FILE* sfile;
int count = 0;

sfile = fopen(filename);

if (sfile == NULL)
    return -1;

while (1)
    
    char c;
    c = fgetc(sfile);
    if (c == EOF)
    
        break;
    
    else
    
        

return count;
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.
TABLE OF CONTENTS
### TRICORE C LANGUAGE 1-1

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Introduction</td>
<td>1–3</td>
</tr>
<tr>
<td>1.2 Data Types</td>
<td>1–4</td>
</tr>
<tr>
<td>1.3 Keywords</td>
<td>1–6</td>
</tr>
<tr>
<td>1.4 Function Qualifiers</td>
<td>1–9</td>
</tr>
<tr>
<td>1.5 Intrinsic Functions</td>
<td>1–12</td>
</tr>
<tr>
<td>1.5.1 Minimum and maximum of (Short) Integers</td>
<td>1–13</td>
</tr>
<tr>
<td>1.5.2 Fractional Arithmetic Support</td>
<td>1–14</td>
</tr>
<tr>
<td>1.5.3 Packed Data Type Support</td>
<td>1–15</td>
</tr>
<tr>
<td>1.5.4 Interrupt Handling</td>
<td>1–19</td>
</tr>
<tr>
<td>1.5.5 Insert Single Assembly Instruction</td>
<td>1–21</td>
</tr>
<tr>
<td>1.5.6 Register Handling</td>
<td>1–22</td>
</tr>
<tr>
<td>1.5.7 Insert / Extract Bit–fields and Bits</td>
<td>1–23</td>
</tr>
<tr>
<td>1.5.8 Miscellaneous Intrinsic Functions</td>
<td>1–25</td>
</tr>
<tr>
<td>1.6 Pragmas</td>
<td>1–26</td>
</tr>
<tr>
<td>1.7 Predefined Macros</td>
<td>1–31</td>
</tr>
</tbody>
</table>

### LIBRARIES 2-1

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Introduction</td>
<td>2–3</td>
</tr>
<tr>
<td>2.2 Library Functions</td>
<td>2–4</td>
</tr>
<tr>
<td>2.2.1 assert.h</td>
<td>2–4</td>
</tr>
<tr>
<td>2.2.2 complex.h</td>
<td>2–4</td>
</tr>
<tr>
<td>2.2.3 ctype.h and wctype.h</td>
<td>2–6</td>
</tr>
<tr>
<td>2.2.4 errno.h</td>
<td>2–7</td>
</tr>
<tr>
<td>2.2.5 fcntl.h</td>
<td>2–9</td>
</tr>
<tr>
<td>2.2.6 fenv.h</td>
<td>2–9</td>
</tr>
<tr>
<td>2.2.7 float.h</td>
<td>2–10</td>
</tr>
<tr>
<td>2.2.8 fss.h</td>
<td>2–10</td>
</tr>
<tr>
<td>2.2.9 inttypes.h and stdint.h</td>
<td>2–11</td>
</tr>
<tr>
<td>2.2.10 iso646.h</td>
<td>2–12</td>
</tr>
<tr>
<td>2.2.11 limits.h</td>
<td>2–12</td>
</tr>
<tr>
<td>2.2.12 locale.h</td>
<td>2–12</td>
</tr>
<tr>
<td>2.2.13 math.h and tgmath.h</td>
<td>2–13</td>
</tr>
<tr>
<td>2.2.14 setjmp.h</td>
<td>2–20</td>
</tr>
</tbody>
</table>
2.2.15 signal.h .................................................. 2–20
2.2.16 stdarg.h .................................................. 2–21
2.2.17 stdbool.h .................................................. 2–21
2.2.18 stddef.h .................................................. 2–22
2.2.19 stdint.h .................................................. 2–22
2.2.20 stdio.h and wchar.h ................................. 2–22
2.2.21 stdlib.h and wchar.h ................................. 2–33
2.2.22 string.h and wchar.h ............................... 2–37
2.2.23 time.h and wchar.h .................................... 2–41
2.2.24 Unistd.h .................................................. 2–44
2.2.25 wchar.h .................................................. 2–45
2.2.26 wctype.h .................................................. 2–46
2.3 C Library Reentrancy .................................... 2–48

TRICORE ASSEMBLY LANGUAGE 3–1

3.1 Introduction .............................................. 3–3
3.2 Built–in Assembly Functions .......................... 3–3
3.2.1 Overview of Built–in Assembly Functions ...... 3–3
3.2.2 Detailed Description of Built–in Assembly Functions ... 3–6
3.3 Assembler Directives and Controls .................. 3–19
3.3.1 Overview of Assembler Directives ............... 3–19
3.3.2 Detailed Description of Assembler Directives ... 3–21
3.3.3 Overview of Assembler Controls .................. 3–64
3.3.4 Detailed Description of Assembler Controls ...... 3–65

RUN–TIME ENVIRONMENT 4–1

4.1 Introduction .............................................. 4–3
4.2 Startup Code .......................................... 4–3
4.3 Stack Usage ............................................ 4–9
4.4 Heap Allocation ........................................ 4–10
4.5 Floating–Point Arithmetic .............................. 4–10
4.5.1 Compliance with IEEE–754 ......................... 4–11
4.5.2 Special Floating–Point Values .................... 4–13
# Table of Contents

## TOOL OPTIONS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Compiler Options</td>
<td>5–3</td>
</tr>
<tr>
<td>5.2</td>
<td>Assembler Options</td>
<td>5–67</td>
</tr>
<tr>
<td>5.3</td>
<td>Linker Options</td>
<td>5–106</td>
</tr>
<tr>
<td>5.4</td>
<td>Control Program Options</td>
<td>5–152</td>
</tr>
<tr>
<td>5.5</td>
<td>Make Utility Options</td>
<td>5–214</td>
</tr>
<tr>
<td>5.6</td>
<td>Archiver Options</td>
<td>5–243</td>
</tr>
</tbody>
</table>

## LIST FILE FORMATS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Assembler List File Format</td>
<td>6–3</td>
</tr>
<tr>
<td>6.2</td>
<td>Linker Map File Format</td>
<td>6–5</td>
</tr>
</tbody>
</table>

## OBJECT FILE FORMATS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>ELF/DWARF Object Format</td>
<td>7–3</td>
</tr>
<tr>
<td>7.2</td>
<td>Motorola S–Record Format</td>
<td>7–4</td>
</tr>
<tr>
<td>7.3</td>
<td>Intel Hex Record Format</td>
<td>7–8</td>
</tr>
</tbody>
</table>

## LINKER SCRIPT LANGUAGE

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1</td>
<td>Introduction</td>
<td>8–3</td>
</tr>
<tr>
<td>8.2</td>
<td>Structure of a Linker Script File</td>
<td>8–3</td>
</tr>
<tr>
<td>8.3</td>
<td>Syntax of the Linker Script Language</td>
<td>8–6</td>
</tr>
<tr>
<td>8.3.1</td>
<td>Preprocessing</td>
<td>8–6</td>
</tr>
<tr>
<td>8.3.2</td>
<td>Lexical Syntax</td>
<td>8–7</td>
</tr>
<tr>
<td>8.3.3</td>
<td>Identifiers</td>
<td>8–7</td>
</tr>
<tr>
<td>8.3.4</td>
<td>Expressions</td>
<td>8–8</td>
</tr>
<tr>
<td>8.3.5</td>
<td>Built–in Functions</td>
<td>8–9</td>
</tr>
<tr>
<td>8.3.6</td>
<td>LSL Definitions in the Linker Script File</td>
<td>8–11</td>
</tr>
<tr>
<td>8.3.7</td>
<td>Memory and Bus Definitions</td>
<td>8–11</td>
</tr>
<tr>
<td>8.3.8</td>
<td>Architecture Definition</td>
<td>8–13</td>
</tr>
</tbody>
</table>

---

4.5.3 Trapping Floating–Point Exceptions 4–13
4.5.4 Floating–Point Trap Handling API 4–15
MANUAL PURPOSE AND STRUCTURE

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

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

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

Structure
The TriCore documentation consists of a User’s Manual which includes a Getting Started section and a separate Reference Manual (this manual).

First you need to install the software. This is described in Chapter 1, Software Installation and Configuration, of the User’s Manual.

After installation you are ready to follow the Getting Started in Chapter 2 of the User’s Manual.

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

Once you are familiar with these tools, you can use the Reference Manual to lookup specific options and details to make fully use of the TriCore toolchain.
SHORT TABLE OF CONTENTS

Chapter 1: TriCore C Language
Contains overviews of all language extensions:
• Data types
• Keywords
• Function qualifiers
• Intrinsic functions
• Pragmas
• Predefined macros

Chapter 2: Libraries
Contains overviews of all library functions you can use in your C source. The libraries are implemented according to the ISO/IEC 9899:1999(E) standard.

Chapter 3: TriCore Assembly Language
Contains an overview of all assembly functions that you can use in your assembly source code.

Chapter 4: Run-time Environment
Contains a description of the C startup code and explains stack and heap usage and floating-point arithmetic.

Chapter 5: Tool Options
Contains a description of all tool options:
• Compiler options
• Assembler options
• Linker options
• Control program options
• Make utility options
• Archiver options

Chapter 6: List File Formats
Contains a description of the following list file formats:
• Assembler List File Format
• Linker Map File Format
Chapter 7: Object File Formats
Contains a description of the following object file formats:

- ELF/DWARF Object Formats
- Motorola S-Record Format
- Intel Hex Record Format

Chapter 8: Linker Script Language
Contains a description of the linker script language (LSL).

Chapter 9: CPU Functional Problems
Contains a description of the TASKING TriCore toolchain software solutions for functional problems and deviations from the electrical specifications and timing specifications for some TriCore derivatives.

Chapter 10: MISRA C Rules
Contains a description the supported and unsupported MISRA C code checking rules.
CONVENTIONS USED IN THIS MANUAL

Notation for syntax

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

**bold** Type this part of the syntax literally.

*italics* Substitute the italic word by an instance. For example:

*filename*

Type the name of a file in place of the word *filename*.

{} Encloses a list from which you must choose an item.

[] Encloses items that are optional. For example

```
ctc [-?]
```

Both *ctc* and *ctc -?* are valid commands.

| Separates items in a list. Read it as OR.

... You can repeat the preceding item zero or more times.

,... You can repeat the preceding item zero or more times, separating each item with a comma.

Example

```
ctc [option]... filename
```

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

```
ctc test.c
ctc -g test.c
ctc -g -E test.c
```

Not valid is:

```
ctc -g
```

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

The following illustrations are used in this manual:

*Note:* notes give you extra information.

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

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

*Command line: type your input on the command line.*

*Reference: follow this reference to find related topics.*
RELATED PUBLICATIONS

C Standards
  More information on the standards can be found at http://www.ansi.org

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

TASKING Tools
- TriCore C Compiler, Assembler, Linker User’s Manual [TASKING, MA060–024–00–00]
- TriCore C++ Compiler User’s Manual [TASKING, MA060–012–00–00]
- TriCore CrossView Pro Debugger User’s Manual [TASKING, MA060–043–00–00]

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

The TASKING TriCore C compiler fully supports the ANSI C standard but adds possibilities to program the special functions of the TriCore.

This chapter contains complete overviews of the following C language extensions of the TASKING TriCore C compiler:

- Data types
- Keywords
- Function qualifiers
- Intrinsic functions
- Pragmas
- Predefined macros
## 1.2 DATA TYPES

The TASKING TriCore C compiler `ctc` supports the following data types:

<table>
<thead>
<tr>
<th>Type</th>
<th>Keyword</th>
<th>Size (bit)</th>
<th>Align (bit)</th>
<th>Ranges</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bit</td>
<td>__bit</td>
<td>8</td>
<td>8</td>
<td>0 or 1</td>
</tr>
<tr>
<td>Boolean</td>
<td>_Bool</td>
<td>8</td>
<td>8</td>
<td>0 or 1</td>
</tr>
<tr>
<td>Character</td>
<td>char</td>
<td>8</td>
<td>8</td>
<td>(-2^7 \ldots 2^7 - 1)</td>
</tr>
<tr>
<td></td>
<td>signed char</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>unsigned char</td>
<td>8</td>
<td>8</td>
<td>(0 \ldots 2^8 - 1)</td>
</tr>
<tr>
<td>Integral</td>
<td>short</td>
<td>16</td>
<td>16</td>
<td>(-2^{15} \ldots 2^{15} - 1)</td>
</tr>
<tr>
<td></td>
<td>signed short</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>unsigned short</td>
<td>16</td>
<td>16</td>
<td>(0 \ldots 2^{16} - 1)</td>
</tr>
<tr>
<td></td>
<td>int</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>signed int</td>
<td></td>
<td></td>
<td>(-2^{31} \ldots 2^{31} - 1)</td>
</tr>
<tr>
<td></td>
<td>long</td>
<td>32</td>
<td>16</td>
<td>(-2^{31} \ldots 2^{31} - 1)</td>
</tr>
<tr>
<td></td>
<td>signed long</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>unsigned int</td>
<td>32</td>
<td>16</td>
<td>(0 \ldots 2^{32} - 1)</td>
</tr>
<tr>
<td></td>
<td>unsigned long</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>enum</td>
<td>8</td>
<td>8</td>
<td>(-2^7 \ldots 2^7 - 1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
<td></td>
<td>(-2^{15} \ldots 2^{15} - 1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>32</td>
<td></td>
<td>(-2^{31} \ldots 2^{31} - 1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8</td>
<td></td>
<td>(-2^7 \ldots 2^7 - 1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
<td></td>
<td>(-2^{15} \ldots 2^{15} - 1)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>32</td>
<td></td>
<td>(-2^{31} \ldots 2^{31} - 1)</td>
</tr>
<tr>
<td>Pointer</td>
<td>pointer to data</td>
<td>32</td>
<td>32</td>
<td>(0 \ldots 2^{32} - 1)</td>
</tr>
<tr>
<td></td>
<td>pointer to func</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Floating-Point</td>
<td>float</td>
<td>32</td>
<td>16</td>
<td>(-3.402e38 \ldots -1.175e-38)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(1.175e-38 \ldots 3.402e38)</td>
</tr>
<tr>
<td></td>
<td>double</td>
<td>64</td>
<td>32</td>
<td>(-1.797e308 \ldots -2.225e-308)</td>
</tr>
<tr>
<td></td>
<td>long double</td>
<td></td>
<td></td>
<td>(2.225e-308 \ldots 1.797e308)</td>
</tr>
<tr>
<td>Fract</td>
<td>__sfract</td>
<td>16</td>
<td>16</td>
<td>([-1, 1])</td>
</tr>
<tr>
<td></td>
<td>__fract</td>
<td>32</td>
<td>32</td>
<td>([-1, 1])</td>
</tr>
<tr>
<td>Type</td>
<td>Keyword</td>
<td>Size (bit)</td>
<td>Align (bit)</td>
<td>Ranges</td>
</tr>
<tr>
<td>-------</td>
<td>----------------</td>
<td>------------</td>
<td>-------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>Accum</td>
<td>__laccum</td>
<td>64</td>
<td>64</td>
<td>[-131072, 131072&gt;</td>
</tr>
<tr>
<td>Packed</td>
<td>__packb</td>
<td>32</td>
<td>16</td>
<td>4x: (-2^7 \ldots 2^7 - 1)</td>
</tr>
<tr>
<td></td>
<td>signed __packb</td>
<td></td>
<td></td>
<td>4x: (0 \ldots 2^8 - 1)</td>
</tr>
<tr>
<td></td>
<td>unsigned __packb</td>
<td></td>
<td></td>
<td>2x: (-2^{15} \ldots 2^{15} - 1)</td>
</tr>
<tr>
<td></td>
<td>__packhw</td>
<td>32</td>
<td>16</td>
<td>2x: (0 \ldots 2^{16} - 1)</td>
</tr>
</tbody>
</table>

*Table 1–1: Data Types*
1.3 Keywords

__a0, __a1, __a8, __a9

The data object is located in a section that is addressable with a sign-extended 16-bit offset from address register A0, A1, A8 or A9 respectively.

__asm() 

With the __asm() keyword you can use assembly instructions in the C source and pass C variables as operands to the assembly code.

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

**instruction_template** Assembly instructions that may contain parameters from the input list or output list in the form: `%parm_nr [regnum]

**%parm_nr[regnum]** Parameter number in the range 0 .. 9. With the optional .regnum you can access an individual register from a register pair or register quad. For example, with register pair d0/d1, .0 selects register d0.

**output_param_list** [[ "=constraint_char\(C\_expression\)],...]

**input_param_list** [[ "constraint_char\(C\_expression\)],...]

**&** Says that an output operand is written to before the inputs are read, so this output must not be the same register as any input.

**constraint_char** Constraint character: the type of register to be used for the C_expression.

**C_expression** Any C expression. For output parameters it must be an lvalue, that is, something that is legal to have on the left side of an assignment.

**register_save_list** [["register_name"],...]

**register_name** Name of the register you want to reserve.
<table>
<thead>
<tr>
<th>Constraint character</th>
<th>Type</th>
<th>Operand</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Address register</td>
<td>a0 .. a15</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>Data register</td>
<td>d0 .. d15</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>Data register pair</td>
<td>e0 .. e7</td>
<td></td>
</tr>
<tr>
<td>m</td>
<td>Memory</td>
<td>variable</td>
<td>Stack or memory operand</td>
</tr>
<tr>
<td>number</td>
<td>Type of operand it is associated with</td>
<td>same as %number</td>
<td>Indicates that %number and number are the same register.</td>
</tr>
</tbody>
</table>

Table 1–2: Available input/output operand constraints

For more information on __asm, see section 3.6, Using Assembly in the C Source, in Chapter TriCore C Language of the User's Manual.

__at()

With the attribute __at() you can place an object at an absolute address.

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

__atbit()

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

```c
__atbit(name, offset)
```

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

__circ

The TriCore C compiler supports the __circ keyword for circular buffers.

For more information see section 3.4.1, Circular Buffers, in Chapter TriCore C Language of the User’s Manual.
With keyword `__near` the declared data object will be located in the first 16 kB of a 256 MB block. These parts of memory are directly addressable with the absolute addressing mode.

With keyword `__far` the data object can be located anywhere in the indirect addressable memory region.

With the data type qualifiers `__sfrbit16` and `__sfrbit32` you can declare bit fields in special function registers. These keywords force 16-bit or 32-bit access.

For more information see section 3.4.2, *Declare an SFR Bit Field: __sfrbit16 and __sfrbit32*, in Chapter *TriCore C Language* of the *User’s Manual*. 

---

```

```
### 1.4 FUNCTION QUALIFIERS

**__enable__**  
**__bisr__()**

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

```c
__interrupt(vector) __enable_isr( void )
__trap(class) __enable_tsr( void )
```

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

```c
__interrupt(vector) __bisr(CCPN) isr( void )
__trap(class) __bisr(CCPN) tsr( void )
```

For more information see section 3.9.2, *Interrupt and Trap Functions*, in Chapter *TriCore C Language* of the *User’s Manual*.

**__indirect**

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

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

**inline**  
**__noinline**

You can use the *inline* qualifier to tell the compiler to inline the function body instead of calling the function. Use the **__noinline** qualifier to tell the compiler *not* to inline the function body.
inline int func1( void )
{
    // inline this function
}

__noinline int func2( void )
{
    // do not inline this function
}

For more information see section 3.9.1, Inlining Functions: inline, in Chapter TriCore C Language of the User’s Manual.

__interrupt()
__interrupt_fast()

You can use the qualifier __interrupt() to declare a function as an interrupt service routine.

    void __interrupt(vector_number) isr(void)
    {
        ...
    }

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

When you define an interrupt service routine with the __interrupt_fast() qualifier, the interrupt handler is directly placed in the interrupt vector table, thereby eliminating the jump code.

For more information see section 3.9.2, Interrupt and Trap Functions, in Chapter TriCore C Language of the User’s Manual.

__trap()
__trap_fast()
__syscallfunc()

The definition of a trap service routine is similar to the definition of an interrupt service routine. Trap functions cannot accept arguments and do not return anything:
void __trap( class ) tsr( void )
{
  ...
}

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

When you define a trap service routine with the `__trap_fast()` qualifier, the trap handler is directly placed in the trap vector table, thereby eliminating the jump code.

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

```
__syscallfunc(TIN)
```

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

For more information see section 3.9.2, *Interrupt and Trap Functions*, in Chapter *TriCore C Language* of the *User’s Manual*.

`__stackparm`

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

```
void __stackparm stack_func ( int );
```
1.5 INTRINSIC FUNCTIONS

The TASKING TriCore C compiler recognizes intrinsic functions that serve the following purposes:

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

All intrinsic functions begin with a double underscore character (__). You can use intrinsic functions as if they were ordinary C functions.
1.5.1 MINIMUM AND MAXIMUM OF (SHORT) INTEGERS

The next table provides an overview of the intrinsic functions that return the minium or maximum of a signed integer, unsigned integer or short integer.

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int __min( int, int )</td>
<td>Return minimum of two integers</td>
</tr>
<tr>
<td>short __mins( short, short )</td>
<td>Return minimum of two short integers</td>
</tr>
<tr>
<td>unsigned int __minu( unsigned int, unsigned int )</td>
<td>Return minimum of two unsigned integers</td>
</tr>
<tr>
<td>int __max( int, int )</td>
<td>Return maximum of two integers</td>
</tr>
<tr>
<td>short __maxs( short, short )</td>
<td>Return maximum of two short integers</td>
</tr>
<tr>
<td>unsigned int __maxu( unsigned int, unsigned int )</td>
<td>Return maximum of two unsigned integers</td>
</tr>
</tbody>
</table>

*Table 1-3: Intrinsic Functions for obtaining min/max values*
1.5.2 FRACTIONAL ARITHMETIC SUPPORT

The next table provides an overview of intrinsic functions to convert fractional values. Note that the TASKING TriCore C compiler fully supports the fractional type so normally you should not need these intrinsic functions (except for __mulfraclong). For compatibility reasons the TASKING TriCore C compiler does support these functions.

**Conversion of Fractional Values**

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>long __mulfraclong( __frac,long )</td>
<td>Integer part of __frac x long</td>
</tr>
<tr>
<td>__sfrac __round16( __frac )</td>
<td>Convert __frac to __sfrac</td>
</tr>
<tr>
<td>__frac __getfrac( __accum )</td>
<td>Convert __accum to __frac</td>
</tr>
<tr>
<td>short __clssf( __sfrac )</td>
<td>Count the consecutive number of bits that have the same value as bit 15 of an __sfrac</td>
</tr>
<tr>
<td>__sfrac __shasfracts( __sfrac,int )</td>
<td>Left/right shift of an __sfrac</td>
</tr>
<tr>
<td>__frac __shafracts( __frac,int )</td>
<td>Left/right shift of an __frac</td>
</tr>
<tr>
<td>__laccum __shaaccum( __laccum,int )</td>
<td>Left/right shift of an __laccum</td>
</tr>
</tbody>
</table>

*Table 1–4: Intrinsic Functions for Conversion of Fractional Values*
1.5.3 PACKED DATA TYPE SUPPORT

The next table provides an overview of the intrinsic functions for initialization of packed data type.

**Initialize Packed Data Types**

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__packb __initpackbl( long )</td>
<td>Initialize __packb with a long integer</td>
</tr>
<tr>
<td>__packb __initpackb( int,int,int,int )</td>
<td>Initialize __packb with four integers</td>
</tr>
<tr>
<td>unsigned __packb __initupackb(</td>
<td>Idem, but unsigned</td>
</tr>
<tr>
<td>unsigned,unsigned,unsigned,unsigned)</td>
<td></td>
</tr>
<tr>
<td>__packhw __initpackhl( long )</td>
<td>Initialize __packhw with a long integer</td>
</tr>
<tr>
<td>__packhw __initpackhw( short,short )</td>
<td>Initialize __packhw with two integers</td>
</tr>
<tr>
<td>unsigned __packhw __initupackhw(</td>
<td>Idem, but unsigned</td>
</tr>
<tr>
<td>unsigned short,unsigned short)</td>
<td></td>
</tr>
</tbody>
</table>

*Table 1-5: Intrinsic Functions to Initialize Packed Data Types*

**Extract Values from Packed Data Types**

The next table provides an overview of the intrinsic functions to extract a single byte or halfword from a __packb or __packhw data type.

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>char __extractbyte1( __packb )</td>
<td>Extract first byte from a __packb</td>
</tr>
<tr>
<td>unsigned char __extractbyte1(</td>
<td>Idem, but unsigned</td>
</tr>
<tr>
<td>__unsigned __packb )</td>
<td></td>
</tr>
<tr>
<td>char __extractbyte2( __packb )</td>
<td>Extract second byte from a __packb</td>
</tr>
<tr>
<td>unsigned char __extractbyte2(</td>
<td>Idem, but unsigned</td>
</tr>
<tr>
<td>__unsigned __packb )</td>
<td></td>
</tr>
<tr>
<td>char __extractbyte3( __packb )</td>
<td>Extract third byte from a __packb</td>
</tr>
<tr>
<td>unsigned char __extractbyte3(</td>
<td>Idem, but unsigned</td>
</tr>
<tr>
<td>__unsigned __packb )</td>
<td></td>
</tr>
<tr>
<td>char __extractbyte4( __packb )</td>
<td>Extract fourth byte from a __packb</td>
</tr>
<tr>
<td>unsigned char __extractbyte4(</td>
<td>Idem, but unsigned</td>
</tr>
<tr>
<td>__unsigned __packb )</td>
<td></td>
</tr>
<tr>
<td>Intrinsic Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>short __extracthw1( __packhw )</td>
<td>Extract first short from a __packhw</td>
</tr>
<tr>
<td>unsigned short __extractuhw1(</td>
<td>Idem, but unsigned</td>
</tr>
<tr>
<td>unsigned __packhw )</td>
<td></td>
</tr>
<tr>
<td>short __extracthw2( __packhw )</td>
<td>Extract second short from a __packhw</td>
</tr>
<tr>
<td>unsigned short __extractuhw2(</td>
<td>Idem, but unsigned</td>
</tr>
<tr>
<td>unsigned __packhw )</td>
<td></td>
</tr>
<tr>
<td>char __getbyte1( __packb * )</td>
<td>Extract first byte from a __packb</td>
</tr>
<tr>
<td>unsigned char __getubyte1(</td>
<td>Idem, but unsigned</td>
</tr>
<tr>
<td>unsigned __packb * )</td>
<td></td>
</tr>
<tr>
<td>char __getbyte2( __packb * )</td>
<td>Extract second byte from a __packb</td>
</tr>
<tr>
<td>unsigned char __getubyte2(</td>
<td>Idem, but unsigned</td>
</tr>
<tr>
<td>unsigned __packb * )</td>
<td></td>
</tr>
<tr>
<td>char __getbyte3( __packb * )</td>
<td>Extract third byte from a __packb</td>
</tr>
<tr>
<td>unsigned char __getubyte3(</td>
<td>Idem, but unsigned</td>
</tr>
<tr>
<td>unsigned __packb * )</td>
<td></td>
</tr>
<tr>
<td>char __getbyte4( __packb * )</td>
<td>Extract fourth byte from a __packb</td>
</tr>
<tr>
<td>unsigned char __getubyte4(</td>
<td>Idem, but unsigned</td>
</tr>
<tr>
<td>unsigned __packb * )</td>
<td></td>
</tr>
<tr>
<td>short __gethw1( __packhw * )</td>
<td>Extract first integer from a __packhw</td>
</tr>
<tr>
<td>unsigned short __getuhw1(</td>
<td>Idem, but unsigned</td>
</tr>
<tr>
<td>unsigned __packhw * )</td>
<td></td>
</tr>
<tr>
<td>short __gethw2( __packhw * )</td>
<td>Extract short integer from a __packhw</td>
</tr>
<tr>
<td>unsigned short __getuhw2(</td>
<td>Idem, but unsigned</td>
</tr>
<tr>
<td>unsigned __packhw * )</td>
<td></td>
</tr>
</tbody>
</table>

Table 1-6: Intrinsic Functions to Extract Values from Packed Data Types
**Insert Values into Packed Data Types**

The next table provides an overview of the intrinsic functions to insert a single byte or halfword into a `__packb` or `__packhw` data type.

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__packb __insertbyte1( __packb, char )</code></td>
<td>Insert char into first byte of a <code>__packb</code></td>
</tr>
<tr>
<td><code>unsigned __packb __insertubyte1( unsigned __packb, unsigned char )</code></td>
<td>Idem, but unsigned</td>
</tr>
<tr>
<td><code>__packb __insertbyte2( __packb, char )</code></td>
<td>Insert char into second byte of a <code>__packb</code></td>
</tr>
<tr>
<td><code>unsigned __packb __insertubyte2( unsigned __packb, unsigned char )</code></td>
<td>Idem, but unsigned</td>
</tr>
<tr>
<td><code>__packb __insertbyte3( __packb, char )</code></td>
<td>Insert char into third byte of a <code>__packb</code></td>
</tr>
<tr>
<td><code>unsigned __packb __insertubyte3( unsigned __packb, unsigned char )</code></td>
<td>Idem, but unsigned</td>
</tr>
<tr>
<td><code>__packb __insertbyte4( __packb, char )</code></td>
<td>Insert char into fourth byte of a <code>__packb</code></td>
</tr>
<tr>
<td><code>unsigned __packb __insertubyte4( unsigned __packb, unsigned char )</code></td>
<td>Idem, but unsigned</td>
</tr>
<tr>
<td><code>__packhw __inserthwl( __packhw, short )</code></td>
<td>Insert short into first halfword of a <code>__packhw</code></td>
</tr>
<tr>
<td><code>unsigned __packhw __insertuhwl( unsigned __packhw, unsigned short )</code></td>
<td>Idem, but unsigned</td>
</tr>
<tr>
<td><code>__packhw __inserthwl2( __packhw, short )</code></td>
<td>Insert short into second halfword of a <code>__packhw</code></td>
</tr>
<tr>
<td><code>unsigned __packhw __insertuhwl2( unsigned __packhw, unsigned short )</code></td>
<td>Idem, but unsigned</td>
</tr>
<tr>
<td><code>void __setbyte1( __packb *, char )</code></td>
<td>Insert first byte into a <code>__packb</code></td>
</tr>
<tr>
<td><code>void __setubyte1( unsigned __packb *, unsigned char )</code></td>
<td>Idem, but unsigned</td>
</tr>
<tr>
<td><code>void __setbyte2( __packb *, char )</code></td>
<td>Insert second byte into a <code>__packb</code></td>
</tr>
<tr>
<td><code>void __setubyte2( unsigned __packb *, unsigned char )</code></td>
<td>Idem, but unsigned</td>
</tr>
</tbody>
</table>
Intrinsic Function | Description
---|---
void _setbyte3( __packb *, char ) | Insert third byte into a __packb
void _setbyte3( unsigned __packb *, unsigned char ) | Idem, but unsigned
void _setbyte4( __packb *, char ) | Insert fourth byte into a __packb
void _setbyte4( unsigned __packb *, unsigned char ) | Idem, but unsigned
void _sethw1( __packhw *, short ) | Insert first integer into a __packhw
void _setuhw1( unsigned __packhw *, unsigned short ) | Idem, but unsigned
void _sethw2( __packhw *, short ) | Insert short integer into a __packhw
void _setuhw2( unsigned __packhw *, unsigned short ) | Idem, but unsigned

Table 1–7: Intrinsic Functions to Insert Values into Packed Data Types

Combine Packed Data Types into a Packed Word

The next table provides an overview of the intrinsic functions to combine the value of packed data types into a packed word. You can combine two __packb (2 x 4 bytes) into a long long or two __packhw (2 x 2 halfwords) into a long long.

The packed word is a double register that is represented by the additional datatype __packw. To access the values in a __packw variable, you can use a union data type: `typedef double __packw;`

These intrinsics are only supported for the TriCore2 (--is-tricore2).

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned long long __transpose_byte( __packb,__packb )</td>
<td>Combine two __packb</td>
</tr>
<tr>
<td>unsigned long long __transpose_hword( __packhw,__packhw )</td>
<td>Combine two __packhw</td>
</tr>
</tbody>
</table>

Table 1–8: Intrinsic Functions to Combine Packed Data Types
**Calculate Absolute Values of Packed Data Type Values**

The next table provides an overview of the intrinsic functions to calculate the absolute value of packed data type values.

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__packb __absb( __packb )</td>
<td>Absolute value of __packb</td>
</tr>
<tr>
<td>__packhw __absh( __packhw )</td>
<td>Absolute value of __packhw</td>
</tr>
<tr>
<td>__sat __packhw __abssh( __sat __packhw )</td>
<td>Absolute value of __packhw using saturation</td>
</tr>
</tbody>
</table>

*Table 1–9: Intrinsic Functions to Calculate Absolute Values*

**Calculate Minimum Packed Data Type Values**

The next table provides an overview of the intrinsic functions to calculate the minimum from two packed data type values.

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__packb __minb( __packb, __packb )</td>
<td>Minimum of two __packb values</td>
</tr>
<tr>
<td>unsigned __packb __minbu( unsigned __packb, unsigned __packb )</td>
<td>Minimum of two unsigned __packb values</td>
</tr>
<tr>
<td>__packhw __minh( __packhw, __packhw )</td>
<td>Minimum of two __packhw values</td>
</tr>
<tr>
<td>unsigned __packhw __minhu( unsigned __packhw, unsigned __packhw )</td>
<td>Minimum of two unsigned __packhw values</td>
</tr>
</tbody>
</table>

*Table 1–10: Intrinsic Functions to Calculate Absolute Values*

**1.5.4 INTERRUPT HANDLING**

The next table provides an overview of the intrinsic functions to read or set interrupt handling:

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void __enable ( void )</td>
<td>Enable interrupts immediately at function entry</td>
</tr>
<tr>
<td>void __disable ( void )</td>
<td>Disable interrupts. Only supported for TriCore1.</td>
</tr>
<tr>
<td>int __disable_and_save ( void )</td>
<td>Disable interrupts and return previous interrupt state (enabled or disabled). Only supported for TriCore2 (--is-tricore2).</td>
</tr>
<tr>
<td>void __restore ( int )</td>
<td>Restore interrupt state. Only supported for TriCore2 (--is-tricore2).</td>
</tr>
<tr>
<td>void __biset ( int )</td>
<td>Set CPU priority number [0..512] and enable interrupts immediately at function entry</td>
</tr>
<tr>
<td>void __syscc ( int )</td>
<td>Call a system call function number</td>
</tr>
</tbody>
</table>

*Table 1-11: Intrinsic Functions for Interrupt Handling*
### 1.5.5 INSERT SINGLE ASSEMBLY INSTRUCTION

The next table provides an overview of the intrinsic functions that you can use to insert a single assembly instruction.

You can also use inline assembly but these intrinsics provide a shorthand for frequently used assembly instructions.

See section 3.6, *Using Assembly in the C Source: `__asm()`* of the *User’s Manual*

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void __debug( void )</td>
<td>Insert DEBUG instruction</td>
</tr>
<tr>
<td>void __dsync( void )</td>
<td>Insert DSYNC instruction</td>
</tr>
<tr>
<td>void __isync( void )</td>
<td>Insert ISYNC instruction</td>
</tr>
<tr>
<td>void __svlcx( void )</td>
<td>Insert SVLCX instruction</td>
</tr>
<tr>
<td>void __rslcx( void )</td>
<td>Insert RSLCX instruction</td>
</tr>
<tr>
<td>void __nop( void )</td>
<td>Insert NOP instruction</td>
</tr>
</tbody>
</table>

*Table 1–12: Intrinsic Functions for Inserting Assembly Instructions*
1.5.6 REGISTER HANDLING

Access Control Registers

The next table provides an overview of the intrinsic functions that you can use to access control registers.

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int __mfcrr(int)</td>
<td>move contents of the addressed core SFR into a data register</td>
</tr>
<tr>
<td>void __mtcr(int,int)</td>
<td>move contents of a data register (second int) to the addressed core SFR (first int)</td>
</tr>
</tbody>
</table>

Table 1-13: Intrinsic Functions for Accessing Control Registers

Perform Register Value Operations

The next table provides an overview of the intrinsic functions that operate on a register and return a value in another register.

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int __clz(int)</td>
<td>Count leading zeros in int</td>
</tr>
<tr>
<td>int __clo(int)</td>
<td>Count leading ones in int</td>
</tr>
<tr>
<td>int __cls(int)</td>
<td>Count number of redundant sign bits (all consecutive bits with the same value as bit 31)</td>
</tr>
<tr>
<td>int __satb(int)</td>
<td>Return saturated byte</td>
</tr>
<tr>
<td>int __satbu(int)</td>
<td>Return saturated unsigned byte</td>
</tr>
<tr>
<td>int __sath(int)</td>
<td>Return saturated halfword</td>
</tr>
<tr>
<td>int __sathu(int)</td>
<td>Return saturated unsigned halfword</td>
</tr>
<tr>
<td>int __abs(int)</td>
<td>Return absolute value</td>
</tr>
<tr>
<td>int __abss(int)</td>
<td>Return absolute value with saturation</td>
</tr>
<tr>
<td>int __parity(int)</td>
<td>Return parity</td>
</tr>
</tbody>
</table>

Table 1-14: Intrinsic Functions for Performing Register Value Operations
1.5.7  INSERT / EXTRACT BIT-FIELDS AND BITS

Insert / Extract Bit-fields

The next table provides an overview of the intrinsic functions to insert or extract a bit-field.

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int __extr ( int value, int pos, int width )</td>
<td>Extract a bit-field (bit pos to bit pos+width) from value</td>
</tr>
<tr>
<td>unsigned int __extru ( int value, int pos, int width )</td>
<td>Same as __extr() but return bit-field as unsigned integer</td>
</tr>
<tr>
<td>int __insert ( int src, int trg, int pos, int width )</td>
<td>Extract bit-field (bit pos to bit pos+width) from src and insert it in trg.</td>
</tr>
<tr>
<td>int _ins( int trg, int trgbit, int src, int srcbit )</td>
<td>Return trg but replace trgbit by srcbit in src.</td>
</tr>
<tr>
<td>int _insn(int trg, int trgbit, int src, int srcbit )</td>
<td>Return trg but replace trgbit by inverse of srcbit in src.</td>
</tr>
</tbody>
</table>

Table 1–15: Intrinsic Functions to Insert / Extract Bit-fields

Atomic Load–Modify–Store

With the next intrinsic function you can perform atomic Load–Modify–Store of a bit-field from an integer value. This function uses the IMASK and LDMST instruction. The intrinsic writes the number of bits of an integer value at a certain address location in memory with a bitoffset. The number of bits must be a constant value.

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void __imaskldmst(int* address, int value, int bitoffset, int bits)</td>
<td></td>
</tr>
</tbody>
</table>

Store a single bit

With the intrinsic macro __putbit() you can store a single bit atomically in memory at a specified bit offset. The bit at offset 0 in value is stored at an address location in memory with a bitoffset.

This intrinsic is implemented as a macro definition which uses the __imaskldmst() intrinsic:

```c
#define __putbit ( value, address, bitoffset ) __imaskldmst ( address, value, bitoffset, 1 )
```
Load a single bit

With the intrinsic macro `__getbit()` you can load a single bit from memory at a specified bit offset. A bit value is loaded from an address location in memory with a bit offset and returned as an unsigned integer value.

This intrinsic is implemented as a macro definition which uses the `__extru()` intrinsic function:

```c
#define __getbit ( address, bitoffset ) __extru ( *(address), bitoffset, 1 )
```

### Intrinsic Macro

| void __putbit( int value, int* address, int bitoffset ) |

| unsigned integer __getbit( int* address, int bitoffset ) |
1.5.8 MISCELLANEOUS INTRINSIC FUNCTIONS

Multiply and Scale Back
The next intrinsic multiplies two 32-bit numbers to an intermediate 64-bit result, and scales back the result to 32 bits. To scale back the result, 32 bits are extracted from the intermediate 64-bit result: bit 63-offset to bit 31-offset.

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>int _mulsc( int a, int b, int offset )</td>
</tr>
</tbody>
</table>

Swap Mask
The next intrinsic exchanges the values of value and memory, but only those bits that are allowed by mask. Before the __swapmsk instruction is generated, the parameters value and mask are moved into a double register.

This intrinsic is only supported for the TriCore2 (is-tricore2).

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>void __swapmsk ( int value, int mask, int * memory )</td>
</tr>
</tbody>
</table>

Initialize Circular Pointer
With the next intrinsic you can initialize a circular pointer with a dynamically allocated buffer at run–time.

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>__circ void * __initcirc( void * buf, unsigned short bufsize, unsigned short byteindex )</td>
</tr>
</tbody>
</table>

See also Section 3.4.1, Circular Buffers, in Chapter TriCore C Language of the User's Manual.
1.6 PRAGMAS

Pragmas are keywords in the C source that control the behavior of the compiler. Pragmas overrule compiler options and keywords.

For general information on pragmas see section 3.7, Pragmas to Control the Compiler, in Chapter TriCore C Language of the User’s Manual.

The syntax is:

```
#pragma name-of-pragma
```

The compiler recognizes the following pragmas, other pragmas are ignored. Sometimes the resemblance of a pragma and a compiler option is so strong, that no explanation is given but instead is referred to the description of the corresponding compiler option.

```
#pragma CPU_functional_problem
#pragma TC112_functional_problem
#pragma TC113_functional_problem
```

Use software work arounds for the specified functional problem.

See compiler option --silicon-bug in section Compiler Options in Chapter Tool Options.

```
#pragma alias symbol=defined_symbol
```

Define symbol as an alias for defined_symbol. It corresponds to an equate directive (.EQU) at assembly level. The symbol should not be defined elsewhere, and defined_symbol should be defined with static storage duration (not extern or automatic).

See also the .EQU directive directive in Section 3.3, Assembler Directives and Controls, in Chapter Assembly Language.

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

See compiler option --align in section Compiler Options in Chapter Tool Options.

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

Performs 'clearing' or no 'clearing' of non-initialized static/public variables.
#pragma default_a0_size [value]
See compiler option `-Z` in section Compiler Options in Chapter Tool Options.

#pragma default_a1_size [value]
See compiler option `-Y` in section Compiler Options in Chapter Tool Options.

#pragma default_near_size [value]
See compiler option `-N` in section Compiler Options in Chapter Tool Options.

#pragma extension isuffix
Enables a language extension to specify imaginary floating-point constants. With this extension, you can use an "i" suffix on a floating-point constant, to make the type _Imaginary:

```c
float 0.5i
```

#pragma extern symbol
Normally, when you use the C keyword extern, the compiler generates an .EXTERN directive in the generated assembly source. However, if the compiler does not find any references to the extern symbol in the C module, it optimizes the assembly source by leaving the .EXTERN directive out.

With this pragma you force the compiler to generate the .EXTERN directive, creating an external symbol in the generated assembly source, even when the symbol is not used in the C module.

See the EXTERN directive directive in Section 3.3, Assembler Directives and Controls, in Chapter Assembly Language.
#pragma for_constant_data_use_memory memory
#pragma forExtern_data_use_memory memory
#pragma for_initialized_data_use_memory memory
#pragma for_uninitialized_data_use_memory memory

Use the specified memory for the type of data mentioned in the pragma name. You can specify the following memories:

near, far, a0, a8 or a9.

For #pragma for_constant_data_use_memory you can also specify the a1 memory.

This pragma overrules the pragmas #pragma default_a0_size, #pragma default_a1_size, #pragma default_near_size, and the memory qualifiers near and far.

#pragma indirect

Generates code for indirect function calling.

See compiler option --indirect in section Compiler Options in Chapter Tool Options.

#pragma indirect_runtime

Generates code for indirect calls to run-time functions.

See compiler option --indirect_runtime in section Compiler Options in Chapter Tool Options.

#pragma inline
#pragma noinline
#pragma smartinline


#pragma macro
#pragma nomacro

Turns macro expansion on or off. Default, macro expansion is turned on.

#pragma message "string"...

Print the message string(s) on standard output.
#pragma object_comment "string" ...

This pragma generates a .comment section in the assembly file with the specified string. After assembling, this string appears in the generated .o or .elf object file. If you specify this pragma more than once in the same module, only the last pragma has effect.

See compiler option --object-comment in section Compiler Options in Chapter Tool Options.

#pragma optimize flags
#pragma endoptimize
#pragma optimize restore

See section 5.3, Compiler Optimizations in Chapter Using the Compiler of the User’s Manual.

#pragma pack 2
#pragma pack 0

See section 3.2.4, Packed Data Types of the User’s Manual.

#pragma section all "section_name"
#pragma section section_type "section_name"
#pragma section code_init
#pragma section const_init
#pragma section vector_init
#pragma section data_overlay

See section 3.10, Compiler Generated Sections and compiler option -R in section Compiler Options in Chapter Tool Options.

#pragma source
#pragma nosource

See compiler option -s in section Compiler Options in Chapter Tool Options.
#pragma switch auto
#pragma switch jumptab
#pragma switch linear
#pragma switch lookup
#pragma switch restore

See section 3.11, Switch Statement of the User’s Manual and compiler option --switch in section Compiler Options in Chapter Tool Options.

#pragma tradeoff level

Specify whether the used optimizations should optimize for more speed (0), regardless of code size or for smaller code size (4), regardless of speed.

See also compiler option -t (--tradeoff) in section Compiler Options in Chapter Tool Options.

#pragma warning [number,...]

With this pragma you can disable warning messages. If you do not specify a warning number, all warnings will be suppressed.

See also compiler option -w (--no-warnings) in section Compiler Options in Chapter Tool Options.

#pragma weak symbol

Mark a symbol as "weak" (WEAK assembler directive). The symbol must have external linkage, which means a global or external object or function. A static symbol cannot be declared weak.

A weak external reference is resolved by the linker when a global (or weak) definition is found in one of the object files. However, a weak reference will not cause the extraction of a module from a library to resolve the reference. When a weak external reference cannot be resolved, the null pointer is substituted.

A weak definition can be overruled by a normal global definition. The linker will not complain about the duplicate definition, and ignore the weak definition.

See the .WEAK directive directive in Section 3.3, Assembler Directives and Controls, in Chapter Assembly Language.
1.7 **PREDEFINED MACROS**

In addition to the predefined macros required by the ISO C standard, the TASKING TriCore C compiler supports the predefined macros as defined in Table 1–16. The macros are useful to make conditional C code.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DOUBLE_FP</strong></td>
<td>Defined when you do not use compiler option -F (Treat double as float)</td>
</tr>
<tr>
<td><strong>SINGLE_FP</strong></td>
<td>Defined when you use compiler option -F (Treat double as float)</td>
</tr>
<tr>
<td><strong>FPU</strong></td>
<td>Defined when you use compiler option --fpu-present (Use hardware floating-point instructions)</td>
</tr>
<tr>
<td><strong>CTC</strong></td>
<td>Identifies the compiler. You can use this symbol to flag parts of the source which must be recognized by the ctc compiler only. It expands to the version number of the compiler.</td>
</tr>
<tr>
<td><strong>TASKING</strong></td>
<td>Identifies the compiler as the TASKING TriCore compiler. It expands to 1.</td>
</tr>
<tr>
<td><strong>DSPC</strong></td>
<td>Indicates conformance to the DSP–C standard. It expands to 1.</td>
</tr>
<tr>
<td><strong>DSPC_VERSION</strong></td>
<td>Expands to the decimal constant 200001L.</td>
</tr>
<tr>
<td><strong>VERSION</strong></td>
<td>Identifies the version number of the compiler. For example, if you use version 2.11 of the compiler, <strong>VERSION</strong> expands to 2001 (dot and revision number are omitted, minor version number in 3 digits).</td>
</tr>
<tr>
<td><strong>REVISION</strong></td>
<td>Identifies the revision number of the compiler. For example, if you use version 2.11 of the compiler, <strong>REVISION</strong> expands to 1.</td>
</tr>
<tr>
<td><strong>BUILD</strong></td>
<td>Identifies the build number of the compiler, composed of decimal digits for the build number, three digits for the major branch number and three digits for the minor branch number. For example, if you use build 1.22.1 of the compiler, <strong>BUILD</strong> expands to 1022001. If there is no branch number, the branch digits expand to zero. For example, build 127 results in 127000000.</td>
</tr>
</tbody>
</table>

*Table 1–16: Predefined macros*
CHAPTER 2

LIBRARIES

TASKING
2.1 INTRODUCTION

This chapter contains an overview of all library functions that you can call in your C source. This includes all functions of the standard C library (libc.a) and some functions of the floating-point library (libfp.a or libfpt.a).

