst=0 bus=address_bus label=rom;
    map src=32k size=32k dst=32k bus=address_bus label=ram;
  }
  .
}
```



From the software part:

```
layout {  
    // define the preferred locating order of sections  
    // in the memory space  
    // (the range is defined in the .cpu file)  
    space address_space {  
        ...  
  
        // define for each sub-area in the space  
        // the locating order of sections  
        block rom {  
            // Memory block starting at chip rom_chip  
  
            // define a cluster for read-only sections  
            cluster code_clstr {  
                ....  
            }  
        }  
        .  
    }  
}
```

# src

## Syntax:

**src** = *address*

(Cpu or memory part)

## Description:

Specify source address as part of the keyword **map** in an **amode**, **space** or **bus** description. For *address* you can use any decimal, hexadecimal or octal number. You can also use the (standard) Delfee suffix **k**, for kilo ( $2^{10}$ ) or **M**, for mega ( $2^{20}$ ). The address is specified in the addressing mode's local MAU (minimum addressable unit) size (default 8 bits).

## Example:

```
cpu {  
  ...  
  amode near_code {  
    attribute Y1;  
    mau 8; // 8-bit addressable  
    map src=0 size=1k dst=0 amode=far_code;  
  }  
}
```

# stack

## Syntax:

**stack** *stack\_description* ; (Software part)  
**stack** ;

## Description:

**stack** is a special form of a section description. The stack is allocated at locate time. The locator only allocates a stack if one is needed. Two special locator labels are associated with the stack space located with keyword **stack**. The begin of the stack area can be obtained by the locator label `_lc_bs`, the end address is accessible by means of label `_lc_es`.

If the stack grows downwards the begin of stack must be the highest address. To accomplish this, you can keep the length positive and set the stack pointer to `end_of_stack`, so the formula:

$$end\_of\_stack = begin\_of\_stack + length$$

is always true.

You can only use the keywords **attribute** and **length** in the stack description. If you specify **stack** without a description, the locator tries to make the stack as big as possible. If you do not specify the keyword **stack** at all, the locator also tries to make the stack as big as possible but at least 100 (MAUs).

## Example:

```
space address_space {
    block ram {
        cluster data_clstr {
            amode far_data {
                section selection=w;
                stack leng=150;
                // stack of 150 MAUs
                ...
            }
        }
    }
}
```

# start

## Syntax:

**start** = *label* ; (Software part)

## Description:

Define a start label for a process.

You can use **start** only within a load module description.

## Example:

```
software {  
    load_mod start = system_start;  
  
    layout {  
        .  
        .  
    }  
}
```

# table

## Syntax:

**table** attr = *attribute* ;  
**table** ; (Software part)

## Description:

Like **stack** and **heap** also **table** is a special kind of section. Normal sections are generated at compile time, and passed via the assembler and linker to the locator. The stack and heap sections are generated at locate time, with a user requested size.

**table** is different. The locator is able to generate a copy table. Normally, this table is put in read-only memory. If you want to steer the table location, you can use the **table** keyword. With table only **attribute** is allowed. The length is calculated at locate time. **table** can occur in a cluster description.

## Example:

```
space address_space {
    block rom {
        ...
        cluster code_clstr {
            attribute r; // cluster attribute
            amode far_code {
                table; // locate copy table here
                section selection=x;
                section selection=r;
                copy; // all ROM copies are located here
            }
        }
    }
}
```

## 7.1 ABBREVIATION OF DELFEE KEYWORDS

The following Delfee keywords can be abbreviated to unique 4 character words:

Keyword	Abbreviation
address	addr
assert	asse
attribute	attr
length	leng

*Table F-3: Abbreviation of Delfee keywords*

## 7.2 DELFEE KEYWORDS SUMMARY

Keyword	Description
address	Specify absolute memory address
amode	Specify the addressing modes
assert	Error if assertion failed
attribute	Assign attributes to clusters, sections, stack or heap
block	Define physical memory area
bus	Specify address bus
chips	Specify cpu chips
cluster	Specify the order and placement of clusters
copy	Define placement of ROM—copies of data sections
cpu	Define cpu part
dst	Destination address
fixed	Define fixed point in memory map
gap	Reserve dynamic memory gap
heap	Define heap
label	Define virtual address label
layout	Start of the layout description
length	Length of stack, heap, physical block or reserved space
load_mod	Define load module (process)

Keyword	Description
map	Map a source address on a destination address
mau	Define minimum addressable unit (in bits)
mem	Define physical start address of a chip
memory	Define memory part
regsfr	Specify register file for use by CrossView
reserved	Reserve memory
section	Define how a section must be located
selection	Specify attributes for grouping sections into clusters
size	Size of address space or memory
software	Define the software part
space	Define an addressing space or specify memory blocks
src	Source address
stack	Define a stack section
start	Give an alternative start label
table	Define a table section

Table F-4: Overview of Delfee keywords

# APPENDIX

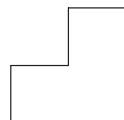
# G

## **DELSEE SYNTAX**

---



**TASKING**





---

**G**

**APPENDIX**

---

This appendix describes the Delfee description language.

## **GENERAL**

*description*  
*partition*  
*description partition*

*partition*  
*memory\_partition*  
*cpu\_partition*  
*software\_partition*

*ident\_list*  
*ident\_list* , *identifier*  
*identifier*

*identifier*  
*STRING*

*file\_name*  
*STRING*

## **CPU**

*cpu\_partition*  
**cpu** { *static\_specs\_list* }  
**cpu** { }  
**cpu** *file\_name*

## **MEMORY**

*memory\_partition*  
**memory** { *static\_specs\_list* }  
**memory** { }  
**memory** *file\_name*  
*static\_specs\_list*  
*static\_specs\_list static\_specs*  
*static\_specs*

*static\_specs*  
*amod\_specs*  
*spce\_specs*  
*bus\_specs*  
*chips\_specs*

*amod\_specs*  
**amode** *ident\_list* { *amod\_list* }

*spce\_specs*  
**space** *ident\_list* { *spce\_list* }

*bus\_specs*  
**bus** *ident\_list* { *bus\_list* }

*chips\_specs*  
**chips** *ident\_list* *chips\_list* ;

*amod\_list*  
*amod\_list* *amod\_def*  
*amod\_def*

*spce\_list*  
*spce\_list* *spce\_def*  
*spce\_def*

*bus\_list*  
*bus\_list* *bus\_def*  
*bus\_def*

*chips\_list*  
*chips\_list* *chips\_def*  
*chips\_def*

*amod\_def*  
*mau\_spec*  
*attribute\_spec*  
*map\_spec*

*spce\_def*  
*mau\_spec*  
*map\_spec*

```
bus_def
    mau_spec
    mem_spec
    map_spec

chips_def
    mau_equ_spec
    attribute_equ_spec
    size_spec

mau_spec
    mau NUMBER ;

mau_equ_spec
    mau = NUMBER

attribute_spec
    attribute STRING ;
    attribute NUMBER ;
    attr STRING ;
    attr NUMBER ;

attribute_equ_spec
    attribute = STRING
    attribute = NUMBER
    attr = STRING
    attr = NUMBER

map_spec
    map map_list ;