Section 2.2, *Library Functions*, gives an overview of all library functions you can use, grouped per header file. A number of functions declared in wchar.h are parallel to functions in other header files. These are discussed together.

Section 2.3, *C Library Reentrancy*, gives an overview of which functions are reentrant and which are not.

The following libraries are included in the TriCore (ctc) toolchain. Both EDE and the control program ctc automatically select the appropriate libraries depending on the specified TriCore derivative.

<table>
<thead>
<tr>
<th>Library to link</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>libc.a</td>
<td>C library</td>
</tr>
<tr>
<td></td>
<td>(Some functions require the floating-point library. Also includes the startup code.)</td>
</tr>
<tr>
<td>libcs.a</td>
<td>C library single precision (compiler option --F)</td>
</tr>
<tr>
<td></td>
<td>(Some functions require the floating-point library. Also includes the startup code.)</td>
</tr>
<tr>
<td>libcs_fpu.a</td>
<td>C library single precision with FPU instructions (compiler option --F and --fpu-present)</td>
</tr>
<tr>
<td>libfp.a</td>
<td>Floating-point library (non-trapping)</td>
</tr>
<tr>
<td>libfpt.a</td>
<td>Floating-point library (trapping)</td>
</tr>
<tr>
<td></td>
<td>(Control program option --fp-trap)</td>
</tr>
<tr>
<td>libfp_fpu.a</td>
<td>Floating-point library (non-trapping, with FPU instructions)</td>
</tr>
<tr>
<td></td>
<td>(Compiler option --fpu-present)</td>
</tr>
<tr>
<td>libfpt_fpu.a</td>
<td>Floating-point library (trapping, with FPU instructions)</td>
</tr>
<tr>
<td></td>
<td>(Control program option --fp-trap, compiler option --fpu-present)</td>
</tr>
<tr>
<td>librt.a</td>
<td>Run-time library</td>
</tr>
</tbody>
</table>

*Table 2-1: Overview of libraries*
2.2 LIBRARY FUNCTIONS

The tables in the sections below list all library functions, grouped per header file in which they are declared. Some functions are not completely implemented because their implementation depends on the context where your application will run. These functions are for example all I/O related functions. Where possible, these functions are implemented using file system simulation (FSS). This system can be used by CrossView Pro to simulate an I/O environment which enables you to debug your application.

2.2.1 ASSERT.H

assert(expr)  Prints a diagnostic message if NDEBUG is not defined.
(Implemented as macro)

2.2.2 COMPLEX.H

The complex number \( z \) is also written as \( x+yi \) where \( x \) (the real part) and \( y \) (the imaginary part) are real numbers of types float, double or long double. The real and imaginary part can be stored in structs or in arrays. This implementation uses arrays because structs may have different alignments.

The header file complex.h also defines the following macros for backward compatibility:

```c
complex _Complex  /* C99 keyword */
imaginary _Imaginary  /* C99 keyword */
```

Parallel sets of functions are defined for double, float and long double. They are respectively named function, functionf, functionl. All long type functions, though declared in complex.h, are implemented as the double type variant which nearly always meets the requirement in embedded applications.

This implementation uses the obvious implementation for complex multiplication; and a more sophisticated implementation for division and absolute value calculations which handles underflow, overflow and infinities with more care. The ISO/IEC 9899 #pragma CX_LIMITED_RANGE therefore has no effect.
**Trigonometric functions**

- `csin` `csinf` `csinl` Returns the complex sine of `z`.
- `ccos` `ccosf` `ccosl` Returns the complex cosine of `z`.
- `ctan` `ctanf` `ctanl` Returns the complex tangent of `z`.
- `casin` `casinf` `casinl` Returns the complex arc sine `sin^{-1}(z)`.
- `cacos` `cacosf` `cacosl` Returns the complex arc cosine `cos^{-1}(z)`.
- `catan` `catanf` `catanl` Returns the complex arc tangent `tan^{-1}(z)`.
- `casinh` `casinhf` `casinhl` Returns the complex hyperbolic sine of `z`.
- `cacosinh` `cacosinhf` `cacosinhl` Returns the complex hyperbolic cosine of `z`.
- `catanh` `catanhf` `catanhll` Returns the complex hyperbolic tangent of `z`.
- `casinhc` `casinhcf` `casinhcfl` Returns the complex arc hyperbolic sine of `z`.
- `cacoshc` `cacoshc` `cacoshcl` Returns the complex arc hyperbolic cosine of `z`.
- `catanhc` `catanhc` `catanhhc` Returns the complex arc hyperbolic tangent of `z`.

**Exponential and logarithmic functions**

- `cexp` `cexpf` `cexpl` Returns the result of the complex exponential function `e^z`.
- `clog` `clogf` `clogl` Returns the complex natural logarithm.

**Power and absolute-value functions**

- `cabs` `cabsf` `cabsl` Returns the complex absolute value of `z` (also known as `norm, modulus or magnitude`).
- `cpow` `cpowf` `cpowl` Returns the complex value of `z` raised to the power `w` (`z^w`) where both `z` and `w` are complex numbers.
- `csqrt` `csqrtf` `csqrtl` Returns the complex square root of `z`.

**Manipulation functions**

- `carg` `cargf` `cargl` Returns the argument of `z` (also known as `phase angle`).
- `cimag` `cimagf` `cimagl` Returns the imaginary part of `z` as a real (respectively as a `double, float, long double`)
- `conj` `conjf` `conjl` Returns the complex conjugate value (the sign of its imaginary part is reversed).
cproj  cprojf  cprojl  Returns the value of the projection of z onto the Riemann sphere.
creal  crealf  creall  Returns the real part of z (respectively as a double, float, long double)

2.2.3  CTYPES.H AND WCTYPES.H

The header file `ctype.h` declares the following functions which take a character `c` as an integer type argument. The header file `wctype.h` declares parallel wide-character functions which take a character `c` of the `wchar_t` type as argument.

<table>
<thead>
<tr>
<th>Ctype.h</th>
<th>Wctype.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isalnum</td>
<td>iswalnum</td>
<td>Returns a non-zero value when <code>c</code> is an alphabetic character or a number ([A-Z][a-z][0-9]).</td>
</tr>
<tr>
<td>isalpha</td>
<td>iswalpha</td>
<td>Returns a non-zero value when <code>c</code> is an alphabetic character ([A-Z][a-z]).</td>
</tr>
<tr>
<td>isblank</td>
<td>iswblank</td>
<td>Returns a non-zero value when <code>c</code> is a blank character (tab, space...)</td>
</tr>
<tr>
<td>iscntrl</td>
<td>iswcntrl</td>
<td>Returns a non-zero value when <code>c</code> is a control character.</td>
</tr>
<tr>
<td>isdigit</td>
<td>iswdigit</td>
<td>Returns a non-zero value when <code>c</code> is a numeric character ([0-9]).</td>
</tr>
<tr>
<td>isgraph</td>
<td>iswgraph</td>
<td>Returns a non-zero value when <code>c</code> is printable, but not a space.</td>
</tr>
<tr>
<td>islower</td>
<td>iswlower</td>
<td>Returns a non-zero value when <code>c</code> is a lowercase character ([a-z]).</td>
</tr>
<tr>
<td>isprint</td>
<td>iswprint</td>
<td>Returns a non-zero value when <code>c</code> is printable, including spaces.</td>
</tr>
<tr>
<td>ispunct</td>
<td>iswpunct</td>
<td>Returns a non-zero value when <code>c</code> is a punctuation character (such as ', ', '!').</td>
</tr>
<tr>
<td>isspace</td>
<td>iswspace</td>
<td>Returns a non-zero value when <code>c</code> is a space type character (space, tab, vertical tab, formfeed, linefeed, carriage return).</td>
</tr>
<tr>
<td>isupper</td>
<td>iswupper</td>
<td>Returns a non-zero value when <code>c</code> is an uppercase character ([A-Z]).</td>
</tr>
<tr>
<td>isxdigit</td>
<td>iswxdigit</td>
<td>Returns a non-zero value when <code>c</code> is a hexadecimal digit ([0-9][A-F][a-f]).</td>
</tr>
<tr>
<td>Ctype.h</td>
<td>Wctype.h</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>tolower</td>
<td>towlower</td>
<td>Returns c converted to a lowercase character if it is an uppercase character, otherwise c is returned.</td>
</tr>
<tr>
<td>toupper</td>
<td>towupper</td>
<td>Returns c converted to an uppercase character if it is a lowercase character, otherwise c is returned.</td>
</tr>
<tr>
<td>_tolower</td>
<td>-</td>
<td>Converts c to a lowercase character, does not check if c really is an uppercase character. Implemented as macro. This macro function is not defined in ISO/IEC 9899.</td>
</tr>
<tr>
<td>_toupper</td>
<td>-</td>
<td>Converts c to an uppercase character, does not check if c really is a lowercase character. Implemented as macro. This macro function is not defined in ISO/IEC 9899.</td>
</tr>
<tr>
<td>isascii</td>
<td></td>
<td>Returns a non-zero value when c is in the range of 0 and 127. This function is not defined in ISO/IEC 9899.</td>
</tr>
<tr>
<td>toascii</td>
<td></td>
<td>Converts c to an ASCII value (strip highest bit). This function is not defined in ISO/IEC 9899.</td>
</tr>
</tbody>
</table>

### 2.2.4 ERRNO.H

```c
int errno    // External variable that holds implementation defined error codes.
```

The following error codes are defined as macros in `errno.h`:
EZERO  0  No error
EPERM  1  Not owner
ENOENT  2  No such file or directory
EINTR  3  Interrupted system call
EIO    4  I/O error
EBADF  5  Bad file number
EAGAIN 6  No more processes
ENOMEM  7  Not enough core
EACCES  8  Permission denied
EFAULT  9  Bad address
EXIST  10  File exists
ENOTDIR 11  Not a directory
EISDIR  12  Is a directory
EINVAL  13  Invalid argument
ENFILE  14  File table overflow
EMFILE  15  Too many open files
ETXTBSY 16  Text file busy
ENOSPC  17  No space left on device
ESPIPE  18  Illegal seek
EROFS  19  Read-only file system
EPIPE  20  Broken pipe
ELOOP  21  Too many levels of symbolic links
ENAMETOOLONG  22  File name too long

Floating-point errors

EDOM  23  Argument too large
ERANGE  24  Result too large

Errors returned by printf/scanf

ERR_FORMAT  25  Illegal format string for printf/scanf
ERR_NOFLOAT 26  Floating-point not supported
ERR_NOLONG  27  Long not supported
ERR_NOPUNCT 28  Pointers not supported

Error returned by file positioning routines

ERR_POS  29  Positioning failure

Encoding error stored in errno by functions like fgetwc, getwc, mbtowc, etc ...

EILSEQ  30  Illegal byte sequence (including too few bytes)
2.2.5 FCNTL.H

The header file fcntl.h contains the function open(), which calls the
low level function _open(), and definitions of flags used by the low level
function _open(). This header file is not defined in ISO/IEC9899.

open  Opens a file a file for reading or writing. Calls _open.
      (FSS implementation)

2.2.6 FENV.H

Contains mechanisms to control the floating-point environment. The functions in this header file are not implemented.

fegetenv  Stores the current floating-point environment.
          (Not implemented)
feholdexcept  Saves the current floating-point environment and installs
               an environment that ignores all floating-point exceptions.
               (Not implemented)
fesetenv  Restores a previously saved (fegetenv or feholdexcept)
          floating-point environment. (Not implemented)
feupdateenv  Saves the currently raised floating-point exceptions, restores a previously saved floating-point environment
           and finally raises the saved exceptions.
            (Not implemented)
feclearexcept  Clears the current exception status flags corresponding to
               the flags specified in the argument. (Not implemented)
fegetexceptflag  Stores the current setting of the floating-point status
                flags. (Not implemented)
feraiseexcept  Raises the exceptions represented in the argument. As a
              result, other exceptions may be raised as well.
               (Not implemented)
fesetexceptflag  Sets the current floating-point status flags.
                (Not implemented)
fetestexcept  Returns the bitwise-OR of the exception macros corre-
               sponding to the exception flags which are currently set
               and are specified in the argument. (Not implemented)

For each supported exception, a macro is defined. The following exceptions are defined:
2.2.7 FLOAT.H

The header file float.h defines the characteristics of the real floating-point types float, double and long double.

Float.h used to contain prototypes for the functions copysign(f), isninf(f), isfinite(f), isnan(f) and scalb(f). These functions have accordingly to the ISO/IEC9899 standard been moved to the header file math.h. See also section 2.2.13, Math.h and Tgmath.h.

2.2.8 FSS.H

The header file fss.h contains definitions and prototypes for low level I/O functions used for CrossView Pro’s file system simulation (FSS). The low level functions are also declared in stdio.h; they are all implemented as FSS functions. This header file is not defined in ISO/IEC9899.

<table>
<thead>
<tr>
<th>Stdio.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_fss_break(void)</td>
<td>Buffer and breakpoint functions for CrossView Pro.</td>
</tr>
<tr>
<td>_fss_init(fd,is_close)</td>
<td>Opens file descriptors 0 (stdin), 1 (stdout) and 2 (stderr) and associates them with terminal window FSS 0 of CrossView Pro.</td>
</tr>
<tr>
<td>_close(fd)</td>
<td></td>
</tr>
<tr>
<td>_lseek(fd,offset,whence)</td>
<td></td>
</tr>
<tr>
<td>_open(fd,flags)</td>
<td></td>
</tr>
<tr>
<td>_read(fd,*buff,cnt)</td>
<td></td>
</tr>
<tr>
<td>_unlink(*name)</td>
<td>See Low Level File Access Functions in section 2.2.20, Stdio.h.</td>
</tr>
<tr>
<td>_write(fd,*buffer,cnt)</td>
<td></td>
</tr>
</tbody>
</table>
2.2.9 INTTYPES.H AND STDINT.H

The header files stdint.h and inttypes.h provide additional declarations for integer types and have various characteristics. The stdint.h header file contains basic definitions of integer types of certain sizes, and corresponding sets of macros. This header file clearly refers to the corresponding sections in the ISO/IEC 9899 standard. The inttypes.h header file includes stdint.h and adds portable formatting and conversion functions. Below the conversion functions from inttypes.h are listed.

```c
intmax_t imaxabs(intmax_t j);
imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);

intmax_t strtoimax(const char * restrict nptr, char ** restrict endptr, int base);
uintmax_t strtomax(const char * restrict nptr, char ** restrict endptr, int base);
intmax_t wcstoiwmax(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);
uintmax_t wcstouwmax(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);
```

- `intmax_t imaxabs(intmax_t j);` Returns the absolute value of j
- `imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);` Computes numer/denom and numer % denom. The result is stored in the quot and rem components of the imaxdiv_t structure type.
- `intmax_t strtoimax(const char * restrict nptr, char ** restrict endptr, int base);` Convert string to maximum sized integer. (Compare strtol)
- `uintmax_t strtomax(const char * restrict nptr, char ** restrict endptr, int base);` Convert string to maximum sized unsigned integer. (Compare strtoul)
- `intmax_t wcstoiwmax(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);` Convert wide string to maximum sized integer. (Compare wcstoll)
- `uintmax_t wcstouwmax(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);` Convert wide string to maximum sized unsigned integer. (Compare wcstoul)
2.2.10 ISO646.H

The header file iso646.h adds tokens that can be used instead of regular operator tokens.

```
#define and   &&
#define and_eq &=
#define bitand &
#define bitor |
#define compl ~
#define not   !
#define not_eq !=
#define or    |
#define or_eq  |
#define xor   ^
#define xor_eq ^=
```

2.2.11 LIMITS.H

Contains the sizes of integral types, defined as macros.

2.2.12 LOCALE.H

To keep C code reasonable portable across different languages and cultures, a number of facilities are provided in the header file local.h.

```
char *setlocale( int category, const char *locale )
```

The function above changes locale-specific features of the run-time library as specified by the category to change and the name of the locale.

The following categories are defined and can be used as input for this function:

```
LC_ALL           0   LC_NUMERIC         3
LC_COLLATE       1   LC_TIME            4
LC_CTYPE         2   LC_MONETARY        5
```
struct *localeconv( void )

Returns a pointer to type `struct lconv` with values appropriate for the formatting of numeric quantities according to the rules of the current locale. The `struct lconv` in this header file is conforming the ISO standard.

### 2.2.13 MATH.H AND TGMATH.H

The header file `math.h` contains the prototypes for many mathematical functions. Before C99, all functions were computed using the double type (the float was automatically converted to double, prior to calculation). In this C99 version, parallel sets of functions are defined for double, float and long double. They are respectively named `function`, `functionf`, `functionl`. All `long` type functions, though declared in `math.h`, are implemented as the `double` type variant which nearly always meets the requirement in embedded applications.

The header file `tgmath.h` contains parallel type generic math macros whose expansion depends on the used type. `tgmath.h` includes `math.h` and the effect of expansion is that the correct `math.h` functions are called. The type generic macro, if available, is listed in the second column of the tables below.

### Trigonometric functions

<table>
<thead>
<tr>
<th>Math.h</th>
<th>Tgmath.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sin</td>
<td>sin</td>
<td>Returns the sine of x.</td>
</tr>
<tr>
<td>cos</td>
<td>cos</td>
<td>Returns the cosine of x.</td>
</tr>
<tr>
<td>tan</td>
<td>tan</td>
<td>Returns the tangent of x.</td>
</tr>
<tr>
<td>asin</td>
<td>asin</td>
<td>Returns the arc sine ( \sin^{-1}(x) ) of x.</td>
</tr>
<tr>
<td>acos</td>
<td>acos</td>
<td>Returns the arc cosine ( \cos^{-1}(x) ) of x.</td>
</tr>
<tr>
<td>atan</td>
<td>atan</td>
<td>Returns the arc tangent ( \tan^{-1}(x) ) of x.</td>
</tr>
<tr>
<td>atan2</td>
<td>atan2f</td>
<td>Returns the result of: ( \tan^{-1}(y/x) ).</td>
</tr>
<tr>
<td>sinh</td>
<td>sinhf</td>
<td>Returns the hyperbolic sine of x.</td>
</tr>
<tr>
<td>cosh</td>
<td>coshf</td>
<td>Returns the hyperbolic cosine of x.</td>
</tr>
<tr>
<td>tanh</td>
<td>tanhf</td>
<td>Returns the hyperbolic tangent of x.</td>
</tr>
<tr>
<td>asinh</td>
<td>asinhf</td>
<td>Returns the arc hyperbolic sinus of x.</td>
</tr>
</tbody>
</table>
### Exponential and logarithmic functions

All of these functions are new in C99, except for \texttt{exp}, \texttt{log} and \texttt{log10}.

<table>
<thead>
<tr>
<th>Math.h</th>
<th>Tgmath.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{acosh}</td>
<td>acosh</td>
<td>Returns the non-negative arc hyperbolic cosine of ( x ).</td>
</tr>
<tr>
<td>\texttt{atanh}</td>
<td>atanh</td>
<td>Returns the arc hyperbolic tangent of ( x ).</td>
</tr>
<tr>
<td>\texttt{exp}</td>
<td>\texttt{exp}</td>
<td>Returns the result of the exponential function ( e^x ).</td>
</tr>
<tr>
<td>\texttt{exp2}</td>
<td>\texttt{exp2}</td>
<td>Returns the result of the exponential function ( 2^x ). \textit{(Not implemented)}</td>
</tr>
<tr>
<td>\texttt{expm1}</td>
<td>\texttt{expm1}</td>
<td>Returns the result of the exponential function ( e^x - 1 ). \textit{(Not implemented)}</td>
</tr>
<tr>
<td>\texttt{log}</td>
<td>\texttt{log}</td>
<td>Returns the natural logarithm ( \ln(x) ), ( x &gt; 0 ).</td>
</tr>
<tr>
<td>\texttt{log10}</td>
<td>\texttt{log10}</td>
<td>Returns the base-10 logarithm of ( x ), ( x &gt; 0 ).</td>
</tr>
<tr>
<td>\texttt{log1p}</td>
<td>\texttt{log1pf}</td>
<td>Returns the base-e logarithm of ( (1+x) ), ( x \neq -1 ). \textit{(Not implemented)}</td>
</tr>
<tr>
<td>\texttt{log2}</td>
<td>\texttt{log2}</td>
<td>Returns the base-2 logarithm of ( x ), ( x &gt; 0 ). \textit{(Not implemented)}</td>
</tr>
<tr>
<td>\texttt{ilogb}</td>
<td>\texttt{ilogbf}</td>
<td>Returns the signed exponent of ( x ) as an integer, ( x &gt; 0 ). \textit{(Not implemented)}</td>
</tr>
<tr>
<td>\texttt{logb}</td>
<td>\texttt{logbf}</td>
<td>Returns the exponent of ( x ) as a signed integer in value in floating-point notation, ( x &gt; 0 ). \textit{(Not implemented)}</td>
</tr>
</tbody>
</table>
## Rounding functions

<table>
<thead>
<tr>
<th>Math.h</th>
<th>Tgmath.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ceil</td>
<td>ceil</td>
<td>Returns the smallest integer not less than x, as a double.</td>
</tr>
<tr>
<td>floor</td>
<td>floor</td>
<td>Returns the largest integer not greater than x, as a double.</td>
</tr>
<tr>
<td>rint</td>
<td>rint</td>
<td>Returns the rounded integer value as an int according to the current rounding direction. See &lt;fenv.h&gt;. <em>(Not implemented)</em></td>
</tr>
<tr>
<td>lrint</td>
<td>lrint</td>
<td>Returns the rounded integer value as a long int according to the current rounding direction. See &lt;fenv.h&gt;. <em>(Not implemented)</em></td>
</tr>
<tr>
<td>llrint</td>
<td>llrint</td>
<td>Returns the rounded integer value as a long long int according to the current rounding direction. See &lt;fenv.h&gt;. <em>(Not implemented)</em></td>
</tr>
<tr>
<td>nearbyint</td>
<td>nearbyint</td>
<td>Returns the rounded integer value as a floating-point according to the current rounding direction. See &lt;fenv.h&gt;. <em>(Not implemented)</em></td>
</tr>
<tr>
<td>round</td>
<td>round</td>
<td>Returns the nearest integer value of x as int. <em>(Not implemented)</em></td>
</tr>
<tr>
<td>lround</td>
<td>lround</td>
<td>Returns the nearest integer value of x as long int. <em>(Not implemented)</em></td>
</tr>
<tr>
<td>llround</td>
<td>llround</td>
<td>Returns the nearest integer value of x as long long int. <em>(Not implemented)</em></td>
</tr>
<tr>
<td>trunc</td>
<td>trunc</td>
<td>Returns the truncated integer value x. <em>(Not implemented)</em></td>
</tr>
</tbody>
</table>
### Remainder after division

<table>
<thead>
<tr>
<th>Math.h</th>
<th>Tgmath.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmod</td>
<td>fmodl</td>
<td>fmodf</td>
</tr>
<tr>
<td>remainder</td>
<td>remainderl</td>
<td></td>
</tr>
<tr>
<td>remainderf</td>
<td></td>
<td></td>
</tr>
<tr>
<td>remquo</td>
<td>remquol</td>
<td>remquof</td>
</tr>
</tbody>
</table>

Returns the remainder \( r \) of \( x - ny \). \( n \) is chosen as \( \text{trunc}(x/y) \). \( r \) has the same sign as \( x \).

Returns the remainder \( r \) of \( x - ny \). \( n \) is chosen as \( \text{trunc}(x/y) \). \( r \) may not have the same sign as \( x \). *(Not implemented)*

Same as remainder. In addition, the argument *quo* is given a specific value (see ISO). *(Not implemented)*

### frexp, ldexp, modf, scalbn, scalbln

<table>
<thead>
<tr>
<th>Math.h</th>
<th>Tgmath.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>frexp</td>
<td>frexpl</td>
<td>frexp</td>
</tr>
<tr>
<td>ldexp</td>
<td>ldexpl</td>
<td>ldexpf</td>
</tr>
<tr>
<td>modf</td>
<td>modfl</td>
<td>modff</td>
</tr>
<tr>
<td>scalbn</td>
<td>scalbnl</td>
<td>scalbnf</td>
</tr>
<tr>
<td>scalbln</td>
<td>scalblnl</td>
<td>scalblnf</td>
</tr>
</tbody>
</table>

Splits a float \( x \) into fraction \( f \) and exponent \( n \), so that:
\[
f = 0.0 \text{ or } 0.5 \leq |f| \leq 1.0 \text{ and } f*2^n = x.
\]
Returns \( f \), stores \( n \).

Inverse of frexp. Returns the result of \( x*2^n \).
\((x \text{ and } n \text{ are both arguments})\).

Splits a float \( x \) into fraction \( f \) and integer \( n \), so that:
\[
|f| < 1.0 \text{ and } f+n=x.
\]
Returns \( f \), stores \( n \).

Computes the result of \( x*\text{FLT\_RADIUS}^n \), efficiently, not normally by computing \( \text{FLT\_RADIUS}^n \) explicitly.

Same as scalbn but with argument \( n \) as long int.
### Power and absolute-value functions

<table>
<thead>
<tr>
<th>Math.h</th>
<th>Tgmath.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cbrt</td>
<td>cbrtl</td>
<td>cbtf</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fabs</td>
<td>fabsl</td>
<td>fabsf</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fma</td>
<td>fmal</td>
<td>fmaf</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>hypot</td>
<td>hypotl</td>
<td>hypotf</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pow</td>
<td>powl</td>
<td>powf</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sqrt</td>
<td>sqrtl</td>
<td>sqrtf</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Manipulation functions: copysign, nan, nextafter, nexttoward

<table>
<thead>
<tr>
<th>Math.h</th>
<th>Tgmath.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>copysign</td>
<td>copysignl</td>
<td></td>
</tr>
<tr>
<td>copysignf</td>
<td></td>
<td>copysign</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Returns the value of (x) with the sign of (y).</td>
</tr>
<tr>
<td>nan</td>
<td>nanl</td>
<td>nanf</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nextafter</td>
<td>nextafterl</td>
<td></td>
</tr>
<tr>
<td>nextafterf</td>
<td></td>
<td>nextafter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Returns the next representable value in the specified format after (x) in the direction of (y). Returns (y) if (x=y). (Not implemented)</td>
</tr>
<tr>
<td>nexttoward</td>
<td>nexttowardl</td>
<td></td>
</tr>
<tr>
<td>nexttowardf</td>
<td></td>
<td>nexttoward</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Positive difference, maximum, minimum

<table>
<thead>
<tr>
<th>Math.h</th>
<th>Tgmath.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fdim</td>
<td>fdiml</td>
<td>fdimf</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Returns the positive difference between:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>(Not implemented)</em></td>
</tr>
<tr>
<td>fmax</td>
<td>fmaxl</td>
<td>fmaxf</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Returns the maximum value of their arguments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>(Not implemented)</em></td>
</tr>
<tr>
<td>fmin</td>
<td>fminl</td>
<td>fminf</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Returns the minimum value of their arguments.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>(Not implemented)</em></td>
</tr>
</tbody>
</table>

### Error and gamma *(Not implemented)*

<table>
<thead>
<tr>
<th>Math.h</th>
<th>Tgmath.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>erf</td>
<td>erfl</td>
<td>erff</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Computes the error function of ( x ).</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>(Not implemented)</em></td>
</tr>
<tr>
<td>erfc</td>
<td>erfcl</td>
<td>erfcf</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Computes the complementary error function of ( x ).</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>(Not implemented)</em></td>
</tr>
<tr>
<td>lgamma</td>
<td>lgamml</td>
<td>lgammaf</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Computes the ( \log_e</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>(Not implemented)</em></td>
</tr>
<tr>
<td>tgamma</td>
<td>tgammal</td>
<td>tgammaf</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Computes ( \Gamma(x) ).</td>
</tr>
<tr>
<td></td>
<td></td>
<td><em>(Not implemented)</em></td>
</tr>
</tbody>
</table>
Comparison macros

The next are implemented as macros. For any ordered pair of numeric values exactly one of the relationships – less, greater, and equal – is true. These macros are type generic and therefor do not have a parallel function in \texttt{tgmath.h}. All arguments must be expressions of real-floating type.

<table>
<thead>
<tr>
<th>Math.h</th>
<th>Tgmath.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isgreater</td>
<td></td>
<td>Returns the value of ((x) &gt; (y))</td>
</tr>
<tr>
<td>isgreaterequal</td>
<td></td>
<td>Returns the value of ((x) \geq (y))</td>
</tr>
<tr>
<td>isless</td>
<td></td>
<td>Returns the value of ((x) &lt; (y))</td>
</tr>
<tr>
<td>islessequal</td>
<td></td>
<td>Returns the value of ((x) \leq (y))</td>
</tr>
<tr>
<td>islessgreater</td>
<td></td>
<td>Returns the value of ((x) &lt; (y) \mid \mid (x) &gt; (y))</td>
</tr>
<tr>
<td>isunordered</td>
<td></td>
<td>Returns 1 if its arguments are unordered, 0 otherwise.</td>
</tr>
</tbody>
</table>

Classification macros

The next are implemented as macros. These macros are type generic and therefor do not have a parallel function in \texttt{tgmath.h}. All arguments must be expressions of real-floating type.

<table>
<thead>
<tr>
<th>Math.h</th>
<th>Tgmath.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fpclassify</td>
<td></td>
<td>Returns the class of its argument: \texttt{FP_INFINITE, FP_NAN, FP_NORMAL,}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\texttt{FP_SUBNORMAL} or \texttt{FP_ZERO}</td>
</tr>
<tr>
<td>isfinite</td>
<td></td>
<td>Returns a nonzero value if and only if its argument has a finite value</td>
</tr>
<tr>
<td>isinf</td>
<td></td>
<td>Returns a nonzero value if and only if its argument has an infinit value</td>
</tr>
<tr>
<td>isnan</td>
<td></td>
<td>Returns a nonzero value if and only if its argument has NaN value.</td>
</tr>
<tr>
<td>isnormal</td>
<td></td>
<td>Returns a nonzero value if an only if its argument has a normal value.</td>
</tr>
<tr>
<td>signbit</td>
<td></td>
<td>Returns a nonzero value if and only if its argument value is negative.</td>
</tr>
</tbody>
</table>
2.2.14 SETJMP.H

The setjmp and longjmp in this header file implement a primitive form of nonlocal jumps, which may be used to handle exceptional situations. This facility is traditionally considered more portable than signal.h.

```c
int setjmp(jmp_buf env)            // Records its caller's environment in env and returns 0.
void longjmp(jmp_buf env, int status) // Restores the environment previously saved with a call to setjmp().
```

2.2.15 SIGNAL.H

Signals are possible asynchronous events that may require special processing. Each signal is named by a number. The following signals are defined:

- `SIGINT` 1 Receipt of an interactive attention signal
- `SIGILL` 2 Detection of an invalid function message
- `SIGFPE` 3 An erroneous arithmetic operation (for example, zero divide, overflow)
- `SIGSEGV` 4 An invalid access to storage
- `SIGTERM` 5 A termination request sent to the program
- `SIGABRT` 6 Abnormal termination, such as is initiated by the abort function.

The next function sends the signal `sig` to the program:

```c
int raise(int sig)
```

The next function determines how subsequent signals will be handled:

```c
signalfunction *signal (int, signalfunction *);
```

The first argument specifies the signal, the second argument points to the signal–handler function or has one of the following values:

- `SIG_DFL` Default behaviour is used
- `SIG_IGN` The signal is ignored

The function returns the previous value of `signalfunction` for the specific signal, or `SIG_ERR` if an error occurs.
The facilities in this header file gives you a portable way to access variable arguments lists, such as needed for as printf and vfprintf. This header file contains the following macros:

va_arg(ap, type)  Returns the value of the next argument in the variable argument list. It's return type has the type of the given argument type. A next call to this macro will return the value of the next argument.

va_end(va_list ap)  This macro must be called after the arguments have been processed. It should be called before the function using the macro 'va_start' is terminated (ANSI specification).

va_start( va_list ap, lastarg );  This macro initializes ap. After this call, each call to va_arg() will return the value of the next argument. In our implementation, va_list cannot contain any bit type variables. Also the given argument lastarg must be the last non-bit type argument in the list.

This header file contains the following macro definitions. These names for boolean type and values are consistent with C++. You are allowed to #undefine or redefine the macros below.

#define bool _Bool
#define true 1
#define false 0
#define __bool_true_false_are_defined 1
2.2.18 STDDEF.H

This header file defines the types for common use:

```plaintext
ptrdiff_t    signed integer type of the result of subtracting two pointers.
size_t       unsigned integral type of the result of the sizeof operator.
wchar_t      integer type to represent character codes in large character sets.
```

Besides these types, the following macros are defined:

```plaintext
NULL                 expands to the null pointer constant
offsetof(_type,_member) expands to an integer constant expression with
type size_t that is the offset in bytes of _member within structure type _type.
```

2.2.19 STDINT.H

See section 2.2.9, inttypes.h and stdint.h

2.2.20 STDIO.H AND WCHAR.H

Types

The header file `stdio.h` contains for performing input and output. A number of also have a parallel wide character function or macro, defined in `wchar.h`. The header file `wchar.h` also `stdio.h`.

In the C language, many I/O facilities are based on the concept of streams. The `stdio.h` header file defines the data type `FILE` which holds the information about a stream. An `FILE` object is created with the function `fopen`. The pointer to this object is used as an argument in many of the in this header file. The `FILE` object can contain the following information:

- the current position within the stream
- pointers to any associated buffers
- indications of for read/write errors
- end of file indication

The header file also defines type `fpos_t` as an `unsigned long`.
## Macros

<table>
<thead>
<tr>
<th>Stdio.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUFSIZE 512</td>
<td>Size of the buffer used by the <code>setbuf/setvbuf</code> function: 512</td>
</tr>
<tr>
<td>EOF -1</td>
<td>End of file indicator.</td>
</tr>
<tr>
<td>WEOF UINTMAX</td>
<td>End of file indicator. \nNOTE: WEOF need not to be a negative number as long as its value does not correspond to a member of the wide character set. (Defined in <code>wchar.h</code>).</td>
</tr>
<tr>
<td>FOPEN_MAX</td>
<td>Number of files that can be opened simultaneously: 4 \nNOTE: According to ISO/IEC 9899 this value must be at least 8.</td>
</tr>
<tr>
<td>FILENAME_MAX 100</td>
<td>Maximum length of a filename: 100</td>
</tr>
<tr>
<td>_IOFBF</td>
<td>Expand to an integer expression, suitable for use as argument to the <code>setvbuf</code> function.</td>
</tr>
<tr>
<td>_IOLBF</td>
<td></td>
</tr>
<tr>
<td>_IONBF</td>
<td></td>
</tr>
<tr>
<td>L_tmpnam</td>
<td>Size of the string used to hold temporary file names: 8 (tmpxxxxxx)</td>
</tr>
<tr>
<td>TMP_MAX 0x8000</td>
<td>Maximum number of unique temporary filenames that can be generated: 0x8000</td>
</tr>
<tr>
<td>stderr</td>
<td>Expressions of type &quot;pointer to FILE&quot; that point to the FILE objects associated with standard error, input and output streams.</td>
</tr>
<tr>
<td>stdin</td>
<td></td>
</tr>
<tr>
<td>stdout</td>
<td></td>
</tr>
</tbody>
</table>
**Low level file access functions**

<table>
<thead>
<tr>
<th>Stdio.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_close(fd)</td>
<td>Used by the functions close and fclose. (FSS implementation)</td>
</tr>
<tr>
<td>_lseek(fd, offset, whence)</td>
<td>Used by all file positioning functions: fgetpos, fseek, fsetpos, ftell, rewind. (FSS implementation)</td>
</tr>
<tr>
<td>_open(fd, flags)</td>
<td>Used by the functions fopen and freopen. (FSS implementation)</td>
</tr>
<tr>
<td>_read(fd, *buff, cnt)</td>
<td>Reads a sequence of characters from a file. (FSS implementation)</td>
</tr>
<tr>
<td>_unlink(*name)</td>
<td>Used by the function remove. (FSS implementation)</td>
</tr>
<tr>
<td>_write(fd, *buffer, cnt)</td>
<td>Writes a sequence of characters to a file. (FSS implementation)</td>
</tr>
</tbody>
</table>

**File access**

<table>
<thead>
<tr>
<th>Stdio.h</th>
<th>Description</th>
</tr>
</thead>
</table>
| fopen(name, mode) | Opens a file for a given mode. Available modes are:  
  "r" read; open text file for reading  
  "w" write; create text file for writing; if the file already exists its contents is discarded  
  "a" append; open existing text file or create new text file for writing at end of file  
  "r+" open text file for update; reading and writing  
  "w+" create text file for update; previous contents if any is discarded  
  "a+" append; open or create text file for update, writes at end of file (FSS implementation) |
| fclose(name) | Flushes the data stream and closes the specified file that was previously opened with fopen. (FSS implementation) |
| fflush(name) | If stream is an output stream, any buffered but unwritten data is written. Else, the effect is undefined. (FSS implementation) |
**Stdio.h**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>freopen(name, mode, stream)</code></td>
<td>Similar to <code>fopen</code>, but rather than generating a new value of type <code>FILE *</code>, the existing value is associated with a new stream. <em>(FSS implementation)</em></td>
</tr>
<tr>
<td><code>setbuf(stream, buffer)</code></td>
<td>If buffer is NULL, buffering is turned off for the stream. Otherwise, <code>setbuf</code> is equivalent to:</td>
</tr>
<tr>
<td></td>
<td><code>(void) setbuf(stream, buf, _IOFBF, BUFSIZ)</code></td>
</tr>
<tr>
<td><code>setvbuf(stream, buffer, mode, size)</code></td>
<td>Controls buffering for the <code>stream</code>; this function must be called before reading or writing. <code>Mode</code> can have the following values:</td>
</tr>
</tbody>
</table>
|                          | `_IOFBF` causes full buffering  
|                          | `_IOLBF` causes line buffering of text files  
|                          | `_IONBF` causes no buffering  
|                          | If buffer is not NULL, it will be used as a buffer; otherwise a buffer will be allocated. `size` determines the buffer size. |

**Character input/output**

The **format** string of **printf** related functions can contain plain text mixed with conversion specifiers. Each conversion specifier should be preceded by a `%` character. The conversion specifier should be build in order:

- Flags (in any order):
  - specifies left adjustment of the converted argument.  
  + a number is always preceded with a sign character.  
+ has higher precedence than **space**.  

**space** a negative number is preceded with a sign, positive numbers with a space.  

0 specifies padding to the field width with zeros (only for numbers).  

# specifies an alternate output form. For o, the first digit will be zero. For x or X, ”0x” and ”0X” will be prefixed to the number. For e, E, f, g, G, the output always contains a decimal point, trailing zeros are not removed.
- A number specifying a minimum field width. The converted argument is printed in a field with at least the length specified here. If the converted argument has fewer characters than specified, it will be padded at the left side (or at the right when the flag ‘-’ was specified) with spaces. Padding to numeric fields will be done with zeros when the flag ‘0’ is also specified (only when padding left). Instead of a numeric value, also ‘*’ may be specified, the value is then taken from the next argument, which is assumed to be of type int.

- A period. This separates the minimum field width from the precision.

- A number specifying the maximum length of a string to be printed. Or the number of digits printed after the decimal point (only for floating-point conversions). Or the minimum number of digits to be printed for an integer conversion. Instead of a numeric value, also ‘*’ may be specified, the value is then taken from the next argument, which is assumed to be of type int.

- A length modifier ‘h’, ‘l’ or ‘L’. ‘h’ indicates that the argument is to be treated as a short or unsigned short number. ‘l’ should be used if the argument is a long integer. ‘L’ indicates that the argument is a long double.

Flags, length specifier, period, precision and length modifier are optional, the conversion character is not. The conversion character must be one of the following, if a character following ‘%’ is not in the list, the behavior is undefined:

<table>
<thead>
<tr>
<th>Character</th>
<th>Printed as</th>
</tr>
</thead>
<tbody>
<tr>
<td>d, i</td>
<td>int, signed decimal</td>
</tr>
<tr>
<td>o</td>
<td>int, unsigned octal</td>
</tr>
<tr>
<td>x, X</td>
<td>int, unsigned hexadecimal in lowercase or uppercase respectively</td>
</tr>
<tr>
<td>u</td>
<td>int, unsigned decimal</td>
</tr>
<tr>
<td>c</td>
<td>int, single character (converted to unsigned char)</td>
</tr>
<tr>
<td>s</td>
<td>char *, the characters from the string are printed until a NULL character is found. When the given precision is met before, printing will also stop</td>
</tr>
<tr>
<td>f</td>
<td>double</td>
</tr>
<tr>
<td>e, E</td>
<td>double</td>
</tr>
<tr>
<td>g, G</td>
<td>double</td>
</tr>
<tr>
<td>Character</td>
<td>Printed as</td>
</tr>
<tr>
<td>-----------</td>
<td>------------</td>
</tr>
<tr>
<td>n</td>
<td>int *, the number of characters written so far is written into the argument. This should be a pointer to an integer in default memory. No value is printed.</td>
</tr>
<tr>
<td>p</td>
<td>pointer (hexadecimal 24-bit value)</td>
</tr>
<tr>
<td>%</td>
<td>No argument is converted, a '%' is printed.</td>
</tr>
</tbody>
</table>

*Table 2–2: Printf conversion characters*

All arguments to the **scanf** related functions should be pointers to variables (in default memory) of the type which is specified in the format string.

The format string can contain:

- Blanks or tabs, which are skipped.
- Normal characters (not '%'), which should be matched exactly in the input stream.
- Conversion specifications, starting with a '%' character.

Conversion specifications should be built as follows (in order):

- A '*', meaning that no assignment is done for this field.
- A number specifying the maximum field width.
- The conversion characters d, i, n, o, u and x may be preceding by 'h' if the argument is a pointer to **short** rather than **int**, or by 'l' (letter ell) if the argument is a pointer to **long**. The conversion characters e, f, and g may be preceding by 'l' if a pointer **double** rather than **float** is in the argument list, and by 'L' if a pointer to a **long double**.
- A conversion specifier. '*', maximum field width and length modifier are optional, the conversion character is not. The conversion character must be one of the following, if a character following '%' is not in the list, the behavior is undefined.