map_list
    map_list map_def
    map_def

map_def
    src_spec
    size_spec
    dst_spec
    align_spec
    page_spec
    amode_spec
    space_spec
    bus_spec
```

*mem\_spec*  
**mem** *mem\_list* ;

*mem\_list*  
*mem\_list* *mem\_def*  
*mem\_def*

*mem\_def*  
*addr\_spec*  
*chips\_spec*

*src\_spec*  
**src** = *NUMBER*

*size\_spec*  
**size** = *NUMBER*

*dst\_spec*  
**dst** = *NUMBER*

*align\_spec*  
**align** = *NUMBER*

*page\_spec*  
**page** = *NUMBER*

*amode\_spec*  
**amode** = *identifier*

*space\_spec*  
**space** = *identifier*

*bus\_spec*  
**bus** = *identifier*

*addr\_spec*  
**address** = *NUMBER*  
**addr** = *NUMBER*

*chips\_spec*  
**chips** = *low\_chip\_list*

*low\_chip\_list*  
*low\_chip\_list* , *low\_chip\_pair*  
*low\_chip\_pair*

*low\_chip\_pair*  
*low\_chip\_pair* | *low\_chip*  
*low\_chip*

*low\_chip*  
*identifier*

## **SOFTWARE**

*software\_partition*  
**software** { *layout\_blocks* }  
**software** { }  
**software** *file\_name*

*layout\_blocks*  
*layout\_blocks* *layout\_block*  
*layout\_block*

*layout\_block*  
*layout*  
*loadmod*

*loadmod*  
**load\_mod** *software\_specs* ;  
**load\_mod** *identifier* *software\_specs* ;

*software\_specs*  
*software\_specs* *software\_spec*  
*software\_spec*

*software\_spec*  
*start*  
*process*

*start*  
**start** = *identifier* ;

*process*  
**process** = *pids*

*pids*  
*NUMBER*  
*pids* , *NUMBER*

*layout*

**layout** { *space\_blocks* }

**layout** { }

**layout** *file\_name*

*space\_blocks*

*space\_blocks* *space\_block*

*space\_block*

*space\_block*

**space** *identifier* { *block\_blocks* }

*block\_blocks*

*block\_blocks* *block\_block*

*block\_block*

*block\_block*

**block** *identifier* { *cluster\_blocks* }

*cluster\_blocks*

*cluster\_blocks* *cluster\_block*

*cluster\_block*

*cluster\_block*

*cluster\_spec*

*p\_gap\_spec*

*p\_fixed\_spec*

*p\_pool\_spec*

*p\_skip\_spec*

*p\_label\_spec*

*cluster\_spec*

**cluster** *identifier* { *amode\_blocks* }

**cluster** *ident\_list* ;

*amode\_blocks*

*amode\_blocks* *amode\_block*

*amode\_block*

*amode\_block*

**amode** *ident\_list* { *section\_blocks* }

**amode** *ident\_list* ;

*section\_block*

*p\_gap\_spec*  
    **gap** *length* ;  
    **gap** ;

*p\_fixed\_spec*  
    **fixed** *address* ;

*p\_pool\_spec*  
    **pool** *length* ;  
    **pool** ;

*p\_label\_spec*  
    **label** *identifier* ;

*p\_skip\_spec*  
    **skip** ;

*attribute*  
    *attribute\_equ\_spec*

*length*  
    **length** = *NUMBER*  
    **leng** = *NUMBER*

*address*  
    **address** = *NUMBER*  
    **addr** = *NUMBER*

*section\_blocks*  
    *section\_blocks* *section\_block*  
    *section\_block*

*section\_block*  
    *section\_spec*  
    *copy\_spec*  
    *v\_fixed\_spec*  
    *v\_gap\_spec*  
    *v\_reserved\_spec*  
    *stack\_spec*  
    *heap\_spec*  
    *table\_spec*  
    *others*  
    *v\_label\_spec*  
    *v\_assert\_spec*  
    *attribute\_spec*



*section\_spec*  
**section** *selection modifiers* ;  
**section** *selection* ;

*modifiers*  
*modifiers modifier*  
*modifier*

*modifier*  
*attribute*  
*address*

*copy\_spec*  
**copy** *selection attribute* ;  
**copy** *selection* ;  
**copy** ;

*selection*  
**selection** = *STRING*  
*identifier*

*v\_fixed\_spec*  
**fixed** *address* ;

*v\_gap\_spec*  
**gap** ;

*v\_reserved\_spec*  
**reserved** *reserved\_options* ;  
**reserved** ;

*reserved\_options*  
*reserved\_options reserved\_option*  
*reserved\_option*

*reserved\_option*  
*attribute*  
*address*  
*length*  
*v\_label\_equ\_spec*

*stack\_spec*  
**stack** *stack\_options* ;  
**stack** ;

*heap\_spec*  
    **heap** *stack\_options* ;  
    **heap** ;

*stack\_options*  
    *stack\_options* *stack\_option*  
    *stack\_option*

*stack\_option*  
    *attribute*  
    *length*

*table\_spec*  
    **table** *attribute* ;  
    **table** ;

*v\_label\_spec*  
    **label** *identifier* ;

*v\_label\_equ\_spec*  
    **label** = *identifier*

*v\_assert\_spec*  
    **assert** ( *bool\_expression* , *STRING* ) ;  
    **asse** ( *bool\_expression* , *STRING* ) ;

*others*  
    **others** ;

*bool\_expression*  
    *term**p* *bool\_op* *term**p*

*term**p*  
    *term* + *term**p*  
    *term* - *term**p*  
    *term*

*term*  
    ( *term* )  
    *identifier*  
    *NUMBER*

*bool\_op*

<

>

==

!=

A **NUMBER** is a series of (hex) digits with optional suffixes 'k' 'M' 'G' which stands for 'kilo', 'mega' and 'giga'. Numbers may be given in hex, octal or decimal with the usual prefix. Where applicable numbers may be preceded by a minus sign.

A **STRING** is a series of characters that is not a number (089 is a **STRING** because it is not a valid octal number) and consists of alphanumeric characters including '\_', '.', '-' and the host dependent directory separators. (For PC '\', '/' and ':')

Any (part of a) token may contain environment variables. If the environment variable A contains the text 'foo' then the sequence:

```
$A/proto.dsc
```

is translated to:

```
foo/proto.dsc
```

Multi character variables must be combined with braces:

```
window = $(MODE);
```

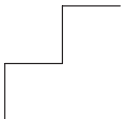
There are three methods to write comments in a delfee script. The first one is the 'C' style comment between '/\*' and '\*/'. The second form is a '#' in the first column. The second form allows preprocessing by the C-preprocessor. Any *#line* or *#file* directive will be ignored by the locator. The third form is the 'C++' style comment; a double slash '//' anywhere on a line introduces comments until the end of line.

# APPENDIX

# H

## **IEEE-695 OBJECT FORMAT**

---



---

# I

# APPENDIX

---

## **1 TIOF AND IEEE-695**

The IEEE-695 standard describes MUFOM: Microprocessor Universal Format for Object Modules. It defines a target independent storage standard for object files. However, this standard does not describe how symbolic debug information should be encoded according to that standard. Symbolic debug information can be a part of an object file. A debugger which reads an object file uses the symbolic debug information to obtain knowledge about the relation between the executable code and the origination high-level language source files. Since the IEEE-695 standard does not describe the representation of debug information, working implementations of this standard show vendor specific and microprocessor specific solutions for this area.

TIOF, which stands for Target Independent Object Format, is specified as a MUFOM based standard including the representation of symbolic debug information for high-level languages, without introducing the microprocessor dependent solutions. The current version of the TASKING debugger is not yet prepared to read TIOF, so you will have to select IEEE-695 as output format of the locator when you want to debug a program.

Since TIOF and IEEE-695 both use the MUFOM concept as their basis both formats are very similar to each other.

## **2 COMMAND LANGUAGE CONCEPT**

Most object formats are record oriented: there are one or more section headers at a fixed position in the file which describe how many sections are present. A section header contains information like start address, file offset, etc. The contents of the section is in some data part, which can only be processed after the header has been read. So the tool that reads such an object uses implicit assumptions how to process such a file. Seeking through the file to get those records which are relevant is usual.

MUFOM ( IEEE-695 ) uses a different approach. It is designed as a command language which steers the linker, locator and object reader in the debugger.

An assembler or compiler may create an object module where most of the data contained in it is relocatable. The next phase in the translation process is linking several object modules into one new object module. A relocatable object uses relocation expressions at places where the absolute values are not yet known. An expression evaluator in the locator transforms the relocation expressions into absolute values.

Finally the object is ready for loading into memory. Since an object file is transformed by several processes, MUFOM implements an object file as a sequence of commands which steers this transformation process.

These commands are created, executed or copied by one of five processes which act on a MUFOM object file:

1. Creation process  
Creation of the object file by an assembler or compiler. The assembler or compiler tells other MUFOM processes what to do, by emitting commands generated from assembly source text or a high-level language.
2. Linkage process  
Linking of several object modules into one module resolving external references by renaming X variables into I variables, and by generating new commands (assigning of R variables).
3. Relocation process  
Relocation, giving all sections an absolute address by assigning their L variable.
4. Expression evaluation process  
Evaluation of loader expressions, generated in one of the three previously mentioned MUFOM processes.
5. Loader process  
Loading the absolute memory image.

The last four processes are in fact command interpreters: the assembler writes an object file which is basically a large sequence of instructions for the linker. For example, instead of writing the contents of a section as a sequence of bytes at a specific position in the file, IEEE-695 defines a load command, LR, which instructs the linker to load a number of bytes. The LR command specifies the number of MAUs (minimum addressable unit) that will be relocated, followed by the actual data. This data can be a number of absolute bytes, or an expression which must be evaluated by the linker.

Transforming relocation expressions into new expressions or absolute data and combining sections is the actual linkage process.

It is possible that one or more of the above MUFOM processes are combined in one tool. For instance, the locator is built from process 3 and process 4 above.

### **3 NOTATIONAL CONVENTIONS**

The following conventions are used in this appendix:

	select one of the items listed between '   '
" "	literal characters are between " "
[ ]+	optional item repeats one time or more
[ ]?	optional item repeats zero times or one time
[ ]*	optional item repeats zero times or more
::=	can be read as "is defined as"

### **4 EXPRESSIONS**

An expression in an IEEE-695 file or a TIOF file is a combination of variables, operators and absolute data.

The variable name always starts with a non-hexadecimal letter (G...Z), immediately followed by an optional hexadecimal number. The first non-hexadecimal letter gives the class of the variable. Reading an object file you encounter the following variables:

- G** – Start address of a program. If not assigned this address defaults to the address of low-level symbol **\_start**.
- I** – An I variable represents a global symbol in an object module.  
  
The I variable is assigned an expression which is to be made available to other modules for the purpose of linkage edition. The name of an I variable is always composed of the letter 'I', followed by a hexadecimal number. An I variable is created only by an NI command.



- L** – Start address of a section. This variable is only used for absolute sections. The 'L' is followed by a section index, which is an hexadecimal number. L variables are created by an assignment command, but the section index must have been defined by an ST command.
- N** – Name of internal symbol. This variable is used to assign values of local symbols, or, to build complex types for use by a high-level language debugger, or for inter-modular type checking during linkage. The N variable is created with a NN command.
- P** – Program pointer per section. This variable always contains the current address of the target memory location. The P variable is followed by a section index, which is a hexadecimal number. The section index must have been defined with an ST command (section type command). The variable is created after its first assignment.
- R** – The R type variable is a relocation reference for a particular section. All references to addresses in this section must be made relative to the R variable. Linking is accomplished by assigning a new value to R. The R variable consists of the letter 'R', followed by an section index, which is a hexadecimal number. The section index must have been defined with an ST command. The default value of an (unassigned) R variable is 0.
- S** – The S type variable is the section size (in MAUs) for a section. There is one S variable per section. The 'S' is followed by an section index. An S variable is created by its first assignment.
- W** – Work variable. This type of variable can be used to assign values to, which can be used in following MUFOM commands. They serve the purpose of maintaining values in a workspace without any additional meaning. A work variable consists of the letter 'W' followed by a hexadecimal number. W variables are created by their first assignment.
- X** – An X type variable refers to an external reference. X-variables cannot have a value assigned to it. An X variable consists of the letter 'X' followed by a hexadecimal number.

The MUFOM language uses the following data types to form expressions:

```

digit          ::=  "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" |
                    "9"

hex_letter     ::=  "A" | "B" | "C" | "D" | "E" | "F"

hex_digit     ::=  digit | hex_letter

hex_number    ::=  [ hex_digit ]+

nonhex_letter  ::=  "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" |
                    "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" |
                    "W" | "X" | "Y" | "Z"

letter        ::=  hex_letter | nonhex_letter

alpha_num     ::=  letter | digit

identifier    ::=  letter [ alpha_num ]*

character     ::=  'value valid within chosen character set'

char_string_length ::=  hex_digit hex_digit

char_string   ::=  char_string_length [ character ]*
```

The numeric value specified in 'char\_string\_length' should be followed by an equal number of characters.

Expressions may be formed out of immediate numbers and MUFOM variables. The MUFOM processes 2 to 4, which form the linker and the locator, contain expression evaluators which parse and calculate the values for the expressions. If a MUFOM process cannot calculate the absolute value of an expression, because the values of the variable are not yet known, it copies the expression (with modifications) into the output file.

Expression are coded in reverse Polish notation. (The operator follows the operands.)

```

expression ::= boolean_function |
               one_operand_function |
               two_operand_function |
               three_operand_function |
               four_operand_function |
               conditional_expr | hex_number | MUFOM_variable
```

## 4.1 FUNCTIONS WITHOUT OPERANDS

@F: false function

@T: true function

boolean\_function ::= "@F" | "@T"

The false and true function produce a boolean result false or true which may be used in logical expressions. Both functions do not have operands.

## 4.2 MONADIC FUNCTIONS

Monadic functions have one operand which precedes the function.

one\_operand\_function ::= operand ", " monop

operand ::= expression

monop ::= "@ABS" | "@NEG" | "@NOT" | "@ISDEF"

@ABS: returns the absolute value of an integer operand

@NEG: returns the negative value of an integer operand

@NOT: returns the negation of a boolean operand or the one's complement value if the operand is an integer

@ISDEF: returns the logical true value if all variable in an expression are defined, return false otherwise.

## 4.3 DYADIC FUNCTIONS AND OPERATORS

Dyadic functions and operators have two operands which precede the operator or function.

two\_operand\_function ::= operand1 ", " operand2 ", " dyadop

operand1 ::= expression

operand2 ::= expression

dyadop ::= "@AND" | "@MAX" | "@MIN" | "@MOD" |  
 "@OR" | "@XOR" |  
 "+" | "-" | "/" | "%" | "<" | ">" | "=" | "#"

- @AND: returns boolean true/false result of logical 'and' operation on operands, when both operands are logical values. When both operands are not logical values the bitwise and is performed.
- @MAX: compares both operands arithmetically and returns the largest value.
- @MIN: compares both operands arithmetically and returns the smallest value.
- @MOD: returns the modulo result of the division of operand1 by operand2. The result is undefined if either operand is negative, or if operand2 is zero.
- @OR: returns boolean true/false result of logical 'or' operation on operands, when both operands are logical values. When both operands are no logical values the bitwise and is performed.
- +, -, \*, /: These are the arithmetic operators for addition, subtraction, multiplication and division. The result is an integer. For division the result is undefined if operand2 equals zero. The result of a division rounds toward zero.
- <, >, =, #: These are operators for the following logical relations: 'less than', 'greater than', 'equals', 'is unequal'. The result is true or false.

4.4 MUFOM VARIABLES

The meaning of the MUFOM variable is explained in section 4. The following syntax rules apply for the MUFOM variables:

- MUFOM\_variable ::= MUFOM\_var | MUFOM\_var\_num MUFOM\_var\_optnum
- MUFOM\_var ::= "G"
- MUFOM\_var\_num ::= "I" | "N" | "W" | "X" hex\_number
- MUFOM\_var\_optnum ::= "L" | "P" | "R" | "S" [ hex\_number ]?

## 4.5 @INS AND @EXT OPERATOR

The @INS operator inserts a bit string.

```
four_operand_function ::= operand1 "," operand2 "," operand3
                        "," operand4 "," @INS
```

operand2 is inserted in operand1 starting at position operand3, and ending at position operand4.

The @EXT operator extracts a bit string.

```
three_operand_function ::= operand1 "," operand2 "," operand3
                        "," @EXT
```

A bit string is extracted from operand1 starting at position operand2 and ending at position operand3.

## 4.6 CONDITIONAL EXPRESSIONS