Length specifier and length modifier are optional, the conversion character is not. The conversion character must be one of the following, if a character following '%' is not in the list, the behavior is undefined.
<table>
<thead>
<tr>
<th>Character</th>
<th>Scanned as</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>int, signed decimal.</td>
</tr>
<tr>
<td>i</td>
<td>int, the integer may be given octal (i.e. a leading 0 is entered) or hexadecimal (leading &quot;0x&quot; or &quot;0X&quot;), or just decimal.</td>
</tr>
<tr>
<td>o</td>
<td>int, unsigned octal.</td>
</tr>
<tr>
<td>u</td>
<td>int, unsigned decimal.</td>
</tr>
<tr>
<td>x</td>
<td>int, unsigned hexadecimal in lowercase or uppercase.</td>
</tr>
<tr>
<td>c</td>
<td>single character (converted to unsigned char).</td>
</tr>
<tr>
<td>s</td>
<td>char *, a string of non white space characters. The argument should point to an array of characters, large enough to hold the string and a terminating NULL character.</td>
</tr>
<tr>
<td>f</td>
<td>float</td>
</tr>
<tr>
<td>e, E</td>
<td>float</td>
</tr>
<tr>
<td>g, G</td>
<td>float</td>
</tr>
<tr>
<td>n</td>
<td>int *, the number of characters written so far is written into the argument. No scanning is done.</td>
</tr>
<tr>
<td>p</td>
<td>pointer; hexadecimal 24–bit value which must be entered without 0x– prefix.</td>
</tr>
<tr>
<td>[...]</td>
<td>Matches a string of input characters from the set between the brackets. A NULL character is added to terminate the string. Specifying [...] includes the ’]’ character in the set of scanning characters.</td>
</tr>
<tr>
<td>[^...]</td>
<td>Matches a string of input characters not in the set between the brackets. A NULL character is added to terminate the string. Specifying [^...] includes the ’]’ character in the set.</td>
</tr>
<tr>
<td>%</td>
<td>Literal ‘%’, no assignment is done.</td>
</tr>
</tbody>
</table>

*Table 2–3: Scanf conversion characters*
<table>
<thead>
<tr>
<th>Stdio.h</th>
<th>Wchar.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fgetc(stream)</td>
<td>fgets(*s,n, stream)</td>
<td>Reads one character from stream. Returns the read character, or EOF/WEOF on error. <em>(FSS implementation)</em></td>
</tr>
</tbody>
</table>
| getc(stream) | getwc(stream) | Same as fgetc/fgetwc except that is implemented as a macro. *(FSS implementation)*    
NOTE: Currently #defined as getchar()/getwchar() because FILE I/O is not supported. Returns the read character, or EOF/WEOF on error. |
<p>| getchar(stdin) | getwchar(stdin) | Reads one character from the stdin stream. Returns the character read or EOF/WEOF on error. Implemented as macro. <em>(FSS implementation)</em> |
| fgets(*s,n, stream) | fgets(*s,n, stream) | Reads at most the next n−1 characters from the stream into array s until a newline is found. Returns s or NULL or EOF/WEOF on error. <em>(FSS implementation)</em> |
| gets(*s,n,stdin) | - | Reads at most the next n−1 characters from the stdin stream into array s. A newline is ignored. Returns s or NULL or EOF/WEOF on error. <em>(FSS implementation)</em> |
| ungetc(c,stream) | ungetwc(c,stream) | Pushes character c back onto the input stream. Returns EOF/WEOF on error. <em>(FSS implementation)</em> |
| fscanf(stream, format,...) | fwscanf(stream, format,...) | Performs a formatted read from the given stream. Returns the number of items converted successfully. <em>(FSS implementation)</em> |</p>
<table>
<thead>
<tr>
<th>Stdio.h</th>
<th>Wchar.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>scanf(format,...)</code></td>
<td><code>wscanf(format,...)</code></td>
<td>Performs a formatted read from stdin. Returns the number of items converted successfully. (FSS implementation)</td>
</tr>
<tr>
<td><code>sscanf(*s, format,...)</code></td>
<td><code>swscanf(*s, format,...)</code></td>
<td>Performs a formatted read from the string s. Returns the number of items converted successfully.</td>
</tr>
<tr>
<td><code>vfscanf(stream, format, arg)</code></td>
<td><code>vfwscanf(stream, format, arg)</code></td>
<td>Same as <code>fscanf/fwsanf</code>, but extra arguments are given as variable argument list <code>arg</code>. (See section 2.2.16, <code>stdarg.h</code>)</td>
</tr>
<tr>
<td><code>vscanf(format, arg)</code></td>
<td><code>vwscanf(format, arg)</code></td>
<td>Same as <code>sscanf/swscanf</code>, but extra arguments are given as variable argument list <code>arg</code>. (See section 2.2.16, <code>stdarg.h</code>)</td>
</tr>
<tr>
<td><code>vsscanf(*s, format, arg)</code></td>
<td><code>vswscanf(*s, format, arg)</code></td>
<td>Same as <code>scanf/wscanf</code>, but extra arguments are given as variable argument list <code>arg</code>. (See section 2.2.16, <code>stdarg.h</code>)</td>
</tr>
<tr>
<td><code>fputc(c,stream)</code></td>
<td><code>fputwc(c,stream)</code></td>
<td>Put character c onto the given <code>stream</code>. Returns EOF/WEOF on error. (FSS implementation)</td>
</tr>
<tr>
<td><code>putc(c,stream)</code></td>
<td><code>putwc(c,stream)</code></td>
<td>Same as <code>fputc/fputwc</code> except that is implemented as a macro. (FSS implementation)</td>
</tr>
<tr>
<td><code>putchar(c,stdout)</code></td>
<td><code>putwchar(c,stdout)</code></td>
<td>Put character c onto the stdout stream. Returns EOF/WEOF on error. Implemented as macro. (FSS implementation)</td>
</tr>
<tr>
<td><code>fputs(*s,stream)</code></td>
<td><code>fputws(*s,stream)</code></td>
<td>Writes string s to the given <code>stream</code>. Returns EOF/WEOF on error. (FSS implementation)</td>
</tr>
<tr>
<td>Stdio.h</td>
<td>Wchar.h</td>
<td>Description</td>
</tr>
<tr>
<td>----------------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>puts(*s)</td>
<td>-</td>
<td>Writes string s to the stdout stream. Returns EOF/WEOF on error. (FSS implementation)</td>
</tr>
<tr>
<td>fprintf(stream, format,...)</td>
<td>fwprintf(stream, format,...)</td>
<td>Performs a formatted write to the given stream. Returns EOF/WEOF on error. (FSS implementation)</td>
</tr>
<tr>
<td>printf(format,...)</td>
<td>wprintf(format,...)</td>
<td>Performs a formatted write to the stream stdout. Returns EOF/WEOF on error. (FSS implementation)</td>
</tr>
<tr>
<td>sprintf(*s, format,...)</td>
<td>-</td>
<td>Performs a formatted write to string s. Returns EOF/WEOF on error.</td>
</tr>
<tr>
<td>snprintf(*s,n format,...)</td>
<td>swprintf(*s,n format,...)</td>
<td>Same as sprintf, but n specifies the maximum number of characters (including the terminating null character) to be written.</td>
</tr>
<tr>
<td>vfprintf(stream, format,arg)</td>
<td>vfprintf(stream, format,arg)</td>
<td>Same as fprintf/fwprintf, but extra arguments are given as variable argument list arg. (See section 2.2.16, stdarg.h) (FSS implementation)</td>
</tr>
<tr>
<td>vprintf(format, arg)</td>
<td>vwprintf(format, arg)</td>
<td>Same as printf/wprintf, but extra arguments are given as variable argument list arg. (See section 2.2.16, stdarg.h) (FSS implementation)</td>
</tr>
<tr>
<td>vsprintf(*s, format,arg)</td>
<td>vswprintf(*s, format,arg)</td>
<td>Same as sprintf/swprintf, but extra arguments are given as variable argument list arg. (See section 2.2.16, stdarg.h)</td>
</tr>
</tbody>
</table>
**Direct input/output**

<table>
<thead>
<tr>
<th>Stdio.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fread(ptr, size, nobj, stream)</code></td>
<td>Reads <code>nobj</code> members of <code>size</code> bytes from the given <code>stream</code> into the array pointed to by <code>ptr</code>. Returns the number of elements successfully read. (FSS implementation)</td>
</tr>
<tr>
<td><code>fwrite((ptr, size, nobj, stream)</code></td>
<td>Writes <code>nobj</code> members of <code>size</code> bytes from to the array pointed to by <code>ptr</code> to the given <code>stream</code>. Returns the number of elements successfully written. (FSS implementation)</td>
</tr>
</tbody>
</table>

**Random access**

<table>
<thead>
<tr>
<th>Stdio.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fseek(stream, offset, origin)</code></td>
<td>Sets the position indicator for <code>stream</code>. (FSS implementation)</td>
</tr>
</tbody>
</table>

When repositioning a binary file, the new position `origin` is given by the following macros:

- `SEEK_SET 0`  offset characters from the beginning of the file
- `SEEK_CUR 1`  offset characters from the current position in the file
- `SEEK_END 2`  offset characters from the end of the file

- `ftell(stream)`  Returns the current file position for `stream`, or −1L on error. (FSS implementation)
- `rewind(stream)`  Sets the file position indicator for the `stream` to the beginning of the file. This function is equivalent to:
  
  ```c
  (void) fseek( stream, 0L, SEEK_SET );
  clearerr( stream );
  ```
  (FSS implementation)
- `fgetpos(stream,pos)`  Stores the current value of the file position indicator for `stream` in the object pointed to by `pos`. (FSS implementation)
- `fsetpos(stream,pos)`  Positions `stream` at the position recorded by `fgetpos` in `*pos`. (FSS implementation)
Operations on files

<table>
<thead>
<tr>
<th>Stdio.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>remove(file)</td>
<td>Removes the named file, so that a subsequent attempt to open it fails. Returns a non-zero value if not successful.</td>
</tr>
<tr>
<td>rename(old,new)</td>
<td>Changes the name of the file from old name to new name. Returns a non-zero value if not successful.</td>
</tr>
<tr>
<td>tmpfile()</td>
<td>Creates a temporary file of the mode &quot;wb+&quot; that will be automatically removed when closed or when the program terminates normally. Returns a file pointer.</td>
</tr>
<tr>
<td>tmpnam(buffer)</td>
<td>Creates new file names that do not conflict with other file names currently in use. The new file name is stored in a buffer which must have room for L_tmpnam characters. Returns a pointer to the temporary name. The file names are created in the current directory and all start with &quot;tmp&quot;. At most TMP_MAX unique file names can be generated.</td>
</tr>
</tbody>
</table>

Error handling

<table>
<thead>
<tr>
<th>Stdio.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clearerr(stream)</td>
<td>Clears the end of file and error indicators for stream.</td>
</tr>
<tr>
<td>ferror(stream)</td>
<td>Returns a non-zero value if the error indicator for stream is set.</td>
</tr>
<tr>
<td>feof(stream)</td>
<td>Returns a non-zero value if the end of file indicator for stream is set.</td>
</tr>
<tr>
<td>perror(*s)</td>
<td>Prints s and the error message belonging to the integer errno. (See section 2.2.4, errno.h)</td>
</tr>
</tbody>
</table>

2.2.21 STDLIB.H AND WCHAR.H

The header file stdlib.h contains general utility functions which fall into the following categories (Some have parallel wide-character, declared in wchar.h)

- Numeric conversions
- Random number generation
- Memory management
- Environment communication
- Searching and sorting
• Integer arithmetic
• Multibyte/wide character and string conversions.

**Macros**

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAND_MAX</td>
<td>32767 Highest number that can be returned by the rand/srand function.</td>
</tr>
<tr>
<td>EXIT_SUCCESS</td>
<td>0 Predefined exit codes that can be used in the exit function.</td>
</tr>
<tr>
<td>EXIT_FAILURE</td>
<td>1 Maximum number of bytes in a multibyte character for the extended character set specified by the current locale (category LC_CTYPE, see section 2.2.12, locale.h).</td>
</tr>
</tbody>
</table>

**Numeric conversions**

Next functions convert the initial portion of a string *s* to a **double**, **int**, **long int** and **long long int** value respectively.

- `double atof(*s)`
- `int atoi(*s)`
- `long atol(*s)`
- `long long atoll(*s)`

Next functions convert the initial portion of the string *s* to a float, double and long double value respectively. *endp* will point to the first character not used by the conversion.

**Stdlib.h**

<table>
<thead>
<tr>
<th>C Type</th>
<th>Function</th>
<th>C Type</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td><code>strtof(*s,**endp)</code></td>
<td>float</td>
<td><code>wctof(*s,**endp)</code></td>
</tr>
<tr>
<td>double</td>
<td><code>strtod(*s,**endp)</code></td>
<td>double</td>
<td><code>wcstod(*s,**endp)</code></td>
</tr>
<tr>
<td>long double</td>
<td><code>strtold(*s,**endp)</code></td>
<td>long double</td>
<td><code>wcstold(*s,**endp)</code></td>
</tr>
</tbody>
</table>
Next functions convert the initial portion of the string *s to a long, long
long, unsigned long and unsigned long long respectively. Base
specifies the radix. *endp will point to the first character not used by the
conversion.

**Stdlib.h**

```c
long strtol (*s,**endp,base)
long long strtoll (*s,**endp,base)
unsigned long strtoul (*s,**endp,base)
unsigned long long strtoull (*s,**endp,base)
```

**Wchar.h**

```c
long wcstol (*s,**endp,base)
long long wcstoll (*s,**endp,base)
unsigned long wcstoul (*s,**endp,base)
unsigned long long wcstoull (*s,**endp,base)
```

---

**Random number generation**

```c
rand
```

Returns a pseudo random integer in the range 0 to RAND_MAX.

```c
srand(seed)
```

Same as rand but uses seed for a new sequence of
pseudo random numbers.

---

**Memory management**

```c
malloc(size)
```

Allocates space for an object with size size.
The allocated space is not initialized. Returns a
pointer to the allocated space.

```c
calloc(nobj,size)
```

Allocates space for n objects with size size.
The allocated space is initialized with zeros. Returns a
pointer to the allocated space.

```c
free(*ptr)
```

Deallocates the memory space pointed to by ptr
which should be a pointer earlier returned by the
malloc or calloc function.

```c
realloc(*ptr,size)
```

Deallocates the old object pointed to by ptr and
returns a pointer to a new object with size size.
The new object cannot have a size larger than the
previous object.
**Environment communication**

abort() Causes abnormal program termination. If the signal SIGABRT is caught, the signal handler may take over control. (See section 2.2.15, `signal.h`).

atexit(*func) *Func* points to a function that is called (without arguments) when the program normally terminates.

exit(status) Causes normal program termination. Acts as if `main()` returns with status as the return value. Status can also be specified with the predefined macros EXIT_SUCCESS or EXIT_FAILURE.

_Exit(status) Same as exit, but not registered by the atexit function or signal handlers registered by the `signal` function are called.

getenv(*s) Searches an environment list for a string *s*. Returns a pointer to the contents of *s*. NOTE: this function is not implemented because there is no OS.

system(*s) Passes the string *s* to the environment for execution. NOTE: this function is not implemented because there is no OS.

**Searching and sorting**

bsearch(*key,*base, *n*,size,*cmp) This function searches in an array of *n* members, for the object pointed to by *key*. The initial base of the array is given by *base*. The size of each member is specified by *size*. The given array must be sorted in ascending order, according to the results of the function pointed to by *cmp*. Returns a pointer to the matching member in the array, or NULL when not found.

qsort(*base,n, *size,*cmp) This function sorts an array of *n* members using the quick sort algorithm. The initial base of the array is given by *base*. The size of each member is specified by *size*. The array is sorted in ascending order, according to the results of the function pointed to by *cmp*. 
Integer arithmetic

int abs(j)  Compute the absolute value of an int, long int, and long long int j resepectively.
long labs(j)
long long llabs(j)
div_t div(x,y)  Compute x/y and x%y in a single operation. X and y have respectively type int, long int and
ldiv_t ldiv(x,y) long long int. The result is stored in the
lldiv_t lldiv(x,y) members quot and rem of struct div_t,
wcstombs wcstombs which have the same types.

Multibyte/wide character and string conversions

mblen(*s,n) Determines the number of bytes in the multi-byte
character pointed to by s. At most n characters will
be examined. (See also mbrlen in section 2.2.25, wchar.h)
mbtowc(*pwc,*s,n) Converts the multi-byte character in s to a
wide-character code and stores it in pwc. At most n
characters will be examined.
wctomb(*s,wc) Converts the wide-character wc into a multi-byte
representation and stores it in the string pointed to
by s. At most MB_CUR_MAX characters are
stored.
mbstowcs(*pwcs,*s,n) Converts a sequence of multi-byte characters in
the string pointed to by s into a sequence of wide
characters and stores at most n wide characters
into the array pointed to by pwcs. (See also
mbstowcs in section 2.2.25, wchar.h)
wctombs(*s,*pwcs,n) Converts a sequence of wide characters in the
array pointed to by pwcs into multi-byte characters
and stores at most n multi-byte characters into the
string pointed to by s. (See also wcstowmb in
section 2.2.25, wchar.h)

2.2.22 STRING.H AND WCHAR.H

This header file provides numerous functions for manipulating strings. By
collection, strings in C are arrays of characters with a terminating null
character. Most functions therefore take arguments of type *char.
However, many functions have also parallel wide-character functions
which take arguments of type *wchar_t. These functions are declared in
wchar.h.
## Copying and concatenation functions

<table>
<thead>
<tr>
<th>Stdio.h</th>
<th>Wchar.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>memcpy(*s1,*s2,n)</td>
<td>wmemcpy(*s1,*s2,n)</td>
<td>Copies n characters from *s2 into *s1 and returns *s1. If *s1 and *s2 overlap the result is undefined.</td>
</tr>
<tr>
<td>memmove(*s1,*s2,n)</td>
<td>wmemmove(*s1,*s2,n)</td>
<td>Same as memcpy, but overlapping strings are handled correctly. Returns *s1.</td>
</tr>
<tr>
<td>strcpy(*s1,*s2)</td>
<td>wcscpy(*s1,*s2)</td>
<td>Copies *s2 into *s1 and returns *s1. If *s1 and *s2 overlap the result is undefined.</td>
</tr>
<tr>
<td>strncpy(*s1,*s2,n)</td>
<td>wcsncpy(*s1,*s2,n)</td>
<td>Copies not more than n characters from *s2 into *s1 and returns *s1. If *s1 and *s2 overlap the result is undefined.</td>
</tr>
<tr>
<td>strcat(*s1,*s2)</td>
<td>wcscat(*s1,*s2)</td>
<td>Appends a copy of *s2 to *s1 and returns *s1. If *s1 and *s2 overlap the result is undefined.</td>
</tr>
<tr>
<td>strncat(*s1,*s2,n)</td>
<td>wcsncat(*s1,*s2,n)</td>
<td>Appends not more than n characters from *s2 to *s1 and returns *s1. If *s1 and *s2 overlap the result is undefined.</td>
</tr>
</tbody>
</table>
## Comparison functions

<table>
<thead>
<tr>
<th>Stdio.h</th>
<th>Wchar.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>memcmp(*s1,*s2,n)</code></td>
<td><code>wmemcmp(*s1,*s2,n)</code></td>
<td>Compares the first ( n ) characters of ( *s1 ) to the first ( n ) characters of ( *s2 ). Returns ( &lt; 0 ) if ( *s1 &lt; *s2 ), ( = 0 ) if ( *s1 = *s2 ), or ( &gt; 0 ) if ( *s1 &gt; *s2 ).</td>
</tr>
<tr>
<td><code>strcmp(*s1,*s2)</code></td>
<td><code>wcscmp(*s1,*s2)</code></td>
<td>Compares string ( *s1 ) to ( *s2 ). Returns ( &lt; 0 ) if ( *s1 &lt; *s2 ), ( = 0 ) if ( *s1 = *s2 ), or ( &gt; 0 ) if ( *s1 &gt; *s2 ).</td>
</tr>
<tr>
<td><code>strncmp(*s1,*s2,n)</code></td>
<td><code>wcsncmp(*s1,*s2,n)</code></td>
<td>Compares the first ( n ) characters of ( *s1 ) to the first ( n ) characters of ( *s2 ). Returns ( &lt; 0 ) if ( *s1 &lt; *s2 ), ( = 0 ) if ( *s1 = *s2 ), or ( &gt; 0 ) if ( *s1 &gt; *s2 ).</td>
</tr>
<tr>
<td><code>strcoll(*s1,*s2)</code></td>
<td><code>wscoll(*s1,*s2)</code></td>
<td>Performs a locale–specific comparison between string ( *s1 ) and string ( *s2 ) according to the LC.COLLATE category of the current locale. Returns ( &lt; 0 ) if ( *s1 &lt; *s2 ), ( = 0 ) if ( *s1 = *s2 ), or ( &gt; 0 ) if ( *s1 &gt; *s2 ). (See section 2.2.12, <code>locale.h</code>)</td>
</tr>
<tr>
<td><code>strxfrm(*s1,*s2,n)</code></td>
<td><code>wcsxfrm(*s1,*s2,n)</code></td>
<td>Transforms (a local) string ( *s2 ) so that a comparison between transformed strings with <code>strcmp</code> gives the same result as a comparison between non–transformed strings with <code>strcoll</code>. Returns the transformed string ( *s1 ).</td>
</tr>
</tbody>
</table>
**Search functions**

<table>
<thead>
<tr>
<th>Stdio.h</th>
<th>Wchar.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>memchr(*s,c,n)</td>
<td>wmemchr(*s,c,n)</td>
<td>Checks the first ( n ) characters of *s on the occurrence of character c. Returns a pointer to the found character.</td>
</tr>
<tr>
<td>strchr(*s,c)</td>
<td>wcschr(*s,c)</td>
<td>Returns a pointer to the first occurrence of character c in *s or the null pointer if not found.</td>
</tr>
<tr>
<td>strrchr(*s,c)</td>
<td>wcstrchr(*s,c)</td>
<td>Returns a pointer to the last occurrence of character c in *s or the null pointer if not found.</td>
</tr>
<tr>
<td>strspn(*s,*set)</td>
<td>wcspn(*s,*set)</td>
<td>Searches *s for a sequence of characters specified in *set. Returns the length of the first sequence found.</td>
</tr>
<tr>
<td>strcspn(*s,*set)</td>
<td>wcscspn(*s,*set)</td>
<td>Searches *s for a sequence of characters <em>not</em> specified in *set. Returns the length of the first sequence found.</td>
</tr>
<tr>
<td>strpbrk(*s,*set)</td>
<td>wcspbrk(*s,*set)</td>
<td>Same as strspn/wcspn but returns a pointer to the first character in *s that also is specified in *set.</td>
</tr>
<tr>
<td>strstr(*s,*sub)</td>
<td>wcsstr(*s,*sub)</td>
<td>Searches for a substring *sub in *s. Returns a pointer to the first occurrence of *sub in *s.</td>
</tr>
<tr>
<td>strtok(*s,*dlm)</td>
<td>wcstok(*s,*dlm)</td>
<td>A sequence of calls to this function breaks the string *s into a sequence of tokens delimited by a character specified in *dlm. The token found in *s is terminated with a null character. Returns a pointer to the first position in *s of the token.</td>
</tr>
</tbody>
</table>
Miscellaneous functions

<table>
<thead>
<tr>
<th>Stdio.h</th>
<th>Wchar.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>memset(*s,c,n)</td>
<td>wmemset(*s,c,n)</td>
<td>Fills the first n bytes of *s with character c and returns *s.</td>
</tr>
<tr>
<td>strerror(errno)</td>
<td>-</td>
<td>Typically, the values for errno come from int errno. This function returns a pointer to the associated error message. (See also section 2.2.4, errno.h)</td>
</tr>
<tr>
<td>strlen(*s)</td>
<td>wcslen(*s)</td>
<td>Returns the length of string *s.</td>
</tr>
</tbody>
</table>

2.2.23 TIME.H AND WCHAR.H

The header file time.h provides facilities to retrieve and use the (calendar) date and time, and the process time. Time can be represented as an integer value, or can be broken-down in components. Two arithmetic data types are defined which are capable of holding the integer representation of times:

```
clock_t    unsigned long long
time_t     unsigned long
```

The type struct tm below is defined according to ISO/IEC9899 with one exception: this implementation does not support leap seconds. The struct tm type is defines as follows:

```
struct tm
{
    int tm_sec;       /* seconds after the minute - [0, 59] */
    int tm_min;       /* minutes after the hour - [0, 59] */
    int tm_hour;      /* hours since midnight - [0, 23] */
    int tm_mday;      /* day of the month - [1, 31] */
    int tm_mon;       /* months since January - [0, 11] */
    int tm_year;      /* year since 1900 */
    int tm_wday;      /* days since Sunday - [0, 6] */
    int tm_yday;      /* days since January 1 - [0, 365] */
    int tm_isdst;     /* Daylight Saving Time flag */
};
```
### Time manipulation

**clock**

Returns the application’s best approximation to the processor time used by the program since it was started. This low-level routine is not implemented because it strongly depends on the hardware. To determine the time in seconds, the result of clock should be divided by the value defined as

```
CLOCKS_PER_SEC 12000000
```

**difftime(t1,t0)**

Returns the difference \( t1 - t0 \) in seconds.

**mktime(tm *tp)**

Converts the broken–down time in the structure pointed to by \( tp \), to a value of type time_t. The return value has the same encoding as the return value of the time function.

**time(*timer)**

Returns the current calendar time. This value is also assigned to *timer.

### Time conversion

**asctime(tm *tp)**

Converts the broken–down time in the structure pointed to by \( tp \) into a string in the form Mon Jan 21 16:15:14 2004 n\0. Returns a pointer to this string.

**ctime(*timer)**

Converts the calendar time pointed to by \( timer \) to local time in the form of a string. This is equivalent to:

```
asctime(localtime(timer))
```

**gmtime(*timer)**

Converts the calendar time pointed to by \( timer \) to the broken–down time, expressed as UTC. Returns a pointer to the broken–down time.

**localtime(*timer)**

Converts the calendar time pointed to by \( timer \) to the broken–down time, expressed as local time. Returns a pointer to the broken–down time.
**Formatted time**

The next function has a parallel function defined in `wchar.h`:

<table>
<thead>
<tr>
<th>Stdio.h</th>
<th>Wchar.h</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>strftime(*s,smax,*fmt,tm *tp)</code></td>
<td><code>wstrftime(*s,smax,*fmt,tm *tp)</code></td>
</tr>
</tbody>
</table>

Formats date and time information from `struct tm *tp` into `*s` according to the specified format `*fmt`. No more than `smax` characters are placed into `*s`. The formatting of `strftime` is locale-specific using the `LC_TIME` category (see section 2.2.12, `locale.h`). You can use the next conversion specifiers:

- `%a` abbreviated weekday name
- `%A` full weekday name
- `%b` abbreviated month name
- `%B` full month name
- `%c` local date and time representation
- `%d` day of the month (01–31)
- `%H` hour, 24-hour clock (00–23)
- `%I` hour, 12-hour clock (01–12)
- `%j` day of the year (001–366)
- `%m` month (01–12)
- `%M` minute (00–59)
- `%p` local equivalent of AM or PM
- `%S` second (00–59)
- `%U` week number of the year, Sunday as first day of the week (00–53)
- `%W` week number of the year, Monday as first day of the week (00–53)
- `%x` local date representation
- `%X` local time representation
- `%y` year without century (00–99)
- `%Y` year with century
- `%Z` time zone name, if any
- `%%` `%
2.2.24 **UNISTD.H**

The file `unistd.h` contains standard UNIX I/O functions. These functions are all implemented using CrossView Pro's file system simulation. This header file is not defined in ISO/IEC9899.

`access(*name, mode)` Use the file system simulation of CrossView Pro to check the permissions of a file on the host. `mode` specifies the type of access and is a bit pattern constructed by a logical OR of the following values:
- `R_OK` Checks read permission.
- `W_OK` Checks write permission.
- `X_OK` Checks execute (search) permission.
- `F_OK` Checks to see if the file exists.

*(FSS implementation)*

`chdir(*path)` Use the file system simulation feature of CrossView Pro to change the current directory on the host to the directory indicated by `path`.

*(FSS implementation)*

`close(fd)` File close function. The given file descriptor should be properly closed. This function calls `_close()`.

*(FSS implementation)*

`getcwd(*buf, size)` Use the file system simulation feature of CrossView Pro to retrieve the current directory on the host. Returns the directory name. *(FSS implementation)*


*(FSS implementation)*

`read(fd, *buff, cnt)` Reads a sequence of characters from a file. This function calls `_read()`.

*(FSS implementation)*

`stat(*name, *buff)` Use the file system simulation feature of CrossView Pro to stat() a file on the host platform.

*(FSS implementation)*

`unlink(*name)` Removes the named file, so that a subsequent attempt to open it fails. Calls `_unlink()`.

*(FSS implementation)*

`write(fd, *buff, cnt)` Write a sequence of characters to a file. Calls `_write()`.

*(FSS implementation)*
### 2.2.25 wchar.h

Many functions in `wchar.h` represent the wide–character variant of other functions so these are discussed together. (See sections 2.2.20, `stdio.h`, 2.2.21, `stdlib.h`, 2.2.22, `strings.h` and 2.2.23, `time.h`).

The remaining functions are described below. They perform conversions between multi-byte characters and wide characters. In these functions, `ps` points to struct `mbstate_t` which holds the conversion state information necessary to convert between sequences of multibyte characters and wide characters:

```c
typedef struct
{
    wchar_t wc_value; /* wide character value solved so far */
    unsigned short n_bytes; /* number of bytes of solved multibyte */
    unsigned short encoding; /* encoding rule for wide character <> multibyte conversion */
} mbstate_t;
```

When multibyte characters larger than 1 byte are used, this struct will be used to store the conversion information when not all the bytes of a particular multibyte character have been read from the source. In this implementation, multi–byte characters are 1 byte long (MB_CUR_MAX and MB_LEN_MAX are defined as 1) and this will never occur.

- `mbsinit(*ps)` Determines whether the object pointed to by `ps`, is an initial conversion state. Returns a non–zero value if so.
- `mbsrtowcs(*pwcs,**src,n,*ps)` Restartable version of `mbstowcs`. See section 2.2.21, `stdlib.h`. The initial conversion state is specified by `ps`. The input sequence of multibyte characters specified indirectly by `src`.
- `wcsrtombs(*s,**src,n,*ps)` Restartable version of `wcstombs`. See section 2.2.21, `stdlib.h`. The initial conversion state is specified by `ps`. The input wide string is specified indirectly by `src`.
- `mbtowc(*pwc,*s,n,*ps)` Converts a multibyte character `*s` to a wide character `*pwc` according to conversion state `ps`. See also `mbtowc` in section 2.2.21, `stdlib`. 

---

**Note:**

- `mbrtowc` and `mbstowcs` are similar in functionality but operate on wide characters.
- `mbtowc` and `wcstombs` are used to convert wide characters to multibyte characters.
- `mbsinit` initializes the state structure for multibyte to wide character conversions.
- The multibyte characters are 1 byte long in this implementation.

---

**References:**

- Sections 2.2.20, `stdio.h`, 2.2.21, `stdlib.h`, 2.2.22, `strings.h` and 2.2.23, `time.h`.
\texttt{wctomb}(*s,wc,\ast ps) \quad \text{Converts a wide character wc to a multi-byte character according to conversion state ps and stores the multi-byte character in *s.}

\texttt{btowc}(c) \quad \text{Returns the wide character corresponding to character c. Returns WEOF on error.}

\texttt{wctob}(c) \quad \text{Returns the multi-byte character corresponding to the wide character c. The returned multi-byte character is represented as one byte. Returns EOF on error.}

\texttt{mbrlen}(*s,n,\ast ps) \quad \text{Inspects up to n bytes from the string *s to see if those characters represent valid multibyte characters, relative to the conversion state held in \ast ps.}

**2.2.26 WCTYPE.H**

Most functions in \texttt{wctype.h} represent the wide-character variant of functions declared in \texttt{ctype.h} and are discussed in section 2.2.3, \texttt{ctype.h}. In addition, this header file provides extensible, locale specific functions and wide character classification.

\texttt{wctype}(*property) \quad \text{Constructs a value of type \texttt{wctype_t} that describes a class of wide characters identified by the string *property. If property identifies a valid class of wide characters according to the LC\_TYPE category (see 2.2.12, \texttt{locale.h}) of the current locale, a non-zero value is returned that can be used as an argument in the \texttt{iswctype} function.}

\texttt{iswctype}(wc,desc) \quad \text{Tests whether the wide character wc is a member of the class represented by \texttt{wctype_t desc}. Returns a non-zero value if tested true.}

<table>
<thead>
<tr>
<th>Function</th>
<th>Equivalent to locale specific test</th>
</tr>
</thead>
<tbody>
<tr>
<td>iswalnum(wc)</td>
<td>iswctype(wc,wctype(&quot;alnum&quot;))</td>
</tr>
<tr>
<td>iswalpha(wc)</td>
<td>iswctype(wc,wctype(&quot;alpha&quot;))</td>
</tr>
<tr>
<td>iswcntrl(wc)</td>
<td>iswctype(wc,wctype(&quot;cntrl&quot;))</td>
</tr>
<tr>
<td>iswdigit(wc)</td>
<td>iswctype(wc,wctype(&quot;digit&quot;))</td>
</tr>
<tr>
<td>iswgraph(wc)</td>
<td>iswctype(wc,wctype(&quot;graph&quot;))</td>
</tr>
</tbody>
</table>
### Function | Equivalent to locale specific test
---|---
iswlower(wc) | iswctype(wc,wctype("lower"))
iswprint(wc) | iswctype(wc,wctype("print"))
iswpunct(wc) | iswctype(wc,wctype("punct"))
iswspace(wc) | iswctype(wc,wctype("space"))
iswupper(wc) | iswctype(wc,wctype("upper"))
iswxdigit(wc) | iswctype(wc,wctype("xdigit"))

**wctrans(**property***)** Constructs a value of type `wctype_t` that describes a mapping between wide characters identified by the string `property`. If `property` identifies a valid mapping of wide characters according to the LC_TYPE category (see 2.2.12, locale.h) of the current locale, a non-zero value is returned that can be used as an argument in the `towctrans` function.

towctrans(wc,desc) Transforms wide character `wc` into another wide-character, described by `desc`.

### Function | Equivalent to locale specific transformation
---|---
towlower(wc) | towctrans(wc,wctrans("tolower"))
towupper(wc) | towctrans(wc,wctrans("toupper"))
2.3 **C LIBRARY REENTRANCY**

Some of the functions in the C library are reentrant, others are not. The table below shows the functions in the C library, and whether they are reentrant or not. A dash means that the function is reentrant. Note that some of the functions are not reentrant because they set the global variable `errno` (or call other functions that eventually set `errno`). If your program does not check this variable and `errno` is the only reason for the function not being reentrant, these functions can be assumed reentrant as well.

The explanation of the cause why a function is not reentrant sometimes refers to a footnote because the explanation is too lengthy for the table.

<table>
<thead>
<tr>
<th>Function</th>
<th>Not reentrant because</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>_close</code></td>
<td>Uses global File System Simulation buffer, <code>fss_buffer</code></td>
</tr>
<tr>
<td><code>_doflt</code></td>
<td>Uses I/O functions which modify <code>iob[]</code>. See (1).</td>
</tr>
<tr>
<td><code>_doprint</code></td>
<td>Uses indirect access to static <code>iob[]</code> array. See (1).</td>
</tr>
<tr>
<td><code>_doscan</code></td>
<td>Uses indirect access to <code>iob[]</code> and calls <code>ungetc</code> (access to local static <code>ungetc[]</code> buffer). See (1).</td>
</tr>
<tr>
<td><code>_Exit</code></td>
<td>See exit.</td>
</tr>
<tr>
<td><code>_filbuf</code></td>
<td>Uses <code>iob[]</code>. See (1).</td>
</tr>
<tr>
<td><code>_filsbuf</code></td>
<td>Uses <code>iob[]</code>. See (1).</td>
</tr>
<tr>
<td><code>_getflt</code></td>
<td>Uses <code>iob[]</code>. See (1).</td>
</tr>
<tr>
<td><code>_iob</code></td>
<td>Defines static <code>iob[]</code>. See (1).</td>
</tr>
<tr>
<td><code>_ioread</code></td>
<td>Depends on low level I/O implementation. Uses <code>iob[]</code>. See (1).</td>
</tr>
<tr>
<td><code>_iowrite</code></td>
<td>Depends on low level I/O implementation. Uses <code>iob[]</code>. See (1).</td>
</tr>
<tr>
<td><code>_lseek</code></td>
<td>Uses global File System Simulation buffer, <code>_fss_buffer</code></td>
</tr>
<tr>
<td><code>_open</code></td>
<td>Uses global File System Simulation buffer, <code>_fss_buffer</code></td>
</tr>
<tr>
<td><code>_read</code></td>
<td>Uses global File System Simulation buffer, <code>_fss_buffer</code></td>
</tr>
<tr>
<td>Function</td>
<td>Not reentrant because</td>
</tr>
<tr>
<td>-----------------</td>
<td>---------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>_unlink</td>
<td>Uses global File System Simulation buffer, _fss_buffer</td>
</tr>
<tr>
<td>_write</td>
<td>Uses global File System Simulation buffer, _fss_buffer</td>
</tr>
<tr>
<td>abort</td>
<td>Calls exit</td>
</tr>
<tr>
<td>abs labs llabs</td>
<td>–</td>
</tr>
<tr>
<td>access</td>
<td>Uses global File System Simulation buffer, _fss_buffer</td>
</tr>
<tr>
<td>acos acosf acosl</td>
<td>Sets errno</td>
</tr>
<tr>
<td>acosh acoshf acoshl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>asctime</td>
<td>asctime defines static array for broken-down time string.</td>
</tr>
<tr>
<td>asin asinf asinl</td>
<td>Sets errno</td>
</tr>
<tr>
<td>asinh asinhf asinhl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>atan atanf atanl</td>
<td>–</td>
</tr>
<tr>
<td>atan2 atan2f atan2l</td>
<td>–</td>
</tr>
<tr>
<td>atanh atanhf atanhl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>atexit</td>
<td>atexit defines static array with function pointers to execute at exit of program.</td>
</tr>
<tr>
<td>atof</td>
<td>–</td>
</tr>
<tr>
<td>atoi</td>
<td>–</td>
</tr>
<tr>
<td>atol</td>
<td>–</td>
</tr>
<tr>
<td>bsearch</td>
<td>–</td>
</tr>
<tr>
<td>btowc</td>
<td>–</td>
</tr>
<tr>
<td>cabs cabsf cabsl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>cacos cacosf cacosl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>cacosh cacoshf cacoshl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>calloc</td>
<td>calloc uses static buffer management structures. See malloc (5).</td>
</tr>
<tr>
<td>carg cargf cargl</td>
<td>–</td>
</tr>
<tr>
<td>casin casinf casinl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>casinh casinhf casinhl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>catan catanf catanl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>Function</td>
<td>Not reentrant because</td>
</tr>
<tr>
<td>-------------------</td>
<td>----------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>catanh, catanhf, catanhl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>cbrt, cbrtl, cbrtf</td>
<td><em>(Not implemented)</em></td>
</tr>
<tr>
<td>ccos, ccosf, ccosl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>ccosh, ccoshf, ccoshl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>ceil, ceilf, ceill</td>
<td>-</td>
</tr>
<tr>
<td>cexp, cexpf, cexpl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>chdir</td>
<td>Uses global File System Simulation buffer, fss_buffer</td>
</tr>
<tr>
<td>cimag, cimagf, cimagl</td>
<td>-</td>
</tr>
<tr>
<td>cleanup</td>
<td>Calls fclose. See (1)</td>
</tr>
<tr>
<td>clearerr</td>
<td>Modifies iob[]. See (1)</td>
</tr>
<tr>
<td>clock</td>
<td>-</td>
</tr>
<tr>
<td>clog, clogf, clogl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>close</td>
<td>Calls_close</td>
</tr>
<tr>
<td>conj, conjf, conjl</td>
<td>-</td>
</tr>
<tr>
<td>copysign, copysignl</td>
<td>-</td>
</tr>
<tr>
<td>cos, cosf, cosl</td>
<td>-</td>
</tr>
<tr>
<td>csh, coshf, coshl</td>
<td>cosh calls exp(), which sets errno. If errno is discarded, cosh is reentrant.</td>
</tr>
<tr>
<td>cpow, cpowf, cpowl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>cproj, cprojf, cprojl</td>
<td>-</td>
</tr>
<tr>
<td>creal, crealf, creall</td>
<td>-</td>
</tr>
<tr>
<td>csin, csinf, csinl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>csinh, csinhf, csinhl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>csqrt, csqrtf, csqrtn</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>ctan, ctanf, ctanl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>ctanh, ctanhf, ctanhl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>ctime</td>
<td>Calls asctime</td>
</tr>
<tr>
<td>difftime</td>
<td>-</td>
</tr>
<tr>
<td>div, ldiv, lldiv</td>
<td>-</td>
</tr>
<tr>
<td>erf, erfl, erff</td>
<td><em>(Not implemented)</em></td>
</tr>
<tr>
<td>Function</td>
<td>Not reentrant because</td>
</tr>
<tr>
<td>----------------</td>
<td>----------------------------------------</td>
</tr>
<tr>
<td>erfc, erfcl, erfcf</td>
<td><em>(Not implemented)</em></td>
</tr>
<tr>
<td>exit</td>
<td>Calls fclose indirectly which uses iob[] calls functions in _atexit array. See (1). To make exit reentrant kernel support is required.</td>
</tr>
<tr>
<td>exp, expf, expl</td>
<td></td>
</tr>
<tr>
<td>exp2, exp2f, exp2l</td>
<td><em>(Not implemented)</em></td>
</tr>
<tr>
<td>expm1, expm1f, expm1l</td>
<td><em>(Not implemented)</em></td>
</tr>
<tr>
<td>fabs, fabsf, fabsl</td>
<td></td>
</tr>
<tr>
<td>fclose</td>
<td>Uses values in iob[]. See (1).</td>
</tr>
<tr>
<td>fdim, fdiml, fdimf</td>
<td><em>(Not implemented)</em></td>
</tr>
<tr>
<td>feclearexcept</td>
<td><em>(Not implemented)</em></td>
</tr>
<tr>
<td>fegetenv</td>
<td><em>(Not implemented)</em></td>
</tr>
<tr>
<td>fegetexceptflag</td>
<td><em>(Not implemented)</em></td>
</tr>
<tr>
<td>fegetround</td>
<td><em>(Not implemented)</em></td>
</tr>
<tr>
<td>feholdexcept</td>
<td><em>(Not implemented)</em></td>
</tr>
<tr>
<td>feof</td>
<td>Uses values in iob[]. See (1).</td>
</tr>
<tr>
<td>ferror</td>
<td><em>(Not implemented)</em></td>
</tr>
<tr>
<td>ferror</td>
<td>Uses values in iob[]. See (1).</td>
</tr>
<tr>
<td>fesetenv</td>
<td><em>(Not implemented)</em></td>
</tr>
<tr>
<td>fesetexceptflag</td>
<td><em>(Not implemented)</em></td>
</tr>
<tr>
<td>fesetround</td>
<td><em>(Not implemented)</em></td>
</tr>
<tr>
<td>fetestexcept</td>
<td><em>(Not implemented)</em></td>
</tr>
<tr>
<td>feupdateenv</td>
<td><em>(Not implemented)</em></td>
</tr>
<tr>
<td>fflush</td>
<td>Modifies iob[]. See (1).</td>
</tr>
<tr>
<td>fgetc, fgetwc</td>
<td>Uses pointer to iob[]. See (1).</td>
</tr>
<tr>
<td>fgetpos</td>
<td>Sets the variable errno and uses pointer to iob[]. See (1) / (2).</td>
</tr>
<tr>
<td>fgets, fgetws</td>
<td>Uses iob[]. See (1).</td>
</tr>
<tr>
<td>floor, floorf, floorl</td>
<td></td>
</tr>
<tr>
<td>fma, fmal, fmaf</td>
<td><em>(Not implemented)</em></td>
</tr>
<tr>
<td>fmax, fmaxl, fmaxf</td>
<td><em>(Not implemented)</em></td>
</tr>
<tr>
<td>fmin, fminl, fminf</td>
<td><em>(Not implemented)</em></td>
</tr>
<tr>
<td>Function</td>
<td>Not reentrant because</td>
</tr>
<tr>
<td>----------</td>
<td>------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>fmod</td>
<td></td>
</tr>
<tr>
<td>fmodl</td>
<td></td>
</tr>
<tr>
<td>fmodf</td>
<td></td>
</tr>
<tr>
<td>fopen</td>
<td>Uses iob[] and calls malloc when file open for buffered IO. See (1)</td>
</tr>
<tr>
<td>fpclassify</td>
<td></td>
</tr>
<tr>
<td>fprintf</td>
<td>Uses iob[]. See (1).</td>
</tr>
<tr>
<td>fscanf</td>
<td>Uses iob[]. See (1).</td>
</tr>
<tr>
<td>fputc</td>
<td>Uses iob[]. See (1).</td>
</tr>
<tr>
<td>puts</td>
<td>Uses iob[]. See (1).</td>
</tr>
<tr>
<td>fread</td>
<td>Calls fgetc. See (1).</td>
</tr>
<tr>
<td>free</td>
<td>free uses static buffer management structures. See malloc (5).</td>
</tr>
<tr>
<td>freopen</td>
<td>Modifies iob[]. See (1).</td>
</tr>
<tr>
<td>frexp</td>
<td></td>
</tr>
<tr>
<td>frexpl</td>
<td></td>
</tr>
<tr>
<td>frexpfl</td>
<td></td>
</tr>
<tr>
<td>fscanf</td>
<td>Uses iob[]. See (1)</td>
</tr>
<tr>
<td>fwscanf</td>
<td>Uses iob[]. See (1)</td>
</tr>
<tr>
<td>fseek</td>
<td>Uses iob[] and calls _doscan. Acesses ungetc[] array. See (1).</td>
</tr>
<tr>
<td>fsetpos</td>
<td>Uses iob[] and sets errno. See (1) / (2).</td>
</tr>
<tr>
<td>ftell</td>
<td>Uses iob[] and sets errno. Calls _lseek. See (1) / (2).</td>
</tr>
<tr>
<td>fwrite</td>
<td>Uses iob[]. See (1).</td>
</tr>
<tr>
<td>getc</td>
<td>Uses iob[]. See (1).</td>
</tr>
<tr>
<td>getwc</td>
<td>Uses iob[]. See (1).</td>
</tr>
<tr>
<td>getgchar</td>
<td>Uses iob[]. See (1).</td>
</tr>
<tr>
<td>getwchar</td>
<td>Uses global File System Simulation buffer, _fss_buffer</td>
</tr>
<tr>
<td>getenv</td>
<td>Skeleton only.</td>
</tr>
<tr>
<td>gets</td>
<td>Uses iob[]. See (1).</td>
</tr>
<tr>
<td>getsws</td>
<td>Uses iob[]. See (1).</td>
</tr>
<tr>
<td>gmtime</td>
<td>gmtime defines static structure</td>
</tr>
<tr>
<td>hypot</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>hypotl</td>
<td></td>
</tr>
<tr>
<td>hypotf</td>
<td></td>
</tr>
<tr>
<td>ilogb</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>ilogbf</td>
<td></td>
</tr>
<tr>
<td>ilogbl</td>
<td></td>
</tr>
<tr>
<td>imaxabs</td>
<td></td>
</tr>
<tr>
<td>imaxdiv</td>
<td></td>
</tr>
<tr>
<td>isalnum</td>
<td></td>
</tr>
<tr>
<td>iswalnum</td>
<td></td>
</tr>
<tr>
<td>isalpha</td>
<td></td>
</tr>
<tr>
<td>iswalpha</td>
<td></td>
</tr>
<tr>
<td>isascii</td>
<td></td>
</tr>
<tr>
<td>iswascii</td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Not reentrant because</td>
</tr>
<tr>
<td>----------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>iscntrl iswcntrl</td>
<td>-</td>
</tr>
<tr>
<td>isdigit iswdigit</td>
<td>-</td>
</tr>
<tr>
<td>isfinite</td>
<td>-</td>
</tr>
<tr>
<td>isgraph iswgraph</td>
<td>-</td>
</tr>
<tr>
<td>isgreater</td>
<td>-</td>
</tr>
<tr>
<td>isgreaterequal</td>
<td>-</td>
</tr>
<tr>
<td>isinf</td>
<td>-</td>
</tr>
<tr>
<td>isless</td>
<td>-</td>
</tr>
<tr>
<td>islessequal</td>
<td>-</td>
</tr>
<tr>
<td>islessgreater</td>
<td>-</td>
</tr>
<tr>
<td>islower iswlower</td>
<td>-</td>
</tr>
<tr>
<td>isnan</td>
<td>-</td>
</tr>
<tr>
<td>isnormal</td>
<td>-</td>
</tr>
<tr>
<td>isprint iswprint</td>
<td>-</td>
</tr>
<tr>
<td>ispunct iswpunct</td>
<td>-</td>
</tr>
<tr>
<td>isspace iswspace</td>
<td>-</td>
</tr>
<tr>
<td>isunordered</td>
<td>-</td>
</tr>
<tr>
<td>isupper iswupper</td>
<td>-</td>
</tr>
<tr>
<td>iswalnum</td>
<td>-</td>
</tr>
<tr>
<td>iswalpha</td>
<td>-</td>
</tr>
<tr>
<td>iswcntrl</td>
<td>-</td>
</tr>
<tr>
<td>iswctype</td>
<td>-</td>
</tr>
<tr>
<td>iswdigit</td>
<td>-</td>
</tr>
<tr>
<td>iswgraph</td>
<td>-</td>
</tr>
<tr>
<td>iswlower</td>
<td>-</td>
</tr>
<tr>
<td>iswprint</td>
<td>-</td>
</tr>
<tr>
<td>iswpunct</td>
<td>-</td>
</tr>
<tr>
<td>iswspace</td>
<td>-</td>
</tr>
<tr>
<td>iswupper</td>
<td>-</td>
</tr>
<tr>
<td>iswxdigit</td>
<td>-</td>
</tr>
<tr>
<td>isxdigit iswxdigit</td>
<td>-</td>
</tr>
<tr>
<td>Function</td>
<td>Not reentrant because</td>
</tr>
<tr>
<td>-------------------</td>
<td>------------------------------------------------------------</td>
</tr>
<tr>
<td>ldexp  ldexpl ldexpf</td>
<td>Sets errno. See (2).</td>
</tr>
<tr>
<td>lgamma lgamma1 lgammaf</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>lrint lrintf lrintl</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>llround llroundl llroundf</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>localeconv</td>
<td>N.A.; skeleton function</td>
</tr>
<tr>
<td>localtime</td>
<td></td>
</tr>
<tr>
<td>log   logl logf</td>
<td>Sets errno. See (2).</td>
</tr>
<tr>
<td>log10 log10l log10f</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>log1p log1pf log1pl</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>log2 log2f log2l</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>logb logbf logbl</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>longjmp</td>
<td></td>
</tr>
<tr>
<td>lrint lrintf lrintl</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>lround lroundl lroundf</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>lseek</td>
<td>Calls _lseek</td>
</tr>
<tr>
<td>malloc</td>
<td>Needs kernel support. See (5).</td>
</tr>
<tr>
<td>mblen</td>
<td>N.A., skeleton function</td>
</tr>
<tr>
<td>mbrlen</td>
<td>Sets errno.</td>
</tr>
<tr>
<td>mbtowc</td>
<td>Sets errno.</td>
</tr>
<tr>
<td>mbsinit</td>
<td></td>
</tr>
<tr>
<td>mbsrtowcs</td>
<td>Sets errno.</td>
</tr>
<tr>
<td>mbstowcs</td>
<td>N.A., skeleton function</td>
</tr>
<tr>
<td>mbtowc</td>
<td>N.A., skeleton function</td>
</tr>
<tr>
<td>memchr wmemchr</td>
<td></td>
</tr>
<tr>
<td>memcmp wmemcmp</td>
<td></td>
</tr>
<tr>
<td>memcpy wmemcpy</td>
<td></td>
</tr>
<tr>
<td>memmove wmemmove</td>
<td></td>
</tr>
<tr>
<td>memset wmemset</td>
<td></td>
</tr>
<tr>
<td>mktime</td>
<td></td>
</tr>
<tr>
<td>modf modfl modff</td>
<td></td>
</tr>
<tr>
<td>nan nanl nanf</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>Function</td>
<td>Not reentrant because</td>
</tr>
<tr>
<td>-----------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>nearbyint</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td></td>
<td>nearbyintf</td>
</tr>
<tr>
<td></td>
<td>nearbyintl</td>
</tr>
<tr>
<td>nextafter</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td></td>
<td>nextafterl</td>
</tr>
<tr>
<td></td>
<td>nextafterf</td>
</tr>
<tr>
<td>nexttoward</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td></td>
<td>nexttowardl</td>
</tr>
<tr>
<td></td>
<td>nexttowardf</td>
</tr>
<tr>
<td>offsetof</td>
<td>-</td>
</tr>
<tr>
<td>open</td>
<td>Calls _open</td>
</tr>
<tr>
<td>perror</td>
<td>Uses errno.</td>
</tr>
<tr>
<td>pow powl powf</td>
<td>Sets errno.</td>
</tr>
<tr>
<td>printf wprintf</td>
<td>Uses iob[].</td>
</tr>
<tr>
<td>putc putwc</td>
<td>Uses iob[].</td>
</tr>
<tr>
<td>putchar putwchar</td>
<td>Uses iob[].</td>
</tr>
<tr>
<td>puts</td>
<td>Uses iob[].</td>
</tr>
<tr>
<td>qsort</td>
<td>-</td>
</tr>
<tr>
<td>raise</td>
<td>Updates the signal handler table</td>
</tr>
<tr>
<td>rand</td>
<td>Uses static variable to remember latest random number. Must diverge from ANSI standard to define reentrant rand. See (4).</td>
</tr>
<tr>
<td>read</td>
<td>Calls _read</td>
</tr>
<tr>
<td>realloc</td>
<td>See malloc (5).</td>
</tr>
<tr>
<td>remainder</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td></td>
<td>remainderl</td>
</tr>
<tr>
<td></td>
<td>remainderf</td>
</tr>
<tr>
<td>remove</td>
<td>N.A; skeleton only.</td>
</tr>
<tr>
<td>remquo remquol remquof</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>rename</td>
<td>N.A; skeleton only.</td>
</tr>
<tr>
<td>rewind</td>
<td>N.A; skeleton only.</td>
</tr>
<tr>
<td>rint rintl rintf</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>round roundl roundf</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>scalblnlscalblnlscalblnf</td>
<td>-</td>
</tr>
<tr>
<td>scalbn scalblnlscalblnf</td>
<td>-</td>
</tr>
<tr>
<td>scanf wscanf</td>
<td>Uses iob[], calls _doscan. See (1).</td>
</tr>
<tr>
<td>setbuf</td>
<td>Sets iob[]. See (1).</td>
</tr>
<tr>
<td>setjmp</td>
<td>-</td>
</tr>
<tr>
<td>Function</td>
<td>Not reentrant because</td>
</tr>
<tr>
<td>-----------------</td>
<td>---------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>setlocale</td>
<td>N.A.; skeleton function</td>
</tr>
<tr>
<td>setvbuf</td>
<td>Sets job and calls malloc. See (1) / (5).</td>
</tr>
<tr>
<td>signal</td>
<td>Updates the signal handler table</td>
</tr>
<tr>
<td>signbit</td>
<td>-</td>
</tr>
<tr>
<td>sin, sinf, sinl</td>
<td>-</td>
</tr>
<tr>
<td>sinh, sinhf, sinh</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>snprintf, swprintf</td>
<td>Sets errno. See (2).</td>
</tr>
<tr>
<td>printf</td>
<td>Sets errno. See (2).</td>
</tr>
<tr>
<td>sqrt, sqrtf, sqrtl</td>
<td>Sets errno. See (2).</td>
</tr>
<tr>
<td>srand</td>
<td>See rand</td>
</tr>
<tr>
<td>sscanf, swscanf</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>stat</td>
<td>Uses global File System Simulation buffer, _fss_buffer</td>
</tr>
<tr>
<td>strcat, wcscat</td>
<td>-</td>
</tr>
<tr>
<td>strchr, wcschr</td>
<td>-</td>
</tr>
<tr>
<td>strcmp, wcscmp</td>
<td>-</td>
</tr>
<tr>
<td>strcoll, wscoll</td>
<td>-</td>
</tr>
<tr>
<td>strcpy, wcsncpy</td>
<td>-</td>
</tr>
<tr>
<td>strcspn, wcsncspn</td>
<td>-</td>
</tr>
<tr>
<td>strlen</td>
<td>-</td>
</tr>
<tr>
<td>strncat, wcsncat</td>
<td>-</td>
</tr>
<tr>
<td>strncmp, wcsncmp</td>
<td>-</td>
</tr>
<tr>
<td>strncpy, wcsncpy</td>
<td>-</td>
</tr>
<tr>
<td>strpbrk, wcspbrk</td>
<td>-</td>
</tr>
<tr>
<td>strrchr, wcsrchr</td>
<td>-</td>
</tr>
<tr>
<td>strspn, wcsspn</td>
<td>-</td>
</tr>
<tr>
<td>strstr, wcsstr</td>
<td>-</td>
</tr>
<tr>
<td>strtok, wcstok</td>
<td>-</td>
</tr>
<tr>
<td>strtol, wcstol</td>
<td>-</td>
</tr>
<tr>
<td>strtod, wcstod</td>
<td>-</td>
</tr>
<tr>
<td>ntoimax</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>Function</td>
<td>Not reentrant because</td>
</tr>
<tr>
<td>---------------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>strtok wcstok</td>
<td>Strtok saves last position in string in local static variable. This function is not reentrant by design. See (4).</td>
</tr>
<tr>
<td>strtol wcstol</td>
<td>Sets errno. See (2).</td>
</tr>
<tr>
<td>strtold wcstold</td>
<td>–</td>
</tr>
<tr>
<td>strtoul wcstoul</td>
<td>Sets errno. See (2).</td>
</tr>
<tr>
<td>strtoull wcstoull</td>
<td>Sets errno. See (2).</td>
</tr>
<tr>
<td>strtoumax</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>strxfrm wcsxfrm</td>
<td>–</td>
</tr>
<tr>
<td>system</td>
<td>N.A; skeleton function</td>
</tr>
<tr>
<td>tan tanf tanl</td>
<td>Sets errno. See (2).</td>
</tr>
<tr>
<td>tanh tanhf tanhl</td>
<td>Sets errno via call to other functions.</td>
</tr>
<tr>
<td>tgamma tgammaf</td>
<td><em>(Not implemented)</em></td>
</tr>
<tr>
<td>time</td>
<td>Uses static variable which defines initial start time</td>
</tr>
<tr>
<td>tmpfile</td>
<td>Uses iob[]. See (1).</td>
</tr>
<tr>
<td>tmpnam</td>
<td>Uses local buffer to build filename. Function can be adapted to use user buffer. This makes the function non ANSI. See (4).</td>
</tr>
<tr>
<td>toascii</td>
<td>–</td>
</tr>
<tr>
<td>tolower</td>
<td>–</td>
</tr>
<tr>
<td>toupper</td>
<td>–</td>
</tr>
<tr>
<td>towctrans</td>
<td>–</td>
</tr>
<tr>
<td>tolower</td>
<td>–</td>
</tr>
<tr>
<td>towlower</td>
<td>–</td>
</tr>
<tr>
<td>towupper</td>
<td>–</td>
</tr>
<tr>
<td>trunc truncl trunctf</td>
<td><em>(Not implemented)</em></td>
</tr>
<tr>
<td>ungetc ungetwc</td>
<td>Uses static buffer to hold ungetted characters for each file. Can be moved into iob structure. See (1).</td>
</tr>
<tr>
<td>unlink</td>
<td>Calls _unlink</td>
</tr>
<tr>
<td>vfprintf vfwprintf</td>
<td>Uses iob[]. See (1).</td>
</tr>
<tr>
<td>vfscanf vfwscanf</td>
<td>Calls doscan</td>
</tr>
<tr>
<td>vprintf vwprintf</td>
<td>Uses iob[]. See (1).</td>
</tr>
</tbody>
</table>
Several functions in the C library are not reentrant due to the following reasons:

- The `iob[]` structure is static. This influences all I/O functions.
- The `ungetc[]` array is static. This array holds the characters (one for each stream) when `ungetc()` is called.
- The variable `errno` is globally defined. Numerous functions read or modify `errno`.
- `_doprint` and `_doscan` use static variables for e.g. character counting in strings.
- Some string functions use locally defined (static) buffers. This is prescribed by ANSI.
- `malloc` uses a static heap space.

The following description discusses these items into more detail. The numbers at the begin of each paragraph relate to the number references in the table above.
(1) iob structures

The I/O part of the C library is not reentrant by design. This is mainly caused by the static declaration of the iob[] array. The functions which use elements of this array access these elements via pointers (FILE *).

Building a multi-process system that is created in one link-run is hard to do. The C language scoping rules for external variables make it difficult to create a private copy of the iob[] array. Currently, the iob[] array has external scope. Thus it is visible in every module involved in one link phase. If these modules comprise several tasks (processes) in a system each of which should have its private copy of iob[], it is apparent that the iob[] declaration should be changed. This requires adaption of the library to the multi-tasking environment. The library modules must use a process identification as an index for determining which iob[] array to use. Thus the library is suitable for interfacing to that kernel only.

Another approach for the iob[] declaration problem is to declare the array static in one of the modules which create a task. Thus there can be more than one iob[] array in the system without having conflicts at link time. This brings several restrictions: Only the module that holds the declaration of the static iob[] can use the standard file handles stdin, stdout and stderr (which are the first three entries in iob[]). Thus all I/O for these three file handles should be located in one module.

(2) errno declaration

Several functions in the C library set the global variable errno. After completion of the function the user program may consult this variable to see if some error occurred. Since most of the functions that set errno already have a return type (this is the reason for using errno) it is not possible to check successful completion via the return type.

The library routines can set errno to the values defined in errno.h. See the file errno.h for more information.

errno can be set to ERR_FORMAT by the print and scan functions in the C library if you specify illegal format strings.

errno will never be set to ERR_NOLONG or ERR_NOPPOINT since the Tricore C library supports long and pointer conversion routines for input and output.
errno can be set to ERANGE by the following functions: exp(), strtol(), strtoul() and tan(). These functions may produce results that are out of the valid range for the return type. If so, the result of the function will be the largest representable value for that type and errno is set to ERANGE.

errno is set to EDOM by the following functions: acos(), asin(), log(), pow() and sqrt(). If the arguments for these functions are out of their valid range (e.g. sqrt( -1 )), errno is set to EDOM.

errno can be set to ERR_POS by the file positioning functions ftell(), fsetpos() and fgetpos().

(3) ungetc
Currently the ungetc buffer is static. For each file entry in the iob[] structure array, there is one character available in the buffer to unget a character.

(4) local buffers
tmpnam() creates a temporary filename and returns a pointer to a local static buffer. This is according to the ANSI definition. Changing this function such that it creates the name in a user specified buffer requires another calling interface. Thus the function would be no longer portable.

strtok() scans through a string and remembers that the string and the position in the string for subsequent calls. This function is not reentrant by design. Making it reentrant requires support of a kernel to store the information on a per process basis.

rand() generates a sequence of random numbers. The function uses the value returned by a previous call to generate the next value in the sequence. This function can be made reentrant by specifying the previous random value as one of the arguments. However, then it is no longer a standard function.

(5) malloc
Malloc uses a heap space which is assigned at locate time. Thus this implementation is not reentrant. Making a reentrant malloc requires some sort of system call to obtain free memory space on a per process basis. This is not easy to solve within the current context of the library. This requires adaption to a kernel.
This paragraph on reentrancy applies to multi-process environments only. If reentrancy is required for calling library functions from an exception handler, another approach is required. For such a situation it is of no use to allocate e.g. multiple \texttt{iob[ ]} structures. In such a situation several pieces of code in the library have to be declared 'atomic': this means that interrupts have to be disabled while executing an atomic piece of code.
CHAPTER 3

TRICORE ASSEMBLY LANGUAGE
3.1 INTRODUCTION

This chapter contains a detailed description of all built-in assembly functions directives and controls. For a description of the TriCore instruction set, refer to the *TriCore Architecture v1.3 Manual* [2000, Infineon].

3.2 BUILT-IN ASSEMBLY FUNCTIONS

3.2.1 OVERVIEW OF BUILT-IN ASSEMBLY FUNCTIONS

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

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

The following tables provide an overview of all built-in assembler functions. *expr* can be any assembly expression resulting in an integer value. Expressions are explained in section 4.6, *Assembly Expressions*, in chapter *Assembly Language* of the *User’s Manual*. 
### Overview of mathematical functions

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>@CVF(expr)</code></td>
<td>Convert integer to floating-point</td>
</tr>
<tr>
<td><code>@CVI(expr)</code></td>
<td>Convert floating-point to integer</td>
</tr>
<tr>
<td><code>@FLD(base,value, width[,start])</code></td>
<td>Shift and mask operation</td>
</tr>
<tr>
<td><code>@FRACT(expr)</code></td>
<td>Convert floating-point to 32-bit fractional</td>
</tr>
<tr>
<td><code>@SFRACT(expr)</code></td>
<td>Convert floating-point to 16-bit fractional</td>
</tr>
<tr>
<td><code>@LNG(expr)</code></td>
<td>Concatenate to double word</td>
</tr>
<tr>
<td><code>@LUN(expr)</code></td>
<td>Convert long fractional to floating-point</td>
</tr>
<tr>
<td><code>@RVB(expr1[,expr2])</code></td>
<td>Reverse order of bits in field</td>
</tr>
<tr>
<td><code>@UNF(expr)</code></td>
<td>Convert fractional to floating-point</td>
</tr>
</tbody>
</table>

### Overview of string functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>@CAT(str1,str2)</code></td>
<td>Concatenate strings</td>
</tr>
<tr>
<td><code>@LEN(string)</code></td>
<td>Length of string</td>
</tr>
<tr>
<td><code>@POS(str1,str2[,start])</code></td>
<td>Position of substring in string</td>
</tr>
<tr>
<td><code>@SCP(str1,str2)</code></td>
<td>Returns 1 if two strings are equal</td>
</tr>
<tr>
<td><code>@SUB(string,expr,expr)</code></td>
<td>Returns a substring</td>
</tr>
</tbody>
</table>

### Overview of macro functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>`@ARG('symbol'</td>
<td>expr)`</td>
</tr>
<tr>
<td><code>@CNT()</code></td>
<td>Return number of macro arguments</td>
</tr>
<tr>
<td><code>@MAC(symbol)</code></td>
<td>Test if macro is defined</td>
</tr>
<tr>
<td><code>@MXP()</code></td>
<td>Test if macro expansion is active</td>
</tr>
</tbody>
</table>
Overview of address calculation functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@HI(expr)</td>
<td>Returns upper 16 bits of expression value</td>
</tr>
<tr>
<td>@HIS(expr)</td>
<td>Returns upper 16 bits of expression value, adjusted for signed addition</td>
</tr>
<tr>
<td>@LO(expr)</td>
<td>Returns lower 16 bits of expression value</td>
</tr>
<tr>
<td>@LOS(expr)</td>
<td>Returns lower 16 bits of expression value, adjusted for signed addition</td>
</tr>
<tr>
<td>@LSB(expr)</td>
<td>Get least significant byte of a word</td>
</tr>
<tr>
<td>@MSB(expr)</td>
<td>Get most significant byte of a word</td>
</tr>
</tbody>
</table>

Overview of assembler mode functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@ASPCP()</td>
<td>Returns the name of the PCP assembler executable</td>
</tr>
<tr>
<td>@ASTC()</td>
<td>Returns the name of the assembler executable</td>
</tr>
<tr>
<td>@CPU(string)</td>
<td>Test if CPU type is selected</td>
</tr>
<tr>
<td>@DEF('symbol'</td>
<td>symbol)</td>
</tr>
<tr>
<td>@EXP(expr)</td>
<td>Expression check</td>
</tr>
<tr>
<td>@INT(expr)</td>
<td>Integer check</td>
</tr>
<tr>
<td>@LST()</td>
<td>LIST control flag value</td>
</tr>
</tbody>
</table>

3.2.2 Detailed Description of Built-in Assembly Functions

@ABS(expression)

Returns the absolute value of expression as an integer value.

Example:

```
AVAL .SET @ABS(-2.1) ; AVAL = 2
```
@ACS(expression)

Returns the arc cosine of expression as a floating-point value in the range zero to pi. The result of expression must be between -1 and 1.

Example:

ACOS .SET @ACS(-1.0) ;ACOS = 3.1415926535897931

@ARG(symbol | expression)

Returns an 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 formal argument name. If the argument is an expression it refers to the ordinal position of the argument in the macro formal argument list. The assembler issues a warning if this function is used when no macro expansion is active.

Example:

.IF @ARG('TWIDDLE') ;twiddle factor provided?
.IF @ARG(1) ;is first argument present?

@ASN(expression)

Returns the arc sine of expression as a floating-point value in the range -pi/2 to pi/2. The result of expression must be between -1 and 1.

Example:

ARCSINE .SET @ASN(-1.0) ;ARCSINE = -1.570796

@ASPCP()

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

Example:

ANAME: .byte @ASPCP() ; ANAME = ‘aspcp’

@ASTC()

Returns the name of the assembler executable. This is ‘astc’ for the TriCore assembler.

Example:

ANAME: .byte @ASTC() ; ANAME = ‘astc’
@AT2(expr1,expr2)

Returns the arc tangent of expr1/expr2 as a floating-point value in the range −pi to pi. Expr1 and expr2 must be separated by a comma.

Example:

    ATAN2   .EQU   @AT2(-1.0,1.0) ; ATAN2 = −0.7853982

@ATN(expression)

Returns the arc tangent of expression as a floating-point value in the range −pi/2 to pi/2.

Example:

    ATAN   .SET   @ATN(1.0) ; ATAN = 0.78539816339744828

@CAT(string1,string2)

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

Example:

    .DEFINE   ID   "@CAT(‘Tri’,‘Core’)" ; ID = ’TriCore’

@CEL(expression)

Returns a floating-point value which represents the smallest integer greater than or equal to expression.

Example:

    CEIL   .SET   @CEL(-1.05) ; CEIL = −1.0

@CNT()

Returns the number of arguments of the current macro expansion as an integer. The assembler issues a warning if this function is used when no macro expansion is active.

Example:

    ARGCNT   .SET   @CNT() ; reserve argument count
@COH(expression)

Returns the hyperbolic cosine of expression as a floating-point value.

Example:

HYCOS .EQU @COH(VAL) ;compute hyperbolic cosine

@COS(expression)

Returns the cosine of expression as a floating-point value.

Example:

.WORD -@COS(@CVF(COUNT)*FREQ) ;compute cosine value

@CPU(string)

Returns an integer 1 if string corresponds to the selected CPU type; 0 otherwise. See also the assembler option -C (Select CPU).

Example:

IF @CPU("tc2") ;TriCore 2 specific part

@CVF(expression)

Converts the result of expression to a floating-point value.

Example:

FLOAT .SET @CVF(5) ;FLOAT = 5.0

@CVI(expression)

Converts the result of expression to an integer value. This function should be used with caution since the conversions can be inexact (e.g., floating-point values are truncated).

Example:

INT .SET @CVI(-1.05) ;INT = -1
@DEF("symbol" | symbol)

Returns an integer 1 if symbol has been defined, 0 otherwise. symbol can be any symbol or label not associated with a .MACRO or .SDECL 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 symbol or label.

Example:

```.
.IF @DEF('ANGLE') ;is symbol ANGLE defined?
.IF @DEF(ANGLE) ;does label ANGLE exist?
```}

@EXP(expression)

Returns 0 if the evaluation of expression would normally result in an error. Returns 1 if the expression can be evaluated correctly. With the @EXP function, you prevent the assembler of generating an error if expression contains an error. No test is made by the assembler for warnings. The expression may be relative or absolute.

Example:

```.
.IF !@EXP(3/0) ;Do the IF on error
;assembler generates no error

.IF !(3/0) ;assembler generates an error
```}

@FLD(base,value,width,[start])

Shift and mask value into base for width bits beginning at bit start. If start is omitted, zero (least significant bit) is assumed. All arguments must be positive integers and none may be greater than the target word size. Returns the shifted and masked value.

Example:

```.
VAR1 .EQU @FLD(0,1,1) ;turn bit 0 on, VAR1=1
VAR2 .EQU @FLD(0,3,1) ;turn bit 0 on, VAR2=1
VAR3 .EQU @FLD(0,3,2) ;turn bits 0 and 1 on, VAR3=3
VAR4 .EQU @FLD(0,3,2,1) ;turn bits 1 and 2 on, VAR4=6
VAR5 .EQU @FLD(0,1,1,7) ;turn eighth bit on, VAR5=0x80
```
@FLR(expression)
Returns a floating-point value which represents the largest integer less than or equal to expression.

Example:
```
FLOOR .SET @FLR(2.5) ;FLOOR = 2.0
```

@FRACT(expression)
This function returns the 32-bit fractional representation 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(expression) is equivalent to ((expression>>16) & 0xffff).

Example:
```
mov.u d2,#LO(COUNT)
addih d2,d2,#HI(COUNT) ; upper 16 bits of COUNT
```

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

Example:
```
movh.a a3,#HIS(label)
lea a3,[a3]@LOS(label)
```

@INT(expression)
Returns an integer 1 if expression has an integer result; otherwise, it returns a 0. The expression may be relative or absolute.

Example:
```
.IF @INT(TERM) ;Test if result is an integer
```
@L10(expression)
Returns the base 10 logarithm of expression as a floating-point value. expression must be greater than zero.
Example:
```assembly
LOG .EQU @L10(100.0) ;LOG = 2
```

@LEN(string)
Returns the length of string as an integer.
Example:
```assembly
SLEN .SET @LEN('string') ;SLEN = 6
```

@LNG(expr1,expr2)
Concatenates the 16-bit expr1 and expr2 into a 32-bit word value such that expr1 is the high half and expr2 is the low half.
Example:
```assembly
LWORD .WORD @LNG(HI,LO) ;build long word
```

@LO(expression)
Returns the lower 16 bits of a value. @LO(expression) is equivalent to expression & 0xffff.
Example:
```assembly
mov.u d2,#@LO(COUNT) ;lower 16 bits of COUNT
addih d2,d2,#@HI(COUNT)
```

@LOG(expression)
Returns the natural logarithm of expression as a floating-point value. expression must be greater than zero.
Example:
```assembly
LOG .EQU @LOG(100.0) ;LOG = 4.605170
```
@LOS(expression)

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

Example:

```assembly
movh.a a3,#@HIS(label)
lea a3,[a3]@LOS(label)
```

@LSB(expression)

Returns the least significant byte of the result of the expression. expression is interpreted as a half word (16 bit).

Example:

```assembly
VAR1 .SET @LSB(0x34) ;VAR1 = 0x34
VAR2 .SET @LSB(0x1234) ;VAR2 = 0x34
VAR3 .SET @LSB(0x654321) ;VAR3 = 0x21
```

@LST()

Returns the value of the $LIST ON/OFF 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:

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

@LUN(expression)

Converts the 32-bit expression to a floating-point value. expression should represent a binary fraction.

Example:

```assembly
DBLFRC1 .EQU @LUN(0x40000000) ;DBLFRC1 = 0.5
DBLFRC2 .EQU @LUN(3928472) ;DBLFRC2 = 0.007354736
DBLFRC3 .EQU @LUN(0xE0000000) ;DBLFRC3 = -0.75
```
@MAC(symbol)
Returns an integer 1 if symbol has been defined as a macro name, 0 otherwise.

Example:

    .IF   @MAC(DOMUL) ;does macro DOMUL exist?

@MAX(expr1,exprN)...
Returns the greatest of expr1,...,exprN as a floating-point value.

Example:

    MAX:  .BYTE @MAX(1,-2.137,3.5) ;MAX = 3.5

@MIN(expr1,exprN)...
Returns the least of expr1,...,exprN as a floating-point value.

Example:

    MIN:  .BYTE @MIN(1,-2.137,3.5) ;MIN = -2.137

@MSB(expression)
Returns the most significant byte of the result of the expression. expression is interpreted as a half word (16 bit).

Example:

    VAR1  .SET @MSB(0x34) ;VAR1 = 0x00
    VAR2  .SET @MSB(0x1234) ;VAR2 = 0x12
    VAR3  .SET @MSB(0x654321) ;VAR3 = 0x43

@MXP()
Returns an integer 1 if the assembler is expanding a macro, 0 otherwise.

Example:

    .IF   @MXP() ;macro expansion active?
@\textit{POS}(str1,str2[,start])

Returns the position of \textit{str2} in \textit{str1} as an integer, starting at position \textit{start}. If \textit{start} is not given the search begins at the beginning of \textit{str1}. If the \textit{start} argument is specified it must be a positive integer and cannot exceed the length of the source string. Note that the first position in a string is position 0.

Example:

\begin{verbatim}
ID    .EQU  @POS('TriCore','Core')    ;ID = 3
ID2   .EQU  @POS('ABCDABCD','B',2)   ;ID2 = 5
\end{verbatim}

@\textit{POW}(expr1,expr2)

Returns \textit{expr1} raised to the power \textit{expr2} as a floating-point value. \textit{expr1} and \textit{expr2} must be separated by a comma.

Example:

\begin{verbatim}
BUF    .EQU  @CVI(@POW(2.0,3.0))     ;BUF = 8
\end{verbatim}

@\textit{RND()}

Returns a random value in the range 0.0 to 1.0.

Example:

\begin{verbatim}
SEED   .EQU  @RND()                   ;save initial SEED value
\end{verbatim}

@\textit{RVB}(expr1,expr2)

Reverse the order of bits in \textit{expr1} delimited by the number of bits in \textit{expr2}. If \textit{expr2} is omitted the field is bounded by the target word size. Both expressions must be 16–bit integer values.

Example:

\begin{verbatim}
VAR1   .SET  @RVB(0x200)            ;reverse all bits, VAR1=0x40
VAR2   .SET  @RVB(0xB02)           ;reverse all bits, VAR2=0x40D0
VAR3   .SET  @RVB(0xB02,2)         ;reverse bits 0 and 1, VAR3=0xB01
\end{verbatim}
@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?```

@SFRACT(expression)

This function returns the 16-bit fractional representation 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:

```
VAR1  .SET  @SGN(-1.2e-92) ;VAR1 = -1
VAR2  .SET  @SGN(0) ;VAR2 = 0
VAR3  .SET  @SGN(28.382) ;VAR3 = 1
```

@SIN(expression)

Returns the sine of expression as a floating-point value.

Example:

```.WORD   @SIN(@CVF(COUNT)*FREQ) ;compute sine value```

@SNH(expression)

Returns the hyperbolic sine of expression as a floating-point value.

Example:

```
HSINE  .EQU   @SNH(VAL) ;hyperbolic sine```
@SQRT(expression)

Returns the square root of expression as a floating-point value. expression must be positive.

Example:

```assembly
SQR1  .EQU @SQRT(3.5) ;SQR1 = 1.870829
SQR2  .EQU @SQRT(16)  ;SQR2 = 4
```

@SUB(string, expression1, expression2)

Returns the substring from string as a string. Expression1 is the starting position within string, and expression2 is the length of the desired string. The assembler issues an error if either expression1 or expression2 exceeds the length of string. Note that the first position in a string is position 0.

Example:

```assembly
.DEFINED @SUB(‘TriCore’,3,4) ;ID = ‘Core’
```

@TAN(expression)

Returns the tangent of expression as a floating-point value.

Example:

```assembly
TANGENT .SET @TAN(1.0) ;TANGENT = 1.5574077
```

@TANH(expression)

Returns the hyperbolic tangent of expression as a floating-point value.

Example:

```assembly
HTAN  .SET @TANH(1) ;HTAN = 0.76159415595
```

@UNF(expression)

Converts expression to a floating-point value. expression should represent a 16-bit binary fraction.

Example:

```assembly
FRC   .EQU @UNF(0x4000) ;FRC = 0.5
```
@XPN(expression)

Returns the exponential function (base e raised to the power of expression) as a floating-point value.

Example:

```
EXP    .EQU  @XPN(1.0)  ;EXP = 2.718282
```
3.3 ASSEMBLER DIRECTIVES AND CONTROLS

3.3.1 OVERVIEW OF ASSEMBLER DIRECTIVES

Assembler directives are grouped in the following categories:

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

The following tables provide an overview of all assembler directives.

**Overview of assembly control directives**

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.COMMENT</td>
<td>Start comment lines</td>
</tr>
<tr>
<td>.DEFINE</td>
<td>Define substitution string</td>
</tr>
<tr>
<td>.END</td>
<td>End of source program</td>
</tr>
<tr>
<td>.FAIL</td>
<td>Programmer generated error message</td>
</tr>
<tr>
<td>.INCLUDE</td>
<td>Include file</td>
</tr>
<tr>
<td>.MESSAGE</td>
<td>Programmer generated message</td>
</tr>
<tr>
<td>.ORG</td>
<td>Initialize memory space and location counters to create a nameless section</td>
</tr>
<tr>
<td>.SDECL</td>
<td>Declare a section with name, type and attributes</td>
</tr>
<tr>
<td>.SECT</td>
<td>Activate a declared section</td>
</tr>
<tr>
<td>.UNDEF</td>
<td>Undefined DEFINE symbol</td>
</tr>
<tr>
<td>.WARNING</td>
<td>Programmer generated warning</td>
</tr>
</tbody>
</table>
## Overview of symbol definition directives

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.EQU</td>
<td>Assign permanent value to a symbol</td>
</tr>
<tr>
<td>.EXTERN</td>
<td>External symbol declaration</td>
</tr>
<tr>
<td>.GLOBAL</td>
<td>Global section symbol declaration</td>
</tr>
<tr>
<td>.LOCAL</td>
<td>Local symbol declaration</td>
</tr>
<tr>
<td>.SET</td>
<td>Set temporary value to a symbol</td>
</tr>
<tr>
<td>.SIZE</td>
<td>Set size of symbol in the ELF symbol table</td>
</tr>
<tr>
<td>.TYPE</td>
<td>Set symbol type in the ELF symbol table</td>
</tr>
<tr>
<td>.WEAK</td>
<td>Mark symbol as 'weak'</td>
</tr>
</tbody>
</table>

## Overview of data definition / storage allocation directives

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.ACCUM</td>
<td>Define 64–bit constant of 18 + 46 bits format</td>
</tr>
<tr>
<td>.ALIGN</td>
<td>Define alignment</td>
</tr>
<tr>
<td>.ASCII / .ASCIIZ</td>
<td>Define ASCII string without / with ending NULL byte</td>
</tr>
<tr>
<td>.BYTE</td>
<td>Define constant byte</td>
</tr>
<tr>
<td>.FLOAT / .DOUBLE</td>
<td>Define a 32–bit / 64–bit floating-point constant</td>
</tr>
<tr>
<td>.FRACT / .SFRACT</td>
<td>Define a 16–bit / 32–bit constant fraction</td>
</tr>
<tr>
<td>.SPACE</td>
<td>Define storage</td>
</tr>
<tr>
<td>.WORD / .HALF</td>
<td>Define a word / half-word constant</td>
</tr>
</tbody>
</table>
Overview of macro and conditional assembly directives

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.DUP / .ENDM</td>
<td>Duplicate sequence of source lines</td>
</tr>
<tr>
<td>.DUPA / .ENDM</td>
<td>Duplicate sequence with arguments</td>
</tr>
<tr>
<td>.DUPC / .ENDM</td>
<td>Duplicate sequence with characters</td>
</tr>
<tr>
<td>.DUPF / .ENDM</td>
<td>Duplicate sequence in loop</td>
</tr>
<tr>
<td>.EXITM</td>
<td>Exit macro</td>
</tr>
<tr>
<td>.IF / .ELIF / .ELSE / .ENDIF</td>
<td>Conditional assembly</td>
</tr>
<tr>
<td>.MACRO / .ENDM</td>
<td>Define macro</td>
</tr>
<tr>
<td>.PMACRO</td>
<td>Undefine (purge) macro definition</td>
</tr>
</tbody>
</table>

Overview of debug directives

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.CALLS</td>
<td>Passes call information to object file. Used by the linker to build a call graph and calculate stack size.</td>
</tr>
</tbody>
</table>

3.3.2 Detailed Description of Assembler Directives

Some assembler directives can be preceeded with a label. If you do not preceede an assembler directive with a label, you must use white space instead (spaces or tabs). The assembler recognizes both upper and lower case for directives.
.ACCUM

Syntax

[label:]    .ACCUM  expression[,expression]...

Description

With the .ACCUM directive (Define 64-bit Constant) the assembler allocates and initializes two words of memory (64 bits) for each argument. Use commas to separate multiple arguments.

An expression can be:

- a fractional fixed point expression (range \([-2^{17}, 2^{17}\))
- NULL (indicated by two adjacent commas: ,,)

Multiple arguments are stored in successive address locations in sets of two bytes. If an argument is NULL its corresponding address location is filled with zeros.

If the evaluated expression is out of the range \([-2^{17}, 2^{17}\), the assembler issues a warning and saturates the fractional value.

Example

ACC: .ACCUM 0.1,0.2,0.3

Related information

.SPACE (Define storage)

.FRACT / .SFRACT (Define 32-bit / 16-bit constant fraction)
.ALIGN

Syntax

.ALIGN expression

Description

With the .ALIGN directive you instruct the assembler to align the location counter. By default the assembler aligns on one byte.

When the assembler encounters the .ALIGN directive, it advances the location counter to an address that is aligned as specified by expression and places the next instruction or directive on that address. The alignment is in minimal addressable units (MAUs). The assembler fills the 'gap' with NOP instructions for code sections or with zeros for data sections. If the location counter is already aligned on the specified alignment, it remains unchanged. The location of absolute sections will not be changed.

The expression must be a power of two: 2, 4, 8, 16, ... If you specify another value, the assembler changes the alignment to the next higher power of two and issues a warning.

The assembler aligns sections automatically to the largest alignment value occurring in that section.

A label is not allowed before this directive.

Example

.ALIGN 16
add d2,d2,d4 ; the assembler aligns
 ; this instruction at 16 bytes and
 ; fills the 'gap' with NOP instructions

.ALIGN 12
add d2,d2,d4 ; WRONG: not a power of two, the
 ; assembler aligns this instruction at
 ; 16 bytes and issues a warning

Related information
.ASCII/.ASCIIIZ

Syntax

[label:] .ASCII  string[string]...
[label:] .ASCIIIZ  string[string]...

Description

With the .ASCII or .ASCIIIZ directive the assembler allocates and initializes memory for each string argument.

The .ASCII directive does not add a NULL byte to the end of the string. The .ASCIIIZ directive does add a NULL byte to the end of the string. The "z" in .ASCIIIZ stands for "zero". Use commas to separate multiple strings.

Example

STRING:  .ASCII  "Hello world"
STRINGZ: .ASCIIIZ "Hello world"

With the .BYTE directive you can obtain exactly the same effect:

STRING:  .BYTE  "Hello world"  ; without a NULL byte
STRINGZ: .BYTE  "Hello world",0  ; with a NULL byte

Related information

.SPACE (Define storage)

.BYTE (Define a constant byte)
.WORD/.HALF (Define a word/halfword)
.BYTE

Syntax

[label] .BYTE argument[argument]...

Description

With the .BYTE directive (Define Constant Byte) the assembler allocates and initializes a byte of memory for each argument.

An argument can be:

- a single or multiple character string constant
- an integer expression
- NULL (indicated by two adjacent commas: ,,)

Multiple arguments are stored in successive byte locations. If an argument is NULL its corresponding byte location is filled with zeros.

If you specify label, it gets the value of the location counter at the start of the directive processing.

Integer arguments are stored as is, but must be byte values (within the range 0–255); floating-point numbers are not allowed. If the evaluated expression is out of the range [-256, +255] the assembler issues an error. For negative values within that range, the assembler adds 256 to the specified value (for example, -254 is stored as 2).

In case of single and multiple character strings, each character is stored in consecutive bytes whose lower seven bits represent the ASCII value of the character. The standard C escape sequences are allowed:

\[
\begin{align*}
\text{.BYTE} & \quad \text{‘R’} \quad ; \quad = 0x52 \\
\text{.BYTE} & \quad \text{‘AB’,,’D’} \quad ; \quad = 0x41420043
\end{align*}
\]

Example

TABLE .BYTE ‘two’,0,’strings’,0
CHARS .BYTE ‘A’,’B’,’C’,’D’

Related information

.SPICE (Define storage)

.ASCII / .ASCIIZ (Define ASCII string without/with ending NULL)
.WORD / .HALF (Define a word / halfword)
.CALLS

Syntax

.CALLS 'caller', 'callee' [call_frequency [stack_usage]...]

Description

Create a flow graph reference between caller and callee. With this information the linker can build a call graph and calculate stack size. Caller and callee are names of functions. The call_frequency shows how many times the callee is called. The stack_usage represents the stack usage in bytes at the location of the call or the maximum stack usage of function caller. A function can use multiple stacks.

The compiler inserts .CALLS directives automatically to pass call tree information. Normally it is not necessary to use the .CALLS directive in hand coded assembly.

A label is not allowed before this directive.

Example

.CALLS 'main', 'func', 1, 8

Indicates that the function main calls the function func 1 time and that the stack usage at the location of the call is 8 bytes.

.CALLS 'main', '', 0, 8

Specifies the maximum stack usage of function main (8 bytes).

Related information
.COMMENT

Syntax

```plaintext
.COMMENT  delimiter
.
delimiter
```

Description

With the .COMMENT directive (Start Comment Lines) you can 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 before this directive.

Example

```plaintext
.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.

* 
```

Related information
.DEFINE

Syntax

.DEFINEx symbol string

Description

With the .DEFINE directive you define a substitution string that you can use on all following source lines. The assembler searches all succeeding lines for an occurrence of symbol, and replaces it with string. If the symbol occurs in a double quoted string it is also replaced. Strings between single quotes are not expanded.

This directive is useful for providing better documentation in the source program. A symbol can consist of letters, digits and underscore characters (_), and the first character cannot be a digit.

The assembler issues a warning if you redefine an existing symbol.

Macros represent a special case. .DEFINE directive translations are 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 before this directive.

Example

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

.DEFINEx LEN '32'

then the source line below:

.SPACE LEN
MSG "The length is: LEN"

would be transformed by the assembler to the following:

.SPACE 32
MSG "The length is: 32"

Related information

.UNDEF (Undefine .DEFINE symbol)
.SET (Set temporary value to a symbol)
.DUP / .ENDM

Syntax

[label] .DUP expression
.
.
.ENDM

Description

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). You can nest the .DUP directive to any level.

If you specify label, it gets the value of the location counter at the start of the .DUP directive processing.

Example

Consider the following source input statements,

```
COUNT .SET 3
  .DUP COUNT ; duplicate NOP count times
  NOP
  .ENDM
```

This is expanded as follows:

```
COUNT .SET 3
  NOP
  NOP
  NOP
```

Related information

.DUPA (Duplicate Sequence with Arguments),
.DUPC (Duplicate Sequence with Characters),
.DUPF (Duplicate Sequence in Loop),
.MACRO (Define Macro)
**.DUPA / .ENDM**

**Syntax**

```
[label] .DUPA formal_arg,argument[argument]...
.
.
.ENDM
```

**Description**

With the `.DUPA` and `.ENDM` directives (Duplicate Sequence with Arguments) you can repeat a block of source statements for each `argument`. For each repetition, every occurrence of the `formal_arg` parameter within the block is replaced with each succeeding `argument` string. If an argument includes an embedded blank or other assembler–significant character, it must be enclosed with single quotes.

If you specify `label`, it gets the value of the location counter at the start of the `.DUPA` directive processing.

**Example**

Consider the following source input statements,

```
.DUPA VALUE,12,,32,34
.BYTE VALUE
.ENDM
```

This is expanded as follows:

```
.BYTE 12
.BYTE VALUE ; results in a warning
.BYTE 32
.BYTE 34
```

The second statement results in a warning of the assembler that the local symbol `VALUE` is not defined in this module and is made external.

**Related information**

- `.DUP` (Duplicate Sequence of Source Lines),
- `.DUPC` (Duplicate Sequence with Characters),
- `.DUPF` (Duplicate Sequence in Loop),
- `.MACRO` (Define Macro)
.DUPC / .ENDM

Syntax

[label] .DUPC formal_arg,string

.

.ENDM

Description

With the .DUPC and .ENDM directives (Duplicate Sequence with Characters) you can repeat a block of source statements for each character within string. For each character in the string, the formal_arg parameter within the block is replaced with that character. If the string is empty, then the block is skipped.

If you specify label, it gets the value of the location counter at the start of the .DUPC directive processing.

Example

Consider the following source input statements,

.DUPC VALUE,'123'
.BYTE VALUE
.ENDM

This is expanded as follows:

.BYTE 1
.BYTE 2
.BYTE 3

Related information

.DUP (Duplicate Sequence of Source Lines),
.DUPE (Duplicate Sequence with Arguments),
.DUPF (Duplicate Sequence in Loop),
.MACRO (Define Macro)
.DUPF / .ENDM

Syntax

[label] .DUPF formal_arg, [start], end, [increment]
.
.
.DENM

Description

With the .DUPF and .ENDM directives (Duplicate Sequence in Loop) you can repeat a block of source statements \((end - start) + 1 / increment\) times. \(Start\) 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 \(formal\_arg\) parameter holds the loop index value and may be used within the body of instructions.

If you specify \(label\), it gets the value of the location counter at the start of the .DUPF directive processing.

Example

Consider the following source input statements,

```
.DUPF   NUM, 0, 7
MOV    D\NUM, #0
 .ENDM
```

This is expanded as follows:

```
MOV    D0, #0
MOV    D1, #0
MOV    D2, #0
MOV    D3, #0
MOV    D4, #0
MOV    D5, #0
MOV    D6, #0
MOV    D7, #0
```

Related information

- .DUP (Duplicate Sequence of Source Lines),
- .DUPA (Duplicate Sequence with Arguments),
- .DUPC (Duplicate Sequence with Characters),
- .MACRO (Define Macro)
.END

Syntax

.END  [expression]

Description

With the optional .END directive you tell the assembler that the logical end
of the source program is reached. If the assembler finds assembly source
lines beyond the .END directive, it ignores those lines and issues a
warning.

The expression is only permitted here for compatibility reasons. It is
ignored during assembly.

You cannot use the .END directive in a macro expansion.

A label is not allowed before this directive.

Example

.END ;End of source program

Related information

–
.EQU

Syntax

\textit{symbol} \texttt{.EQU} \textit{expression}

Description

With the \texttt{.EQU} directive you assign the value of \textit{expression} to \textit{symbol} permanently. Once defined, you cannot redefine the \textit{symbol}.

The \textit{expression} can be relocatable or absolute and forward references are allowed.

Example

To assign the value 0x4000 permanently to the symbol \texttt{A\_D\_PORT}:

\begin{verbatim}
A_D_PORT .EQU 0x4000
\end{verbatim}

You cannot redefine the symbol \texttt{A\_D\_PORT} after this.

Related information

\texttt{.SET} (Set temporary value to a symbol)
**.EXITM**

**Syntax**

```
.EXITM
```

**Description**

With the `.EXITM` directive (Exit Macro) the assembler will immediately terminate a macro expansion. It is useful when you use it with the conditional assembly directive `.IF` to terminate macro expansion when, for example, error conditions are detected.

A label is not allowed before this directive.

**Example**

```
CALC .MACRO XVAL,YVAL
    .IF XVAL<0
        .FAIL 'Macro parameter value out of range'
        .EXITM ; Exit macro
    .ENDIF
. .
. .
. ENDM
```

**Related information**

- `.DUP` (Duplicate Sequence of Source Lines),
- `.DUPA` (Duplicate Sequence with Arguments),
- `.DUPC` (Duplicate Sequence with Characters),
- `.DUPF` (Duplicate Sequence in Loop),
- `.MACRO` (Define Macro)
.EXTERN

Syntax

.EXTERN  symbol[,...]

Description

With the .EXTERN directive (External Symbol Declaration) you 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 with the .GLOBAL directive.

If you do not use the .EXTERN directive to specify that a symbol is defined externally and the symbol is not defined within the current module, the assembler issues a warning and inserts the .EXTERN directive for that symbol.

A label is not allowed before this directive.

Example

.EXTERN AA,CC,DD  ;defined elsewhere

Related information

.GLOBAL (Global symbol declaration)
.LOCAL  (Local symbol declaration)
**.FAIL**

**Syntax**

```
.FAIL  [(string | exp),(string | exp)...]
```

**Description**

With the `.FAIL` directive (Programmer Generated Error) you tell the assembler to output an error message during the assembling process.

The total error count will be incremented as with any other error. The `.FAIL` directive is for example useful in combination with conditional assembly for exceptional condition checking. The assembly process proceeds normally after the error has been printed.

Optionally, you can specify an arbitrary number of strings and expressions, in any order but separated by commas, to describe the nature of the generated error. If you use expressions, the assembler outputs the result. The assembler outputs a space between each argument.

With this directive the assembler exits with exit code 1 (an error).

A label is not allowed before this directive.

**Example**

```
.FAIL  ‘Parameter out of range’
```

This results in the error:

```
E143: ["filename" line] Parameter out of range
```

**Related information**

- `.MESSAGE` (Programmer Generated Message),
- `.WARNING` (Programmer Generated Warning)
**.FLOAT/.DOUBLE**

**Syntax**

```
[label] .FLOAT expression[,expression]...
[label] .DOUBLE expression[,expression]...
```

**Description**

With the .FLOAT or .DOUBLE directive the assembler allocates and initializes a floating-point number (32 bits) or a double (64 bits) in memory for each argument.

An expression can be:

- a floating-point expression
- NULL (indicated by two adjacent commas: ,)

You can represent a constant as a signed whole number with fraction or with the 'e' format as used in the C language. 12.457 and +0.27E-13 are legal floating-point constants.

If you specify label, it gets the value of the location counter at the start of the directive processing.

If the evaluated argument is too large to be represented in a single word / double-word, the assembler issues an error and truncates the value.

**Examples**

```
FLT: .FLOAT 12.457,+0.27E-13
DBL: .DOUBLE 12.457,+0.27E-13
```

**Related information**

.SPACES (Define storage)
**.FRACT/.SFRACT**

**Syntax**

```
[label:] .FRACT  expression[,expression]...
```

```
[label:] .SFRACT  expression[,expression]...
```

**Description**

With the .FRACT or .SFRACT directive the assembler allocates and initializes one word of memory (32 bits) or a halfword (16 bits) for each argument. Use commas to separate multiple arguments.

An *expression* can be:

- a fractional fixed point expression (range [-1, +1>)
- NULL (indicated by two adjacent commas: `,,`)  

Multiple arguments are stored in successive address locations in sets of two bytes. If an argument is NULL its corresponding address location is filled with zeros.

If the evaluated argument is out of the range [-1, +1>, the assembler issues a warning and saturates the fractional value.

**Example**

```
FRCT: .FRACT  0.1,0.2,0.3
SFRCT: .SFRACT  0.1,0.2,0.3
```

**Related information**

- **.SPACE** (Define storage)
- **.ACCUM** (Define 64-bit constant fraction in 18+46 bits format)
.GLOBAL

Syntax

.GLOBAL  symbol[,symbol]...

Description

All symbols or labels defined in the current section or module are local to the module by default. You can change this default behavior with assembler option -ig.

With the .GLOBAL directive (Global Section Symbol Declaration) you declare one of more symbols as global. This means that the specified symbols are defined within the current section or module, and that those definitions should be accessible by all modules, using the EXTERN directive.

Only symbols that are defined with the .EQU directive or program labels can be made global.

If the symbols that appear in the operand field are not used in the module, the assembler gives a warning.

A label is not allowed before this directive.

Example

.SDECL "data.io",DATA
.SECT "data.io"
.GLOBAL LOOPA  ; LOOPA will be globally accessible by other modules
LOOPA .HALF 0x100  ; assigns the value 0x100 to LOOPA

Related information

.EXTERN (External symbol declaration)
.LOCAL  (Local symbol declaration)
.IF / .ELIF / .ELSE / .ENDIF

Syntax

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

Description

With the .IF/ .ENDIF directives you can create a part of conditional assembly code. The assembler assembles only the code that matches a specified condition.

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

You can nest .IF directives to any level. The .ELSE, .ELIF and .ENDIF directives always refer to the nearest previous .IF directive.

A label is not allowed before this directive.

Example

Suppose you have an assemble source file with specific code for a test version, for a demo version and for the final version. Within the assembly source you define this code conditionally as follows:

```
.IF   TEST
  ...
  ; code for the test version
.ENDIF
```

```
.ELIF DEMO
  ...
  ; code for the demo version
.ENDIF
```

```
.ELSE
  ...
  ; code for the final version
.ENDIF
```
Before assembling the file you can set the values of the symbols `.TEST` and `.DEMO` in the assembly source before the `.IF` directive is reached. For example, to assemble the demo version:

```
TEST .SET 0
DEMO .SET 1
```

You can also define the symbols on the command line with the option `-D`:

```
astc -DDIMO -DTEST=0 test.src
```
.INCLUDE

Syntax

.INCLUDE 'filename' | <filename>

Description

With the .INCLUDE directive you include another file at the exact location in the source where the .INCLUDE occurs. The .INCLUDE directive works similarly to the #include statement in C. The source from the include file is assembled as if it followed the point of the .INCLUDE directive. When the end of the included file is reached, assembly of the original file continues.