```
conditional_expr ::= err_expr | if_else_expr
```

```
err_expr ::= value "," condition "," err_num "," "@ERR"
```

```
value ::= expression
```

```
condition ::= expression
```

```
err_num ::= expression
```

```
if_else_expr ::= condition "," "@IF" "," expression ","
               "@ELSE" "," expression "," "@END"
```

## **5 MUFOM COMMANDS**

### **5.1 MODULE LEVEL COMMANDS**

At module level there are four commands: one command to start and one to end a module, one command to set the date and time of creation of the module, and one command to specify address formats.

#### **5.1.1 MB COMMAND**

The MB command is the first command in a module. It specifies the target machine configuration and an optional command with the module name.

MB\_command ::= "MB" machine\_identifier [ "," module\_name ]? "."

Example: MB TriCore.

#### **5.1.2 ME COMMAND**

The module end command is the last command in an object file. It defines the end of the object module.

ME\_command ::= "ME."

#### **5.1.3 DT COMMAND**

The DT command sets the date and time of creation of an object module.

DT\_command ::= "DT" [ digit ]\* "."

Example: DT19930120120432.

The format of display of the date and time is "YYYYMMDDHHMMSS":

4 digits for the year, 2 digits for the month, 2 digits for the day, 2 digits for the hour, 2 digits for the minutes and 2 digits for the seconds.

### 5.1.4 AD COMMAND

The AD command specifies the address format of the target execution environment.

AD\_command ::= "AD" bits\_per\_MAU [ "," MAU\_per\_address  
[ "," order ]? ]?

MAU\_per\_address ::= hex\_number

bits\_per\_MAU ::= hex\_number

order ::= "L" | "M"

MAU stands for minimum addressable unit. This is target processor dependant.

L means least significant byte at lowest address ( little endian )

M means most significant byte at lowest address ( big endian )

Example:

AD8 , 3 , L .

Specifies a 3-byte addressable 8-bit processor running in little endian mode.

## 5.2 COMMENT AND CHECKSUM COMMAND

The comment command offers the possibility to store information in an object module about the object module and the translators that created it. The comment may be used to record the file name of the source file of the object module or the version number of the translator that created it. Because the standard supports several layers each of which has its own revision number an object module may contain several comment commands which specify which revision of the standard has been used to create the module. The contents of a comment is not prescribed by the standard and thus it is implementation defined how a MUFOM process handles a comment command.

CO\_command ::= "CO" [comment\_level]? "," comment\_text ."

comment\_level ::= hex\_number

comment\_text ::= char\_string

The comment levels 0 – 6 are reserved to pass information about the revision number of the layers in this standard.

The checksum command starts and checks the checksum calculation of an object module.

## 5.3 SECTIONS

A section is the smallest unit of code or data that can be controlled separately. Each section has a unique number which is introduced at the first section begin (SB) command. The contents of a section may follow its introduction. A section ends at the next SB command with a number different from the current number. A section resumes at an SB command with a number that has been introduced before.

### 5.3.1 SB COMMAND

SB\_command ::= "SB" hex\_number "."

The maximum number of sections in an object module is implementation defined.

### 5.3.2 ST COMMAND

The ST command specifies the type of a section.

ST\_command ::= "ST" section\_number [ "," section\_type ]\*  
[ "," section\_name ]? "."

section\_type ::= letter

section\_name ::= char\_string

A section can be named or unnamed. If section\_name is omitted a section is unnamed. A section can be relocatable or absolute. If the section start address is an absolute number the section is called absolute. If the section start address is not yet known, the section is called relocatable. In relocatable sections all addresses are specified relative to the relocation base of that section. The relocation phase of the linker or locator may map the relocation base of a section onto a fixed address.



During linkage edition the section name and the section attributes identify a section and thus the actions to be taken. If a section is defined in several modules, the linkage editor must determine how to act on sections with the same name. This can be either one of the following strategies:

- several sections are to be joined into a single one
- several sections are to be overlapped
- sections are not to coexist

A section type gives additional information to the linkage editor about the section, which may be used to layout a section in memory. Section type information is encoded with letters, which may be combined in one ST command. Some combinations of letters are invalid or may be meaningless.

letter	meaning	class	explanation
A	absolute	access	section has absolute address assigned to corresponding L-variable
R	read only	access	no write access to this section
W	writable	access	section may be read and written
X	executable	access	section contains executable code
Z	zero page	access	if target has zero page or short addressable page Z-section map into it
<i>Ynum</i>	addressing mode	access	section must be located in addressing mode <i>num</i>
B	blank	access	section must be initialized to '0' (cleared)
F	not filled	access	section is not filled or cleared (scratch)
I	initialize	access	section must be initialized in rom
E	equal	overlap	if sections in two modules have different length an error must be raised
M	max	overlap	Use largest value as section size
U	unique	overlap	The section name must be unique
C	cumulative	overlap	Concatenate sections if they appear in several modules. The section alignment for partial section must be preserved

letter	meaning	class	explanation
O	overlay	overlap	sections with the name <i>name@func</i> must be combined to one section <i>name</i> , according to the rules for <i>func</i> obtained from the call graph
S	separate	overlap	multiple sections can have the same name and they may be relocated at unrelated addresses
N	now	when	section is located before normal sections (without N or P)
P	postpone	when	section is located after normal sections (without N or P)

Table H-1: Section types

### 5.3.3 SA COMMAND

SA\_command ::= "SA" section\_number "," [MAU\_boundary ]? [ "," page\_size ]? "."

MAU\_boundary ::= expression

page\_size ::= expression

The MAU boundary value forces the relocater to align a section on the number of MAUs specified. If page\_size is present the relocater checks that the section does not exceed a page boundary limit when it is relocated.

## 5.4 SYMBOLIC NAME DECLARATION AND TYPE DEFINITION

### 5.4.1 NI COMMAND

The NI command defines an internal symbol. An internal symbol is visible outside the module. Thus it may resolve an undefined external in another module.

NI\_command ::= "N" I\_variable "," char\_string "."

The NI\_command must precede any reference to the I\_variable in a module. There may not be more than one I\_variable with the same name or number.

**5.4.2 NX COMMAND**

The NX command defines an external symbol which is undefined in the current module. The NX command must precede all occurrences of the corresponding X variable.

NX\_command ::= "N" X\_variable "," char\_string "."

The unresolved reference corresponding to an NX-command can be resolved by an internal symbol definition ( NI\_command ) in another module.

**5.4.3 NN COMMAND**

The NN command defines a local name which may be used for defining a name of a local symbol in a module or a name in a type definition.

A name defined with an NN command is not visible outside the scope of the module. The NN command must precede all occurrences of the corresponding N variable.

NN\_command ::= "N" N\_variable "," char\_string "."

**5.4.4 AT COMMAND**

The attribute command may be used to define debugging related information of a symbol, such as the symbol type number. Level 2 of the standard does not prescribe the contents of the optional fields of the AT command. The language dependent layer (level 3) describes how these fields can be used to pass high-level symbol information with the AT command.

AT\_command ::= "AT" variable "," type\_table\_entry [ "," lex\_level [ "," hex\_number ]\* ]? "."

variable ::= I\_variable | N\_variable | X\_variable

`type_table_entry ::= hex_number`

`lex_level ::= hex_number`

The `type_table_entry` is a type number introduced with a `type` command (TY). References to type numbers in the AT command may precede the definition of the type in the TY command.

The meaning of the `lex_level` field is defined at layer 3 or higher. The same applies to the optional `hex_number` fields.

### 5.4.5 TY COMMAND

The TY-command defines a new type table entry. The type number introduced by the type command can be seen as a reference index to this type. The TY-command defines the relation between the newly introduced type and other types that are defined in other places in the object module. It also establishes a relation between a new type index and symbols (`N_variable`).

`TY_command ::= "TY" type_table_entry [ "," parameter ]+ "."`

`type_table_entry ::= hex_number`

`parameter ::= hex_number | N_variable | "T" type_table_entry`

Level 2 does not define the semantics of the parameters. These are defined at level 3, the language layer. A linkage editor which does not have knowledge of the semantics of the parameter in a type command can still perform type comparison: Two types are considered to compare equal when the following conditions hold:

- both types have an equal number of parameters.
- the numeric values in the types are equal
- `N_variables` in both types have the same name
- the type entries referenced from both types compare equal

Variable `N0` is supposed to compare equal to any other name.

Type table entry `T0` is supposed to compare equal to any other type.

## **5.5 VALUE ASSIGNMENT**

### **5.5.1 AS COMMAND**

The assignment command assigns a value to a variable.

AS\_command ::= "AS" MUFOM\_variable "," expression "."

## **5.6 LOADING COMMANDS**

The contents of a section is either absolute data (code) or relocatable data (code). Absolute data can be loaded with the LD command. The address where loading takes place depends on the value of the P-variable belonging to the section. Data which is contiguous in a LD command is supposed to be loaded contiguously in memory.

If data is not absolute it contains expressions which must be evaluated by the expression evaluator. The LR command allows a relocation expression to be part of the loading command.

### **5.6.1 LD COMMAND**

LD\_command ::= "LD" [ hex\_digit ]+ "."

The constants loaded with the LD command are loaded with the most significant part first.

### **5.6.2 IR COMMAND**

A relocation base is an expression which can be associated with a relocation letter. This relocation letter can be used in subsequent load relocate commands.

IR\_command ::= "IR" relocation\_letter "," relocation\_base  
[ "," number\_of\_bits ]? "."

relocation\_letter ::= nonhex\_letter

relocation\_base ::= expression

number\_of\_bits ::= expression

Example:

```
IRV,X20,16.
ITM,R2,40,+,8.
```

The number\_of\_bits must be less than or equal to the number of bits per address, which is the product of the number of MAUs per address and the number of bits per MAU, both of which are specified in the AD command. If the number\_of\_bits is not specified it equals the number of bits per address.

### 5.6.3 LR COMMAND

LR\_command ::= "LR" [ load\_item ]+ "

load\_item ::= relocation\_letter offset "," | load\_constant |  
 "(" expression [ "," number\_of\_MAUs ]? ")"

load\_constant ::= [ hex\_digit ]+

number\_of\_MAUs ::= expression

Examples:

```
LR002000400060.
LRT80,0020.
LR(R2,100,+,4).
```

The first example shows immediate constants which may be loaded as a part of an LR command.

The second example shows the use of the relocation base defined in the previous paragraph, followed by a constant.

The third example shows how the value of the expression  $R2 + 100$  is used to load 4 MAUs.

The three commands in this example may be combined into one LR command:

```
LR002000400060T80,0020(R2,100,+,4).
```

### **5.6.4 RE COMMAND**

The replicate command defines the number of times a LR command must be replicated:

`RE_command ::= "RE" expression "."`

The LR command must immediately follow the RE command.

Example:

```
RE04 .
LR (R2, 200, +, 4) .
```

The commands above load 16 MAUs: 4 times the 4 MAU value of R2 + 200

## **5.7 LINKAGE COMMANDS**

### **5.7.1 RI COMMAND**

The retain internal symbol command indicates that the symbolic information of an NI command must be retained in the output file.

`RI_command ::= "R" I_variable [ ", " level_number ]? "."`

`level_number ::= hex_number`

### **5.7.2 WX COMMAND**

The weak external command flags a previously defined external (NX\_command) as weak. This means that if the external remains unresolved, the value of the expression in the WX command is assigned to the X variable.

`WX_command ::= "W" X_variable [ ", " default_value ]? "."`

`default_value ::= expression`

### **5.7.3 LI COMMAND**

The LI command specifies a default library search list. The library names specified in the LI\_command are searched for unresolved references.

LI\_command ::= "LI" char\_string [ "," char\_string ]\* "."

### **5.7.4 LX COMMAND**

The LX command specifies a library to search for a named unresolved variable.

LX\_command ::= "L" X\_variable [ "," char\_string ]+ "."

The paragraphs above showed the commands and operators as ASCII strings. In an object file they are binary encoded. The following tables show the binary representation.



6 MUFOM FUNCTIONS

The following table lists the first byte of MUFOM elements. Each value between 0 and 255 classifies the MUFOM language element that follows, or it is a language element itself. E.g. numbers outside the range 0–127 are preceded by a length field: 0x82 specifies that a 2 byte integer follows. 0xE4 is the function code for the LR command.

Overview of first byte of MUFOM language elements

Value	Description
0x00 – 0x7F	Start of regular string, or one byte numbers ranging from 0 – 127
0x80	Code for omitted optional number field
0x81 – 0x88	Numbers outside the range 0 – 127
0x89 – 0x8F	Unused
0x90 – 0xA0	User defined function codes
0xA0 – 0xBF	MUFOM function codes
0xC0	Unused
0xC1 – 0xDA	MUFOM letters
0xDB – 0xDF	Unused
0xE0 – 0xF9	MUFOM commands
0xFA – 0xFF	Unused

Table H-2: Overview of first byte of MUFOM language elements

Binary encoding of MUFOM letters and function codes

Function code		Identifiers	
Function	code	Letter	code
@F	0xA0		
@T	0xA1	A	0xC1
@ABS	0xA2	B	0xC2
@NEG	0xA3	C	0xC3
@NOT	0xA4	D	0xC4
+	0xA5	E	0xC5

Function code		Identifiers	
Function	code	Letter	code
–	0xA6	F	0xC6
/	0xA7	G	0xC7
*	0xA8	H	0xC8
@MAX	0xA9	I	0xC9
@MIN	0xAA	J	0xCA
@MOD	0xAB	K	0xCB
<	0xAC	L	0xCC
>	0xAD	M	0xCD
=	0xAE	N	0xCE
!= <>	0xAF	O	0xCF
@AND	0xB0	P	0xD0
@OR	0xB1	Q	0xD1
@XOR	0xB2	R	0xD2
@EXT	0xB3	S	0xD3
@INS	0xB4	T	0xD4
@ERR	0xB5	U	0xD5
@IF	0xB6	V	0xD6
@ELSE	0xB7	W	0xD7
@END	0xB8	X	0xD8
@ISDEF	0xB9	Y	0xD9
		Z	0xDA

Table H-3: Binary encoding of MUFOM letters and function codes

### ***MUFOM Command codes***

Command	Code	Description
MB	0xE0	Module begin
ME	0xE1	Module end
AS	0xE2	Assign
IR	0xE3	Initialize relocation base

Command	Code	Description
LR	0xE4	Load with relocation
SB	0xE5	Section begin
ST	0xE6	Section type
SA	0xE7	Section alignment
NI	0xE8	Internal name
NX	0xE9	External name
CO	0xEA	Comment
DT	0xEB	Date and time
AD	0xEC	Address description
LD	0xED	Load
CS (with sum)	0xEE	Checksum followed by sum value
CS	0xEF	Checksum (reset sum to 0 )
NN	0xF0	Name
AT	0xF1	Attribute
TY	0xF2	Type
RI	0xF3	Retain internal symbol
WX	0xF4	Weak external
LI	0xF5	Library search list
LX	0xF6	Library external
RE	0xF7	Replicate
SC	0xF8	Scope definition
LN	0xF9	Line number
	0xFA	Undefined
	0xFB	Undefined
	0xFC	Undefined
	0xFD	Undefined
	0xFE	Undefined
	0xFF	Undefined

Table H-4: MUFOM Command codes

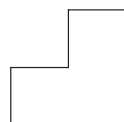
# APPENDIX



## MOTOROLA S-RECORDS



TASKING



---

# — APPENDIX

---



The locator has an option that controls the length of the output buffer for generating S1 records. The default buffer length is 32 code bytes.

The checksum calculation of S1 records is identical to S0.

### **S2 - record**

With the **-f2S2** option of the locator, which is the default, the actual program code and data is supplied with S2 records, with the following layout:

'S' '2' <length\_byte> <address> <code bytes> <checksum\_byte>

For the TriCore the locator generates 3-byte addresses.

Example:

```
S213FF002000232222754E00754F04AF4FAE4E22BF
  | |           | _ code
  | |           | _ address
  | |           | _ length
```

The locator has an option that controls the length of the output buffer for generating S2 records. The default buffer length is 32 code bytes.

The checksum calculation of S2 records is identical to S0.

### **S3 - record**

With the **-f2S3** option of the locator, the actual program code and data is supplied with S3 records, with the following layout:

'S' '3' <length\_byte> <address> <code bytes> <checksum\_byte>

This record is used for 4-byte addresses.

Example:

```
S3070000FFFE6E6825
  | |           | | _ checksum
  | |           | | _ code
  | |           | _ address
  | |           | _ length
```

The locator has an option that controls the length of the output buffer for generating S3 records.

The checksum calculation of S3 records is identical to S0.

### **S7 - record**

With the **-f2S3** option of the locator, at the end of an S-record file, the locator generates an S7 record, which contains the program start address. S7 is the corresponding termination record for S3 records.

Layout:

'S' '7' <length\_byte> <address> <checksum\_byte>

Example:

```
S70500006E6824
| |         | _checksum
| | _address
| _ length
```

The checksum calculation of S7 records is identical to S0.

### **S8 - record**

With the **-f2S2** option of the locator, which is the default, at the end of an S-record file, the locator generates an S8 record, which contains the program start address.

Layout:

'S' '8' <length\_byte> <address> <checksum\_byte>

Example:

```
S804FF0003F9
| |         | _checksum
| | _address
| _ length
```

The checksum calculation of S8 records is identical to S0.

### **S9 - record**

With the **-f2S1** option of the locator, at the end of an S-record file, the locator generates an S9 record, which contains the program start address. S9 is the corresponding termination record for S1 records.

Layout:

'S' '9' <length\_byte> <address> <checksum\_byte>



Example:

```
S9030210EA
| |   |_checksum
| |_ address
|_ length
```

The checksum calculation of S9 records is identical to S0.

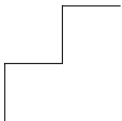
# APPENDIX



## INTEL HEX RECORDS



TASKING





# APPENDIX



Intel Hex records describe the hexadecimal object file format for 8-bit, 16-bit and 32-bit microprocessors. The hexadecimal object file is an ASCII representation of an absolute binary object file. There are six different types of records:

- Data Record (8-, 16, or 32-bit formats)
- End of File Record (8-, 16, or 32-bit formats)
- Extended Segment Address Record (16, or 32-bit formats)
- Start Segment Address Record (16, or 32-bit formats)
- Extended Linear Address Record (32-bit format only)
- Start Linear Address Record (32-bit format only)

For the TriCore the locator generates records in the 32-bit format (4-byte addresses).

**General Record Format**

In the output file, the record format is:

:	<i>length</i>	<i>offset</i>	<i>type</i>	<i>content</i>	<i>checksum</i>
---	---------------	---------------	-------------	----------------	-----------------

Where:

: is the record header.

*length* is the record length which specifies the number of bytes of the *content* field. This value occupies one byte (two hexadecimal digits). The locator outputs records of 255 bytes (32 hexadecimal digits) or less; that is, *length* is never greater than FFH.

*offset* is the starting load offset specifying an absolute address in memory where the data is to be located when loaded by a tool. This field is two bytes long. This field is only used for Data Records. In other records this field is coded as four ASCII zero characters ('0000').

*type* is the record type. This value occupies one byte (two hexadecimal digits). The record types are:

Byte Type	Record type
00	Data
01	End of File
02	Extended segment address (not used)
03	Start segment address (not used)
04	Extended linear address (32-bit)
05	Start linear address (32-bit)

- content*

is the information contained in the record. This depends on the record type.
- checksum*

is the record checksum. The locator computes the checksum by first adding the binary representation of the previous bytes (from *length* to *content*). The locator then computes the result of sum modulo 256 and subtracts the remainder from 256 (two's complement). Therefore, the sum of all bytes following the header is zero.

**Extended Linear Address Record**

The Extended Linear Address Record specifies the two most significant bytes (bits 16–31) of the absolute address of the first data byte in a subsequent Data Record:

:	02	0000	04	<i>upper_address</i>	<i>checksum</i>
---	----	------	----	----------------------	-----------------

The 32-bit absolute address of a byte in a Data Record is calculated as:

$( address + offset + index ) \text{ modulo } 4G$

where:

- address*

is the base address, where the two most significant bytes are the *upper\_address* and the two least significant bytes are zero.
- offset*

is the 16-bit offset from the Data Record.
- index*

is the index of the data byte within the Data Record (0 for the first byte).

Example:

```
:0200000400FFFB
| | | | | _ checksum
| | | | | _ upper_address
| | | | | _ type
| | | | | _ offset
| | | | | _ length
```

**Data Record**

The Data Record specifies the actual program code and data.

:	<i>length</i>	<i>offset</i>	00	<i>data</i>	<i>checksum</i>
---	---------------	---------------	----	-------------	-----------------

The *length* byte specifies the number of *data* bytes. The locator has an option that controls the length of the output buffer for generating Data records. The default buffer length is 32 bytes.

The *offset* is the 16-bit starting load offset. Together with the address specified in the Extended Address Record it specifies an absolute address in memory where the data is to be located when loaded by a tool.

Example:

```
:0F00200000232222754E00754F04AF4FAE4E22C3
| | | | | _ data
| | | | | _ type
| | | | | _ offset
| | | | | _ length
| | | | | _ checksum
```

**Start Linear Address Record**

The Start Linear Address Record contains the 32-bit program execution start address.

Layout:

:	04	0000	05	<i>address</i>	<i>checksum</i>
---	----	------	----	----------------	-----------------

Example:

```
:0400000500FF0003F5
| | | | |
| | | | | _ checksum
| | | | | _ address
| | | | | _ type
| | | | | _ offset
| | | | | _ length
```

**End of File Record**

The hexadecimal file always ends with the following end-of-file record:

```
:00000001FF
| | | | |
| | | | | _ checksum
| | | | | _ type
| | | | | _ offset
| | | | | _ length
```

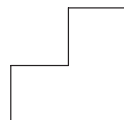
# INDEX

## INDEX

---



**TASKING**





---

# INDEX

---

# Symbols

- .a extension, 9-25
- .accum, 7-6
- .addr, F-33
- .align, assembler directive, 7-7
- .ascii, 7-8
- .asciiz, 7-9
- .asm, file extension, 2-4
- .byte, 7-10
- .calls, 7-11
- .comment, assembler directive, 7-12
- .DEFAULT, 11-22
- .define, assembler directive, 7-13
- .DONE, 11-22
- .double, 7-14
- .dsc extension, 10-4
- .dup, 7-15
- .dupa, 7-16
- .dupc, 7-17
- .dupf, 7-18
- .elc extension, 10-27
- .else, assembler directive, 7-31
- .elseif, assembler directive, 7-31
- .end, assembler directive, 7-19
- .endif, 7-20
- .endm, 7-21
- .equ, 7-22
- .ers extension, 2-13
- .exitm, 7-23
- .extern, assembler directive, 7-24
- .fail, assembler directive, 7-25
- .float, 7-26
- .fract, 7-27
- .global, assembler directive, 7-28
- .half, 7-29
- .if, assembler directive, 7-31
- .IGNORE, 11-22
- .include, assembler directive, 7-33
- .INIT, 11-22
- .local, assembler directive, 7-34
- .lst, file extension, 2-4, 2-6
- .macro, assembler directive, 7-35
- .message, 7-37
- .name, assembler directive, 7-38
- .obj file extension, 2-4, 2-6
- .org, assembler directive, 7-39
- .out extension, 10-4
- .pcp, file extension, 2-6
- .pmacro, 7-41
- .PRECIOUS, 11-22
- .sdecl, 3-4
  - assembler directive*, 7-42
- .sect, assembler directive, 7-44
- .set, assembler directive, 7-45
- .sfract, 7-46
- .SILENT, 11-22
- .space, 7-47
- .src, file extension, 2-4
- .SUFFIXES, 11-22
- .symb, 7-48
- .undef, assembler directive, 7-49
- .warning, 7-50
- .word, 7-51
- #define, F-28
- #elif, F-31
- #endif, F-32
- #if, F-31
- #ifdef, F-31
- #ifndef, F-31
- #include, F-29
- #undef, F-29
- @ character, 3-4
- \ character, 4-6
- \_ASPCP, 5-8
- \_ASTRI, 5-8
- \_lc\_b\_section, 10-31
- \_lc\_bh, 10-32, F-51
- \_lc\_bs, 10-33, F-70
- \_lc\_cp, 10-28, 10-34
- \_lc\_e\_section, 10-31
- \_lc\_eh, 10-32, F-51
- \_lc\_es, 10-33, F-70
- \_lc\_u\_identifier, 10-35

\_lc\_ub\_identifier, 10-36  
 \_lc\_ue\_identifier, 10-36  
 \_PCP\_, 2-34

## A

action attribute, F-39  
 actions, 10-28  
 adding files to a project, 1-11  
 addition and subtraction, 5-11  
 addr, F-34  
 address, F-34  
   *sorted ascending*, 10-24  
   *unsorted*, 10-24  
 address mapping, F-8  
 address space, definition, F-10  
 addressing mode, F-36  
   *definition*, F-11  
 addressing modes  
   *absolute*, 5-3  
   *base + offset*, 5-4  
   *bit-reverse*, 5-5  
   *circular*, 5-4  
   *PCP assembler*, 5-5  
   *post-increment*, 5-4  
   *pre-increment*, 5-4  
   *TriCore assembler*, 5-3  
 alignment, F-39  
 amode, F-36  
   *definition*, F-11, F-24  
   *manipulating sections*, F-25  
 archiver, 11-4  
 arithmetic operators, 5-11  
 artri, 1-3, 11-4  
 ASPCPINC, 2-22, 2-44  
 asse, F-38  
 assembler  
   *absolute list file*, 2-45  
   *controls*  
     *case*, 8-6  
     *debug*, 8-7

*fpu*, 8-8  
   *hw\_only*, 8-9  
   *ident*, 8-10  
   *list*, 8-11, 8-12  
   *mmu*, 8-13  
   *nopaging*, 8-20  
   *object*, 8-14  
   *overview of*, 8-4-8-26  
   *page*, 8-15, 8-16  
   *pagelength*, 8-18  
   *pagewidth*, 8-19  
   *paging*, 8-20  
   *prctl*, 8-21  
   *print*, 8-22  
   *stitle*, 8-23  
   *TC*, 8-24  
   *title*, 8-25  
   *warning*, 8-26  
   *general controls*, 8-3  
   *input files and output files*, 2-4, 2-6  
   *invocation*, 2-3  
   *invocation aspcp*, 2-6  
   *invocation astri*, 2-4  
   *list file*, 2-44  
   *options summary aspcp*, 2-7  
   *options summary astri*, 2-5  
   *page header*, 2-45  
   *primary controls*, 8-3  
   *source listing*, 2-46  
 assembler controls, overview of,  
   8-4-8-26  
 assembler directives  
   *.accum*, 7-6  
   *.align*, 7-7  
   *.ascii*, 7-8  
   *.asciiz*, 7-9  
   *.byte*, 7-10  
   *.calls*, 7-11  
   *.comment*, 7-12  
   *.define*, 7-13  
   *.double*, 7-14  
   *.dup*, 7-15

- .dupa*, 7-16
- .dupc*, 7-17
- .dupf*, 7-18
- .else*, 7-31
- .elseif*, 7-31
- .end*, 7-19
- .endif*, 7-20
- .endm*, 7-21
- .equ*, 7-22
- .exitm*, 7-23
- .extern*, 7-24
- .fail*, 7-25
- .float*, 7-26
- .fract*, 7-27
- .global*, 7-28
- .half*, 7-29
- .if*, 7-31
- .include*, 7-33
- .local*, 7-34
- .macro*, 7-35
- .message*, 7-37
- .name*, 7-38
- .org*, 7-39
- .pmacro*, 7-41
- .sdecl*, 7-42
- .sect*, 7-44
- .set*, 7-45
- .sfract*, 7-46
- .space*, 7-47
- .symb*, 7-48
- .undef*, 7-49
- .warning*, 7-50
- .word*, 7-51
- assembly control, 7-3
- conditional assembly, 7-5
- debugging, 7-3
- macros, 7-5
- overview, 7-3
- symbol definition, 7-4
- assembler options
  - ?, 2-8
  - C, 2-9
  - c, 2-10
  - D, 2-11
  - e, 2-12
  - err, 2-13
  - f, 2-14
  - FPU, 2-16
  - g, 2-17, 2-18
  - H, 2-20, 2-21
  - I, 2-22
  - i, 2-23
  - L, 2-24
  - l, 2-26
  - ll, 2-27
  - ln, 2-28
  - lt, 2-29
  - lw, 2-30
  - MMU, 2-31
  - o, 2-33
  - P, 2-34
  - p, 2-35
  - pRAPTOR, 2-36
  - pTC1775, 2-37
  - t, 2-38
  - TC2, 2-32
  - V, 2-39
  - v, 2-40
  - w, 2-41
  - WAE, 2-42
  - z, 2-43
- assembly source file, 2-4, 2-6
- assert, F-38
- astri, 1-3
- ASTRIINC, 2-22, 2-44
- AT keyword, 3-6
- attr, F-39
- attribute, F-39
  - action*, F-39
  - b*, F-40
  - defaults*, F-40
  - f*, F-40
  - g*, F-39
  - i*, F-40
  - r*, F-39
  - s*, F-39

*section*, 7-42

*w*, F-39

*x*, F-39

*y*, F-39

## B

binary operator, 5-10

bitwise and operator, 5-13

bitwise not operator, 5-13

bitwise operators, 5-13

bitwise or operator, 5-13

bitwise xor operator, 5-13

block, F-42

*definition*, F-20

buffer size, 10-24

bus, F-43

*definition*, F-13

## C

call graph, 9-3, 9-7, 9-28, 11-32

*stack usage*, 9-28

case, assembler control, 8-6

case sensitivity, 9-6

cctri, 1-3, 11-7

CCTRIBIN, 11-12

CCTRIOPT, 11-12

character, 4-4

chip, *definition*, F-15

chips, F-44

clear, section attribute, 3-5

cluster

*cleared sections*, F-40

*definition*, F-23

*executable sections*, F-39

*global sections*, F-39

*read-only*, F-39

*scratch sections*, F-40

*writable*, F-39

cluster keyword, F-45

command file, 2-14, 9-11, 10-11, 11-9

command line processing, 2-14, 9-11,  
10-11

comment, 4-5

conditional assembly, 6-12

conditional statements, F-31

continuation, 3-4

control lines, 8-3

control program, 10-27, 11-7

control program options

-?, 11-8

-c, 11-9

-c++, 11-8

-cc, 11-9

-cl, 11-9

-cm, 11-9

-cp, 11-9

-cs, 11-9

-elf, 11-11

-f, 11-9

-fptrap, 11-11

-ieee, 11-11

-ibex, 11-11

-nolib, 11-11

-o, 11-11

-srec, 11-11

-tiof, 11-11

-tmp, 11-11

-V, 11-8