The filename specifies the filename of the file to be included. The filename must be compatible with the operating system (forward/backward slashes) and can include a directory specification.

If an absolute pathname is specified, the assembler searches for that file. If a relative path is specified or just a filename, the order in which the assembler searches for include files is:

1. The current directory if you used the 'filename' construction.
   The current directory is not searched if you use the <filename> syntax.

2. The path that is specified with the assembler option -I.

3. The path that is specified in the environment variable ASTCINC when the product was installed.

4. The include directory relative to the installation directory.

A label is not allowed before this directive.

Example

.INCLUDE 'storage\mem.asm' ; include file
.INCLUDE <data.asm> ; Do not look in
; current directory

Related information

Assembler option -I (Add directory to include file search path) in section 5.2, Assembler Options, of Chapter Tool Options.
.LOCAL

Syntax

.LOCAL  symbol[symbol]...

Description

All symbols or labels defined in the current section or module are local to the module by default. You can change this default behavior with assembler option `-ig`.

With the .LOCAL directive (Local Section Symbol Declaration) you declare one of more symbols as local. This means that the specified symbols are explicitly local to the module in which you define them.

If the symbols that appear in the operand field are not used in the module, the assembler gives a warning.

A label is not allowed before this directive.

Example

```assembly
.SDEC  
."data.io",DATA
.SECT  
."data.io"
.LOCAL  LOOPA  ; LOOPA is local to this section

LOOPA .HALF  0x100  ; assigns the value 0x100 to LOOPA
```

Related information

- .EXTERN (External symbol declaration)
- .GLOBAL (Global symbol declaration)
.MACRO / .ENDM

Syntax

\textit{macro\_name} \texttt{.MACRO \{dumarg\,[dumarg]...\}}
\texttt{.macr\_definition\_statements}
\texttt{.}
\texttt{.ENDM}

Description

With the \texttt{.MACRO} directive you define a macro. Macros provide a shorthand method for handling a repeated pattern of code or group of instructions. You can define the pattern as a macro, and then call the macro at the points in the program where the pattern would repeat. The \texttt{.ENDM} directive indicates the end of the macro.

The definition of a macro consists of three parts:

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

The arguments are symbolic names that the macro preprocessor replaces with the literal arguments when the macro is expanded (called). Each formal \textit{argument} must follow the same rules as symbol names: the name can consist of letters, digits and underscore characters (_). The first character cannot be a digit. Argument names cannot start with a percent sign (\%).

Macro definitions can be nested but the nested macro will not be defined until the primary macro is expanded.

You can use the following operators in macro definition statements:
<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\</td>
<td>Macro argument concatenation</td>
<td>Concatenates a macro argument with adjacent alphanumeric characters.</td>
</tr>
<tr>
<td>?</td>
<td>Return decimal value of symbol</td>
<td>Substitutes the ?sequence with a character string that represents the decimal value of the symbol.</td>
</tr>
<tr>
<td>%</td>
<td>Return hex value of symbol</td>
<td>Substitutes the %sequence with a character string that represents the hexadecimal value of the symbol.</td>
</tr>
<tr>
<td>&quot;</td>
<td>Macro string delimiter</td>
<td>Allows the use of macro arguments as literal strings.</td>
</tr>
<tr>
<td>^</td>
<td>Macro local label override</td>
<td>Causes local labels in its term to be evaluated at normal scope rather than at macro scope.</td>
</tr>
</tbody>
</table>

**Example**

The macro definition:

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

The macro call:

```
.SDECL "".text",code
.SECT "".text"
CONSTD d4,0x12345678
```

The macro expands as follows:

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

**Related information**

- **DUP** (Duplicate Sequence of Source Lines),
- **DUPA** (Duplicate Sequence with Arguments),
- **DUPC** (Duplicate Sequence with Characters),
- **DUPF** (Duplicate Sequence in Loop)

.MESSAGE

Syntax

.MESEGGE  [\textit{string} | \textit{exp}]\ldots

Description

With the .MESSAGE directive (Programmer Generated Message) you tell the assembler to output an information message during assembly.

The error and warning counts will not be affected. The .MESSAGE directive is for example useful in combination with conditional assembly for informational purposes. The assembly proceeds normally after the message has been printed.

Optionally, you can specify an arbitrary number of strings and expressions, in any order but separated by commas, to describe the nature of the message. If you use expressions, the assembler outputs the result. The assembler outputs a space between each argument.

This directive has no effect on the exit code of the assembler.

A label is not allowed before this directive.

Example

\texttt{.DEFIN\texttt{E LONG "SHORT"}}
\texttt{.MESSAGE 'This is a LONG string'\newline \texttt{.MESSAGE "This is a LONG string"}}

Within single quotes, the defined symbol LONG is not expanded. Within double quotes the symbol LONG is expanded. So, the actual message is printed as:

\begin{verbatim}
This is a LONG string
This is a SHORT string
\end{verbatim}

Related information

- \texttt{.FAIL} (Programmer Generated Error)
- \texttt{.WARNING} (Programmer Generated Warning)
\textbf{.ORG}

Syntax

\texttt{.ORG [abs-loc][,sect_type][,attribute]...}

Description

With the \texttt{.ORG} directive you can specify an absolute location (\textit{abs-loc}) in memory of a section. This is the same as a \texttt{.SDECL/.SECT} without a section name.

This directive uses the following arguments:

\texttt{abs-loc} \hspace{1em} Initial value to assign to the run-time location counter. \textit{abs-loc} must be an absolute expression. If \textit{abs-loc} is not specified, then the value is zero.

\texttt{sect_type} \hspace{1em} An optional section type:
\begin{itemize}
  \item \texttt{code} \hspace{1em} code section
  \item \texttt{data} \hspace{1em} data section
\end{itemize}

\texttt{attribute} \hspace{1em} An optional section attribute:

\begin{itemize}
  \item \texttt{init} \hspace{1em} section is copied from ROM to RAM at startup
  \item \texttt{noread} \hspace{1em} section can be executed from but not read
  \item \texttt{noclear} \hspace{1em} section is not cleared during startup
  \item \texttt{max} \hspace{1em} data overlay with other parts with the same name, is implicit a type of 'noclear'
  \item \texttt{rom} \hspace{1em} data section remains in ROM
\end{itemize}

A label is not allowed before this directive.

Example

\begin{verbatim}
; define a section on location 100 decimal
.org 100

; define a relocatable nameless section
.org

; define a relocatable data section
.org ,data
\end{verbatim}
; define a data section on 0x8000
.org 0x8000, data

Related information

.SDECL (Declare section name and attributes)
.SECT  (Activate a declared section)
.PMACRO

Syntax

`PMACRO symbol[,symbol]...`

Description

With the `.PMACRO` directive (Purge Macro) you tell the assembler to undefine the specified macro, so that later uses of the symbol will not be expanded.

A label is not allowed before this directive.

Example

```
.PMACRO MAC1,MAC2
```

This statement causes the macros named `MAC1` and `MAC2` to be undefined.

Related information

`.MACRO` (Define Macro)
.SDECL

Syntax

.SDECL "name", type [, attribute ]... [AT address]

Description

With the .SDECL directive you can define a section with a name, type and optional attributes. Before any code or data can be placed in a section, you must use the .SECT directive to activate the section.

This directive uses the following arguments:

type: A section type:

- code: code section
- data: data section

attribute: An optional section attribute:

Code attributes:

- init: section is copied from ROM to RAM at startup
- noread: section can be executed from but not read

Data attributes:

- noclear: section is not cleared during startup
- max: data overlay with other parts with the same name, is implicit a type of 'noclear'
- rom: data section remains in ROM

Sections with attribute noclear are not zeroed at startup. This is a default attribute for data sections. You can only use this attribute with a data type section. This attribute is only useful with BSS sections, which are cleared at startup by default.

The attribute init defines that the code section contains initialization data, which is copied from ROM to RAM at program startup.

Sections with the attribute rom 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 attribute max, the linker generates a section with a size that is the largest of the sizes in the individual object modules. The attribute max only applies to data sections.
The name of a section can have a special meaning for locating sections. The name of code sections should always start with ".text" (or ".pcptext" for PCP code). With data sections, the prefix in the name is important. The prefix determines if the section is initialized, constant or uninitialized and which addressing mode is used.

<table>
<thead>
<tr>
<th>Name prefix</th>
<th>Type of DATA section</th>
</tr>
</thead>
<tbody>
<tr>
<td>.data</td>
<td>initialized</td>
</tr>
<tr>
<td>.zdata</td>
<td>initialized, abs 18 addressing</td>
</tr>
<tr>
<td>.sdata</td>
<td>initialized, a0 addressing</td>
</tr>
<tr>
<td>.data_a8</td>
<td>initialized, a8 addressing</td>
</tr>
<tr>
<td>.data_a9</td>
<td>initialized, a9 addressing</td>
</tr>
<tr>
<td>.rodata</td>
<td>constant data</td>
</tr>
<tr>
<td>.zrodata</td>
<td>constant data, abs 18 addressing</td>
</tr>
<tr>
<td>.srodata</td>
<td>constant data, a0 addressing</td>
</tr>
<tr>
<td>.rodata_a8</td>
<td>constant data, a8 addressing</td>
</tr>
<tr>
<td>.rodata_a9</td>
<td>constant data, a9 addressing</td>
</tr>
<tr>
<td>.bss</td>
<td>uninitialized</td>
</tr>
<tr>
<td>.zbss</td>
<td>uninitialized, abs 18 addressing</td>
</tr>
<tr>
<td>.sbss</td>
<td>uninitialized, a0 addressing</td>
</tr>
<tr>
<td>.bss_a8</td>
<td>uninitialized, a8 addressing</td>
</tr>
<tr>
<td>.bss_a9</td>
<td>uninitialized, a9 addressing</td>
</tr>
<tr>
<td>.ldata</td>
<td>a1 addressing (read only constants, literal data)</td>
</tr>
<tr>
<td>.pcpdata</td>
<td>pcp data</td>
</tr>
</tbody>
</table>

Table 3-1: Data section name prefixes

Note that the compiler uses the following name convention:

`prefix.module-name.function-or-object-name`

Examples:

`.sdecl "text.t.main", CODE ; declare code section
.sect "text.t.main" ; activate section

`.sdecl "data.t.var1", DATA ; declare data section
.sect "data.t.var1" ; activate section
.sdecl "text.intvec.00a", CDOE ; declare interrupt
          ; vector table entry for interrupt 10
.sect "text.intvec.00a" ; activate section

.SECT (Activate a declared section)
.ORG (Initialize a nameless section)
.SECT

Syntax

```
.SECT "name" [ , RESET]
```

Description:

With the .SECT directive you activate a previously declared section with the name *name*. Before you can activate a section, you must define the section with the .SDECL directive. You can activate a section as many times as you need.

With the section attribute **RESET** you can reset counting storage allocation in **data** sections that have section attribute **max**.

Examples:

```
.sdecl "zdata.t.var2", DATA ; declare data section
.sect "zdata.t.var2" ; activate section
```

**.SDECL** (Declare a section with name, type and attributes)

**.ORG** (Initialize a nameless section)
.SET

Syntax

\[ \text{symbol .SET expression} \]
\[ .SET \text{ symbol expression} \]

Description

With the .SET directive you assign the value of \textit{expression} to \textit{symbol} temporarily. If a symbol was defined with the .SET directive, you can redefine that symbol in another part of the assembly source, using another .SET directive.

The .SET directive is useful in establishing temporary or reusable counters within macros. \textit{Expression} must be absolute and forward references are allowed.

Symbols that are set with the .EQU directive, cannot be redefined.

Example

\[ \text{COUNT .SET 0 ; Initialize COUNT. Later on you can} \]
\[ \quad \text{; assign other values to the symbol COUNT.} \]

Related information

\[ .EQU \quad \text{(Assign permanent value to a symbol)} \]
.SIZE

Syntax

```
.SIZE  symbol, expression
```

Description

With the .SIZE directive you set the size of the specified symbol to the value represented by expression.

The .SIZE directive may occur anywhere in the source file unless the specified symbol is a function. In this case, the .SIZE directive must occur after the function has been defined.

Example

```
main: .type  func
    .      ; function main
.
    ret16
main_function_end:
    .size  main,main_function_end-main
```

Related information

(TYPE) (Set Symbol Type)
.SPACE

Syntax

[<label>] .SPACE <expression>

Description

With the .SPACE directive (Define Storage) the assembler reserves a block of memory. The reserved block of memory is not initialized to any value.

With expression you specify the number of MAUs (Minimum Addressable Units) you want to reserve, and how much the location counter will advance. 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). For the TriCore assembler astc, the MAU size is 1 byte. For the PCP assembler aspcp, the MAU size is 2 bytes for pcp code sections and 4 bytes for pcp data sections.

If you specify label, it gets the value of the location counter at the start of the directive processing.

Example

To reserve 12 bytes (not initialized) of memory in a TriCore data section:

```
sdecl "zbss.tst.uninit",DATA
.sect "zbss.tst.uninit"
uninit .SPACE 12 ; Sample buffer
```

Related information

- .ASCII / .ASCIIZ (Define ASCII string without/with ending NULL)
- .BYTE (Define a constant byte)
- .FLOAT / .DOUBLE (Define a 32-bit / 64-bit floating-point constant)
- .WORD / .HALF (Define a word / halfword)
.TYPE

Syntax

symbol .TYPE typeid

Description

With the .TYPE directive you set a symbol’s type to the specified value in the ELF symbol table. Valid symbol types are:

FUNC The symbol is associated with a function or other executable code.

OBJECT The symbol is associated with an object such as a variable, an array, or a structure.

FILE The symbol name represents the filename of the compilation unit.

Labels in code sections have the default type FUNC. Labels in data sections have the default type OBJECT.

Example

Afunc .TYPE FUNC

Related information

.SIZE (Set Symbol Size)
.UNDEF

Syntax

```
.UNDEF symbol
```

Description

With the .UNDEF directive you can undefine a substitution string that was previously defined with the .DEFINE directive. The substitution string associated with symbol is released, and symbol will no longer represent a valid .DEFINE substitution.

A label is not allowed before this directive.

Example

```
.UNDEF LEN    ; Undefines the LEN substitution string
               ; that was previously defined with the
               ; .DEFINE directive
```

Related information

.DEFINE (Define Substitution String)
.WARNING

**Syntax**

```
.WARNINg [\(string \mid exp\)][\(string \mid exp\)]...
```

**Description**

With the `.WARNING` directive (Programmer Generated Warning) you tell the assembler to output a warning message during the assembling process.

The total warning count will be incremented as with any other warning. The `.WARNING` directive is for example useful in combination with conditional assembly for exceptional condition checking. The assembly process proceeds normally after the warning has been printed.

Optionally, you can specify an arbitrary number of strings and expressions, in any order but separated by commas, to describe the nature of the generated warning. If you use expressions, the assembler outputs the result. The assembler outputs a space between each argument.

This directive has no effect on the exit code of the assembler, unless you use the assembler option `--warnings-as-errors`. In that case the assembler exits with exit code 1 (an error).

A label is not allowed before this directive.

**Example**

```
.WARNINg ‘parameter too large’
```

This results in the warning:

```
W144: ["filename" line] Parameter out of range
```

**Related information**

- `.FAIL` (Programmer Generated Error),
- `.MESSAGE` (Programmer Generated Message)
**.WEAK**

**Syntax**

`.WEAK symbol[,symbol]...`

**Description**

With the `.WEAK` directive you mark one of more symbols as ‘weak’. The symbol can be defined in the same module with the `.GLOBAL` directive or the `.EXTERN` directive. If the symbol does not already exist, it will be created.

A ‘weak’ external reference is resolved by the linker when a global (or weak) definition is found in one of the object files. However, a weak reference will not cause the extraction of a module from a library to resolve the reference.

You can overrule a weak definition with a `.GLOBAL` definition in another module. The linker will not complain about the duplicate definition, and ignore the weak definition.

Only program labels and symbols defined with `EQU` can be made weak.

**Example**

```
LOOPA .EQU 1 ; definition of symbol LOOPA
GLOBAL LOOPA ; LOOPA will be globally
    ; accessible by other modules
.WEAK LOOPA ; mark LOOPA as weak
```

**Related information**
.WORD/.HALF

Syntax

[label] .WORD argument[,argument]...

[label] .HALF argument[,argument]...

Description

With the .WORD or .HALF directive the assembler allocates and initializes one word (32 bits) or a halfword (16 bits) of memory for each argument.

An argument can be:

- a single or multiple character string constant
- an expression
- NULL (indicated by two adjacent commas: ,,)

Multiple arguments are stored in sets of four or two bytes. If an argument is NULL its corresponding address locations are filled with zeros.

If you specify label, it gets the value of the location counter at the start of the directive processing.

In case of single and multiple character strings, each character is stored in consecutive bytes whose lower seven bits represent the ASCII value of the character. The standard C escape sequences are allowed:

```
.WORD 'R' ; = 0x52
.WORD 'ABCD' ; = 0x41424344

.HALF 'R' ; = 0x52
.HALF 'AB' ; = 0x4142
.HALF 'ABCD' ; = 0x4142
                0x4344
```

If the evaluated argument is too large to be represented in a word / halfword, the assembler issues an error and truncates the value.

Examples

WRD: .WORD 14,1635,0x34266243,'ABCD'

HLF: .HALF 14,1635,0x2662,'AB'
With the `.BYTE` directive you can obtain exactly the same effect:

```
WRD:    .BYTE  14,0,0,0,1635%256,6,0,0,
        0x43,0x62,0x26,0x34,’D’,’C’,’B’,’A’

HLF:    .BYTE  14,0,1635%256,6,0x62,0x26,’B’,’A’
```

**Related information**

- `.SPACE` (Define storage)
- `.ASCII` / `.ASCIIZ` (Define ASCII string without/with ending NULL)
- `.BYTE` (Define a constant byte)
### 3.3.3 Overview of Assembler Controls

The following tables provide an overview of all assembler controls.

**Overview of assembler listing controls**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$LIST ON / OFF</td>
<td>Generation of assembly list file temporary ON/OFF</td>
</tr>
<tr>
<td>$LIST &quot;flags&quot;</td>
<td>Exclude / include lines in assembly list file</td>
</tr>
<tr>
<td>$PAGE</td>
<td>Generate formfeed in assembly list file</td>
</tr>
<tr>
<td>$PAGE settings</td>
<td>Define page layout for assembly list file</td>
</tr>
<tr>
<td>$PRCTL</td>
<td>Send control string to printer</td>
</tr>
<tr>
<td>$STITLE</td>
<td>Set program subtitle in header of assembly list file</td>
</tr>
<tr>
<td>$TITLE</td>
<td>Set program title in header of assembly list file</td>
</tr>
</tbody>
</table>

**Overview of miscellaneous assembler controls**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$CASE ON / OFF</td>
<td>Case sensitive user names ON/OFF</td>
</tr>
<tr>
<td>$DEBUG ON / OFF</td>
<td>Generation of symbolic debug ON/OFF</td>
</tr>
<tr>
<td>$DEBUG &quot;flags&quot;</td>
<td>Select debug information</td>
</tr>
<tr>
<td>$FPU</td>
<td>Allow single precision floating-point instructions</td>
</tr>
<tr>
<td>$HW_ONLY</td>
<td>Prevent substitution of assembly instructions by smaller or faster instructions</td>
</tr>
<tr>
<td>$IDENT LOCAL / GLOBAL</td>
<td>Assembler treats labels by default as local or global</td>
</tr>
<tr>
<td>$MMU</td>
<td>Allow memory management instructions</td>
</tr>
<tr>
<td>$OBJECT</td>
<td>Alternative name for the generated object file</td>
</tr>
<tr>
<td>$TCdefect ON / OFF</td>
<td>Enable/disable assembler check for specified functional problem</td>
</tr>
<tr>
<td>$TC2</td>
<td>Allow TriCore 2 instructions</td>
</tr>
<tr>
<td>$WARNING OFF [num]</td>
<td>Suppress all or some warnings</td>
</tr>
</tbody>
</table>
3.3.4 DETAILED DESCRIPTION OF ASSEMBLER CONTROLS

The assembler recognizes both upper and lower case for controls.
$CASE ON / OFF

Syntax

$CASE ON  (default)
$CASE OFF

Description

With the $CASE ON and $CASE OFF controls you specify whether the assembler operates in case sensitive mode or not. By default the assembler operates in case sensitive mode. This means that all user-defined symbols and labels are treated case sensitive, so LAB and Lab are distinct. Note that instruction mnemonics, register names, directives and controls are always treated case insensitive.

Example

; begin of source
$CASE OFF    ; assembler in case insensitive mode

Related option

Assembler option –c (Switch to case insensitive mode) in section 5.2, Assembler Options, of Chapter Tool Options.

Related information
$DEBUG ON / OFF

Syntax

$DEBUG ON
$DEBUG OFF
$DEBUG "flags"

Description

With the $DEBUG ON and $DEBUG OFF controls you turn the generation of debug information on or off. ($DEBUG ON is similar to the assembler option -g).

If you use $DEBUG control with flags, you can set the following flags:

- a/A assembler source line information
- h/H pass HLL debug information

You cannot use these two types of debug information both. So, $DEBUG "ah" is not allowed.

- l/L local symbols debug information
- s/S always debug; either "AhL" or "aHl"

Debug information that is generated by the C compiler, is always passed to the object file.

Example

;begin of source
$DEBUG ON ; generate local symbols debug information

Related option

Assembler option -g (Select debug information) in section 5.2, Assembler Options, of Chapter Tool Options.

Related information

-
$FPU

Syntax

$FPU

Description

With the $FPU control you instruct the assembler to accept and encode single precision floating-point instructions in the assembly source file.

When you use this control, the define __FPU__ is set to 1. By default the define __FPU__ is set to 0 which tells the assembler not to accept single precision floating-point instructions.

Example

; begin of source
$FPU ; the use of single precision FPU instructions
 ; in this source is allowed.

Related option

Assembler option --fpu-present (Allow the use of single precision floating-point instructions) in section 5.2, Assembler Options, of Chapter Tool Options.

Related information
$HW\_ONLY

Syntax

$HW\_ONLY

Description

Normally the assembler replaces 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.

Example

```
;begin of source
$HW\_ONLY  ; the assembler does not substitute
            ; instructions with other, smaller or
            ; faster instructions.
```

Related option

Assembler option `–Og` (Allow generic instructions) in section 5.2, `Assembler Options`, of Chapter `Tool Options`.

Related information

-
$IDENT

Syntax

$IDENT LOCAL
$IDENT GLOBAL

Description

With the controls $IDENT LOCAL and $IDENT GLOBAL you tell the assembler how to treat symbols that you have not specified explicitly as local or global with the assembler directives .LOCAL or .GLOBAL.

By default the assembler treats all symbols as local symbols unless you have defined them explicitly as global.

Example

; begin of source
$IDENT GLOBAL ; assembly labels are global by default

Related option

Assembler option -i (Treat labels by default local / global) in section 5.2, Assembler Options, of Chapter Tool Options.

Related information

Assembler directive .LOCAL (Local symbol declaration)
Assembler directive .GLOBAL (Global symbol declaration)
$LIST ON / OFF

Syntax

$LIST ON
.
  ; assembly source lines
.
$LIST OFF

Description

If you generate a list file with the assembler option -l, you can use the $LIST ON and $LIST OFF controls to specify which source lines the assembler must write to the list file. Without the command line option -l, the $LIST ON and $LIST OFF controls have no effect.

The $LIST ON control actually increments a counter that is checked for a positive value and is symmetrical with respect to the $LIST OFF control. Note the following sequence:

; Counter value currently 1
$LIST ON ; Counter value = 2
$LIST ON ; Counter value = 3
$LIST OFF ; Counter value = 2
$LIST OFF ; Counter value = 1

The listing still would not be disabled until another $LIST OFF control was issued.

Example

Suppose you assemble the following assembly source with the assembler option -l:

.SDECL ".text", CODE
.SECT ".text"
  ... ; source line in list file
$LIST OFF
  ... ; source line not in list file
$LIST ON
  ... ; source line also in list file
.END
The assembler generates a list file with the following lines:

```
.SDECL "..text",CODE
.SECT "..text"
... ; source line in list file
$LIST ON
... ; source line also in list file
.END
```

**Related option**

Assembler option `-l` (Generate list file) in section 5.2, *Assembler Options*, of Chapter *Tool Options*.

**Related information**

Assembler control `$LIST` (Exclude / include lines in assembly list file)

Assembler function `@LST()` in section 3.2, *Built-in Assembly Functions*. 
$LIST flags

Syntax

Begin of assembly file

$LIST "flags"

Description

If you generate a list file with the assembler option -l, you can use the $LIST controls to specify which type of source lines the assembler must exclude from the list file. Without the command line option -l, the $LIST control has no effect.

You can set the following flags to remove or include lines:

- c/C Lines with assembler controls
- d/D Lines with section directives (.SECT and .SDECL)
- e/E Lines with symbol definition directives (.EXTERN, .GLOBAL, .LOCAL, .CALLS)
- g/G Lines with generic instruction expansion
- i/I Lines with generic instructions
- l/L #Line source lines
- m/M Lines with macro definitions (.MACRO and .DUP)
- n/N Empty source lines
- p/P Lines with conditional assembly
- q/Q Lines with the .EQU or .SET directive
- r/R Lines with relocation characters (’r’)
- v/V Lines with .EQU or .SET values
- w/W Wrapped part of a line
- x/X Lines with expanded macros
- y/Y Lines with cycle counts

If you do not specify this control or the assembler option -Iflag, the assembler uses the default: -LcDEGilMnPqrVWXy.

Example

To exclude assembly files with controls from the list file:

;begin of source
$LIST "c"
Related option

Assembler option `-L` (List file formatting options) in section 5.2, *Assembler Options*, of Chapter *Tool Options*.

Related information

Assembler control `$LIST ON / OFF` (Assembly list file ON / OFF)

Assembler function `@LST()` in section 3.2, *Built-in Assembly Functions*. 
$MMU

Syntax

$MMU

Description

With the $MMU control you instruct the assembler to accept and encode memory management instructions in the assembly source file.

When you use this control, the define ___MMU___ is set to 1.

Example

;begin of source
$MMU ; the use of memory management instructions
; in this source is allowed.

Related option

Assembler option --mmu-present (Allow the use of memory management instructions) in section 5.2, Assembler Options, of Chapter Tool Options.

Related information

—
$OBJECT

Syntax

$OBJECT "file"
$OBJECT OFF

Description

With the $OBJECT control you can specify an alternative name for the generated object file. With the $OBJECT OFF control, the assembler does not generate an object file at all.

Example

;Begin of source
$object "x1.o" ; generate object file x1.o

Related option

Assembler option -o (Define output filename) in section 5.2, Assembler Options, of Chapter Tool Options.

Related information
$PAGE

Syntax

$PAGE [width,length,blanktop,blankbtm,blankleft]

Description

If you generate a list file with the assembler option -l, you can use the $PAGE control to format the generated list file.

width Number of characters on a line (1–255). Default is 132.

length Number of lines per page (10–255). Default is 66. As a special case a page length of 0 (zero) turns off all headers, titles, subtitles, and page breaks.

blanktop Number of blank lines at the top of the page. Default = 0. Specify a value so that blanktop + blankbtm ≤ length − 10.

blankbtm Number of blank lines at the bottom of the page. Default = 0. Specify a value so that blanktop + blankbtm ≤ length − 10.

blankleft Number of blank columns at the left of the page. Default = 0. Specify a value smaller than width.

If you use the $PAGE control without arguments, it causes a 'formfeed': the next source line is printed on the next page in the list file. The $PAGE control itself is not printed.

You can omit an argument by using two adjacent commas. If the remaining arguments after an argument are all empty, you can omit them.

Example

$PAGE ; formfeed, the next source line is printed
       ; on the next page in the list file.

$PAGE 96 ; set page width to 96. Note that you can
         ; omit the last four arguments.

$PAGE ,,3,3; use 3 line top/bottom margins.

Related option

-
Related information

Assembler control \texttt{$STITLE$} (Set program subtitle in header of list file)
Assembler control \texttt{$TITLE$} (Set program title in header of list file)

Assembler option \texttt{-l} (Generate list file) in Section 5.2, \textit{Assembler Options}, of Chapter \textit{Tool Options}.

Assembler option \texttt{-L} (List file formatting options) in Section 5.2, \textit{Assembler Options}, of Chapter \textit{Tool Options}.
$\text{PRCTRL}$

Syntax

\[\text{PRCTRL} \ exp | \text{string}|exp | \text{string}]...\]

Description

If you generate a list file with the assembler option `-l`, you can use the $\text{PRCTRL}$ control to send control strings to the printer.

The $\text{PRCTRL}$ control simply concatenates its arguments and sends them to the listing file (the control line itself is not printed unless there is an error).

You can specify the following arguments:

- **exp**
  - A byte expression which may be used to encode non-printing control characters, such as ESC.

- **string**
  - An assembler string, which may be of arbitrary length, up to the maximum assembler-defined limits.

The $\text{PRCTRL}$ control can appear anywhere in the source file; the assembler sends out the control string at the corresponding place in the listing file.

If a $\text{PRCTRL}$ control 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. In this manner, you can use a $\text{PRCTRL}$ control to restore a printer to a previous mode after printing is done.

Similarly, if the $\text{PRCTRL}$ control appears as the first line in the first input file, the assembler sends out the control string before page headings or titles.

Example

\[\text{PRCTRL} \ \$1B,'E' \ ; \text{Reset HP LaserJet printer}\]

Related option

- `-`

Related information

Assembler option `-l` (Generate list file) in Section 5.2, *Assembler Options*, of Chapter *Tool Options*.
**$STITLE**

**Syntax**

\[ $STITLE \ "title" \]

**Description**

If you generate a list file with the assembler option \(-l\), you can use the $STITLE control to specify the program subtitle which is printed at the top of all succeeding pages in the assembler list file below the title.

The specified subtitle is valid until the assembler encounters a new $STITLE control. By default, the subtitle is empty.

The $STITLE control itself will not be printed in the source listing.

If the page width is too small for the title to fit in the header, it will be truncated.

**Example**

\[
\begin{align*}
$TITLE & \ 'This is the title' \\
$STITLE & \ 'This is the subtitle'
\end{align*}
\]

The header of the second page in the list file will now be:

```
TASKING TriCore Assembler vx.yrz Build nnn SN 00000000
This is the title
This is the subtitle
```

**Related option**

\(-l\)

**Related information**

Assembler control $TITLE (Set program title in header of list file)

Assembler option \(-l\) (Generate list file) in Section 5.2, *Assembler Options*, of Chapter *Tool Options*. 
## $TC

### Syntax

```
$TC\text{defect \hspace{1em} ON}
$TC\text{defect \hspace{1em} 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 `$TC112\_DEFECTS` (respectively `$TC113\_DEFECTS`).

### Example

```
$TC112\_COR1 ON ; enable assembler check for CPU functional problem TC112\_COR1
```

### Related option

Assembler option `--silicon-bug` (Check on CPU functional defect) in section 5.2, Assembler Options, of Chapter Tool Options.

### Related information

See Chapter 9, CPU Functional Problems, for more information about the individual problems.
$TC2

Syntax

$TC2

Description

With the $TC2 control you instruct the assembler to accept and encode TriCore 2 instructions in the assembly source file.

When you use this control, the define _TC2_ is set to 1.

Example

;begin of source
$TC2 ; the use of TriCore 2 instructions
; in this source is allowed.

Related option

Assembler option --is-tricore2 (Allow the use of TriCore 2 instructions) in section 5.2, Assembler Options, of Chapter Tool Options.

Related information
$TITLE

Syntax

$TITLE "title"

Description

If you generate a list file with the assembler option -l, you can use the $TITLE control to specify the program title which is printed at the top of each page in the assembler list file.

By default, the title is empty.

If the page width is too small for the title to fit in the header, it will be truncated.

Example

$TITLE 'This is the title'

The header of the list file will now be:

TASKING TriCore Assembler vx.yrz Build nnn SN 00000000
This is the title

Related option

Related information

$TITLE (Set program subtitle in header of assembly list file)
$\text{WARNING OFF}$

Syntax

$\text{WARNING OFF}$
$\text{WARNING OFF number}$

Description

With the $\text{WARNING OFF}$ control you can suppresses all warning messages or specific warning messages.

- By default, all warnings are reported.
- If you specify this option but without numbers, all warnings are suppressed.
- If you specify this option with a number, only the specified warning is suppressed.

Example

$\text{WARNING OFF} \quad ; \text{all warning messages are suppressed}$

$\text{WARNING OFF 135} \quad ; \text{supress warning message 135}$

Related option

Assembler option $-w$ (Suppress some or all warnings) in section 5.2, $\textit{Assembler Options}$, of Chapter $\textit{Tool Options}$.

Related information
CHAPTER 4

RUN-TIME ENVIRONMENT

TASKING
4.1 INTRODUCTION

This chapter describes the startup code used by the TASKING TriCore C compiler, the stack layout and the heap; and the floating-point arithmetic.

4.2 STARTUP CODE

You need the run-time startup code to build an executable application. The default startup code consists of the following components:

- **Initialization code.** This code is executed when the program is initiated and before the function `main()` is called.
- **Exit code.** This controls the closedown of the application after the program’s main function terminates.
- **The trap vector table.** This contains default trap vectors.

The startup code is part of the C library `libc.a`, and the source is present in the file `cstart.asm` in the directory `lib\src`.

If the default run-time startup code does not match your configuration, you need to modify the startup code accordingly.

See also section 7.6, *Linking the C Startup Code* in Chapter *Using the Linker* of the *User’s Manual*.

The entry point of the startup code (power–on vector) is label `_START`. This global label should not be removed, since the C compiler refers to it. It is also used as the default start address of the application.

**Initialization code**

The following initialization actions are executed before the application starts:

1. Re-enable and reset the call depth counter and make A0, A1, A8, A9 write–able. It is required for CrossView Pro that these RESET values are restored for each time the startup code is executed.

2. Initialize the user stack pointer. The user stack pointer is loaded into memory by the stack address, located at `_lc_ue_ustack`. This label is defined in the Linker Script File. See section 4.3, *Stack Usage* for detailed information on the stack.
3. Clear Previous Context Pointer Segment Address and Offset Field. It is required for CrossView Pro's stack trace that these RESET values are restored for each time the startup code is executed.

4. Setup the context save area lists. Tables with start/end addresses go in a separate 'csa_areas' section.

5. Initialize registers and bus configuration. In the file cstart.asm the actual location of several special function registers is required. These addresses are specified in the regcpu_name.def SFR system include files. You can include such a file with the assembler option -Ccpu_name. In EDE the appropriate file is included when you have selected a CPU type. If you do not specify an SFR file, the default SFR regtcllib.def file is included.

6. Load Base Address of Trap Vector Table. This address is indicated by the linker label _lc_u_trap_tab as defined in the Linker Script File.

7. Load Base Address of Interrupt Vector Table. This address is indicated by the linker label _lc_u_int_tab as defined in the Linker Script File.

8. Initialize the interrupt stack pointer. The interrupt stack pointer is loaded into memory by the interrupt stack address, located at _lc_ue_istack. This label is defined in the Linker Script File.

9. Initialize and clear C variables.

10. Copy initialized sections from ROM to RAM, using a linker generated table (also known as the 'copy table') and a run-time library function _c_init.

11. Initialize the argc and argv arguments to zero.

12. Call the entry point of your application with a call to function main().

**Exit code**

When the C application 'returns', which is not likely to happen in an embedded environment, the program ends with a DEBUG16 instruction, at the assembly label _exit. When using a debugger, it can be useful to set a breakpoint on this label to indicate that the program has reached the end, or that the library function exit() has been called.
**Trap vector table**

The default startup code makes sure that the trap vectors for exceptions 0 to 7 are filled in. Default trap vectors are resolved from the C library. You can overrule these routines with your own exception routines.

To disable a default trap vector from EDE:

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

   *The Project Options dialog appears.*

2. Expand the **Processor** entry, expand **Startup**, expand **Startup Code** and select **Trap Vectors**.

3. Disable the trap vectors you do not want to be automatically defined in the startup code.

See also section 3.9.2, *Interrupt and Trap Functions* in Chapter TriCore C Language of the User’s Manual.

**Control Startup Code from EDE**

To control `cstart.asm` from within EDE, you first have to add `cstart.asm` to your project:

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

   *The Project Options dialog appears.*

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

3. Enable the option **Automatically copy and link cstart.asm to your project**.

   *The file cstart.asm is copied to your project directory and added to your project.*

Now you can specify all your startup settings in the pages **Startup Code**, **Boot Memory** and **Memory Control**.

You can specify CPU settings in the same dialog:

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

   *The Project Options dialog appears.*

2. Expand the **Processor** entry and select **Bus Configuration**.
3. Select the appropriate bus configuration settings.

*EDE automatically defines macros according to the selected settings.*

**Macro Preprocessor Symbols**

A number of macro preprocessor symbols are used in the startup code. These can be enabled or disabled using the assembler command line option `-D` with the following syntax:

`-D<identifier>[=replacement]`.

In the startup file (*cstart.asm*) the following macro preprocessor symbols are used:

<table>
<thead>
<tr>
<th>Define</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>External Boot Memory Configuration (BOOTCFG)</strong></td>
<td></td>
</tr>
<tr>
<td><code>_BOOTCFG_VALUE</code></td>
<td>Boot memory Offset Address + 0x4</td>
</tr>
<tr>
<td><strong>Memory Control (PMUCON/DMUCON)</strong></td>
<td></td>
</tr>
<tr>
<td><code>_PMU_CON_VALUE</code></td>
<td>If defined, value of PMU Control register</td>
</tr>
<tr>
<td><code>_PMU_EIFCON_VALUE</code></td>
<td>If defined, value of PMU External Instruction Fetch Control register</td>
</tr>
<tr>
<td><code>_DMU_CON_DCAON</code></td>
<td>If defined, Enable data cache</td>
</tr>
<tr>
<td><strong>Startup</strong></td>
<td></td>
</tr>
<tr>
<td><code>_NO_BTV_INIT</code></td>
<td>If defined, Base Address of Trap Vector Table is not initialized with trap table start address (<code>trap_tab</code>).</td>
</tr>
<tr>
<td><code>_NO_BIV_INIT</code></td>
<td>If defined, Base Address of Interrupt Vector Table is not initialized with interrupt table start address (<code>lc_u_int_table</code>).</td>
</tr>
<tr>
<td><code>_NO_ISP_INIT</code></td>
<td>If defined, Interrupt Stack Pointer is not initialized with end address of interrupt stack (<code>lc_ue_istack</code>).</td>
</tr>
<tr>
<td><code>_NO_USP_INIT</code></td>
<td>If defined, User Stack Pointer is not initialized with end address of user stack (<code>lc_ue_istack</code>).</td>
</tr>
<tr>
<td><code>_NO_PCX_RESET</code></td>
<td>If defined, the Previous Context is not explicitly cleared.</td>
</tr>
<tr>
<td><code>_NO_PSW_RESET</code></td>
<td>If defined, the Call Depth Counter is not explicitly cleared.</td>
</tr>
<tr>
<td>Define</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>_NO_A0A1_ADDRESSING</td>
<td>If defined, global address register A0/A1 is not initialized with start address of the _a0/_a1 addressable area (_lc_gb_a0/1).</td>
</tr>
<tr>
<td>_NO_A8A9_ADDRESSING</td>
<td>If defined, global address register A8/A9 is not initialized with start address of the _a8/_a9 addressable area (_lc_gb_a8/9).</td>
</tr>
<tr>
<td>_NO-CSA_INIT</td>
<td>If defined, Context Save Area lists are not initialized.</td>
</tr>
<tr>
<td>_NO_WATCHDOG_INIT</td>
<td>If defined, Watchdog timer disabled.</td>
</tr>
<tr>
<td>_NO_BUS_CONF</td>
<td>If defined, bus configuration registers are not initialized.</td>
</tr>
<tr>
<td>_NO_C_INIT</td>
<td>If defined, C variables are not initialized.</td>
</tr>
<tr>
<td>_NO_ARG_INIT</td>
<td>If defined, disable initialization of argc and argv[].</td>
</tr>
<tr>
<td>_NO_EXIT</td>
<td>If defined, C library function exit() or abort() not supported.</td>
</tr>
<tr>
<td>_USERDEFINED_TRAP_n</td>
<td>If defined, the default trap vector n is disabled.</td>
</tr>
</tbody>
</table>

**Miscellaneous**

<table>
<thead>
<tr>
<th>Define</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_CALL_INIT</td>
<td>Can be set to a function to be called before main. This function cannot have a return or arguments. This function can be used, for example, to initialize the serial port before main is called. This is useful for building programs without making any modifications to the original source.</td>
</tr>
<tr>
<td>_CALL_ENDINIT</td>
<td>Can be set to a function to be called before the ENDINIT instruction is executed. Like the CALLINIT function, it cannot not have a return value or arguments.</td>
</tr>
</tbody>
</table>

**CPU functional bypasses**

<table>
<thead>
<tr>
<th>Define</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_TC112_XXX</td>
<td>If defined, TC112 CPU functional defect XXX is bypassed and/or checked.</td>
</tr>
<tr>
<td>_TC113_XXX</td>
<td>If defined, TC113 CPU functional defect XXX is bypassed and/or checked. See Chapter 9 <em>CPU Functional Problems</em> for a complete list of these macros.</td>
</tr>
</tbody>
</table>

*Table 4-1: Defines used in cstart.src*
The following table shows the linker labels used in the startup code.

<table>
<thead>
<tr>
<th>Define</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_START</td>
<td>start label, mentioned in LSL file (tc_arch.lsl)</td>
</tr>
<tr>
<td>_c_init</td>
<td>label copy table init function</td>
</tr>
<tr>
<td>main</td>
<td>start label user C program</td>
</tr>
<tr>
<td>exit</td>
<td>start label of exit() function</td>
</tr>
<tr>
<td>_exit</td>
<td>exit() function jumps to this place</td>
</tr>
<tr>
<td>_CALL_ENDINIT</td>
<td>label called before ENDINIT</td>
</tr>
<tr>
<td>_CALL_INIT</td>
<td>_CALL_INIT label called before main()</td>
</tr>
<tr>
<td>_lc_gb_a0</td>
<td>linker label start of A0 addressable area</td>
</tr>
<tr>
<td>_lc_gb_a1</td>
<td>linker label start of A1 addressable area</td>
</tr>
<tr>
<td>_lc_gb_a8</td>
<td>linker label start of A8 addressable area</td>
</tr>
<tr>
<td>_lc_gb_a9</td>
<td>linker label start of A9 addressable area</td>
</tr>
<tr>
<td>_lc_u_int_tab</td>
<td>linker label interrupt table</td>
</tr>
<tr>
<td>_lc_ub_csa</td>
<td>linker label context save area begin</td>
</tr>
<tr>
<td>_lc_ue_csa</td>
<td>linker label context save area end</td>
</tr>
<tr>
<td>_lc_ue_istack</td>
<td>linker label interrupt stack end</td>
</tr>
<tr>
<td>_lc_ue_ustack</td>
<td>linker label user stack end</td>
</tr>
</tbody>
</table>

*Table 4-2: Linker labels used in startup code*
4.3 STACK USAGE

The stack is used for local automatic variables and function parameters. The following diagram shows the structure of a stack frame.

![Stack Diagram]

Figure 4-1: Stack diagram

The stack size is defined in the linker script file (tc_arch.lsl in directory include.lsl) with the macro USTACK and ISTACK, which results in sections called ustack and istack.

The linker defined label lc_ue_ustack refers to the top of the user stack area and is used in the file cstart.asm to initialize the user stack pointer register (SP). The linker defined label lc_ue_istack refers to the top of the interrupt stack area and is used in the file cstart.asm to initialize the interrupt stack pointer register (ISP).

As long as the user program does not change the IS bit in the program status word (PSW), only the user stack is used. Refer to the TriCore Architecture (v1.3) Manual for the implications of an IS bit change.
4.4 HEAP ALLOCATION

The heap is only needed when you use one or more of the dynamic memory management library functions: `malloc()`, `calloc()`, `free()` and `realloc()`. The heap is a reserved area in memory. Only if you use one of the memory allocation functions listed above, the linker automatically allocates a heap, as specified in the linker script file with the keyword `heap`.

A special section called `heap` is used for the allocation of the heap area. The size of the heap is defined in the linker script file (`tc_arch.lsl` in directory `include.lsl`) with the macro `HEAP`, which results in a section called `heap`. The linker defined labels `_lc_bh` and `_lc_ah` (begin and end of heap) are used by the library function `sbrk()`, which is called by `malloc()` when memory is needed from the heap.

The special `heap` section is only allocated when its linker labels are used in the program.

4.5 FLOATING-POINT ARITHMETIC

Floating-point arithmetic support for the compiler `ctc` is included in the software as a separate set of libraries or in the hardware when available (only single precision). During linking you have to specify the desired floating-point library after the C library. The libraries are reentrant, and only use temporary program stack memory.

To ensure portability of floating-point arithmetic, floating-point arithmetic for the compiler `ctc` has been implemented complying with the IEEE-754 standard for floating-point arithmetic. See the *IEEE Standard Binary for Floating-Point Arithmetic* document [IEEE Computer Society, 1985] for more details on the floating-point arithmetic definitions. This document is referred to as IEEE-754 in this manual.

The compiler `ctc` supports both single and double precision floating-point operations using the ISO C types `float` and `double` respectively. To optimize for speed, also a non-trapping library is included.

It is possible to intercept floating-point exceptional cases and, if desired, handle them with an application defined exception handler. The intercepting of floating-point exceptions is referred to as 'trapping'. Examples of how to install a trap handler are included.
4.5.1 COMPLIANCE WITH IEEE-754

The level to which the floating-point implementation complies with the IEEE-754 standard, depends on the chosen configuration.

All floating-point calculations are executed using the ‘round to nearest (even)’ rounding mode, since this is required by ANSI-C 89. This is conform IEEE-754. Because there are no double precision floating-point hardware instructions, an emulating library is always needed for double precision calculation.

When the use of hardware FPU instructions is chosen (--fpu-present), the available hardware instructions for single precision floating-point will be used either in the compiler or in one of the libraries. For double precision floating-point calculations the chosen floating-point emulaton library will be used. When no hardware FPU instructions are allowed, all floating-point operations will be used from the chosen floating-point emulaton library.

In EDE you can specify to use the single precision floating-point:

1. From the Project menu, select Project Options...
   
   The Project Options dialog appears.

2. Expand the C Compiler entry and select Miscellaneous.

3. Enable the option Single precision floating-point only: treat type ‘double’ as ‘float’.

Compiler option --fpu-present in Chapter 5, Tool Options.

Compliance with IEEE-754: TriCore hardware FPU instructions

The following implementation issues for the single precision hardware instructions (optionally implemented on the TriCore chip), are important:

- subnormals are not supported (hardware design decision).
- when converting single precision floats to integers, rounding is done to the nearest (even) integer. This does not comply with ANSI-C 89 or ISO-C 99, but does comply with IEEE-754, since this is the current rounding mode (hardware design decision).
- when a converted single precision float overflows the target integer type, the value is saturated to MAX_INT or MIN_INT (hardware design decision).
- whenever a double precision float is involved, the results are determined by the chosen emulation library.

**Compliance with IEEE-754: Trapping emulation library**

The following implementation issues for the trapping floating-point library are important:

- subnormals are not supported. This is conform the TriCore hardware design.
- when converting floats to integers, the value is truncated. This complies with ANSI-C 89 and ISO-C 99, but does not comply with IEEE-754, since the current rounding mode is 'round to nearest (even)'.
- when a converted float overflows the target integer type, a predictable value is assigned to the target integer.

**Compliance with IEEE-754: Hand-optimized non-trapping emulation library**

The following implementation issues for the non-trapping floating-point library are important:

- when calculating with floats, rounding is done to the nearest integer (rounding towards infinity when equally near).
- there is no distinction between -0 and +0
- when an operand of a calculation is a NaN, Inf or subnormal, the result is undefined.
- when the result of a calculation would be a subnormal, the result is 0.
- whenever a NaN or Inf would be the result of a calculation, the result is undefined
- when converting single precision floats to integers, rounding is done to the nearest integer (rounding towards infinity when equally near). This is similar to the TriCore FPU hardware.
- when converting double precision floats to integers, the value is truncated. This is similar to the trapping emulation library.
- when a converted float overflows the target integer type, the value is saturated to MAX_INT or MIN_INT.
4.5.2 SPECIAL FLOATING-POINT VALUES

Below is a list of special, IEEE-754 defined, floating-point values as they can occur during run-time.

<table>
<thead>
<tr>
<th>Special value</th>
<th>Sign</th>
<th>Exponent</th>
<th>Mantissa</th>
</tr>
</thead>
<tbody>
<tr>
<td>+0.0 (Positive Zero)</td>
<td>0</td>
<td>all zeros</td>
<td>all zeros</td>
</tr>
<tr>
<td>-0.0 (Negative Zero)</td>
<td>1</td>
<td>all zeros</td>
<td>all zeros</td>
</tr>
<tr>
<td>+INF (Positive Infinite)</td>
<td>0</td>
<td>all ones</td>
<td>all zeros</td>
</tr>
<tr>
<td>-INF (Negative Infinite)</td>
<td>1</td>
<td>all ones</td>
<td>all zeros</td>
</tr>
<tr>
<td>NaN (Not a number)</td>
<td>0</td>
<td>all ones</td>
<td>not all zeros</td>
</tr>
</tbody>
</table>

*Table 4–3: Special floating-point values*

4.5.3 TRAPPING FLOATING-POINT EXCEPTIONS

Four floating-point libraries are delivered:

<table>
<thead>
<tr>
<th>Library to link</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>libfp.a</td>
<td>Floating-point library (non–trapping, no FPU). This is the default.</td>
</tr>
<tr>
<td>libfpt.a</td>
<td>Floating-point library (trapping, no FPU) (Control program option --fp–trap)</td>
</tr>
<tr>
<td>libfp_fpu.a</td>
<td>Floating-point library (non–trapping, with FPU instructions) (Compiler option --fpu–present)</td>
</tr>
<tr>
<td>libfpt_fpu.a</td>
<td>Floating-point library (trapping, with FPU instructions) (Control program option --fp–trap, compiler option --fpu–present)</td>
</tr>
</tbody>
</table>

*Table 4–4: Floating-point libraries*

Both EDE and the control program cctc automatically select the appropriate libraries depending on the specified TriCore derivative. By specifying the --fp–trap option to the control program cctc, the trapping type floating-point library is linked into your application. By specifying the --fpu–present option to the control program cctc, a floating-point library with single precision FPU instructions is linked into your application. If these options are not specified, the floating-point library without trapping mechanism and without FPU instructions is used.
In EDE you can specify to use the trapping type floating-point library as follows:

1. From the **Project** menu, select **Project Options...**  
   *The Project Options dialog appears.*

2. Expand the **Linker** entry and select **Libraries**.

3. Enable the option **Use trapping floating-point library**.

**IEEE-754 Trap Handler**

In the IEEE-754 standard a trap handler is defined, which is invoked on (specified) exceptional events, passing along much information about the event. To install your own trap handler, use the library call `_fp_install_trap_handler`. When installing your own exception handler, you must select on which types of exceptions you want to have your handler invoked, using the function call `_fp_set_exception_mask`. See below for more details on the floating-point library exception handling function interface.

**SIGFPE Signal Handler**

In ANSI-C the regular approach of dealing with floating-point exceptions is by installing a so-called signal handler by means of the ANSI-C library call `signal`. If such a handler is installed, floating-point exceptions cause this handler to be invoked. To have the signal handler for the **SIGFPE** signal actually become operational with the provided floating-point libraries, a (very) basic version of the IEEE-754 exception handler must be installed (see example below) which will raise the desired signal by means of the ANSI-C library function call `raise`. For this to be achieved, the function call `_fp_install_trap_handler` is present. When installing your own exception handler, you will have to select on which types of exceptions you want to receive a signal, using the function call `_fp_set_exception_mask`. See further below for more details on the floating-point library exception handling function interface.

There is no way to specify any information about the context or nature of the exception to the signal handler. Just that a floating-point exception occurred can be detected. See therefor the IEEE-754 trap handler discussion above if you want more control over floating-point results.
4.5.4  FLOATING-POINT TRAP HANDLING API

For purposes of dealing with floating-point arithmetic exceptions, the following library calls are available:

```c
#include <float.h>

static void pass_fp_exception_to_signal(
    _fp_exception_info_t *info )
{
    info;  /* suppress parameter not used warning */

    /* cause SIGFPE signal to be raised */

    raise( SIGFPE );
    /*
     * now continue the program
     * with the unaltered result
     */
}
```

A pair of functions to get or set the mask which controls which type of floating-point arithmetic exceptions are either ignored or passed on to the trap handler. The types of possible exception flag bits are defined as:

- `EFINVOP`
- `EFDIVZ`
- `EFOVFL`
- `EFUNFL`
- `EFINEXCT`

while,

- `EFALL`

is the OR of all possible flags. See below for an explanation of each flag.
#include <float.h>

int _fp_get_exception_status( void );
void _fp_set_exception_status( int );

A pair of functions for examining or presetting the status word containing
the accumulation of all floating-point exception types which occurred so
far. See the possible exception type flags above.

#include <float.h>

void _fp_install_trap_handler( void (*)
       (_fp_exception_info_t * ) );

This function call expects a pointer to a function, which in turn expects a
pointer to a structure of type _fp_exception_info_t. The members of
_fiexception_info_t are:

exception

This member contains one of the following (numerical) values:

EFINVOP
EFDIVZ
EFOVFL
EFUNFL
EFINEXCT

operation

This member contains one of the following numbers:

_OP_ADDITION
_OP_SUBTRACTION
_OP_COMPARISON
_OP_EQUALITY
_OP_LESS_THAN
_OP_LARGER_THAN
_OP_MULTIPLICATION
_OP_DIVISION
_OP_CONVERSION

source_format
destination_format
Numerical values of these two members are:

```c
_TYPE_SIGNED_CHARACTER
_TYPE_UNSIGNED_CHARACTER
_TYPE_SIGNED_SHORT_INTEGER
_TYPE_UNSIGNED_SHORT_INTEGER
_TYPE_SIGNED_INTEGER
_TYPE_UNSIGNED_INTEGER
_TYPE_SIGNED_LONG_INTEGER
_TYPE_UNSIGNED_LONG_INTEGER
_TYPE_FLOAT
_TYPE_DOUBLE
```

```c
operand1 /* left side of binary or */
    /* right side of unary */
operand2 /* right side for binary */
result
```

These three are of the following type, to receive and return a value of arbitrary type:

```c
typedef union _fp_value_union_t
{
    char c;
    unsigned char uc;
    short s;
    unsigned short us;
    int i;
    unsigned int ui;
    long l;
    unsigned long ul;
    float f;
    #if !_SINGLE_FP
    double d;
    #endif
} _fp_value_union_t;
```

The member d is not present when specifying the `-F` option (treat double as float) to the C compiler.
The following table lists all the exception code flags, the corresponding error description and result:

<table>
<thead>
<tr>
<th>Error Description</th>
<th>Exception Flag</th>
<th>Default Result with Trapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid Operation</td>
<td>EFINVOP</td>
<td>NaN</td>
</tr>
<tr>
<td>Division by zero</td>
<td>EFDIVZ</td>
<td>+INF or −INF</td>
</tr>
<tr>
<td>Overflow</td>
<td>EFOVFL</td>
<td>+INF or −INF</td>
</tr>
<tr>
<td>Underflow</td>
<td>EFUNFL</td>
<td>zero</td>
</tr>
<tr>
<td>Inexact</td>
<td>EFINEST</td>
<td>undefined</td>
</tr>
<tr>
<td>INF</td>
<td>Infinite which is the largest absolute floating-point number, which is always: −INF &lt; every finite number &lt; +INF</td>
<td></td>
</tr>
<tr>
<td>NAN</td>
<td>Not a Number, a symbolic entity encoded in floating-point format.</td>
<td></td>
</tr>
</tbody>
</table>

*Table 4-5: Exception Type Flag Codes*

To ensure all exception types are specified, you can specify **EFALL** to a function, which is the binary OR of all above enlisted flags.
CHAPTER 5

TOOL OPTIONS
5.1 COMPILER OPTIONS

This section lists all compiler options.

Options in EDE versus options on the command line

Most command line options have an equivalent option in EDE but some options are only available on the command line. If there is no equivalent option in EDE, you can specify a command line option in EDE as follows:

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

   The Project Options dialog appears.

2. Expand the C Compiler entry and select Miscellaneous.

3. Enter one or more command line options in the Additional options field.

   Be aware that some command line options are not useful in EDE or just do not have any effect. For example, the option `–n` sends output to stdout instead of a file and has no effect in EDE.

Short and long option names

Options have both short and long names. Short option names always begin with a single minus (`-`) character, long option names always begin with two minus (`--`) characters. You can abbreviate long option names as long as it forms a unique name. You can mix short and long option names on the command line.

Options can have flags or suboptions. To switch a flag 'on', use a lowercase letter or a `longflag`. To switch a flag off, use an uppercase letter or a `~longflag`. Separate `longflags` with commas. The following two invocations are equivalent:

```
ctc -Oac test.c
ctc --optimize=+coalesce,+cse test.c
```

When you do not specify an option, a default value may become active.
-? (--help)

EDE

Command line syntax

-?

--help[=item]

You can specify the following arguments:

- **intrinsics** Show the list of intrinsic functions
- **options** Show extended option descriptions
- **pragmas** Show the list of supported pragmas

Description

Displays an overview of all command line options. When you specify an argument you can list extended information such as a list of intrinsic functions, pragmas or option descriptions.

Example

The following invocations all display a list of the available command line options:

```
ctc -?
ctc --help
ctc
```

The following invocation displays a list of the available pragmas:

```
ctc --help=pragmas
```

Related information
-A (--language)

EDE
1. From the Project menu, select Project Options...
   *The Project Options dialog appears.*
2. Expand the C Compiler entry and select Language.
3. Enable or disable the options *Allow C++ style comments in ISO C90 mode* and *Allow relaxed const check for string literals*.

Command line syntax
- `A[flags]`
- `--language=[flags]`

You can set the following flags:

```
   p/P  (+/-comments)  Allow C++ style (//)comments in ISO C90
   x/X  (+/-strings)   Relaxed const check for string literals
```

Default
- `-Apx`

Description
With this option you control the language extensions the compiler can accept. Default the TriCore C compiler allows all language extensions.

The option `-A (--language)` without flags is the equivalent of `-APX` and disables all language extensions.

With `-Ap` you tell the compiler to allow C++ style comments (`//`) in ISO C90 mode (option `-c90`). In ISO C99 mode this style of comments is always accepted.

With `-Ax` you tell the compiler not to check for assignments of a constant string to a non-constant string pointer. With this option the following example does not produces a warning:

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

```
ctc -APx -c90 test.c
ctc --language=-comments,+strings --iso=90 test.c
```

The compiler compiles in ISO C90 mode, accepts assignments of a constant string to a non-constant string pointer but ignores C++ style comments.

Related information

- Compiler option `–c` (ISO C standard)
--align

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

   The Project Options dialog appears.

2. Expand the C Compiler entry and select Code Generation.
3. Specify a value in the Minimum alignment field.

Command line syntax
--align=value

Default
--align=1

Description
By default the TriCore compiler aligns objects to the minimum alignment required by the architecture. With this option you can increase this alignment for objects of four bytes or larger. The value must be a power of two.

Example
To align all objects of four bytes or larger on a 4-byte boundary, enter:

   ctc --align=4 test.c

Instead of this option you can also specify the following pragma in your C source:

   #pragma align 4

With #pragma align restore you can return to the previous alignment setting.

Related information
Section 3.7, Controlling the Compiler: Pragmas, in Chapter TriCore C Language of the TriCore User’s Manual.
-C (--cpu)

EDE

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

   The Project Options dialog appears.

2. Expand the Processor entry and select Processor Definition.

3. Select a processor from the Target processor list.

Command line syntax

   -C cpu

   --cpu=cpu

Description

With this option you define the target processor for which you create your application. Default the control program generates an object file for the TC11IB.

Based on the target processor the compiler automatically detects whether a FPU-unit is present and whether the architecture is a TriCore2. This means you do not have to specify the compiler options --fpu-present and --is-tricore2 explicitly when one of the supported derivatives is selected.

The compiler always includes the register file regcpu.sfr, unless you disable the option Automatic inclusion of '.sfr' file on the Preprocessing page of the Compiler options (command line option --no-tasking-sfr).

Example

To compile the file test.c for the TC11IB processor and use the SFR file regtc11lib.sfr:

   ctc -Ctc11lib test.c
   ctc --cpu=tc11lib test.c

To avoid conflicts, make sure you specify the same target processor to the assembler.
Related information

Compiler option `--no-tasking-sfr` (Do not include SFR file)

Assembler option `-C` (Select CPU)
Control program option `--C` (Use SFR definitions for CPU)

Section 5.4, *Calling the Compiler*, in Chapter *Using the Compiler* of the *User’s Manual*. 
-c (--iso)

EDE

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

   The Project Options dialog appears.

2. Expand the C Compiler entry and select Language.

3. Select the ISO C standard C90 or C99.

Command line syntax

   -c{90|99}
   --iso={90|99}

Default

   -c99

Description

   With this option you select the ISO C standard. C90 is also referred to as the "ANSI C standard". C99 refers to the newer ISO/IEC 9899:1999 (E) standard. C99 is the default.

Example

   To select the ISO C C90 standard on the command line:

   ctc -c90 test.c
   ctc --iso=90 test.c

Related information

   Compiler option -A (Language extensions)
**--check**

**EDE**

1. In the project window, select the file you want to check.
2. From the **Build** menu, select **Check Syntax**.

**Command line syntax**

--check

**Description**

With this option you can check the source code for syntax errors, without generating code. This saves time in developing your application.

The compiler reports any warnings and/or errors.

**Example**

To check for syntax errors, without generating code:

```
ctc --check test.c
```

**Related information**

Assembler option **--check** (Check syntax)
**--cse-all-addresses**

**EDE**

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

   *The Project Options dialog appears.*

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

3. Add the option **--cse-all-addresses** to the **Additional options** field.

**Command line syntax**

   **--cse-all-addresses**

**Description**

With this option you tell the compiler to make all addresses available for common subexpression evaluation.

Normally the compiler ignores **__near** and **__ax** addresses for common subexpressions. However, depending on the use of address registers and whether stack and/or addressed memory are internal or external, it might be wise to consider them for CSE.

**Example**

   `ctc --cse-all-addresses -Oc test.c`

   The compiler makes all addresses available for common subexpression evaluation.

**Related information**

   Compiler option **-Oc** (Common subexpression elimination)
\textbf{–D (\texttt{--define})}

\section*{EDE}

1. From the \textbf{Project} menu, select \textbf{Project Options...}

   \textit{The Project Options dialog appears.}

2. Expand the \textbf{C Compiler} entry and select \textbf{Preprocessing}.

3. Enter a macro name and/or definition in the \textbf{Define user macros} field.

   \textit{Use commas to separate multiple macro definitions.}

\subsection*{Command line syntax}

\begin{itemize}
  \item \texttt{–Dmacro\_name[=macro\_definition]}
  \item \texttt{--define=macro\_name[=macro\_definition]}
\end{itemize}

\subsection*{Description}

With this option you can define a macro and specify it to the preprocessor. If you only specify a macro name (no macro definition), the macro expands as '\texttt{1}'.

You can specify as many macros as you like. In EDE, use commas to separate multiple macro definitions. On the command line, use the option \texttt{–D} multiple times. If the command line exceeds the limit of the operating system, you can define the macros in an \textit{option file} which you then must specify to the compiler with the option \texttt{–f file}.

Defining macros with this option (instead of in the C source) is, for example, useful to compile conditional C source as shown in the example below.
Example

Consider the following C program with conditional code to compile a demo program and a real program:

```c
void main( void )
{
#if DEMO
    demo_func();    /* compile for the demo program */
#else
    real_func();    /* compile for the real program */
#endif
}
```

You can now use a macro definition to set the DEMO flag:

```bash
ctc -DDEMO test.c
ctc -DDEMO=1 test.c

c.tc --define=DEMO test.c
c.tc --define=DEMO=1 test.c
```

Note that all four invocations have the same effect.

The next example shows how to define a macro with arguments. Note that the macro name and definition are placed between double quotes because otherwise the spaces would indicate a new option.

```bash
ctc -D"MAX(A,B)=((A) > (B) ? (A) : (B))"
```

Related information

- Compiler option `-U` (Undefine macro)
- Compiler option `-f` (Specify an option file)
--diag

EDE

1. In the Help menu, enable the option Show Help on Tool Errors.

2. In the Build tab of the Output window, double-click on an error or warning message.

   A description of the selected message appears.

Command line syntax

```
--diag=[format;][all | number,number]...
```

Optionally, you can use one of the following display formats (format):

- **text**: The default is plain text
- **html**: Display explanation in HTML format
- **rtf**: Display explanation in RTF format

Description

With this option the compiler displays a description and explanation of the specified error message(s) on stdout (usually the screen). The compiler does not compile any files.

To create a file with the descriptions, you must redirect the output.

With the suboption all, the descriptions of all error messages are given. If you want the description of one or more selected error messages, you can specify the error message numbers, separated by commas.

With this option the compiler does not compile any files.

Example

To display an explanation of message number 282, enter:

```
ctc --diag=282
```

This results in the following message and explanation:

```
E282: unterminated comment

Make sure that all every comment starting with /* has a matching */. Nested comments are not possible.
```
To write an explanation of all errors and warnings in HTML format to file `errors.html`, enter:

```
tc --diag=html:all > errors.html
```

**Related information**

Section 5.8, *C Compiler Error Messages*, in Chapter *Using the Compiler* of the *User’s Manual*. 
-E (--preprocess)

EDE

1. From the **Project** menu, select **Project Options**...
   
   *The Project Options dialog appears.*

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

3. Enable the option **Store the C compiler preprocess output** (<file>.pre).

Command line syntax

```
-E[flags]
--preprocess[=flags]
```

You can set the following flags (when you specify `-E` without flags, the default is `-ECMP`):

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>c/C (+/-comments)</td>
<td>Keep comments</td>
</tr>
<tr>
<td>m/M (+/-make)</td>
<td>Generate dependencies for make</td>
</tr>
<tr>
<td>p/P (+/-noline)</td>
<td>Strip #line source position info</td>
</tr>
</tbody>
</table>

Description

With this option you tell the compiler to preprocess the C source. EDE stores the preprocess output in the file `name.pre` (where `name` is the name of the C source file to compile). EDE also compiles the C source.

On the command line, the compiler sends the preprocessed file to stdout. To capture the information in a file, specify an output file with the option `-o`.

With `-Ec` you tell the preprocessor to keep the comments from the C source file in the preprocessed output.

With `-Em` the compiler will generate dependency lines that can be used in a Makefile. The preprocessor output is discarded.

With `-Ep` you tell the preprocessor to strip the #line source position information (lines starting with `#line`). These lines are normally processed by the assembler and not needed in the preprocessed output. When you leave these lines out, the output is easier to read.
Example

```bash
ctc -EcMP test.c -o test.pre

ctc --preprocess=+comments,-make,-noline test.c
    --output=test.pre
```

The compiler preprocesses the file `test.c` and sends the output to the file `test.pre`. Comments are included but no dependencies are generated and the line source position information is not stripped from the output file.

Related information
--error-file

EDE

Command line syntax

--error-file[=file]

Description
With this option the compiler redirects error messages to a file.

If you do not specify a filename, the error file will be named after the input file with extension .err.

Example
To write errors to errors.err instead of stderr, enter:

    ctc --error-file=errors.err test.c

Related information

Compiler option --warnings-as-errors (Treat warnings as errors)
-F (--no-double)

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

   The Project Options dialog appears.

2. Expand the C Compiler entry and select Language.

3. Enable the option Single precision floating-point only.

Command line syntax

   -F

   --no-double

Description

With this option you tell the compiler to treat variables of the type double as float. Because the float type takes less space, execution speed increases and code size decreases, both at the cost of less precision.

Example

   ctc -F test.c

The file test.c is compiled where variables of the type double are treated as float.

Related information
\[-f \text{ (\texttt{--option-file})}\]

**EDE**

1. From the **Project** menu, select **Project Options...**
   
   *The Project Options dialog appears.*

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

3. Add the option \(-f\) to the **Additional options** field.

   In EDE you can save your options in a file and restore them to call the compiler with those options:

   - From the **Project** menu, select **Save Options...** or **Load Options...**

   Be aware that when you specify the option \(-f\) in the **Additional options** field, the options are *added* to the compiler options you have set in the Project Options dialog. Only in extraordinary cases you may want to use them in combination.

**Command line syntax**

\[-f \text{ file,}...\]

\[\text{--option-file=}\text{file,}...\]

**Description**

Instead of typing all options on the command line, you can create an option file which contains all options and files you want to specify. With this option you specify the option file to the compiler.

Use an option file when the length of the command line would exceed the limits of the operating system, or just to store options and save typing.

You can specify the option \(-f\) multiple times.
Format of an option file

- Multiple command line arguments on one line in the option file are allowed.
- To include whitespace in an argument, surround the argument with single or double quotes.
- If you want to use single quotes as part of the argument, surround the argument by double quotes and vice versa:
  
  "This has a single quote ' embedded"

  'This has a double quote " embedded'

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

- When a text line reaches its length limit, use a \ to continue the line. Whitespace between quotes is preserved.
  
  "This is a continuation \ line"
  -> "This is a continuation line"

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

Example

Suppose the file myoptions contains the following lines:

    -Ctcllib
    -s
    test.c

Specify the option file to the compiler:

    ctc -f myoptions

This is equivalent to the following command line:

    ctc -Ctcllib -s test.c

Related information
--fpu-present

EDE
1. From the Project menu, select Project Options...
   The Project Options dialog appears.
2. Expand the Processor entry and select Processor Definition.
3. In the Target processor list select (user defined TriCore−1 v1.3) or (user defined TriCore−2).
4. Enable the option FPU present.

Command line syntax
--fpu-present

Description
With this option the compiler can generate single precision floating–point instructions in the assembly file. When you select this option, the macro __FPU__ is defined in the C source file.

If you choose a valid target processor (command line option −C (--cpu)), this option is automatically set, based on the chosen target processor.

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

    ctc --fpu-present test.c

Related information

Compiler option --is-tricore2 (Tricore2 instructions allowed)

Compiler option −C (Use SFR definitions for CPU)
-g (--debug-info)

EDE
1. From the Project menu, select Project Options...
   The Project Options dialog appears.
2. Expand the C Compiler entry and select Debug Information.
3. Enable the option Generate symbolic debug information

Command line syntax
- g
   --debug-info

Description
With this option you tell the compiler to add directives to the output file for including symbolic information. This facilitates high level debugging but increases code size. For the final application, compile your C files without debug information.

When you specify a high optimization level, the debug comfort may decrease. Therefore, the compiler issues warning W555 if the debug comfort would be decreased as a result of the chosen optimizations.

Example
To add symbolic debug information to the output file, enter:

c tc -g test.c

Related information
-H (--include-file)

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

   The Project Options dialog appears.
2. Expand the C Compiler entry and select Preprocessing.
3. Enter the name of the file in the Include this file before source field.

Command line syntax
- H/file,...

   --include-file=file,...

Description
With this option you include one extra file at the beginning of each C source file, before other includes. This is the same as specifying #include "file" at the beginning of each of your C sources.

Example
ctc -Hstdio.h test1.c test2.c

The file stdio.h is included at the beginning of both test1.c and test2.c.

Related information

Compiler option -I (Add directory to include file search path)

Section 5.5, How the Compiler Searches Include Files, in Chapter Using the Compiler of the User’s Manual.
-I (--include-directory)

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

   The Directories dialog appears.

2. Enter one or more search paths in the Include Files Path field.

Command line syntax
-\l path,...

--include-directory=path,...

Description
With this option you can specify the path where your include files are located. A relative path will be relative to the current directory.

The order in which the compiler searches for include files is:

1. The pathname in the C source file and the directory of the C source
   (only for #include files that are enclosed in "")

2. The path that is specified with this option.

3. The path that is specified in the environment variable CTCINC when
   the product was installed.

4. The default directory $(PRODDIR)\include.

Example
Suppose that the C source file test.c contains the following lines:

    #include <stdio.h>
    #include "myinc.h"

You can call the compiler as follows:

    ctc -Imyinclude test.c

First the compiler looks for the file stdio.h in the directory myinclude
relative to the current directory. If it was not found, the compiler searches
in the environment variable and then in the default include directory.
The compiler now looks for the file `myinc.h` in the directory where `test.c` is located. If the file is not there the compiler searches in the directory `myinclude`. If it was still not found, the compiler searches in the environment variable and then in the default `include` directory.

**Related information**

- Compiler option `-H` (Include file at the start of a compilation)


**--indirect**

**EDE**

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

   *The Project Options dialog appears.*

2. Expand the **C Compiler** entry and select **Code Generation**.

3. Enable the option **Call functions indirectly**.

**Command line syntax**

```
--indirect
```

**Description**

With this option you tell the compiler to generate code for indirect function calling.

**Example**

```
ctc --indirect test.c
```

The compiler generates far calls for all functions.

Instead of this option you can also specify the following pragma in your C source:

```
#pragma indirect
```

**Related information**

Compiler option **--indirect-runtime**

See also section 3.9.3, *Function Calling Modes: __indirect*, in Chapter *TriCore C Language* of the *User’s Manual*. 
--indirect-runtime

EDE
1. From the Project menu, select Project Options...
   The Project Options dialog appears.
2. Expand the C Compiler entry and select Code Generation.
3. Enable the option Call run-time functions indirectly.

Command line syntax
--indirect-runtime

Description
With this option you tell the compiler to generate code for indirect calls to run-time functions. Use this option if you locate the entire run-time library in far memory.

Example
```c
ctc --indirect-runtime test.c
```

The compiler generates far calls for all run-time functions.

Instead of this option you can also specify the following pragma in your C source:
```
#pragma indirect_runtime
```

Related information

Compiler option --indirect

See also section 3.9.3, Function Calling Modes: __indirect, in Chapter TriCore C Language of the User’s Manual.
--inline

EDE

Command line syntax

--inline

Description
With this option you instruct the compiler to inline all functions, regardless whether they have the keyword `inline` or not. This option has the same effect as a `#pragma inline` at the start of the source file.

This option can be useful to increase the possibilities for code compaction (option `-Or`).

Example
To inline all functions:

```
ctc --inline test.c
```

Related information

Compiler option `-Or` (Optimization: code compaction)
--inline-max-incr /
--inline-max-size

EDE

Command line syntax

--inline-max-incr=percentage
--inline-max-size=thresbold

Default

--inline-max-incr=25
--inline-max-size=10

Description

With these options you can control the function inlining optimization process of the compiler. These options have only effect when you have enabled the inlining optimization (option -Oi).

Regardless of the optimization process, the compiler always inlines all functions that have the function qualifier inline.

With the option --inline-max-size you can specify the maximum size of functions that the compiler inlines as part of the optimization process. The compiler always inlines all functions that are smaller than the specified thresbold. The threshold is measured in compiler internal units and the compiler uses this measure to decide which functions are small enough to inline. The default threshold is 10.

After the compiler has inlined all functions that have the function qualifier inline and all functions that are smaller than the specified threshold, the compiler looks whether it can inline more functions without increasing the code size too much. With the option --inline-max-incr you can specify how much the code size is allowed to increase. Default, this is 25% which means that the compiler continues inlining functions until the resulting code size is 25% larger than the original size.
Example

ctc --inline-max-incr=40 --inline-max-size=15 test.c

The compiler first inlines all functions with the function qualifier inline and all functions that are smaller than the specified threshold of 15. If the code size has still not increased with 40%, the compiler decides which other functions it can inline.

Related information

Compiler option -O (Specify optimization level)

Section 3.9.1, Inlining Functions, in Chapter TriCore C Language of the User’s Manual.
--integer-enumeration

EDE

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

   *The Project Options dialog appears.*

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

3. Enable the option **Use 32-bit integers for enumeration**.

**Command line syntax**

    --integer-enumeration

**Description**

With this option you tell the compiler to use (32-bit) integers for enumerations. Without this option, the compiler treats small enumerated types as a smaller integer, a `char` or even a `__bit` type to reduce code size.

**Example**

To treat enumerated types always as 32-bit integer, enter:

    ctc --integer-enumeration test.c

**Related information**
is-tricore2

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

*The Project Options dialog appears.*

2. Expand the Processor entry and select Processor Definition.

3. In the Target processor list select (user defined TriCore-2).

4. Optionally enable or disable the options FPU present and MMU present.

Command line syntax

--is-tricore2

Description
With this option the compiler can generate TriCore 2 instructions in the assembly file. When you select this option, the macro __TC2__ is defined in the C source file.

If you choose a valid target processor (command line option --C (--cpu)), this option is automatically set, based on the chosen target processor.

Example
To allow the use of TriCore 2 instructions in the assembly code, enter:

```
ctc --is-tricore2 test.c
```

Related information

- Compiler option --fpu-present (Use hardware floating-point instructions)
- Assembler option --mmu-present (Allow use of MMU instructions)
- Compiler option --C (Use SFR definitions for CPU)
–k (--keep-output-files)

EDE

EDE always removes the .src file when errors occur during compilation.

Command line syntax

–k

--keep-output-files

Description

If an error occurs during compilation, the resulting .src file may be incomplete or incorrect. With this option you keep the generated output file (.src) when an error occurs.

By default the compiler removes the generated output file (.src) when an error occurs. This is useful when you use the make utility mktc. If the erroneous files are not removed, the make utility may process corrupt files on a subsequent invocation.

Use this option when you still want to inspect the generated assembly source. Even if it is incomplete or incorrect.

Example

ctc -k test.c

When an error occurs during compilation, the generated output file test.src will not be removed.

Related information

Compiler option --warnings-as-errors (Treat warnings as errors)
--misrac

EDE

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

   The Project Options dialog appears.

2. Expand the C Compiler entry and select MISRA C.

3. Select a MISRA C configuration.

4. (Optional) In the MISRA C Rules entry, specify the individual rules.

Command line syntax

   --misrac={all | number [-number],... }

Description

With this option you specify to the compiler which MISRA C rules must be checked. With the option --misrac=all the compiler checks for all supported MISRA C rules.

Example

   ctc --misrac=9-13 test.c

   The compiler generates an error for each MISRA C rule 9, 10, 11, 12 or 13 violation in file test.c.

Related information

   See Chapter 10 MISRA C Rules for a list of all supported MISRA C rules.
   
   Linker option --misra-c-report.
-N (--default-near-size)

EDE
1. From the Project menu, select Project Options...
   The Project Options dialog appears.
2. Expand the C Compiler entry and select Allocation.
3. Enable the option Default __near allocations for objects below threshold and enter a threshold value.

Command line syntax
- N[threshold]
   --default-near-size[=threshold]

Default
- N8

Description
With this option you can specify a threshold value for __near allocation. If you do not specify __near or __far in the declaration of an object, the compiler chooses where to place the object. The compiler allocates objects smaller or equal to the threshold in __near sections. Larger objects are allocated in __a0, __a1 or __far sections.

The default threshold is eight bytes.

If you specify -N without a threshold value, all objects will be allocated __near, including arrays and string constants.

Instead of this option you can also use #pragma default_near_size in the C source.

Example
   ctc -N12 test.c

Data elements smaller than or equal to 12 bytes are allocated in __near sections.
Related information

- Compiler option `-Y` (maximum size in bytes for data elements that are default located in `__a1` sections)
- Compiler option `-Z` (maximum size in bytes for data elements that are default located in `__a0` sections)

Section 3.3.1, *Declare a Data Object in a Special Part of Memory*, in Chapter *TriCore C Language* of the *User’s Manual.*
-n (--stdout)

**EDE**

-  

**Command line syntax**

  -n
  
  --stdout

**Description**

With this option you tell the compiler to send the output to stdout (usually your screen). No files are created.

This option is for example useful to quickly inspect the output or to redirect the output to other tools.

**Example**

```bash
ctc -n test.c
```

The compiler sends the output (normally `test.src`) to stdout and does not create the file `test.src`.

**Related information**

-
**--no-tasking-sfr**

**EDE**

1. From the **Project** menu, select **Project Options**...
   
   *The Project Options dialog appears.*

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

3. Disable the option **Automatic inclusion of `.sfr` file**.

**Command line syntax**

```
--no-tasking-sfr
```

**Description**

Normally, the compiler includes a special function register (SFR) file before compiling. The compiler automatically selects the SFR file belonging to the target you select on the **Processor definition** page of the Processor options (compiler option `-C`).

With this option the compiler does *not* include the register file `regcpu.sfr` as based on the selected target processor.

Use this option if you want to use your own set of SFR files.

**Example**

```
ctc -Ctcllib --no-tasking-sfr test.c
```

The register file `regtcllib.sfr` is not included.

**Related information**

- Compiler option `-C` (Use SFR definitions for CPU)
-O (--optimize)

EDE

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

   The Project Options dialog appears.

2. Expand the C Compiler entry and select Optimization.

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

Command line syntax

-0[flags]
--optimize[=flags]

You can set the following flags:

- a/A (+/-coalesce) Coalescer: remove unnecessary moves
- c/C (+/-cse) Common subexpression elimination
- e/E (+/-expression) Expression simplification
- f/F (+/-flow) Control flow optimization and code reordering
- g/G (+/-glo) Generic assembly optimizations
- i/I (+/-inline) Function inlining
- k/K (+/-schedule) Instruction scheduler
- l/L (+/-loop) Loop transformations
- m/M (+/-simd) Perform SIMD optimizations
- o/O (+/-forward) Forward store
- p/P (+/-propagate) Constant propagation
- s/S (+/-subscript) Subscript strength reduction
- v/V (+/-ifconvert) Convert IF statements using predicates
- w/W (+/-pipeline) Software pipelining
- y/Y (+/-peephole) Peephole optimizations

Use the following options for predefined sets of flags:

-O0 (--optimize=0) No optimization.

   Alias for: -OACEFGIKLMO PSVWY

No optimizations are performed. The compiler tries to achieve a 1-to-1 resemblance between source code and produced code. Expressions are evaluated in the same order as written in the source code, associative and commutative properties are not used.
-O1  (--optimize=1)  Debug purpose optimization  
   Alias for:  -Oaefglklmopsvwy

Enables optimizations that do not affect the debug-ability of the source code. Use this level when you are developing/debugging new source code.

-02  (--optimize=2)  Release purpose optimization (default)  
   Alias for:  -Oacefglklmopsvwy

Enables more optimizations to reduce code size and/or execution time. The debugger can handle this code but the relation between source code and generated instructions may be hard to understand. Use this level for those modules that have already been debugged. This is the default optimization level.

-03  (--optimize=3)  Aggressive optimization  
   Alias for:  -Oacefglklmopsvwy

Enables aggressive global optimization techniques. Although in theory debugging is still possible, the relation between source code and generated instructions is complex and hard to understand. Use this level to compress your program into the system memory, or to decrease execution time to meet your real-time requirements.

**Default**

-02

**Description**

With this option you can control the level of optimization. If you do not use this option, the default optimization level is medium optimization (option -02 or -O or -Oacefglklmopsvwy).

When you use this option to specify a set of optimizations, you can overrule these settings in your C source file with

```
#pragma optimize flag
#pragma endoptimize
```

In addition to the option -O, you can specify the option -t. With this option you specify whether the used optimizations should optimize for more speed (regardless of code size) or for smaller code size (regardless of speed).
Example

The following invocations are equivalent and result all in the default medium optimization set:

```
ctc test.c
ctc -02 test.c
ctc --optimize=2 test.c
ctc -0 test.c
ctc --optimize test.c

ctc -OacefgIklpswy test.c
ctc --optimize=+coalesce,+cse,+expression,+flow,
+glo,-inline,+schedule,+loop,+propagate,
+subscript,+ifconvert,+pipeline,+peephole test.c
```

Related information

Compiler option `-t` (Trade off between speed (`-t0`) and size (`-t4`))

```
#pragma optimize flag
#pragma endoptimize
```

Section 5.3, *Compiler Optimizations*, in Chapter *Using the Compiler* of the *User’s Manual*.
-o (---output)

EDE

Command line syntax

-ofile

--output=file

Description

With this option you can specify another filename for the output file of the compiler. Without this option the basename of the C source file is used with extension .src.

EDE names the output file always after the C source file.

Example

    ctc -o output.src test.c
    ctc --output=output.src test.c

The compiler creates the file output.src for the compiled file test.c.

Without the option -o, like EDE, the compiler uses the names of the input file and creates test.src.

Related information
--object-comment

EDE

Command line syntax

--object-comment=comment

Description

With this option the compiler generates a .comment section at the end of the assembly file. The section contains the comment specified with this option. After assembling, this text is included in the .o object and .elf files. Place the comment between double quotes.

Example

ctc --object-comment="Created by TASKING" test.c

The compiler creates the file test.src with a .comment section at the end of the file. After assembling this file, the text "Created by TASKING" is incorporated in the generated object file.

Instead of this option you can also specify the following pragma in your C source:

#pragma object_comment comment

Related information
-R (--rename-sections)

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

   The Project Options dialog appears.

2. Expand the C Compiler entry and select Miscellaneous.

3. Add the option -R to the Additional options field.

Command line syntax

   -R [name]

   --rename-sections[=name]

Description
The compiler defaults to a section naming convention, using a prefix indicating the section type, the module name and a symbol name:

   section_type_pref.module_name.symbol_name

For example .text.module_name.symbol_name for code sections.

If a module must be loaded at a fixed address or if a data section needs a special place in memory, you can use the -R option to generate a different section name (section_type_pref.name where name replaces the part module_name.symbol_name). You can now use the new unique section name in the linker script file for locating.

When you use -R without a value, the compiler uses the default section naming.

Example
To generate the section name section_type_pref.NEW instead of the default section name section_type_pref.module_name.symbol_name, enter:

   ctc -RNEW test.c

To generate the section name section_type_pref instead of the default section name section_type_pref.module_name.symbol_name, enter:

   ctc -R" " test.c (note the space between the quotes)
Related information

Section 3.10, Compiler Generated Sections, in Chapter TriCore C Language of the User’s Manual.
-s (---source)

EDE

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

   *The Project Options dialog appears.*

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

3. Enable the option **Merge C source code with assembly in output file (.src)**.

**Command line syntax**

```bash
-s
--source
```

**Description**

With this option you tell the compiler to merge C source code with generated assembly code in the output file. The C source lines are included as comments.

**Example**

```bash
ctc -s test.c
```

The output file `test.src` contains the original C source lines as comments, besides the generated assembly code.

**Related information**
--section-per-data-object

EDE

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

   The Project Options dialog appears.

2. Expand the C Compiler entry and select Code Generation.

3. Enable the option Generate a section for each data object.

Command line syntax

   --section-per-data-object

Description

Normally the compiler generates one section for each data type in a module (such as .data, .rodata, .bss, .zdata, ...).

With this option you force the compiler to generate a separate section for each data object. This provides more control about allocation during the linking process.

Example

   ctc --section-per-data-object test.c

   For each data object in test.c the compiler generates a separate section.

Related information
## silicon-bug

**EDE**

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

   *The Project Options dialog appears.*

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

3. Select the bypasses you want to enable.

### Command line syntax

`--silicon-bug=arg,...`

You can give one or more of the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>all-tc112</td>
<td>All TriCore 1 v1.2 (TC112) workarounds</td>
<td>all-tc112</td>
</tr>
<tr>
<td>all-tc113</td>
<td>All TriCore 1 v1.3 (TC113) workarounds</td>
<td>all-tc113</td>
</tr>
<tr>
<td>cor1</td>
<td>workaround for TC112 COR1 (#pragma TC112_COR1)</td>
<td>cor1</td>
</tr>
<tr>
<td>cor4</td>
<td>workaround for TC112 COR4 (#pragma TC112_COR4)</td>
<td>cor4</td>
</tr>
<tr>
<td>cor7</td>
<td>workaround for TC112 COR7 (#pragma TC112_COR7)</td>
<td>cor7</td>
</tr>
<tr>
<td>cor10</td>
<td>workaround for TC112 COR10 (#pragma TC112_COR10)</td>
<td>cor10</td>
</tr>
<tr>
<td>cor13</td>
<td>workaround for TC112 COR13 (#pragma TC112_COR13)</td>
<td>cor13</td>
</tr>
<tr>
<td>cor14</td>
<td>workaround for TC112 COR14 (#pragma TC112_COR14)</td>
<td>cor14</td>
</tr>
<tr>
<td>cor16</td>
<td>workaround for TC112 COR16 (#pragma TC112_COR16)</td>
<td>cor16</td>
</tr>
<tr>
<td>cor17</td>
<td>workaround for TC112 COR17 (#pragma TC112_COR17)</td>
<td>cor17</td>
</tr>
<tr>
<td>cpu-tc013</td>
<td>workaround for CPU_TC.013 (#pragma CPU_TC013)</td>
<td>cpu-tc013</td>
</tr>
<tr>
<td>cpu-tc030</td>
<td>workaround for CPU_TC.030 (#pragma CPU_TC030)</td>
<td>cpu-tc030</td>
</tr>
<tr>
<td>cpu-tc031</td>
<td>workaround for CPU_TC.031 (#pragma CPU_TC031)</td>
<td>cpu-tc031</td>
</tr>
<tr>
<td>cpu-tc034</td>
<td>workaround for CPU_TC.034 (#pragma CPU_TC034)</td>
<td>cpu-tc034</td>
</tr>
<tr>
<td>cpu-tc048</td>
<td>workaround for CPU_TC.048 (#pragma CPU_TC048)</td>
<td>cpu-tc048</td>
</tr>
<tr>
<td>cpu-tc050</td>
<td>workaround for CPU_TC.050 (#pragma CPU_TC050)</td>
<td>cpu-tc050</td>
</tr>
<tr>
<td>cpu-tc060</td>
<td>workaround for CPU_TC.060 (#pragma CPU_TC060)</td>
<td>cpu-tc060</td>
</tr>
</tbody>
</table>
### Description

With this option you tell the compiler to use software workarounds for some CPU functional problems.

### Example

```bash
ctc --silicon-bug=cpu5,cpu9 test.c
```

The compiler uses workarounds for TC113 problems CPU5 and CPU9.

### Related information

See Chapter 9, *CPU Functional Problems*, for more information about the individual problems and workarounds.

```c
#pragma CPU_functional_problem
#pragma TC112_functional_problem
#pragma TC113_functional_problem
```
**--static**

EDE

Command line syntax

    --static

Description

With this option, the compiler treats external definitions at file scope (except for `main`) as if they were declared `static`. As a result, unused functions will be eliminated, and the alias checking algorithm assumes that objects with static storage cannot be referenced from functions outside the current module.

This option only makes sense when you specify all modules of an application on the command line.

Example

    ctc --static module1.c module2.c module3.c

Related information
--switch

EDE

1. From the Project menu, select Project Options...
   *The Project Options dialog appears.*

2. Expand the C Compiler entry and select Code generation.

3. Select an Algorithm for switch statements.

Command line syntax

   --switch=arg

You can give one of the following arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>auto</td>
<td>Choose most optimal code</td>
</tr>
<tr>
<td>jumptab</td>
<td>Generate jump tables</td>
</tr>
<tr>
<td>linear</td>
<td>Use linear jump chain code</td>
</tr>
<tr>
<td>lookup</td>
<td>Generate lookup tables</td>
</tr>
</tbody>
</table>

Default

   --switch=auto

Description

With this option you tell the compiler which code must be generated for a switch statement: a jump chain (linear switch), a jump table or a lookup table. By default, the compiler will automatically choose the most efficient switch implementation based on code and data size and execution speed.

Example

   ctc --switch=jumptab test.c

The compiler uses a table filled with target addresses for each possible switch value.

Instead of this option you can also specify the following pragma in your C source:

   #pragma switch jumptab
Related information

See also section 3.11, *Switch Statement*, in Chapter *TriCore C Language* of the *User's Manual*. 
−t (--tradeoff)

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

   The Project Options dialog appears.

2. Expand the C Compiler entry and select Optimization.


Command line syntax

−t{0 | 1 | 2 | 3 | 4}

   --tradeoff={0 | 1 | 2 | 3 | 4}

Default

   −t2

Description

If the compiler uses certain optimizations (option −O), you can use this option to specify whether the used optimizations should optimize for more speed (regardless of code size) or for smaller code size (regardless of speed).

   Default: the compiler balances speed and size while optimizing (−t0).

   If you have not used the option −O, the compiler uses default medium optimization, so you can still specify the option −t.

Example

   To set the trade–off level for the used optimizations:

         ctc −t4 test.c
         ctc --tradeoff=4 test.c

   The compiler uses the default medium optimization level and optimizes for code size rather than for speed.

Related information

   Compiler option −O (Specify optimization level)
–U (--undefine)

EDE

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

   The Project Options dialog appears.

2. Expand the C Compiler entry and select Preprocessing.

3. Disable one or both predefined macros.

Command line syntax

- U macro_name

--undefine=macro_name

Description

With this option you can undefine an earlier defined macro as with #undef. The TriCore compiler predefines the following macros:

```c
#define ___TASKING__ 1
#define ___CTC__ compiler_version_nr
```

This option is for example useful to undefine predefined macros.

The following predefined ISO C standard macros cannot be undefined:

```c
__FILE__ current source filename
__LINE__ current source line number (int type)
__TIME__ hh:mm:ss
__DATE__ MMM dd yyyy
__STDC__ level of ANSI standard
```

Example

To undefine the predefined macro ___TASKING___:

```
ctc -U ___TASKING__ test.c
ctc --undefine=___TASKING__ test.c
```
Related information

Compiler option `-D` (Define macro)

Section 3.8, *Predefined Macros*, in Chapter *Using the Compiler* of the *User’s Manual*. 
-u (--)uchar

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

   *The Project Options dialog appears.*

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

3. Enable the option **Treat 'char' variables as unsigned instead of signed**.

Command line syntax
- `--u`
- `--uchar`

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

Example
With the following command `char` is treated as `unsigned char`:

- `ctc -u test.c`
- `ctc --uchar test.c`

Related information
-V (--version)

EDE
-

Command line syntax
- V
--version

Description
Display version information. The compiler ignores all other options or input files.

Example
  ctc -v
  ctc --version

The compiler does not compile any files but displays the following version information:

  TASKING TriCore VX-toolset C compiler  vxx.yrz Build 000
  Copyright 2002-2004 Altium BV  Serial# 00000000

Related information
-
−w (--no-warnings)

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

   The Project Options dialog appears.

2. Expand the C Compiler entry and select Warnings.

3. Enable one of the options Report all warnings, Suppress all warnings, or Suppress specific warnings.

   If you select Suppress specific warnings:

4. Enter the numbers, separated by commas, of the warnings you want to suppress.

Command line syntax

−w[nr]

--no-warnings[=nr]

Description
With this option you can suppresses all warning messages or specific warning messages.

• If you do not specify this option, all warnings are reported.
• If you specify this option but without numbers, all warnings are suppressed.
• If you specify this option with a number, only the specified warning is suppressed. You can specify the option −w multiple times.

Example
To suppress all warnings:

c tc test.c −w
ctc test.c --no-warnings

To suppress warnings 135 and 136:

c tc test.c −w135 −w136
c tc test.c --no-warnings=135 --no-warnings=136
Related information

Compiler option **--warnings-as-errors** (Treat warnings as errors)
--warnings-as-errors

EDE

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

   The Project Options dialog appears.

2. Expand the C Compiler entry and select Warnings.

3. Enable the option Treat warnings as errors.

Command line syntax

   --warnings-as-errors

Description

With this option you tell the compiler to treat warnings as errors.

Example

   ctc --warnings-as-errors test.c

When a warning occurs, the compiler considers it as an error.

Related information

   Compiler option -w (suppress some or all warnings)
-Y (--default-a1-size)

EDE

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

*The Project Options dialog appears.*

2. Expand the C Compiler entry and select Allocation.

3. Enable the option Default __a1 allocations for objects below threshold and enter a threshold value.

Command line syntax

```
--default-a1-size[=threshold]
```

Default

-Y0

Description

With this option you can specify a threshold value for __a0 allocation. If you do not specify a memory qualifier such as __near or __far in the declaration of an object, the compiler chooses where to place the object based on the size of the object.

First, the size of the object is checked against the -N threshold, according to the description of the -N option. If the size is larger than the -N threshold, but lower or equal to the -Y threshold, the object is allocated in __a1 memory. Larger objects, arrays and strings will be allocated __far.

The default -Y threshold is zero, which means that the compiler will never use __a1 memory unless you specify the -Y option. When you use the -Y option without a threshold value, all objects not allocated __near, including arrays and string constants, will be allocated in __a1 memory.

Allocation in __a1 memory means that the object is addressed indirectly, using A1 as the base pointer. The total amount of memory that can be addressed this way is 64 Kbytes.

Instead of this option you can also use #pragma default_a1_size in the C source.
Example

```
ctc -Y12 test.c
```

Data elements smaller than or equal to 12 bytes are allocated in `__a1` sections.

Related information

Compiler option `-Z` (max size (in bytes) for rodata elements located in `__a1` sections)

Compiler option `-N` (maximum size in bytes for data elements that are default located in `__near` sections)

Section 3.3.1, *Declare a Data Object in a Special Part of Memory*, in Chapter *TriCore C Language* of the *User’s Manual.*
-Z (--default-a0-size)

EDE

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

The Project Options dialog appears.

2. Expand the C Compiler entry and select Allocation.

3. Enable the option Default __a0 allocations for objects below threshold and enter a threshold value.

Command line syntax

- \texttt{-Z[threshold]}

- \texttt{--default-a0-size[=threshold]}

Default

- \texttt{-Z0}

Description

With this option you can specify a threshold value for __a0 allocation. If you do not specify a memory qualifier such as __near or __far in the declaration of an object, the compiler chooses where to place the object based on the size of the object.

First, the size of the object is checked against the -N threshold, according to the description of the -N option. If the size is larger than the -N threshold, but lower or equal to the -Z threshold, the object is allocated in __a0 memory. Larger objects, arrays and strings will be allocated __far.

The default -Z threshold is zero, which means that the compiler will never use __a0 memory unless you specify the -Z option. When you use the -Z option without a threshold value, all objects not allocated __near, including arrays and string constants, will be allocated in __a0 memory.

Allocation in __a0 memory means that the object is addressed indirectly, using A0 as the base pointer. The total amount of memory that can be addressed this way is 64 Kbytes.

Instead of this option you can also use \#pragma default_a0_size in the C source.
Example

ctc -Z12 test.c

Data elements smaller than or equal to 12 bytes are allocated in __a0 sections.

Related information

Compiler option -Y (max size (in bytes) for data elements located in __a0 sections)

Compiler option -N (maximum size in bytes for data elements that are default located in __near sections)

Section 3.3.1, Declare a Data Object in a Special Part of Memory, in Chapter TriCore C Language of the User’s Manual.
5.2 ASSEMBLER OPTIONS

This section lists all assembler options.

Options in EDE versus options on the command line

Most command line options have an equivalent option in EDE but some options are only available on the command line. If there is no equivalent option in EDE, you can specify a command line option in EDE as follows:

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

   The Project Options dialog appears.

2. Expand the Assembler entry and select Miscellaneous.

3. Enter one or more command line options in the Additional options field.

   Be aware that some command line options are not useful in EDE or just do not have any effect. For example, the option -V displays version header information and has no effect in EDE.

Short and long option names

Options have both short and long names. Short option names always begin with a single minus (-) character, long option names always begin with two minus (--) characters. You can abbreviate long option names as long as it forms a unique name. You can mix short and long option names on the command line.

Options can have flags or suboptions. To switch a flag 'on', use a lowercase letter or a +longflag. To switch a flag off, use an uppercase letter or a –longflag. Separate longflags with commas. The following two invocations are equivalent:

   astc -Lmx test.src
   astc --list-format=+macro,+macro-expansion test.src

When you do not specify an option, a default value may become active.
-? (---help)

EDE

Command line syntax

-?

--help[=options]

Description
Displays an overview of all command line options. When you specify the options argument, a list with option descriptions is displayed.

Example
The following invocations all display a list of the available command line options:

astc -?
astc --help
astc

The following invocation displays extended information about all options:

astc --help=options

Related information
EDE
1. From the Project menu, select Project Options...

   The Project Options dialog appears.

2. Expand the Processor entry and select Processor Definition.

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

Command line syntax

   `-C cpu`

   `--cpu=cpu`

Description

With this option you define the target processor for which you create your application.

Based on the target processor the assembler automatically detects whether a MMU or FPU-unit is present and whether the architecture is a TriCore2. This means you do not have to specify the assembler options `--mmu-present`, `--fpu-present` and `--is-tricore2` explicitly when one of the supported derivatives is selected.

The assembler automatically includes the register file `regcpu.def`, unless you specify assembler option `--no-tasking-sfr`.

Example

In EDE, the target CPU has the following settings:

- Target processor: TC11IB

To define this on the command line:

   `astc -Ctc11lib test.src`
   `astc --cpu=tc11lib test.src`

The assembler assembles `test.src` for the TC11IB processor and includes the register file `regtc11lib.def`. Furthermore the assembler allows MMU instructions to be used.
To avoid conflicts, make sure you specify the same target processor as you did for the compiler.

Related information

Assembler option --no-tasking-sfr (Do not include .def file)

Compiler option -C (Use SFR definitions for CPU)
Control program option -C (Use SFR definitions for CPU)

Section 6.4, Calling the Assembler, in Chapter Using the Assembler of the User's Manual.
–c (––case-insensitive)

EDE
1. From the Project menu, select Project Options...
   
   The Project Options dialog appears.

2. Expand the Assembler entry and select Miscellaneous.

3. Disable the option Assemble case sensitive.

Command line syntax

–c

--case-insensitive

Description

With this option you tell the assembler not to distinguish between upper and lower case characters. By default the assembler considers upper and lower case characters in labels and user-defined symbols as different characters. Note that instruction mnemonics, register names, directives and controls are always treated case insensitive.

Disabling the option Assemble case sensitive in EDE is the same as specifying the option –c on the command line.

Assembly source files that are generated by the compiler must always be assembled case sensitive. When you are writing your own assembly code, you may want to specify the case insensitive mode.

Example
To assemble case insensitive:

    astc -c test.src
    astc --case-insensitive test.src

The assembler considers upper and lower case characters as being the same. So, for example, the label LabelName is the same label as labelname.

Related information

Linker option --case-sensitive (Link case insensitive)
**--check**

**EDE**
1. In the project window, select the file you want to check.
2. From the **Build** menu, select **Check Syntax**.

**Command line syntax**

`--check`

**Description**

With this option you can check the source code for syntax errors, without generating code. This saves time in developing your application.

The assembler reports any warnings and/or errors.

**Example**

To check for syntax errors, without generating code:

```
astc --check test.src
```

**Related information**

Compiler option `--check` (Check syntax)
**–D (—define)**

**EDE**

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

   *The Project Options dialog appears.*

2. Expand the **Assembler** entry and select **Preprocessing**.

3. Enter a macro name and/or definition in the **Define user macros** field.

   *Use commas to separate multiple macro definitions.*

**Command line syntax**

\[-Dmacro\_name[=macro\_definition]\]

\[—define=macro\_name[=macro\_definition]\]

**Description**

With this option you can define a macro and specify it to the assembler preprocessor. If you only specify a macro name (no macro definition), the macro expands as ’1’.

You can specify as many macros as you like. In EDE, use commas to separate multiple macro definitions. On the command line you can use the option \(-D\) multiple times. If the command line exceeds the limit of the operating system, you can define the macros in an option file which you then must specify to the assembler with the option \(-f\) file.

Defining macros with this option (instead of in the assembly source) is, for example, useful in combination with conditional assembly as shown in the example below.

This option has the same effect as defining symbols via the \(\text{.SET}\), and \(\text{.EQU}\) directives. (similar to \#define in the C language). With the \text{.MACRO} directive you can define more complex macros.
Example

Consider the following C program with conditional code to compile a demo program and a real program:

```
 IF DEMO == 1
     ...
     ; instructions for demo application
 ELSE
     ...
     ; instructions for the real application
ENDIF
```

You can now use a macro definition to set the DEMO flag:

```
astc -DDEMO test.src
astc -DDEMO=1 test.src
astc --define=DEMO test.src
astc --define=DEMO=1 test.src
```

Note that all four invocations have the same effect.

Related information

Assembler option `-f` (Specify an option file)

Section 4.10.5, Conditional Assembly, in Chapter TriCore Assembly Language of the User's Manual.
**--diag**

**EDE**

1. In the **Help** menu, enable the option **Show Help on Tool Errors**.

2. In the **Build** tab of the **Output** window, double-click on an error or warning message.

   *A description of the selected message appears.*

**Command line syntax**

```bash
--diag=[/format;][all | number,number]...
```

Optionally, you can use one of the following display formats (**format***):

- **text**  
  The default is plain text
- **html**  
  Display explanation in HTML format
- **rtf**  
  Display explanation in RTF format

**Description**

With this option you can ask for an extended description of error messages in the format you choose. The output is directed to **stdout** (normally your screen) and in the format you specify.

To create a file with the descriptions, you must redirect the output.

With the suboption **all**, the descriptions of all error messages are given. If you want the description of one or more selected error messages, you can specify the error message numbers, separated by commas.

With this option the assembler does not assemble any files.

**Example**

To display an explanation of message number 241, enter:

```bash
astc --diag=241
```

This results in the following message and explanation:

```
W241: additional input files will be ignored

The assembler supports only a single input file. All other input files are ignored.
```
To write an explanation of all errors and warnings in HTML format to file `aserrors.html`, enter:

```
astc --diag=html:all > aserrors.html
```

**Related information**

--emit-locals

EDE

Command line syntax

--emit-locals

Description

With this option the assembler also emits local symbols to the object file’s symbol table. Normally, only global symbols are emitted.

Example

To emit local symbols, enter:

    astc --emit-locals test.src

Related information
**--error-file**

**EDE**

**Command line syntax**

```
--error-file=[file]
```

**Description**

With this option the assembler redirects error messages to a file.

If you do not specify a filename, the error file will be named after the input file with extension `.ers`.

**Example**

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

```
astc --error-file=errors.ers test.src
```

**Related information**

Assembler option **--warnings-as-errors** (Treat warnings as errors)
-f (--option-file)

EDE

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

   *The Project Options dialog appears.*

2. Expand the **Assembler** entry and select **Miscellaneous**.

3. Add the option `-f` to the **Additional options** field.

In EDE you can save your options in a file and restore them to call the assembler with those options:

1. From the **Project** menu, select **Save Options**... or **Load Options**...

Be aware that when you specify the option `-f` in the **Additional options** field, the options are *added* to the assembler options you have set in the Project Options dialog. Only in extraordinary cases you may want to use them in combination.

**Command line syntax**

- `f file,...`

- `--option-file=file,...`

**Description**

Instead of typing all options on the command line, you can create an option file which contains all options and files you want to specify. With this option you specify the option file to the assembler.

Use an option file when the length of the command line would exceed the limits of the operating system, or just to store options and save typing.

You can specify the option `-f` multiple times.
Format of an option file

- Multiple command line arguments on one line in the option file are allowed.
- To include whitespace in an argument, surround the argument with single or double quotes.
- If you want to use single quotes as part of the argument, surround the argument by double quotes and vise versa:

  "This has a single quote ' embedded"

  'This has a double quote " embedded'

  'This has a double quote " and \ a single quote ‘”’ embedded"

Note that adjacent strings are concatenated.
- When a text line reaches its length limit, use a \ to continue the line. Whitespace between quotes is preserved.

  "This is a continuation \ line"

  -> "This is a continuation line"

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

Example

Suppose the file myoptions contains the following lines:

- -Ctc1lib
  test.src

Specify the option file to the assembler:

  astc -f myoptions
  astc --option-file=myoptions

This is equivalent to the following command line:

  astc -Ctc1lib test.src

Related information
--fpu-present

EDE

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

   *The Project Options dialog appears.*

2. Expand the Processor entry and select Processor Definition.

3. In the Target processor list select a (user defined TriCore) option.

4. Enable the option FPU present.

Command line syntax

   --fpu-present

Description

With this option you can use single precision floating-point instructions in the assembly code. When you select this option, the define __FPU__ is set to 1. By default the define __FPU__ is set to 0.

Example

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

   astc --fpu-present test.src

Related information

   Assembler option -C (Select CPU)
-g (--debug-info)

EDE

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

   The Project Options dialog appears.

2. Expand the Assembler entry and select Debug Information.

3. Enable one or more debug options.

You cannot use Assembly source line information and Pass HLL debug information simultaneously.

Command line syntax

- `g[flag]`

- `--debug-info[=flag]`

You can set the following flags:

- **a/A (+/-asm)**: Assembly source line information
- **h/H (+/-hll)**: Pass HLL debug information
- **l/L (+/-local)**: Local symbols debug information
- **s/S (+/-smart)**: Smart debug information

Default

- `gs`

Description

With this option you tell the assembler to generate debug information. If you do not use this option or if you specify `-g` without any flags, the default is `-gs`.

You cannot specify `-gah`. Either the assembler generates assembly source line information, or it passes HLL debug information.

When you specify `-gs`, the assembler selects which flags to use. If high level language information is available in the source file, the assembler passes this information (same as `-gAhL`). If not, the assembler generates assembly source line information and local symbols debug information (same as `-gaHl`).
With **-gAHLS** the assembler does not generate any debug information.

**Example**

To disable symbolic debug information, turn all flags off:

```
astc -gAHLS test.src
astc --debug-info=-asm,-hll,-local,-smart test.src
```

To enable smart debugging, enter:

```
astc -gs test.src
astc --debug-info=+smart test.src
```

**Related information**

-
-H (---include-file)

EDE
1. From the Project menu, select Project Options...
   
   The Project Options dialog appears.
2. Expand the Assembler entry and select Preprocessing.
3. Enter the name of the file in the Include this file before source field.

Command line syntax

   -Hfile,...
   
   --include-file=file,...

Description
With this option you include one extra file at the beginning of the assembly source file, before other includes. This is the same as specifying .INCLUDE 'file' at the beginning of your assembly sources.

Example

   astc -Hmyinc.inc test1.src
   astc --include-file=myinc.inc test1.src

   The file myinc.inc is included at the beginning of test1.src before it is assembled.

Related information

Assembler option -I (Add directory to include file search path)

Section 6.5, How the Assembler Searches Include Files, in Chapter Using the Assembler of the User's Manual.
-l (--include-directory)

EDE

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

The Directories dialog appears.

2. Enter one or more search paths in the Include Files Path field.

Command line syntax

-l path,...

--include-directory=path,...

Description

With this option you can specify the path where your include files are located. A relative path will be relative to the current directory.

The order in which the assembler searches for include files is:

1. The absolute pathname, if specified in the .INCLUDE directive. Or, if no path or a relative path is specified, the same directory as the source file.

2. The path that is specified with this option.

3. The path that is specified in the environment variable ASTCINC when the product was installed.

4. The default include directory relative to the installation directory.

Example

Suppose that your assembly source file test.src contains the following line:

```
.INCLUDE ‘myinc.inc’
```

You can call the assembler as follows:

```
astc -Ic:\proj\include test.src
astc --include-directory=c:\proj\include test.src
```
First the assembler looks in the directory where `test.src` is located for the file `myinc.inc`. If it does not find the file, it looks in the directory `c:\proj\include` for the file `myinc.inc` (this option).

**Related information**


Assembler option `-H` (Include file at the start of the input files)
-i (---symbol-scope)

EDE

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

   The Project Options dialog appears.

2. Expand the Assembler entry and select Miscellaneous.

3. Select the default label mode: Local or Global.

Command line syntax

- i{g | l}

   --symbol-scope={global | local}

Default

- il

Description

With this option you tell the assembler how to treat symbols that you have not specified explicitly as global or local.

By default the assembler treats all symbols as local symbols unless you have defined them explicitly as global.

Example

   astc -ig test.src
   astc --symbol-scope=global test.src

The assembler treats all symbols as global symbols unless they are defined as local symbols in the assembly source file.

Related information
--is-tricore2

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

*The Project Options dialog appears.*

2. Expand the Processor entry and select Processor Definition.

3. In the Target processor list select (user defined TriCore-2).

Command line syntax

```bash
--is-tricore2
```

Description

With this option you can use TriCore 2 instructions in the assembly code. When you select this option, the define __TC2__ is set to 1.

Example

To allow the use of TriCore 2 instructions in the assembly code, enter:

```bash
astc --is-tricore2 test.src
```

Related information

Assembler option -C (Select CPU)
-k (--keep-output-files)

EDE

EDE always removes the .o file when errors occur during assembly.

Command line syntax

- k

--keep-output-files

Description

If an error occurs during assembly, the resulting .o file may be incomplete or incorrect. With this option you keep the generated object file (.o) when an error occurs.

By default the assembler removes the generated object file (.o) when an error occurs. This is useful when you use the make utility \texttt{mktc}. If the erroneous files are not removed, the make utility may process corrupt files on a subsequent invocation.

Use this option when you still want to use the generated object. For example when you know that a particular error does not result in a corrupt object file.

Example

\texttt{astc -k test.src}

When an error occurs during assembly, the generated output file \texttt{test.o} will not be removed.

Related information

\textbf{Assembler option \texttt{--warnings-as-errors}} (Treat warnings as errors)
-L (--list-format)

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

The Project Options dialog appears.

2. Expand the Assembler entry and select List File.

3. Enable the options to include that information in the list file.

Command line syntax

-L flags

--list-format=flags

You can set the following flags:

0 same as -LCDEGILMNPQRVWXy
1 same as -Lcdegilmnpqrwxxy

c/C (+/-control) Assembler controls
d/D (+/-section) Section directives
e/E (+/-symbol) Symbol definition directives
g/G (+/-generic-expansion) Generic instruction expansion
i/I (+/-generic) Generic instructions
l/L (+/-line) #line source lines
m/M (+/-macro) Macro definitions
n/N (+/-empty-line) Empty source lines
p/P (+/-conditional) Conditional assembly
q/Q (+/-equate) Assembler .EQU and .SET directives
r/R (+/-relocations) Relocation characters ('r')
v/V (+/-equate-values) Assembler .EQU and .SET values
w/W (+/-wrap-lines) Wrapped source lines
x/X (+/-macro-expansion) Macro expansions
y/Y (+/-cycle-count) Cycle counts

Default

-LcDEGIlMnPqrVWXy
Description

With this option you specify which information you want to include in the list file. Use this option in combination with the option `-l (--list-file)`.

If you do not specify this option, the assembler uses the default: `-LcDEGiloMnPqrvWXY`.

With option `-tl`, the assembler also writes section information to the list file.

Example

```
astc -l -Ldm test.src
astc --list-file --list-format=+section,+macro test.src
```

The assembler generates a list file that includes all default information plus section directives and macro definitions.

Related information

- Assembler option `-l` (Generate list file)
- Assembler option `-tl` (Display section information in list file)
- Linker option `-M` (Generate map file)

-I (--list-file)

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

The Project Options dialog appears.
2. Expand the Assembler entry and select List File.
3. Enable the option Generate list file.

Command line syntax
-l
--list-file

Description
With this option you tell the assembler to generate a list file. A list file shows the generated object code and the relative addresses. Note that the assembler generates a relocatable object file with relative addresses.

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

astc -l test.src
astc --list-file test.src

Related information
Assembler option -L (List file formatting options)
Linker option -M (Generate map file)
Section 6.1, Assembler List File Format, in Chapter List File Formats.
-m (--preprocessor-type)

EDE

1. From the Project menu, select Project Options...
   
   The Project Options dialog appears.

2. Expand the Assembler entry and select Preprocessing.

3. Select No preprocessor or the TASKING preprocessor.

Command line syntax

- m{n | t}

--preprocessor-type={none | tasking}

Default

- mt

Description

With this option you select the preprocessor that the assembler will use. By default, the assembler uses the TASKING preprocessor.

When the assembly source file does not contain any preprocessor symbols, you can specify the assembler not to use a preprocessor.

Example

astc test.src
astc -mt test.src
astc --preprocessor=tasking test.src

These invocations have the same effect: the assembler preprocesses the file test.src with the TASKING preprocessor.

Related information
--mmu-present

EDE

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

   The Project Options dialog appears.

2. Expand the Processor entry and select Processor Definition.

3. In the Target processor list select (user defined TriCore–1 v1.3) or (user defined TriCore–2).

4. Enable the option MMU present.

   This option is only available (and relevant) for specific target processors. See option –C (--cpu) to select a target processor.

Command line syntax

   --mmu-present

Description

With this option you can use memory management instructions in the assembly code. When you select this option, the define __MMU__ is set to 1.

Example

To allow the use of memory management unit (MMU) instructions in the assembly code, enter:

   astc --mmu-present test.src

Related information

Assembler option –C (Select CPU)
--no-tasking-sfr

EDE

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

   *The Project Options dialog appears.*

2. Expand the **Assembler** entry and select **Preprocessing**.

3. Disable the option **Include '.def' file**.

**Command line syntax**

```
--no-tasking-sfr
```

**Description**

Normally, the assembler includes a special function register (SFR) file before compiling. The assembler automatically selects the SFR file belonging to the target you select on the **Processor options** page of the Processor options (assembler option -C).

With this option the assembler does not include the register file `regcpu.def` as based on the selected target processor.

Use this option if you want to use your own set of SFR '.def' files.

**Example**

```
astc -Ctc1lib --no-tasking-sfr test.src
```

The register file `regctc1lib.def` is not included, but the assembler allows the use of MMU instructions due to -C.

**Related information**

* Assembler option -C (Select CPU)
-O (--optimize)

EDE

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

   *The Project Options dialog appears.*

2. Expand the Assembler entry and select Optimization.

3. Enable or disable the optimization suboptions.

Command line syntax

-Oflags

--optimize=flags

You can set the following flags:

- **g/G** (+/--generics)  
  Allow generic instructions

- **s/S** (+/--instr--size)  
  Optimize instruction size

Default

-Ogs

Description

With this option you can control the level of optimization. If you do not use this option, -Ogs is the default.

Example

The following invocations are equivalent and result all in the default optimizations:

- astc test.src
- astc -Ogs test.src
- astc --optimize=+generics,+instr-size test.src

Related information

-o (--output)

EDE

Command line syntax

-ofile

--output=file

Description

With this option you can specify another filename for the output file of the assembler. Without this option, the basename of the assembly source file is used with extension .o.

EDE names the output file always after the assembly source file.

Example

astc -o relobj.o asm.src
astc --output=relobj.o asm.src

The assembler creates the file relobj.o for the assembled file asm.src.

Without the option -o, like EDE, the assembler uses the name of the input file and creates asm.o.

Related information

-
**--silicon-bug**

**EDE**

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

   The Project Options dialog appears.

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

3. Select the CPU functional problems you want to check for.

**Command line syntax**

   **--silicon-bug=arg,...**

You can give one or more of the following arguments:

- **all-tc112** All TriCore 1 v1.2 (TC112) checks:
  alias for cor1, cor4, cor6,
  cor7, cor10, cor13, cor15, cor16, cor17,
  cpu-tc013, cpu-tc030, cpu-tc031, cpu-tc034,
  cpu-tc048, cpu-tc050, cpu-tc060

- **all-tc113** All TriCore 1 v1.3 (TC113) checks:
  alias for pmu1, pmu3, cpu9,
  cpu11, cpu13, cpu14, cpu15, cpu16,
  dmu1,lfi2, lfi3, cpu-tc060

- **cor1** check for TC112 COR1
- **cor4** check for TC112 COR4
- **cor6** check for TC112 COR6
- **cor7** check for TC112 COR7
- **cor10** check for TC112 COR10
- **cor13** check for TC112 COR13
- **cor15** workaround for TC112 COR15
- **cor16** workaround for TC112 COR16
- **cor17** check for TC112 COR17
cpu-tc013  check for CPU TC013
cpu-tc030  check for CPU TC030
cpu-tc031  check for CPU TC031
cpu-tc034  check for CPU TC034
cpu-tc048  check for CPU TC048
cpu-tc050  check for CPU TC050
cpu-tc060  check for CPU TC060

cpu9  check for TC113 CPU9
cpu11  check for TC113 CPU11
cpu13  workaround for TC113 CPU13
cpu14  check for TC113 CPU14
cpu15  check for TC113 CPU15
cpu16  check for TC113 CPU16
dmu1  check for TC113 DMU1
lfi2  check for TC113 LFI2
lfi3  check for TC113 LFI3
pmu1  workaround for TC113 PMU1
pmu3  workaround for TC113 PMU3

Description

With this option you tell the assembler to check for some CPU functional problems. The assembles gives a warning when the specified problem is present.

Example

    astc --silicon-bug=cpu5,cpu9 test.src

The assembler checks for TC113 problems CPU5 and CPU9 and gives a warning when the problem is present.

Related information

See Chapter 9, CPU Functional Problems, for more information about the individual problems.
-t (--section-info)

EDE

1. From the Project menu, select Project Options...
   
   The Project Options dialog appears.

2. Expand the Assembler entry and select List File.

3. Enable the option Generate list file.

4. Enable the option Display section information.
   
   EDE always writes the section information to the list file.

Command line syntax

-flags

--section-info=flags

You can set the following flags:

- c/C (+/-console) Display section information on stdout.
- l/L (+/-list) Write section information to the list file.

Description

With this option you tell the assembler to display section information. For each section its memory space, size, total cycle counts and name is listed on stdout and/or in the list file.

The cycle count consists of two parts: the total accumulated count for the section and the total accumulated count for all repeated instructions. In the case of nested loops it is possible that the total supersedes the section total.

With -tl, the assembler writes the section information to the list file. You must specify this option in combination with the option -l (generate list file).
Example

```
astc -l -tcl test.src
astc -l --section-info=+console,+list test.src
```

The assembler generates a list file and writes the section information to this file. The section information is also displayed on stdout.

Section summary:

```
REL        4 .zbss_clr_test1
REL        46 .text_test1
REL        4 .zdata_rom_test1
```

Related information

- Assembler option -l (Generate list file)
-V (---version)

**EDE**

---

**Command line syntax**

- \(-V\)
- \(--version\)

**Description**

Display version information. The assembler ignores all other options or input files.

**Example**

\(\text{astc } -V\)
\(\text{astc } --version\)

The assembler does not assemble any files but displays the following version information:

```
TASKING TriCore VX-toolset Assembler   vxx.yrz Build nnn
Copyright years Altium BV              Serial# 00000000
```

**Related information**

---
-w (--no-warnings)

**EDE**

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

   *The Project Options dialog appears.*

2. Expand the **Assembler** entry and select **Warnings**.

3. Enable one of the options **Report all warnings**, **Suppress all warnings**, or **Suppress specific warnings**.

   *If you select **Suppress specific warnings**:*

4. Enter the numbers, separated by commas, of the warnings you want to suppress.

**Command line syntax**

- `--w[nr,...]`

- `--no-warnings[nr,...]`

**Description**

With this option you can suppresses all warning messages or specific warning messages.

- If you do not specify this option, all warnings are reported.
- If you specify this option but without numbers, all warnings are suppressed.
- If you specify this option with a number, only the specified warning is suppressed. You can specify the option `-w` multiple times.

**Example**

To suppress all warnings:

```
astc -w test.src
astc --no-warnings test.src
```

To suppress warnings 135 and 136:

```
astc -w135,136 test.src
astc --no-warnings=135,136 test.src
```
Related information

Assembler option `--warnings-as-errors` (Treat warnings as errors)
--warnings-as-errors

EDE

1. From the Project menu, select Project Options...
   The Project Options dialog appears.

2. Expand the Assembler entry and select Warnings.

3. Enable the option Treat warnings as errors.

Command line syntax

   --warnings-as-errors

Description

With this option you tell the assembler to treat warnings as errors.

Example

   astc --warnings-as-errors test.src

When a warning occurs, the assembler considers it as an error. No object file is generated, unless you specify option -k (--keep-output-files).

Related information

   Assembler option -w (suppress some or all warnings)
5.3 LINKER OPTIONS

Options in EDE versus options on the command line

Most command line options have an equivalent option in EDE but some options are only available on the command line. EDE invokes the linker via the control program. Therefore, it uses the syntax of the control program to pass options and files to the linker.

See section 5.4, Control Program Options.

If necessary, you can specify a command line option in EDE.

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

   The Project Options dialog appears.

2. Expand the Linker entry and select Miscellaneous.

3. Enter one or more command line options in the Additional options field.

   Because EDE uses the control program, EDE automatically precedes the option with -Wl to pass the option via the control program directly to the linker.

   For example, if you enter the option -DDEMO in the Additional options field, EDE generates the option -Wl-DDEMO for the control program which tells the control program to pass the option -DDEMO to the linker.

   Be aware that some options are not useful in EDE or just will not have any effect. For example, the option -k keeps files after an error occurred. When you specify this option in EDE, it will have no effect because EDE always removes the output file after an error had occurred.

Short and long option names

Options can have both short and long names. Short option names always begin with a single minus (-) character, long option names always begin with two minus (-- )characters. You can abbreviate long option names as long as it forms a unique name. You can mix short and long option names on the command line.
Options can have flags or suboptions. To switch a flag 'on', use a lowercase letter or a +longflag. To switch a flag off, use an uppercase letter or a -longflag. Separate longflags with commas. The following two invocations are equivalent:

```
ltc -mfkl test.o
ltc --map-file-format=+files,+link,+locate test.o
```

When you do not specify an option, a default value may become active.
-? (---help)

EDE
-

Command line syntax
-?
--help

Description
Displays an overview of all command line options.

Example
The following invocations all display a list of the available command line options:

ltc -?
ltc --help
ltc

Related information
–c (--chip-output)

EDE

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

   The Project Options dialog appears.

2. Expand the Linker entry and select Output Format.

3. Enable one or more output formats.

4. Enable the option Create file for each memory chip.

   For some output formats you can specify a number of suboptions.

Command line syntax

   –c[basename]:format[|addr_size],...

   --chip-output=[basename]:format[|addr_size],...

You can specify the following formats:

   IHEX     Intel Hex
   SREC     Motorola S–records

The addr_size specifies the size of the addresses in bytes (record length). For Intel Hex you can use the values: 1, 2 and 4 (default). For Motorola S you can specify: 2 (S1 records), 3 (S2 records, default) or 4 bytes (S3 records).

Description

With this option you specify the Intel Hex or Motorola S–record output format for loading into a PROM–programmer. The linker generates a file for each ROM memory defined in the LSL file, where sections are located:

   memory memname
   {   type=rom;
      }

The name of the file is the name of the EDE project or, on the command line, the name of the memory space that was emitted with extension .hex or .sre. Optionally you can specify a basename which prepends the generated file name.
Examples

To generate Intel Hex output files for each defined memory, enter the following on the command line:

```
ltc -cmyfile:IHEX test1.o
ltc --chip-output=myfile:IHEX test1.o
```

This generates the file `myfile_memspace.hex`.

Related information

- Linker option `-o` (output file),
- Section 7.2, *Motorola S-Record Format*,
- Section 7.3, *Intel Hex Record Format*, in Chapter *Object File Formats*.
--case-insensitive

EDE
1. From the Project menu, select Project Options...
   The Project Options dialog appears.
2. Expand the Linker entry and select Miscellaneous.
3. Disable the option Link case sensitive.

Command line syntax
--case-insensitive

Description
With this option you tell the linker not to distinguish between upper and lower case characters. By default the linker considers upper and lower case characters as different characters.

Disabling the option Link case sensitive in EDE is the same as specifying the option --case-insensitive on the command line.

Assembly source files that are generated by the compiler must always be assembled and thus linked case sensitive. When you have written your own assembly code and specified to assemble it case insensitive, you must also link the .o file case insensitive.

Example
To link case insensitive:

    ltc --case-insensitive test.o

The linker considers upper and lower case characters as being the same. So, for example, the label LabelName is considered the same label as labelname.

Related information
Assembler option -c (Assemble case insensitive)
-D (--define)

EDE

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

   *The Project Options dialog box appears.*

2. Expand the **Linker** entry and select **Miscellaneous**.

3. Add the option `-D` to the **Additional options** field.

**Command line syntax**

- `Dmacro_name[=macro_definition]`

- `--define=macro_name[=macro_definition]`

**Description**

With this option you can define a macro and specify it to the linker preprocessor. If you only specify a macro name (no macro definition), the macro expands as '1'.

You can specify as many macros as you like: you can use the option `-D` multiple times. If the command line exceeds the limit of the operating system, you can define the macros in an **option file** which you then must specify to the linker with the option `-ffile`.

Define `macro` to the preprocessor, as in `#define`. Any number of symbols can be defined. The definition can be tested by the preprocessor with `#if`, `#ifdef` and `#ifndef`, for conditional locating.

**Example**

To define the RESET vector, interrupt table start address and trap table start address which is used in the linker script file `tclvl_3.lsl`, enter:

```
ltc test.o -otest.elf -dtclvl_3.lsl -DRESET=0xa0000000 -DINTTAB=0xa000f0000 --define=TRAPTAB=0xa000f2000
```

**Related information**

- Linker option `-f` (Name of invocation file)
-d (--lsl-file)

EDE

1. From the Project menu, select Project Options...
   
   The Project Options dialog appears.

2. Expand the Linker entry and select Script File.

3. Select Use project specific memory and section LSL file and specify a name.

Command line syntax

- d/file
  --lsl-file=file

Description

With this option you specify a linker script file to the linker. If you do not specify this option, the linker uses a default script file. You can specify the existing file target.lsl or the name of a manually created linker script file. You can use this option multiple times. The linker processes the LSL files in the order in which they appear on the command line.

The linker script file contains vital information about the core for the locating phase of the linker. A linker script file is coded in LSL and contains the following types of information:

- the architecture definition describes the core's hardware architecture.
- the memory definition describes the physical memory in the system.
- the section layout definition describes how to locate sections in memory.

Example

To read linker script file information from file tc1v1_3.lsl:

```
ltc -dtc1v1_3.lsl test.o
ltc --lsl-file=tc1v1_3.lsl test.o
```
Related information

Linker option --lsl-check (Check LSL file(s) and exit)

Section 7.7, Controlling the Linker with a Script in Chapter Linker of the User’s Manual.
--diag

EDE

1. In the Help menu, enable the option Show Help on Tool Errors.

2. In the Build tab of the Output window, double-click on an error or warning message.

A description of the selected message appears.

Command line syntax

`--diag=[format;][all | number,number]...`

Optionally, you can use one of the following display formats (`format`):

- text: The default is plain text
- html: Display explanation in HTML format
- rtf: Display explanation in RTF format

Description

With this option you can ask for an extended description of error messages in the format you choose. The output is directed to `stdout` (normally your screen) and in the format you specify.

To create a file with the descriptions, you must redirect the output.

With the suboption `all`, the descriptions of all error messages are given. If you want the description of one or more selected error messages, you can specify the error message numbers, separated by commas.

With this option the linker does not link any files.

Example

To display an explanation of message number 106, enter:

`ltc --diag=106`

This results in the following message and explanation:

`E106: unresolved external: message`

The linker could not resolve all external symbols. This is an error when the incremental linking option is disabled. The `<message>` indicates the symbol that is unresolved.
To write an explanation of all errors and warnings in HTML format to file `errors.html`, enter:

```
ltc --diag=html:all > errors.html
```

**Related information**

-e (--extern)

EDE

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

   The Project Options dialog appears.

2. Expand the Linker entry and select Miscellaneous.

3. Add the option -e in the Additional options field.

Command line syntax

- e symbol
  --extern=symbol

Description

With this option you force the linker to consider the given symbol as an undefined reference. The linker tries to resolve this symbol by extracting the corresponding symbol definition from a library. If the symbol is defined in an object file, this option has no influence on the link process.

Suppose you are linking from a library. Because the library itself already has been compiled and assembled, the linker does not find any unresolved symbols. Hence, the linker will not extract any module from the library. When you force a symbol to be undefined, the linker extracts those modules that contain the symbol.

This option is, for example, useful if the startup code is part of a library. Because your own application does not refer to the startup code, you can force the startup code to be extracted by specifying the symbol `_START` as an unresolved external.

Example:

Consider the following invocation:

```
ltc mylib.a
```

Nothing is linked and no output file will be produced, because there are no unresolved symbols when the linker searches through `mylib.a`.

```
ltc -e _START mylib.a
ltc --extern=_START mylib.a
```
In this case the linker searches for the symbol \texttt{START} in the library and (if found) extracts the object that contains \texttt{START}, the startup code. If this module contains new unresolved symbols, the linker looks again in \texttt{mylib.a}. This process repeats until no new unresolved symbols are found.

**Related information**

Section 7.4.1, *Specifying Libraries to the Linker*, in Chapter *Using the Linker* of the *User’s Manual*. 

--error-file

EDE

Command line syntax

--error-file [=file]

Description

With this option the linker redirects error messages to a file.

If you do not specify a filename, the error file is ltc.elk.

Example

ltc --error-file=my.elk test.o

The linker writes error messages to the file my.elk instead of stderr.

Related information

Linker option --warnings-as-errors (Treat warnings as errors)
-f (--option-file)

EDE
1. From the Project menu, select Project Options...
   The Project Options dialog appears.
2. Expand the Linker entry and select Miscellaneous.
3. Add the option -f to the Additional options field.

In EDE you can save your options in a file and restore them to call the linker with those options:
1. From the Project menu, select Save Options... or Load Options...

Be aware that when you specify the option -f in the Additional options field, the options are added to the linker options you have set in the Project Options dialog. Only in extraordinary cases you may want to use them in combination.

Command line syntax

- f file,...
--option-file=file,...

Description

Instead of typing all options on the command line, you can create an option file which contains all options and files you want to specify. With this option you specify the option file to the linker.

Use an option file when the length of the command line would exceed the limits of the operating system, or just to store options and save typing.

You can specify the option -f multiple times.

Format of an option file

- Multiple command line arguments on one line in the option file are allowed.
- To include whitespace in an argument, surround the argument with single or double quotes.
- If you want to use single quotes as part of the argument, surround the argument by double quotes and vice versa:
“This has a single quote ’ embedded”

‘This has a double quote “ embedded’

‘This has a double quote ” and \ a single quote ’’ embedded”

Note that adjacent strings are concatenated.

- When a text line reaches its length limit, use a ‘\’ to continue the line. Whitespace between quotes is preserved.

  "This is a continuation \ line”

  -> "This is a continuation line”

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

Example

Suppose the file myoptions contains the following lines:

- Mmymap (generate a map file)
test.o (input file)
-Lc:\mylibs (additional search path for system libraries)

Specify the option file to the linker:

  ltc -f myoptions
  ltc --option-file=myoptions

This is equivalent to the following command line:

  ltc -Mmymap test.o -Lc:\mylibs

Related information
--first-library-first

EDE
-

Command line syntax

--first-library-first

Description
When the linker processes a library it searches for symbols that are referenced by the objects and libraries processed so far. If the library contains a definition for an unresolved reference the linker extracts the object that contains the definition from the library.

By default the linker processes object files and libraries in the order in which they appear on the command line. If you specify the option --first-library-first the linker always tries to take the symbol definition from the library that appears first on the command line before scanning subsequent libraries.

This is for example useful when you are working with a newer version of a library that partially overlaps the older version. Because they do not contain exactly the same functions, you have to link them both. However, when a function is present in both libraries, you may want the linker to extract the most recent function.

With this option, you tell the linker to scan the libraries from left to right, and extract the symbol from the first library where the linker finds it.

Example:

ltc --first-library-first a.a test.o b.a

If the file test.o calls a function which is both present in a.a and b.a, normally the function in b.a would be extracted. With this option the linker first tries to extract the symbol from the first library a.a.

Related information

Linker option --no-rescan (Do not rescan libraries)
-l (--include-directory)

EDE

Command line syntax

- filepath...

--include-directory=filepath,...

Description

With this option you can specify the path where your LSL include files are located. A relative path will be relative to the current directory.

The order in which the linker searches for LSL include files is:

1. The path name in the LSL file and the directory where the LSL files are located (only for #include files that are enclosed in """)
2. The path that is specified with this option.
3. The default ..\include.lsl directory relative to the installation directory.

Example

Suppose that the LSL file lslfile.lsl contains the following lines:

```
#include "mypart.lsl"
```

You can call the linker as follows:

```
ltc -Imyinclude -dlslfile.lsl test.o

ltc --include-directory=myinclude
   --lsl-file=lslfile.lsl test.o
```

First the linker looks in the directory where lslfile.lsl is located for the file mypart.lsl. If it does not find the file, it looks in myinclude subdirectory relative to the current directory for the file mypart.lsl (this option). Finally it looks in the directory ..\include.lsl.

Related information

- Linker option -d (Linker script file)
-i

(--user-provided-initialization-code)

EDE
1. From the Project menu, select Project Options...
   The Project Options dialog appears.
2. Expand the Linker entry and select Miscellaneous.
3. Disable the option Use standard copy-table for initialization.

Command line syntax
- i
   --user-provided-initialization-code

Description
It is possible to use your own initialization code, for example, to save ROM space. With this option you tell the linker not to generate a copy table for initialize/clear sections. Use linker labels in your source code to access the positions of the sections when located.

If the linker detects references to the TASKING initialization code, an error is emitted: it is either the TASKING initialization routine or your own, not both.

Note that the options --no-rom-copy and --non-romable, may vary independently. The 'copytable-compression' optimization is automatically disabled when you enable this option.

Example:
To link with your own startup code:
   ltc -i test.o
   ltc --user-provided-initialization-code test.o

Related information
-k (--keep-output-files)

EDE

EDE always removes the output files when errors occurred.

Command line syntax

- \( k \)
  --keep-output-files

Description

If an error occurs during linking, the resulting output file may be incomplete or incorrect. With this option you keep the generated output files when an error occurs.

By default the linker removes the generated output files when an error occurs. This is useful when you use the make utility `mk tc`. If the erroneous files are not removed, the make utility may process corrupt files on a subsequent invocation.

Use this option when you still want to use the generated file. For example when you know that the error(s) do not result in a corrupt output file, or when you want to inspect the output file, or send it to Altium support.

Example

```bash
ltc -k test.o
ltc --keep-output-files test.o
```

When an error occurs during linking, the generated output file `test.elf` will not be removed.

Related information
-L (\textit{--library-directory} /  
\textit{--ignore-default-library-path})

\textbf{EDE}

1. From the \textbf{Project} menu, select \textbf{Directories}...

   \textit{The Directories dialog appears.}

2. Add a pathname in the \textbf{Library Files Path} field.

3. In the \textbf{Library Files Path} field, add the pathnames of the directories where the linker should look for library files.

   \textit{If you enter multiple paths, separate them with a semicolon (;).}

\textbf{Command line syntax}

- \texttt{-L path,...}
- \texttt{--library-directory=path,...}
- \texttt{-L}
- \texttt{--ignore-default-library-path}

\textbf{Description}

With this option you can specify the path(s) where your system libraries, specified with the \texttt{-l} option, are located. If you want to specify multiple paths, use the option \texttt{-L} for each separate path.

The default path is \texttt{$(PRODDIR)/lib$}.

If you specify only \texttt{-L} (without a pathname) or the long option \texttt{--ignore-default-library-path}, the linker will \textit{not} search the default path and also not in the paths specified in the environment variable \texttt{LIBTC1V1_2}, \texttt{LIBTC1V1_3} or \texttt{LIBTC2}. So, the linker ignores steps 2, 3 and 4 as listed below.

The priority order in which the linker searches for system libraries specified with the \texttt{-l} option is:

1. The path that is specified with the \texttt{-L} option.

2. The path that is specified in the environment variable \texttt{LIBTC1V1_2},  
   \texttt{LIBTC1V1_3} or \texttt{LIBTC2} when the product was installed.
3. The default directory \$(PRODDIR)\lib.

4. The processor specific directory, for example \$(PRODDIR)\lib\tcl.

**Example**

Suppose you call the linker as follows:

```bash
ltc test.o -Lc:\mylibs -lc
```

First the linker looks in the directory `c:\mylibs` for library `libc.a` (this option).

If it does not find the requested libraries, it looks in the directory that is set with the environment variable `LIBTCIV2`, `LIBTCIV1_2`, `LIBTCIV1_3` or `LIBTC2`.

Then the linker looks in the default directory \$(PRODDIR)\lib for libraries.

**Related information**

- Linker option `-l` (Link system library)


  Section 1.3.2, *Configuring the Command Line Environment*, in Chapter *Software Installation* of the *User’s Manual*. 
-l (--library)

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

   The Project Options dialog appears.

2. Expand the Linker entry and select Libraries.

3. Enable the option Link default C libraries.

Command line syntax

-lname
--library=name

Description

With this option you tell the linker to search also in system library
libname.a, where name is a string. The linker first searches for system
libraries in any directories specified with -Lpath, then in the directories
specified with the environment variable LIBTC1V1_2, LIBTC1V1_3 or
LIBTC2, unless you used the option -L without a directory.

If you use the libc.a library, you must always link the libfp.a library
as well. Remember that the order of the specified libraries is important!

Example

To search in the system library libfp.a (floating-point library):

   ltc test.o mylib.a -lfp

The linker links the file test.o and first looks in mylib.a (in the current
directory only), then in the system library libfp.a to resolve unresolved
symbols.

Related information

Linker option -L (Additional search path for system libraries)

Section 7.4.1, Specifying Libraries to the Linker, in Chapter Using the Linker
of the User’s Manual.
--link-only

EDE

Command line syntax

--link-only

Description

With this option you suppress the locating phase. The linker stops after linking. The linker complains if any unresolved references are left.

Example:

ltc --link-only hello.o

The linker checks for unresolved symbols and creates the file taks1.out.

Related information

Control program option --cl (Stop after linking)
--lsl-check

EDE

Command line syntax

```
--lsl-check
```

Description
With this option the linker just checks the syntax of the LSL file(s) and exits. No linking or locating is performed.

Example:

To check the LSL file(s) and exit:

```
ltc --lsl-check --lsl-file=mylslfile.lsl
```

Related information

- Linker option **-d** (Linker script file)
  
  Linker option **--lsl-dump** (Dump LSL info)

Chapter 8, *Linker Script Language*. 
--lsl-dump

**EDE**

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

   *The Project Options dialog appears.*

2. Expand the **Linker** entry and select **Script File**.

3. Enable the option **Dump processor and memory info from LSL file**.

**Command line syntax**

```
--lsl-dump=[file]
```

**Description**

With this option you tell the linker to dump the LSL part of the map file in a separate file, independent of the --M (generate map file) option. If you do not specify a filename, the file `ltc.ldf` is used.

**Example**

```
ltc --lsl-dump=mydump.ldf test.o
```

The linker dumps the processor and memory info from the LSL file in the file `mydump.ldf`.

**Related information**

- Linker option `-m` (Map file formatting options)
-M (--map-file)

**EDE**

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

   *The Project Options dialog appears.*

2. Expand the **Linker** entry and select **Map File**.

3. Enable the option **Generate a map file (.map)**.

**Command line syntax**

```
-M/[file]
--map-file=[/file]
```

**Description**

With this option you tell the linker to generate a linker map file. If you do not specify a filename, the linker uses the same basename as the output file with the extension `.map`.

A linker map file is a text file that shows how the linker has mapped the sections and symbols from the various object files (.o) to the linked object file. A locate part shows the absolute position of each section. External symbols are listed per space with their absolute address, both sorted on symbol and sorted on address.

With the option `-m` (map file formatting) you can specify which parts you want to place in the map file.

**Example**

To generate a map file (`test.map`):

```
ltc -Mtest.map test.o
ltc --map-file=test.map test.o
```

The control program by default tells the linker to generate a map file.

**Related information**

- [Linker option `-m` (Map file formatting options)]


-m (--map-file-format)

EDE
1. From the Project menu, select Project Options...
   
   The Project Options dialog appears.
2. Expand the Linker entry and select Map File.
3. Enable the options to include that information in the map file.

Command line syntax

-mflags

--map-file-format=flags

You can set the following flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>same as -mcfklMoQrSU (link info)</td>
</tr>
<tr>
<td>1</td>
<td>same as -mCfKlmQrSU (locate info)</td>
</tr>
<tr>
<td>2</td>
<td>same as -mcfkldmQsru (all info)</td>
</tr>
<tr>
<td>c/C</td>
<td>(+/-callgraph)</td>
</tr>
<tr>
<td>f/F</td>
<td>(+/-files)</td>
</tr>
<tr>
<td>k/K</td>
<td>(+/-link)</td>
</tr>
<tr>
<td>I/L</td>
<td>(+/-locate)</td>
</tr>
<tr>
<td>m/M</td>
<td>(+/-memory)</td>
</tr>
<tr>
<td>o/O</td>
<td>(+/-overlay)</td>
</tr>
<tr>
<td>q/Q</td>
<td>(+/-statics)</td>
</tr>
<tr>
<td>r/R</td>
<td>(+/-crossref)</td>
</tr>
<tr>
<td>s/S</td>
<td>(+/-lsl)</td>
</tr>
<tr>
<td>u/U</td>
<td>(+/-rules)</td>
</tr>
</tbody>
</table>

Default

-mCfklMORSU

Description

With this option you specify which information you want to include in the map file. Use this option in combination with the option -M (--map-file). If you do not specify this option, the linker uses the default: -mCfklMORSU
Example

```
ltc -Mtest.map -mFr test.o
ltc --map-file=test.map
    --map-file-format=+crossref,-files test.o
```

The linker generates the map file `test.map` that includes all default information plus the cross reference part, but not the processed files part.

Related information

Linker option `-M` (Generate map file)

--misra-c-report

EDE

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

   The Project Options dialog appears.

2. Expand the C Compiler entry and select MISRA C.

3. Select a MISRA C configuration.

4. Enable the option Produce a MISRA C report.

Command line syntax

   --misra-c-report [=file]

Description

With this option you tell the linker to create a MISRA C Quality Assurance report. This report lists the various modules in the project with the respective MISRA C settings at the time of compilation. If you do not specify a filename, the file name.mcr is used.

Example

   ltc --misra-c-report test.o

   The linker creates a MISRA C report file test.mcr.

Related information

   Compiler option --misracc
--munch

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

   *The Project Options dialog appears.*

2. Expand the **Linker** entry and select **Miscellaneous**.

3. Add the option **--munch** to the **Additional options** field.

Command line syntax

   **--munch**

Description

With this option you tell the linker to activate the muncher in the pre-locate phase.

The data sections are initialized when the application is downloaded. The data sections are not re-initialized when the application is restarted.

Example

   ltc --munch test.o

   The linker activates the muncher in the pre-locate phase while linking the file *test.o*.

Related information
**–N (---no-rom-copy)**

**EDE**

1. From the **Project** menu, select **Project Options**...
   
   *The Project Options dialog appears.*

2. Expand the **Linker** entry and select **Miscellaneous**.

3. Add the option **–N** to the **Additional options** field.

**Command line syntax**

```plaintext
--N
--no-rom-copy
```

**Description**

With this option the linker will not generate a ROM copy for data sections. A copy table is generated and contains entries to clear BSS sections. However, no entries to copy data sections from ROM to RAM are placed in the copy table.

The data sections are initialized when the application is downloaded. The data sections are not re-initialized when the application is restarted.

**Example**

```plaintext
ltc -N test.o
ltc --no-rom-copy test.o
```

The linker does not generate a copy table.

**Related information**

---
**--no-rescan**

**EDE**

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

   *The Project Options dialog appears.*

2. Expand the **Linker** entry and select **Libraries**.

3. Disable the option **Rescan libraries to solve unresolved externals**.

**Command line syntax**

   ```
   --no-rescan
   ```

**Description**

When the linker processes a library, it searches for symbol definitions that are referenced by the objects and libraries processed so far. If the library contains a definition for an unresolved reference, the linker extracts the object that contains the definition from the library. The linker processes object files and libraries in the order in which they appear on the command line.

When all objects and libraries are processed the linker checks if there are unresolved symbols left. If so, the default behavior of the linker is to rescan all libraries in the order given on the command line. The linker stops rescanning the libraries when all symbols are resolved, or when the linker could not resolve any symbol(s) during the rescan of all libraries. Notice that resolving one symbol may introduce new unresolved symbols.

With this option, you tell the linker to scan the object files and libraries only once. When the linker has not resolved all symbols after the first scan, it reports which symbols are still unresolved. This option is useful if you are building your own libraries. The libraries are most efficiently organized if the linker needs only one pass to resolve all symbols.

**Example:**

To scan the libraries only once:

   ```
   ltc --no-rescan test.o a.a b.a
   ```

The linker resolves all unresolved symbols while scanning the object files and libraries and reports all remaining unresolved symbols after this scan.
Related information

Linker option **--first-library-first** (Scan libraries in the specified order)
--non-romable

EDE

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

   The Project Options dialog appears.

2. Expand the Linker entry and select Miscellaneous.

3. Add the option to the Additional options field.

Command line syntax

   --non-romable

Description

With this option you tell the linker that the application is not romable. The linker will locate all ROM sections in RAM. A copy table is generated and is located in RAM. When the application is started, that data and BSS sections are re-initialized.

Example

   ltc --non-romable test.o

   The linker locates all ROM sections in RAM.

Related information

   --
-O (--optimize)

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

   The Project Options dialog appears.

2. Expand the Linker entry and select Optimization.

3. Enable or disable the optimization suboptions.

Command line syntax

-Oflags

--optimize=flags

You can set the following flags:

c/C (+/-delete-unreferenced-code-sections)
   Delete unreferenced code sections from the output file
   (no effect on sources compiled with debug information)

 d/D (+/-delete-unreferenced-data-sections)
   Delete unreferenced data sections from the output file
   (no effect on sources compiled with debug information)

 l/L (+/-first-fit-decreasing)
   Use a 'first fit decreasing' algorithm to locate
   unrestricted sections in memory.

 s/S (+/-delete-unreferenced-symbols)
   Delete unreferenced symbols from the output file

 t/T (+/-copytable-compression)
   Locate (unrestricted) sections in such a way that
   the size of the copy table is as small as possible.

 x/X (+/-delete-duplicate-code)
   Delete duplicate code from the output file

 y/Y (+/-delete-duplicate-data)
   Delete duplicate constant data from the output file
Use the following options for predefined sets of flags:

- **-O0 (--optimize=0)** No optimization.
  Alias for: **-OCDLSTXY**

- **-O1 (--optimize=1)** Normal optimization (default).
  Alias for: **-OCDLStXY**

- **-O2 (--optimize=2)** All optimizations.
  Alias for: **-OcdlStxy**

**Default**

- **-O1**

**Description**

With this option you can control the level of optimization. If you do not use this option, **-OCDLStXY** (**-O1**) is the default.

**Example**

The following invocations are equivalent and result all in the default optimizations.