-v, 11-11

-v0, 11-12

-Wa, 11-8

-Wc, 11-8

-wc++, 11-12

-Wcp, 11-8

-Wlc, 11-8

-Wlk, 11-8

-Wpl, 11-8

copy, F-46

copy table, 10-28, 10-34

*start of*, 10-29

cp\_bss, 10-28

cp\_copy, 10-28

cpu, F-47  
 creating a makefile, 1-12  
 CTRLIB, 9-13, 9-25

## D

debug, assembler control, 8-7  
 debug information, 11-39  
 debugger, starting, 1-10  
 debugging, 1-17, 7-3  
 declaration attribute, 7-42  
 Delfee, 10-3  
   *abbreviation of keywords*, F-73  
   *basic structure*, F-3  
   *comments*, G-12  
   *cpu part*, F-6, G-3  
   *getting started*, F-3  
   *keyword reference*, F-32  
   *keyword summary*, F-73  
   *memory part*, F-27, G-3  
   *preprocessing*, F-28  
   *software part*, F-17, G-7  
 description file, F-5  
 development flow, 1-4  
 directive, 4-4  
 directory separator, 9-25  
 division, 5-12  
 double, 7-14  
 dst keyword, F-48  
 dummy argument string, 6-10  
 dyadic functions, H-8

## E

EDE, 1-5  
   *build an application*, 1-9  
   *load files*, 1-7  
   *open a project*, 1-7  
   *select a toolchain*, 1-6  
   *start a new project*, 1-11  
   *starting*, 1-5

else, 11-16  
 embedded development environment.  
   *See* EDE  
 endif, 11-16  
 environment variable  
   *ASPCPINC*, 2-22, 2-44  
   *ASTRIINC*, 2-22, 2-44  
   *CCTRIBIN*, 11-12  
   *CCTRIOPT*, 11-12  
   *CTRLIB*, 9-13, 9-25  
   *HOME*, 11-15  
   *overview of*, 1-16  
   *TMPDIR*, 1-17, 2-44, 11-12  
   *used by control program*, 11-12  
   *used by tool chain*, 1-16  
 equal operator, 5-13  
 error list file, 2-4, 2-6  
 example  
   *starting EDE*, 1-5  
   *using EDE*, 1-5  
   *using the control program*, 1-12  
   *using the makefile*, 1-15  
 exit macro, 7-23  
 expression evaluator, H-4  
 expression string, 5-7  
 expressions, 5-5  
   *absolute*, 5-5  
   *relocatable*, 5-5  
   *type of*, 5-8  
 extension, 1-18  
   *.a*, 1-19, 9-25  
   *.abs*, 1-19  
   *.asm*, 1-19  
   *.c*, 1-19  
   *.cal*, 1-19  
   *.cc*, 1-19  
   *.dsc*, 1-19  
   *.elf*, 1-19  
   *.ers*, 2-13  
   *.hex*, 1-19  
   *.ic*, 1-19  
   *.lhl*, 1-19  
   *.lst*, 1-19

*.map*, 1-19  
*.obj*, 1-19, 2-4, 2-6  
*.out*, 1-19  
*.pcp*, 1-19  
*.src*, 1-19  
*.sre*, 1-19  
 external memory, F-16  
 external part, 10-15, 11-34

## F

file extensions, 1-18  
 file inclusion, F-29  
 fixed, F-49  
 float, 7-26  
 floating point, single precision, 2-16  
 flow graph, 2-3, 7-11  
 format  
   *specifier*, 10-24  
   *suboptions*, 10-24  
 fpu, assembler control, 8-8  
 fraction  
   16-bit, 7-46  
   32-bit, 7-27  
 function, 5-15  
   *abs*, 5-17  
   *address calculation*, 5-16  
   *arg*, 5-17  
   *aspcp*, 5-17  
   *assembler mode*, 5-16  
   *astri*, 5-18  
   *cat*, 5-18  
   *cnt*, 5-18  
   *def*, 5-18  
   *detailed description*, 5-17  
   *fract*, 5-18  
   *fraction*, 5-17  
   *hi*, 5-19  
   *his*, 5-19  
   *len*, 5-19  
   *lo*, 5-19  
   *los*, 5-19

*lst*, 5-20  
*mac*, 5-20  
*macro*, 5-16  
*mathematical*, 5-15  
*max*, 5-20  
*min*, 5-20  
*mvp*, 5-20  
*pos*, 5-20  
*result type*, 5-9  
*scp*, 5-21  
*sfract*, 5-21  
*sgn*, 5-21  
*string*, 5-15  
*sub*, 5-21  
*syntax*, 5-15  
 function delimiter, 4-13  
 Functional problem checks, 2-43, 8-24

## G

gap, F-50  
 global symbol, 3-3  
 global type info, 11-36  
 greater than operator, 5-13  
 greater than or equal operator, 5-13

## H

heap, 10-32, F-51  
 HOME, 11-15  
 hw\_only, assembler control, 8-9

## I

ident, assembler control, 8-10  
 identifier, 4-3  
 IEEE  
   *archiver*, 11-4  
   *command language concept*, H-3

*conditional expressions, H-10*  
*expressions, H-5*  
*notational conventions, H-5*  
*variables in object file, H-5*  
*viewer, 11-26*  
 ifdef, 11-16  
 ifndef, 11-16  
 include file, 2-22  
 incremental linkage, 9-29  
 incremental linking, 9-3  
 input specification, 4-3  
 instruction, 4-3  
 Intel hex, record type, J-3  
 invocation  
   *assembler, 2-3*  
   *assembler aspcp, 2-6*  
   *assembler atri, 2-4*  
   *linker, 9-4*  
   *locator, 10-4*

## K

keyword  
   *abbreviation of, F-73*  
   *amode, F-24*  
   *block, F-20*  
   *bus, F-13*  
   *chips, F-15*  
   *cluster, F-23*  
   *map, F-8*  
   *mem, F-10*  
   *selection, F-22*  
   *stack, 10-29*  
   *summary of, F-73*

## L

label, 3-3, 4-3, F-52  
   *local, 4-3*  
   *locator, 10-29*  
   *numeric, 4-3*

layout, F-17, F-54  
   *definition, F-18*  
   *example, F-18*  
 lctri, 1-3  
 leng, F-55  
 length, F-55  
 less than operator, 5-13  
 less than or equal operator, 5-13  
 level, mixed, 11-47  
 level 0, 11-47  
 level 1, 11-46  
 level option -ln, 11-46  
 library  
   *linking, 9-27*  
   *position, 9-27*  
   *system, 9-25*  
   *user, 9-25*  
 library maintainer, 11-4  
 library member, search algorithm, 9-27  
 library search path, 9-25  
 line continuation, 4-6  
 linker  
   *invocation, 9-4*  
   *linking with libraries, 9-27*  
   *map file, 9-28*  
   *messages, 9-37*  
   *options summary, 9-4*  
   *output, 9-28*  
 linker options  
   -?, 9-5  
   -C, 9-6  
   -c, 9-7  
   -d, 9-8  
   -e, 9-9  
   -err, 9-10  
   -f, 9-11  
   -H, 9-5  
   -L, 9-13  
   -l, 9-14  
   -M, 9-15  
   -N, 9-16  
   -O, 9-17  
   -o, 9-18



-r, 9-19  
 -t, 9-22  
 -u, 9-20  
 -V, 9-21  
 -v, 9-22  
 -w, 9-23  
 -WAE, 9-24  
 linking, incremental, 9-29  
 list, assembler control, 8-11, 8-12  
 list file, 2-4, 2-6, 2-26, 2-27, 2-28,  
     2-29, 2-30, 2-44, 9-15  
     *absolute*, 2-45  
     *file name*, 2-28  
     *header title*, 2-29  
     *page length*, 2-27  
     *page width*, 2-30  
     *removing lines from*, 2-24  
 lktri, 1-3  
 load module, F-17  
 load\_mod, F-17, F-56  
 local label, 4-3, 6-11  
 local symbol, 3-3  
 locating, 10-25  
 location counter, 4-14  
 locator  
     *calling via control program*, 10-27  
     *error output file*, 10-27  
     *invocation*, 10-4  
     *labels*, 10-29  
     *labels reference*, 10-30  
     *locating*, 10-25  
     *messages*, 10-27  
     *options summary*, 10-4  
     *output*, 10-27  
 locator control file, 10-25  
 locator options  
     -?, 10-5  
     -c, 10-6  
     -d, 10-7  
     -e, 10-8  
     -em, 10-9  
     -err, 10-10  
     -f, 10-11

-f *format*, 10-13  
 -H, 10-5  
 -M, 10-14  
 -N, 10-15  
 -o, 10-16  
 -p, 10-17  
 -S, 10-18  
 -s, 10-19  
 -V, 10-20  
 -v, 10-21  
 -w, 10-22  
 -WAE, 10-23  
 logical and, 5-14  
 logical not, 5-14  
 logical operators, 5-14  
 logical or, 5-14

## M

macro, 4-4  
     *.dup directive*, 6-11  
     *argument concatenation*, 4-6, 6-7  
     *call*, 6-5  
     *conditional assembly*, 6-12  
     *definition*, 6-4  
     *dummy argument operator*, 6-7  
     *dummy argument string*, 6-10  
     *local label*, 4-10, 6-11  
     *return hex value operator*, 6-9  
     *return value operator*, 6-8  
     *string delimiter*, 4-11  
 macro definition, F-28  
 macros  
     *parameterless*, F-28  
     *user defined*, F-28  
     *with parameters*, F-29  
 makefile, 11-13  
     *automatic creation of*, 1-12  
     *updating*, 1-12  
 map, Delfee keyword, F-8

- map file, 9-28, 10-14
  - default basename*, 9-17
- map keyword, F-57
- MAU, H-4
- mau, F-58
- mau (minimum addressable unit), F-9, F-10
- max, section attribute, 3-5
- mem, Delfee keyword, F-10
- mem keyword, F-59
- memory, F-60
  - external*, F-16
  - layout*, F-54
  - reserve*, F-62
  - scratch*, 11-44
- memory management instructions, 2-31
- messages
  - linker*, 9-37
  - locator*, 10-27
- minimum addressable unit, F-58
- minus operator, 5-11
- mktri, 11-13
  - .DEFAULT target*, 11-22
  - .DONE target*, 11-22
  - .IGNORE target*, 11-22
  - .INIT target*, 11-22
  - .PRECIOUS target*, 11-22
  - .SILENT target*, 11-22
  - .SUFFIXES target*, 11-22
  - comment lines*, 11-15
  - conditional processing*, 11-16
  - exist function*, 11-20
  - export line*, 11-16
  - functions*, 11-19
  - ifdef*, 11-16
  - implicit rules*, 11-24
  - include line*, 11-16
  - macro definition*, 11-15
  - macro MAKE*, 11-17
  - macro MAKEFLAGS*, 11-18
  - macro PRODDIR*, 11-18
  - macro SHELLCMD*, 11-18
  - macro TMP\_CCOPT*, 11-18
  - macro TMP\_CCPRG*, 11-18
  - macros*, 11-17
  - makefiles*, 11-15
  - match function*, 11-19
  - nexist function*, 11-21
  - protect function*, 11-20
  - rules in makefile*, 11-22
  - separate function*, 11-20
  - special macros*, 11-17
  - special targets*, 11-22
  - targets*, 11-21
- mmu, assembler control, 8-13
- module, 3-3
  - symbols*, 3-3
- modulo, 5-12
- monadic functions, H-8
- MUFOM, H-3
  - AD command*, H-12
  - AS command*, H-18
  - AT command*, H-16
  - checksum command*, H-12
  - command codes*, H-23
  - comment command*, H-12
  - data types*, H-7
  - DT command*, H-11
  - first byte of language elements*, H-22
  - function codes*, H-22
  - functions*, H-22
  - IR command*, H-18
  - LD command*, H-18
  - letters*, H-22
  - LI command*, H-21
  - loading commands*, H-18
  - LR command*, H-19
  - LX command*, H-21
  - MB command*, H-11
  - ME command*, H-11
  - module level commands*, H-11
  - NI command*, H-15
  - NN command*, H-16
  - NX command*, H-16
  - processes*, H-4

*RE command*, H-20  
*RI command*, H-20  
*SA command*, H-15  
*SB command*, H-13  
*sections*, H-13  
*ST command*, H-13  
*TY command*, H-17  
*value assignment*, H-18  
*WX command*, H-20  
multiplication, 5-12

## N

nuclear, section attribute, 3-5  
nopaging, assembler control, 8-20  
not equal operator, 5-13  
number, 5-6  
    *binary*, 5-6  
    *decimal*, 5-6  
    *hexadecimal*, 5-7  
numeric label, 4-3

## O

object, assembler control, 8-14  
object file, 2-4, 2-6  
    *displaying parts of*, 11-29  
    *external part*, 11-34  
object layer, 11-45  
object reader (prtri), 11-26  
operands, 5-3  
operators, 5-10  
    *precedence list*, 5-10  
options summary  
    *assembler aspcp*, 2-7  
    *assembler astri*, 2-5  
    *linker*, 9-4  
    *locator*, 10-4  
output buffer size, 10-24  
output file, 2-33, 9-18, 10-16  
output format, 10-13

## P

page, assembler control, 8-15, 8-16  
pagelength, assembler control, 8-18  
pagewidth, assembler control, 8-19  
paging, assembler control, 8-20  
parameterless macros, F-28  
parentheses, 5-6  
PCP assembler, 2-3  
plus operator, 5-11  
prctl, assembler control, 8-21  
predefined symbol  
    *\_ASPCP*, 5-8  
    *\_ASTRI*, 5-8  
prefix for global/external symbols,  
    2-34  
preprocessing, F-28  
preprocessor directives  
    *#define*, F-28  
    *#elif*, F-31  
    *#endif*, F-32  
    *#if*, F-31  
    *#ifdef*, F-31  
    *#ifndef*, F-31  
    *#include*, F-29  
    *#undef*, F-29  
print, assembler control, 8-22  
procedures, 11-40  
program development, 1-3  
project files, adding files, 1-11  
prototype, 9-36  
prtri, 1-3, 11-26  
    *display options*, 11-28  
    *file info*, 11-29  
    *global type info*, 11-36  
    *input control option*, 11-26  
    *options*  
        *-c*, 11-32  
        *-d*, 11-39  
        *-e*, 11-34  
        *-f file*, 11-26  
        *-g*, 11-36  
        *-h*, 11-29

- i*, 11-43
- ln*, 11-46
- s*, 11-30
- vn*, 11-49
- output control options*, 11-28
- section info*, 11-30

## R

- regsfr, F-61
- relational operators, 5-13
- relocatable object file, 9-3
- relocatable object module, 2-4, 2-6
- reserved, F-62
- reset, section activation attribute, 3-5
- return hex value operator, 6-9
- return value operator, 6-8

## S

- scratch cluster, F-40
- scripts, 1-20
- sdecl, 3-4
- section, 3-4, 10-31, F-63
  - absolute*, 3-6
  - activation*, 3-6, 7-44
  - activation attribute*, 3-5
  - alignment*, F-39
  - attribute*, 3-5, 7-42
    - max*, 3-5
  - attributes*, F-21
  - characteristics*, F-21
  - declaration*, 3-4
  - definition*, 7-42
  - examples*, 3-7
  - image*, 11-43
  - manipulation*, F-25
  - name*, 3-4
  - placing algorithm*, F-26
  - selection*, F-21
    - by attribute*, F-22

- by name*, F-22
- by special section*, F-22
- excluding*, F-23
- summary*, 2-38
- type*, 3-4, 7-42
- selection, F-64
- separator character, 9-25
- shift left operator, 5-12
- shift operators, 5-12
- shift right operator, 5-12
- sign operators, 5-11
- size, F-65
- software, F-66
- source line, removing from list file, 2-24
- source listing, addr field, 2-46
- space, F-67
  - definition*, F-10, F-19
  - generate code for*, 10-18
- src keyword, F-69
- stack, 10-33, F-70
- stack usage, 9-28
- start, F-71
- statement, 4-3
- stitle, assembler control, 8-23
- string, 5-7
  - DEFINE expansion*, 4-11
- symbol, 3-3, 5-7
  - global*, 3-3
  - local*, 3-3
  - predefined*, 5-8
- symbol character
  - return hex value*, 4-9
  - return value*, 4-8
- symbols, 11-40
- syntax for PCP assembler, 2-35, 2-36
- syntax of an expression, 5-6
- system libraries, 9-13, 9-14

# T

table, F-72  
 TC, assembler control, 8-24  
 temporary files, 1-17, 11-12  
 TIOF, H-3  
 title, assembler control, 8-25  
 TMPDIR, 1-17, 2-44, 11-12  
 toolchain, 1-3  
 TriCore2 instructions, 2-32  
 type  
   *basic*, 11-37  
   *mnemonic*, 11-38  
 type checking, 9-33  
   *between functions*, 9-35  
   *missing types*, 9-37  
   *recursive*, 9-34

# U

unary operator, 5-10  
 UNIX, scripts, 1-20

updating makefile, 1-12  
 user defined macros, F-28  
 utilities

*artri*, 11-4  
*cctri*, 11-7  
*mktri*, 11-13  
*prtri*, 11-26

# V

verbose, 2-40, 9-22, 10-21  
 verbose option, linker, 9-28  
 verbose option -vn, 11-49  
 version information, 2-39, 9-21, 10-20

# W

warning, assembler control, 8-26  
 warnings (suppress), 2-41, 9-23, 10-22  
 warnings as errors, 2-42, 9-24, 10-23