```plaintext
ltc test.o
ltc -0 test.o
ltc -O1 test.o
ltc -OCDLStXY test.o

ltc --optimize test.o
ltc --optimize=1 test.o
ltc --optimize=-delete-unreferenced-code-sections,
     -delete-unreferenced-code-sections,
     -first-fit-decreasing,+copytable-compression,
     -delete-duplicate-code,-delete-duplicate-data test.o
```

**Related information**

Section 7.2.3, *Linker Optimizations*, in Chapter *Using the Linker* of the *User’s Manual*. 
-o (--output-file)

**EDE**

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

   The Project Options dialog appears.

2. Expand the Linker entry and select Output Format.

3. Enable one or more output formats.

   For some output formats you can specify a number of suboptions.

   EDE and the control program name the output file always after the first input file with the extension .elf.

**Command line syntax**

```
-o [filename][|format|addr_size][space_name]|...
```

```
--output=[filename][|format|addr_size][space_name]|...
```

You can specify the following formats:

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELF</td>
<td>ELF/DWARF</td>
</tr>
<tr>
<td>IEEE</td>
<td>IEEE-695</td>
</tr>
<tr>
<td>IHEX</td>
<td>Intel Hex</td>
</tr>
<tr>
<td>SREC</td>
<td>Motorola S−records</td>
</tr>
</tbody>
</table>

**Description**

By default, the linker generates an output file in ELF/DWARF format, with the name task1.elf.

With this option you can specify an alternative filename, and an alternative output format. The default output format is the format of the first input file.

You can use the -o option multiple times. This useful to generate multiple output formats or to link multiple address spaces. With the first occurrence of the -o option you must specify the filename without extension. If you do not specify a filename, or you do not specify the -o option at all, the linker uses the default basename taskn.


**IHEX and SREC formats**

If you specify the Intel Hex format or the Motorola S–records format, you can use the argument `addr_size` to specify the size of addresses in bytes (record length). For Intel Hex you can use the values: **1**, **2**, and **4** (default). For Motorola S–records you can specify: **2** (S1 records), **3** (S2 records, default) or **4** bytes (S3 records).

With the argument `space_name` you can specify the name of the address space. The name of the output file will be `filename` with the extension `.hex` or `.sre` and contains the specified space. (Remember to use the `-o` option multiple times to link multiple address spaces.)

If you do not specify `space_name`, the default address space is emitted. In this case the name of the output file will be `filename spacename` with the extension `.hex` or `.sre`.

Use option `-c` (**--chip-output**) to create Intel Hex or Motorola S–record output files for each chip (suitable for loading into a PROM–programmer).

**Example**

To create the output file `myfile.hex` of the address space named `linear`:

```
ltc test.o -omyfile.hex:IHEX:2,linear
ltc test.o --output-file=myfile.hex:IHEX:2,linear
```

**Related information**

*Linker option `-c` (Generate an output file for each chip)*

Section 7.1, *ELF/DWARF Object Format*, in Chapter *Object File Formats*. 
-r (--incremental)

EDE

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

   The Project Options dialog appears.

2. Expand the Linker entry and select Miscellaneous.

3. Add the option -r in the Additional options field.

Command line syntax

- r
  --incremental

Description

Normally the linker links and locates the specified object files. With this option you tell the linker only to link the specified files. The linker creates a linker output file .out. You then can link this file again with other object files until you have reached the final linker output file that is ready for locating.

In the last pass, you call the linker without this option with the final linker output file .out. The linker will now locate the file.

Example

In this example, the files test1.o, test2.o and test3.o are incrementally linked:

- ltc -r test1.o -test.out (test1.o and test2.o are linked)
- ltc -r test3.o test.out (test3.o is linked)
- ltc test.out (test.out is located)

Related information

-S (--strip-debug)

EDE

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

   The Project Options dialog appears.

2. Expand the Linker entry and select Miscellaneous.

3. Enable the options to include that information in the map file.

4. Disable the option Include symbolic debug information.

Command line syntax

- S

   --strip-debug

Description

With this option you specify not to include symbolic debug information in the resulting output file.

Example

   ltc -S test.o -otest.elf
   ltc --strip-debug test.o --output=test.elf

   The linker generates the object file test.elf without symbolic debug information.
-V (--version)

EDE
-

Command line syntax
- V

Description
Display version information. The linker ignores all other options or input files.

Example
ltc -V
ltc --version

The linker does not link any files but displays the following version information:

    TASKING TriCore VX-toolset object linker     vx.yrz Build 000
    Copyright years Altium BV                      Serial# 00000000

Related information
-v (--verbose)

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

   The Project Options dialog appears.

2. Expand the Linker entry and select Miscellaneous.

3. Enable the option Print the name of each file as it is processed.

Command line syntax
-v

Description
With this option you put the linker in verbose mode. The linker prints the filenames and the link passes while it processes the files. It also shows which objects are extracted from libraries. With this option you can monitor the current status of the linker.

Example

   ltc test.o -lc -lfp -lrt -v

The linker links the file test.o and displays the steps it performs.

   ltc I405: activating pre link phase
   ltc I406: activating link phase
   ltc I401: start linking task "task1"
   ltc I415: reading file "/test.o"
   ltc I413: start processing library "/tc/ctc/lib/tcl/libc.a"
   ltc I416: reading file "cstart.o" from library "libc.a"
   ...
   ltc I414: start rescanning libraries
   ...
   ltc I407: activating post link phase
   ltc I408: activating pre locate phase
   ltc I409: activating locate phase
   ...
   ltc I418: binding locator symbols
   ltc I411: activating post locate phase
   ltc I410: activating file producing phase
   ltc I401: start producing files for task "task1"

Related information
-w (--no-warnings)

EDE

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

   The Project Options dialog appears.

2. Expand the Linker entry and select Warnings.

3. Enable one of the options Report all warnings, Suppress all warnings, or Suppress specific warnings.

   If you select Suppress specific warnings:

4. Enter the numbers, separated by commas, of the warnings you want to suppress.

Command line syntax

   -w[nr[,nr]...]
   --no-warnings=[nr[,nr]...]

Description

With this option you can suppresses all warning messages or specific warning messages.

- If you do not specify this option, all warnings are reported.
- If you specify this option but without numbers, all warnings are suppressed.
- If you specify this option with a number, only the specified warnings are suppressed. Separate multiple warnings by commas.

Example:

To suppress all warnings:

   ltc -w test.o
   ltc --no-warnings test.o

To suppress warnings 113 and 114:

   ltc -w113,114 test.o
   ltc --no-warnings=113,114 test.o
Related information

Linker option `--warnings-as-errors` (Treat warnings as errors)
EDE

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

   The Project Options dialog appears.

2. Expand the Linker entry and select Warnings.

3. Enable the option Treat warnings as errors.

Command line syntax

   --warnings-as-errors

Description

With this option you tell the linker to treat warnings as errors.

When the linker detects an error, it tries to continue the link process and reports other errors and warnings. However, the linker will exit with an exit status not equal zero (!= 0) and will not produce any output files.

Example

   ltc --warnings-as-errors test.o

   When a warning occurs, the linker considers it as an error.

Related information

   Linker option -w (Suppress some or all warnings)
5.4 CONTROL PROGRAM OPTIONS

The control program **cctc** facilitates the invocation of the various components of the TriCore toolchain from a single command line. The control program is a command line tool so there are no equivalent options in EDE.

For the linker options in EDE, EDE invokes the linker via the control program. Therefore, it uses the syntax of the control program to pass options and files to the linker. See section 5.3, *Linker Options*, for an overview of the EDE linker options and the corresponding command line linker options.

Some options are interpreted by the control program itself; other options are passed to those programs in the toolchain that accept the option.

**Recognized input files**

The control program recognizes the following input files:

- Files with a `.cc`, `.cxx` or `.cpp` suffix are interpreted as C++ source programs and are passed to the C++ compiler.
- Files with a `.c` suffix are interpreted as C source programs and are passed to the compiler.
- Files with a `.asm` suffix are interpreted as hand-written assembly source files which have to be passed to the assembler.
- Files with a `.src` suffix are interpreted as compiled assembly source files. They are directly passed to the assembler.
- Files with a `.a` or `.elb` suffix are interpreted as library files and are passed to the linker.
- Files with a `.o` suffix are interpreted as object files and are passed to the linker.
- Files with a `.out` suffix are interpreted as linked object files and are passed to the locating phase of the linker. The linker accepts only one `.out` file in the invocation.
- An argument with a `.lsl` suffix is interpreted as a linker script file and is passed to the linker.

Normally, the control program tries to compile, assemble, link and locate all source files to absolute object files. There are however, options to suppress the assembler, link or locate stage.
-? (–help)

Command line syntax

-? [options]

--help [options]

Description
Displays an overview of all command line options. When you specify the suboption options, you receive extended information.

Example
The following invocations all display a list of the available command line options:

cctc -?
cctc

Related information
-A (--language)

Command line syntax

-A[flags]

--language[=flags]

You can set the following flags:

p/P  (+/-comments)  Allow C++ style comments in ISO C90
x/X  (+/-strings)    Relaxed const check for string literals

Default

-Apx

Description

With this option you control the language extensions the compiler can accept. Default the C compiler allows all language extensions.

-A (--language) is the equivalent of -APX and disables all language extensions.

With -Ap you tell the compiler to allow C++ style comments (//) in ISO C90 mode (option -c90). In ISO C99 mode this style of comments is always accepted.

With -Ax you tell the compiler not to check for assignments of a constant string to a non-constant string pointer. With this option the following example does not produce a warning:

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

Example

```c
crc -APx test.c
crc --language=-comments,+strings test.c
```

The control program calls the compiler in such a way that it accepts assignments of a constant string to a non-constant string pointer but ignores C++ style comments.
Related information

Compiler option -A (Control language extensions)
---address-size

Command line syntax
---address-size=addr_size

Description
If you specify IHEX or SREC with the control option --format, you can additionally specify the record length and the address space to be emitted in the output files.

With this option you can specify the size of addresses in bytes (record length). For Intel Hex you can use the values: 1, 2, and 4 (default). For Motorola S-records you can specify: 2 (S1 records), 3 (S2 records, default) or 4 bytes (S3 records).

If you do not specify addr_size, the default address size is generated.

Example
To create the SREC file test.s with S1 records, type:

cctc --format=SREC --address-size=2

Related information
Control program option --format (Set linker output format)
Control program option --space (Set linker output space name)

Linker option -o (Specify an output object file)
**–C (--cpu)**

**Command line syntax**

–C<cpu>

**Description**

With this option you define the target processor for which you create your application. Default the control program generates an object file for the TC11IB.

Based on the specified target processor it is automatically detected whether a MMU or FPU–unit is present and whether the architecture is a TriCore2. This means you do not have to specify the assembler options --mmu–present, --fpu–present and --is-tricore2 explicitly when one of the supported derivatives is selected.

Based on the specified target processor the control program always includes the correct register files, unless you specify control program option --no-tasking-sfr.

**Example**

To generate the file test.elf for the TC11IB processor:

```bash
cctc -Ctc11lib test.c
cctc --cpu=tc11lib test.c
```

**Related information**

Compiler option –C (Use SFR definitions for CPU)

Assembler option –C (Select CPU)

Section 5.4, *Calling the Compiler*, in Chapter *Using the Compiler* of the *User’s Manual*. 
--case-insensitive

Command line syntax

--case-insensitive

Description

With this option you tell the control program not to distinguish between upper and lower case characters. By default upper and lower case characters are considered as different characters. Note that in assembly instruction mnemonics, register names, directives and controls are always treated case insensitive.

Assembly source files that are generated by the compiler must always be assembled and linked case sensitive. When you are writing your own assembly code, you may want to specify the case insensitive mode.

Example

To create the file test.elf with case insensitive assembling and linking:

    cctc -c test.c
    cctc --case-insensitive test.c

The assembler and linker now consider upper and lower case characters as being the same. So, for example, the label LabelName is the same label as labelname.

Related information

Assembler option --case-sensitive (Assemble case insensitive)
Linker option --case-sensitive (Link case insensitive)
-cc/-cs/-co/-cl

Command line syntax

-cc
  --create=c

-cs
  --create=assembly

-co
  --create=object

-cl
  --create=relocatable

Description

Normally the control program generates an absolute object file of the specified output format from the file you supplied as input.

With this option you tell the control program to stop after a certain number of phases.

-cc  Stop after C++ files are compiled to intermediate C files (.ic)
-cs  Stop after C files are compiled to assembly (.src)
-co  Stop after the files are assembled to object files (.obj)
-cl  Stop after the files are linked to a linker object file (.eln)

To generate the object file test.o:

  cctc -c test.c
  cctc --create=object test.c

The control program stops after the file is assembled. It does not link nor locate the generated output.

Related information
---check

Command line syntax

   --check

Description

With this option you can check the source code for syntax errors, without generating code. This saves time in developing your application.

The compiler/assembler reports any warnings and/or errors.

Example

To check for syntax errors, without generating code:

   cctc --check test.c

Related information

 Compiler option **--check** (Check syntax)
 Assembler option **--check** (Check syntax)
**-D (--define)**

**Command line syntax**

```plaintext
-D macro_name[=macro_definition]
--define=macro_name[=macro_definition]
```

**Description**

With this option you can define a macro and specify it to the preprocessor. If you only specify a macro name (no macro definition), the macro expands as '1'.

You can specify as many macros as you like. On the command line, use the option `-D` multiple times. If the command line exceeds the length limit of the operating system, you can define the macros in an option file which you then must specify to the control program with the option `-f file`.

Defining macros with this option (instead of in the C source) is, for example, useful to compile or assemble conditional source as shown in the example below.

The control program passes the option `-D (--define)` to the compiler and the assembler.

**Example**

Consider the following C program with conditional code to compile a demo program and a real program:

```plaintext
void main( void )
{
    #if DEMO == 1
        demo_func(); /* compile for the demo program */
    #else
        real_func(); /* compile for the real program */
    #endif
}
```

You can now use a macro definition to set the DEMO flag. With the control program this looks as follows:

```plaintext
cctc -DDEMO test.c
cctc -DDEMO=1 test.c
```
cctc --define=DEMO test.c

Note that all four invocations have the same effect.

The next example shows how to define a macro with arguments. Note that the macro name and definition are placed between double quotes because otherwise the spaces would indicate a new option.

```
cctc -D"MAX(A,B)=((A) > (B) ? (A) : (B))"
cctc --define="MAX(A,B)=((A) > (B) ? (A) : (B))"
```

**Related information**

- Control program option `-U` (Undefine preprocessor macro)
- Control program option `-f` (Read options from file)
-d (--lsl-file)

Command line syntax

-d/file

--lsl-file=file

Description

A linker script file contains vital information about the core for the locating phase of the linker. A linker script file is coded in LSL and contains the following types of information:

• the architecture and derivative definition describe the core’s hardware architecture and its internal memory.
• the board specification describes the physical memory available in the system.
• the section layout definition describes how to locate sections in memory.

With this option you specify a linker script file via the control program to the linker. If you do not specify this option, the linker does not use a script file. You can specify the existing file tctarget.lsl or the name of a manually written linker script file. You can use this option multiple times. The linker processes the LSL files in the order in which they appear on the command line.

Example

To read linker script file information from file mylslfile.lsl:

cctc -dmylslfile.lsl test.obj

cctc --lsl-file=mylslfile.lsl test.obj

Related information

Section 7.7, Controlling the Linker with a Script, in the User’s Manual.
--diag

Command line syntax

--diag=[format;][all | nr,...]

Description

With this option you can ask for an extended description of error messages in the format you choose. The output is directed to stdout (normally your screen) and in the format you specify. You can specify the following formats: html, rtf or text (default). To create a file with the descriptions, you must redirect the output.

With the suboption all, the descriptions of all error messages are given. If you want the description of one or more selected error messages, you can specify the error message numbers, separated by commas.

With this option the control program does not process any files.

Example

To display an explanation of message number 103, enter:

   cctc --diag=103

This results in message 103 with explanation.

To write an explanation of all errors and warnings in HTML format to file ccerrors.html, enter:

   cctc --diag=html;all > ccerrors.html

Related information
–E (–preprocess)

Command line syntax

–E[flags]

–preprocess=[flags]

You can set the following flags:

- c/C  (+/-comments)  Keep comments
- p/P  (+/-noline)    Strip #line source position info

Description

With this option you tell the control program to preprocess the C source.

The compiler sends the preprocessed file to stdout. To capture the information in a file, specify an output file with the option –o.

With –Ec you tell the preprocessor to keep the comments from the C source file in the preprocessed output.

With –Ep you tell the preprocessor to strip the #line source position information (lines starting with #line). These lines are normally processed by the assembler and not needed in the preprocessed output. When you leave these lines out, the output is more orderly to read.

Example

```
cctc -EcP test.c -o test.pre
```
```
cctc --preprocess +comments,-noline test.c
    --output=test.pre
```

The compiler preprocesses the file test.c and sends the output to the file test.pre. Comments are included but the line source position information is not stripped from the output file.

Related information
--error-file

Command line syntax

    --error-file [=file]

Description

    With this option the control program redirects error messages to a file.

    If you do not specify a filename, the error file will be named after the input file with extension .err.

Example

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

        cctc --error-file=errors.err test.c

Related information

    Control program option --warnings-as-errors (Warnings as errors)
--exceptions

Command line syntax

--exceptions

Description

With this option you enable support for exception handling in the C++ compiler.

Example

To enable exception handling, enter:

    cctc --exceptions test.cc

Related information
\textbf{\texttt{-F (--no-double)}}

\textbf{Command line syntax}

\begin{verbatim}
-F
--no-double
\end{verbatim}

\textbf{Description}

With this option you tell the control program to treat variables of the type double as float. Because the float type takes less space, execution speed increases and code size decreases, both at the cost of less precision.

\textbf{Example}

\begin{verbatim}
cctc -F test.c
cctc --no-double test.c
\end{verbatim}

The file \texttt{test.c} is processed where variables of the type double are treated as float in the compilation phase.

\textbf{Related information}
-f (--option-file)

Command line syntax

-f file

--option-file=file

Description

Instead of typing all options on the command line, you can create an option file which contains all options and file you want to specify. With this option you specify the option file to the control program.

Use an option file when the length of the command line would exceed the limits of the operating system, or just to store options and save typing.

You can specify the option -f multiple times.

Format of an option file

- Multiple command line arguments on one line in the option file are allowed.
- To include whitespace in an argument, surround the argument with single or double quotes.
- If you want to use single quotes as part of the argument, surround the argument by double quotes and vice versa:

  "This has a single quote ' embedded"

  'This has a double quote ” embedded’

  ‘This has a double quote ” and 
  a single quote ‘”’ embedded”

Note that adjacent strings are concatenated.

- When a text line reaches its length limit, use a \ to continue the line. Whitespace between quotes is preserved.

  "This is a continuation 
  line”

  \ -> "This is a continuation line”

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

Suppose the file `myoptions` contains the following lines:

- `g`
- `k`
- `test.c`

Specify the option file to the control program:

- `ctc -f myoptions`
- `ctc --option-file=myoptions`

This is equivalent to the following command line:

- `ctc -g -k test.c`

Related information

-
---force-c

Command line syntax
---force-c

Description
With this option you tell the control program to treat all .cc files as C files instead of C++ files. This means that the control program does not call the C++ compiler and forces the linker to link C libraries.

Example
```plaintext
cctc --force-c test.cc
```

The C++ file test.cc is considered to be a normal C file.

Related information
Control program option --force-c++ (Force C++ compilation and linking)
--force-c++

Command line syntax

--force-c++

Description

With this option you tell the control program to treat all .c files as C++ files instead of C files. This means that the control program calls the C++ compiler prior to the C compiler and forces the linker to link C++ libraries.

Example

```bash
cctc --force-c++ test.c
```

The file `test.c` is considered to be a C++ file.

Related information

Control program option `--force-c` (Treat C++ files as C files)
--force-munch

Command line syntax

  --force-munch

Description

With this option you force the control program to activate the muncher in the pre-locate phase.

Example

To force the muncher phase in the pre-locate phase, type:

  cctc --force-munch test.cc

Related information

  —
--force-prelink

Command line syntax
   --force-prelink

Description
   With this option you force the control program to invoke the C++ pre-linker.

Example
   cctc --force-prelink test.cc

   The control program always invokes the C++ pre-linker when generating test.elf.

Related information
--format

Command line syntax

```
--format=format
```

You can specify the following formats:

- **ELF**    ELF/DWARF
- **IEEE**   IEEE–695
- **IHEX**   Intel Hex
- **SREC**   Motorola S–records

**Description**

With this option you specify the output format for the resulting (absolute) object file. The default output format is ELF/DWARF, which can directly be used by the CrossView Pro debugger.

If you choose IHEX or SREC, you can additionally specify the address size of the chosen format (option **--address-size**) and the address space to be emitted (option **--space**).

**Example**

To generate an IEEE output file:

```
cctc --format=IEEE test1.c test2.c --output=test.abs
```

**Related information**

- Control program option **--address-size** (For linker IHEX,SREC files)
- Control program option **--space** (Set linker output space name)

Linker option **–o** (output file)
Linker option **–c** (generate hex file)

Section 7.1, *ELF/DWARF Object Format*, in Chapter *Object File Formats*.
--fp-trap

Command line syntax

--fp-trap

Description

Default the control program uses the non-trapping floating-point library (libfp.a). With this option you tell the control program to use the trapping floating-point library (libfpt.a).

If you use the trapping floating-point library, exceptional floating-point cases are intercepted and can be handled separately by an application defined exception handler. Using this library decreases the execution speed of your application.

Example

    cctc --fp-trap test.c

    Link the trapping floating-point library when generating the object file test.elf.

Related information
**--fpu-present**

**Command line syntax**

    --fpu-present

**Description**

With this option the compiler can generate single precision floating-point instructions in the assembly file. When you select this option, the macro \_FPU\_ is defined in the C source file.

If you choose a valid target processor (command line option -C (--cpu)), this option is automatically set, based on the chosen target processor.

**Example**

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

    ctc --fpu-present test.c

**Related information**

- Compiler option -C (Use SFR definitions for CPU)
-g (\texttt{--debug-info})

Command line syntax

\begin{verbatim}
-g
--debug-info
\end{verbatim}

Description

With this option you tell the control program to include debug information in the generated object file.

Example

\begin{verbatim}
cctc -g test.c
cctc --debug-info test.c
\end{verbatim}

The control program includes symbolic debug information in the generated object file \texttt{test.elf}.

Related information
-I (--include-directory)

Command line syntax

-\texttt{I}path

\texttt{--include-directory=path}

Description

With this option you can specify the path where your include files are located. A relative path will be relative to the current directory.

Example

Suppose that the C source file \texttt{test.c} contains the following lines:

\begin{verbatim}
#include <stdio.h>
#include "myinc.h"
\end{verbatim}

You can call the control program as follows:

\begin{verbatim}
cctc -Imyinclude test.c
cctc --include-directory=myinclude
\end{verbatim}

First the compiler looks for the file \texttt{stdio.h} in the directory \texttt{myinclude} relative to the current directory. If it was not found, the compiler searches in the environment variable and then in the default include directory.

The compiler now looks for the file \texttt{myinc.h} in the directory where \texttt{test.c} is located. If the file is not there the compiler searches in the directory \texttt{myinclude}. If it was still not found, the compiler searches in the environment variable and then in the default include directory.

Related information

- Compiler option \texttt{-I} (Add directory to include file search path)
- Compiler option \texttt{-H} (Include file at the start of a compilation)
--instantiate

Command line syntax

---instantiate=mode

Description

Normally, when a file is compiled, no template entities are instantiated (except those assigned to the file by automatic instantiation). The overall instantiation mode can, however, be changed with this option. You can specify the following modes:

- **none**  Do not automatically create instantiations of any template entities. This is the default. It is also the usually appropriate mode when automatic instantiation is done.

- **used**  Instantiate those template entities that were used in the compilation. This will include all static data members for which there are template definitions.

- **all**   Instantiate all template entities declared or referenced in the compilation unit. For each fully instantiated template class, all of its member functions and static data members will be instantiated whether or not they were used. Non-member template functions will be instantiated even if the only reference was a declaration.

- **local** Similar to **--instantiate=used** except that the functions are given internal linkage. This is intended to provide a very simple mechanism for those getting started with templates. The compiler will instantiate the functions that are used in each compilation unit as local functions, and the program will link and run correctly (barring problems due to multiple copies of local static variables). However, one may end up with many copies of the instantiated functions, so this is not suitable for production use.

You cannot use **--instantiate=local** in conjunction with automatic template instantiation.

Example

To specify instantiation mode **used**, type

```
cctc --instantiate=used test.cc
```
Related information
**--instantiation-dir**

**Command line syntax**

```
--instantiation-dir=dir
```

**Description**

With this option the C++ compiler generates additional files for template instantiations in the specified directory.

If you do not specify this option, files are created in the current directory.

If you specify the control program option

```
--no-one-instantiation-per-object
```

this option remains without effect.

**Example**

To specify the directory for instantiation files, type

```
cctc --instantiation-dir=instant test.cc
```

**Related information**

Control program option

```
--no-one-instantiation-per-object
```
--instantiation-file

Command line syntax

    --instantiation-file=file

Description

With this option the C++ compiler generates a list of all generated template instantiation files and writes it to the specified file. You can use this file for example to use as an option file for the archiver artc.

Example

To create a file with a list of all generated instantiation files, type

    cctc --instantiation-file=instlist.ii test.cc

Related information

-
--is-tricore2

Command line syntax

--is-tricore2

Description

With this option you allow the control program to use TriCore 2 instructions in the generated output file.

If you choose a valid target processor (command line option --cpu), this option is automatically set, based on the chosen target processor.

Example

To allow the use of TriCore 2 instructions in the assembly code, enter:

cctc --is-tricore2 test.c

Related information

Compiler option --C (Use SFR definitions for CPU)
--iso

Command line syntax

    --iso={90|99}

Description

With this option you specify to the control program against which ISO standard it should check your C source. C90 is also referred to as the "ANSI C standard". C99 refers to the newer ISO/IEC 9899:1999 (E) standard and is the default.

⚠️ Independent of the chosen ISO standard, the control program always links libraries with C99 support.

Example

To compile the file test.c conform the ISO C90 standard:

    cctc --iso=90 test.c

Related information

💡 Compiler option -c (ISO C standard)
-k (--keep-output-files)

Command line syntax

- k

--keep-output-files

Description

If an error occurs during the compilation, assembling or linking process, the resulting output file may be incomplete or incorrect. With this option you keep the generated output files when an error occurs.

By default the control program removes generated output files when an error occurs. This is useful when you use the make utility. If the erroneous files are not removed, the make utility may process corrupt files on a subsequent invocation.

Use this option when you still want to use the generated files. For example when you know that a particular error does not result in a corrupt file, or when you want to inspect the output file, or send it to Altium support.

Example

    cctc -k test.c
    cctc --keep-output-files test.c

When an error occurs during compiling, assembling or linking, the erroneous generated output files will not be removed.

Related information
**-L (--library-directory / --ignore-default-library-path)**

**Command line syntax**

- `L path`
  --library-directory=path

- `-L`
  --ignore-default-library-path

**Description**

With this option you can specify the path(s) where your system libraries, specified with the `-l` option, are located. If you want to specify multiple paths, use the option `-L` for each separate path.

The default path is `$(PRODDIR)/lib/tcl`.

If you specify only `-L` (without a pathname) or the long option `--ignore-default-library-path`, the linker will not search the default path and also not in the paths specified in the environment variables LIBTC1V1_2, LIBTC1V1_3 or LIBTC2. So, the linker ignores steps 2 and 3 as listed below.

The priority order in which the linker searches for system libraries specified with the `-l` option is:

1. The path that is specified with the `-L` option.
2. The path that is specified in the environment variables LIBTC1V1_2, LIBTC1V1_3 or LIBTC2 when the product was installed.
3. The default directory `$(PRODDIR)/lib/tcl` or `$(PRODDIR)/lib/tc2`.

**Example**

Suppose you call the control program as follows:

```
cctc test.c -Lc:\mylibs -lcs
```

First the linker looks in the directory `c:\mylibs` for library `libc.a` (this option).
If it does not find the requested libraries, it looks in the directory that is set with the environment variables LIBTC1V1_2, LIBTC1V1_3 or LIBTC2.

Then the linker looks in the default directory $(PRODDIR)/lib/tc1 or $(PRODDIR)/lib/tc2 for libraries.

**Related information**

- Linker option `-l` (Search also in system library `libname`)
-l (---library)

Command line syntax

-lname

--library=name

Description

With this option you tell the linker via the control program to search also in system library libname.a, where name is a string. The linker first searches for system libraries in any directories specified with -Lpath, then in the directories specified with the environment variables LIBTC1V1_2, LIBTC1V1_3 or LIBTC2, unless you used the option -L without a directory.

Example

To search in the system library libfp.a (floating-point library):

    cctc test.obj mylib.a -lf
    cctc test.obj mylib.a --library=fp

The linker links the file test.obj and first looks in mylib.a (in the current directory only), then in the system library libfp.a to resolve unresolved symbols.

Related information

Control program option -L (Add library directory)

--list-object-files

Command line syntax

    --list-object-files

Description

With this option the list of object files that are handled by the prelinker, is
displayed at stdout. The list is shown when it is changed by the
prelinker.

Example

To show the list of object files handled by the prelinker, enter:

    cctc --list-object-files test.cc

Related information


--mmu-present

Command line syntax

--mmu-present

Description

With this option you can use memory management instructions in the assembly code. When you select this option, the define __MMU__ is set to 1.

Example

To allow the use of memory management unit (MMU) instructions in the assembly code, enter:

    cctc --mmu-present test.c

Related information

Assembler option -C (Select CPU)
-n (--dry-run)

Command line syntax

- `n`
  --dry-run

Description

With this option you put the control program `verbose` mode. The control program prints the invocations of the tools it would use to process the files.

Example

To see how the control program will invoke the tools it needs to process the file `test.c`:

```
cctc -n test.c
```
```
cctc --dry-run test.c
```

The control program only displays the invocations of the tools it would use to create the final object file but does not actually perform the steps.

Related information

Control program option -v (Verbose output)
--no-auto-instantiation

Command line syntax

--no-auto-instantiation

Description
Default, the c++ compiler automatically instantiates templates. With this option automatic instantiation of templates is disabled.

Example
To disable automatic instantiation, type

cctc --no-auto-instantiation test.cc

Related information
--no-default-libraries

Command line syntax

    --no-default-libraries

Description

Default the control program specifies the standard C libraries and run-time library to the linker.

With this option you tell the control program not to specify the standard C libraries and run-time library to the linker.

In this case you must specify the libraries you want to link to the linker with the option `-library_name`. The control program recognizes the option `-l` as an option for the linker.

Example

    cctc --no-default-libraries test.c

The control program does not specify any libraries to the linker. In normal cases this would result in unresoved externals.

To specify your own libraries (`libmy.a`) and avoid unresolved externals:

    cctc --no-default-libraries -lmy test.c

Related information

  ![Linker option -l](image) (Search also in system library `libx.a`)
--no-map-file

Command line syntax

--no-map-file

Description

By default the control program generates a linker map file (.map).

A linker map file is a text file that shows how the linker has mapped the sections and symbols from the various object files (.obj) to the linked object file. A locate part shows the absolute position of each section. External symbols are listed per space with their absolute address, both sorted on symbol and sorted on address.

With this option you prevent the generation of a map file.

Example

To prevent the generation of the linker map file test.map:

    cctc --no-map-file test.c

Related information

警惕 Linker option -M (Generate map file)
--no-one-instantiation-per-object

Command line syntax

    --no-one-instantiation-per-object

Description

With this option, the C++ compiler writes template instantiations into a single object file. If you do not specify this option, the C++ compiler creates multiple files. In that case you can specify a directory for those files with the control program option \(--\)instantiation--dir.\)

Example

To create a file with a list of all generated instantiation files, type

    cctc --no-one-instantiation-per-object test.cc

Related information

Control program option \(--\)instantiation--dir\)
--no-tasking-sfr

Command line syntax

--no-tasking-sfr

Description

Normally, the compiler and assembler include a special function register (SFR) file before compiling. This file is automatically selected based on the target you select on the Processor definition page of the Processor options (compiler option --C).

With this option the compiler and assembler do not automatically include a register file.

Use this option if you want to use your own set of SFR files.

Example

    cctc -Ctcllib --no-tasking-sfr test.c

The register file regtcllib.sfr is not included.

Related information

- Compiler option --C (Use SFR definitions for CPU)
-o (--output)

Command line syntax

-ofile

--output=file

Description

By default, the control program generates a file with the same basename as the first specified input file. With this option you specify another name for the resulting absolute object file.

Example

```c
cctc test.c prog.c
```

The control program generates an ELF/DWARF object file (default) with the name test.elf.

To generate the file result.elf:

```c
cctc -oresult.elf test.c prog.c
cctc --output=result.elf test.c prog.c
```

Related information
--prelink-copy-if-non-local

Command line syntax

--prelink-copy-if-non-local

Description

If a file must be recompiled and it is not in the current directory, with this option the C/C++ prelinker copies the prelink file (.ii) to the current directory and rewrites that .ii file so it can find its associated .cc file. As a result, the .cc file is recompiled in the current directory.

With this option you prevent that previously compiled files are overwritten during recompilation.

Example

To copy all files for recompilation to the current directory:

    cctc --prelink-copy-if-non-local test.cc

Related information
--prelink-local-only

Command line syntax

   --prelink-local-only

Description

   With this option the C++ prelinker ignores all files that are outside the current directory.

Example

   To prelink only files in the current directory:

   cctc --prelink-local-only test.cc

Related information


--prelink-remove-instantiation-flags

Command line syntax
--prelink-remove-instantiation-flags

Description
With this option the C++ prelinker removes all instantiation flags from the generated object files.

Example
To remove instantiation flags from the generated object files:

cctc --prelink-remove-instantiation-flags test.cc

Related information
--show-c++-warnings

Command line syntax

   --show-c++-warnings

Description

The C++ compiler may generate a compiled C++ file (.ic) that causes warnings during compilation or assembling. With this option you tell the control program to show these warnings. Default C++ warnings are suppressed.

Example

   cctc --show-c++-warnings test.cc

The control program calls the C++ compiler which generates the C file (test.ic). If this file causes warnings during compilation or assembling, these warnings are shown.

Related information

   -
--silicon-bug

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

The Project Options dialog appears.

2. Expand the Processor entry and select Bypasses.

3. Select the bypasses you want to enable.

Command line syntax
--silicon-bug=arg,...

You can give one or more of the following arguments (corresponding pragmas are listed to be complete):

all-tc112 All TriCore 1 v1.2 (TC112) workarounds
all-tc113 All TriCore 1 v1.3 (TC113) workarounds

cor1 workaround for TC112 COR1 (#pragma TC112_COR1 )
cor4 workaround for TC112 COR4 (#pragma TC112_COR4 )
cor7 workaround for TC112 COR7 (#pragma TC112_COR7 )
cor10 workaround for TC112 COR10 (#pragma TC112_COR10)
cor13 workaround for TC112 COR13 (#pragma TC112_COR13)
cor14 workaround for TC112 COR14 (#pragma TC112_COR14)
cor16 workaround for TC112 COR16 (#pragma TC112_COR16)
cor17 workaround for TC112 COR17 (#pragma TC112_COR17)
cpu-tc013 workaround for CPU_TC.013 (#pragma CPU_TC013)
cpu-tc030 workaround for CPU_TC.030 (#pragma CPU_TC030)
cpu-tc031 workaround for CPU_TC.031 (#pragma CPU_TC031)
cpu-tc034 workaround for CPU_TC.034 (#pragma CPU_TC034)
cpu-tc048 workaround for CPU_TC.048 (#pragma CPU_TC048)
cpu-tc050 workaround for CPU_TC.050 (#pragma CPU_TC050)
cpu-tc060 workaround for CPU_TC.060 (#pragma CPU_TC060)
Description
With this option the control program tells the compiler/assembler/linker to use software workarounds for some CPU functional problems.

Example

cctc --silicon-bug=cpu5,cpu9 test.c

The compiler uses workarounds for TC113 problems CPU5 and CPU9.

Related information
See Chapter 9, CPU Functional Problems, for more information about the individual problems and workarounds.
--space

Command line syntax

    --space=space_name

Description

If you specify IHEX or SREC with the control option --format, you can additionally specify the record length and the address space to be emitted in the output files.

With this option you can specify which address space must be emitted. With the argument space_name you can specify the name of the address space. The name of the output file will be filename with the extension .hex or .s.

If you do not specify space_name, the default address space is emitted. In this case the name of the output file will be filename spacename with the extension .hex or .s.

Example

To create the IHEX file test.hex, type:

    cctc --format=IHEX --space=far test.c

If the specified memory space does not exist, the control program emits the default space name and reflects this in the output file name.

Related information

- Control program option --format (Set linker output format)
- Linker option -o (Specify an output object file)
--static

Command line syntax

--static

Description

This option is directly passed to the compiler.

With this option, the compiler treats external definitions at file scope (except for main) as if they were declared static. As a result, unused functions will be eliminated, and the alias checking algorithm assumes that objects with static storage cannot be referenced from functions outside the current module.

This option only makes sense when you specify all modules of an application on the command line.

Example

```
cctc --static module1.c module2.c module3.c
```

Related information
-t (---keep-temporary-files)

Command line syntax

- \( -t \)

---keep-temporary-files

Description

By default, the control program removes intermediate files like the .src file (result of the compiler phase) and the .eln file (result of the linking phase).

With this option you tell the control program to keep temporary files it generates during the creation of the absolute object file.

Example

To keep all temporary files:

```c
  cctc -t test.c
  cctc --keep-temporary-files test.c
```

The control program keeps all intermediate files it generates while creating the absolute object file test.elf.

Related information

-
-U (--undefine)

Command line syntax

- \texttt{-Umacro\_name}
- \texttt{--undefine=macro\_name}

Description

With this option you can undefine an earlier defined macro as with \texttt{#undef}.

This option is for example useful to undefine predefined macros.

However, the following predefined ISO C standard macros cannot be undefined:

\begin{itemize}
\item \_\_FILE\_ current source filename
\item \_\_LINE\_ current source line number (int type)
\item \_\_TIME\_ hh:mm:ss
\item \_\_DATE\_ mmm dd yyyy
\item \_\_STDC\_ level of ANSI standard
\end{itemize}

The control program passes the option \texttt{-U (--undefine)} to the compiler.

Example

To undefine the predefined macro \_\_TASKING\_: 

\begin{verbatim}
cctc -U\_\_TASKING\_ test.c
cctc --undefine=\_\_TASKING\_ test.c
\end{verbatim}

Related information

\begin{itemize}
\item Control Program option \texttt{-D} (Define preprocessor macro)
\end{itemize}
-V (--version)

Command line syntax

- V
--version

Description

Display version information. The control program ignores all other options or input files.

Example

```bash
cctc -V
```
```bash
cctc --version
```

The control program does not call any tools but displays the following version information:

```
TASKING TriCore VX-toolset control program vx.yrz Build nnn
Copyright 2003-year Altium BV Serial# 00000000
```

Related information
-v (---verbose)

Command line syntax

- v
  --verbose

Description

With this option you put the control program in *verbose* mode. With the option `-v` the control program performs its tasks while it prints the steps it performs to `stdout`.

Example

```
cctc -v test.c
cctc --verbose test.c
```

The control program processes the file `test.c` and displays the invocations of the tools it uses to create the final object file.

Related information

Control program option `-n` (Verbose output and suppress execution)
-Wtool (--pass)

Command line syntax

- Wc\texttt{option} \texttt{--pass-c++=option} \quad \text{Pass option directly to the C++ compiler}
- W\texttt{option} \texttt{--pass-c=option} \quad \text{Pass option directly to the C compiler}
- Wa\texttt{option} \texttt{--pass-assembler=option} \quad \text{Pass option directly to the assembler}
- Wp\texttt{loption} \texttt{--pass-prelinker=option} \quad \text{Pass option directly to the C++ prelinker}
- Wl\texttt{option} \texttt{--pass-linker=option} \quad \text{Pass option directly to the linker}

Description

With this option you tell the control program to call a tool with the specified option. The control program does not use the option itself, but specifies it directly to the tool which the control program calls.

Example

```bash
 cctc -Wl-r test.c
```

The control program does not use the option -r but calls the linker with the option -r (lcc -r).

Related information
**-w (--no-warnings)**

**Command line syntax**

```
-w[nr]  --no-warnings[=nr]
```

**Description**

With this option suppresses all warning messages or a specific warning. If you do not specify this option, all warnings are reported.

**Example**

To suppress all warnings:

```
cctc -w test.c
```

```
cctc --no-warnings test.c
```

To suppress warnings 100:

```
cctc -w100 test.c
```

```
cctc --no-warnings=100 test.c
```

**Related information**

Control program option **--warnings-as-errors** (Warnings as errors)
--warnings-as-errors

Command line syntax

   --warnings-as-errors

Description

With this option you tell the control program to treat warnings as errors.

Example

   cctc --warnings-as-errors test.c

When a warning occurs, the control program considers it as an error.

Related information

   Control program option -w (Suppress all warnings)
5.5 MAKE UTILITY OPTIONS

When you build a project in EDE, EDE generates a makefile and uses the graphical make utility `wmk` to build all your files. However, you can also use the make utility `mktc` from the command line to build your project.

The invocation syntax is:

```
mktc [option...] [target...] [macro=def]
```

This section describes all options for the make utility. The make utility is a command line tool so there are no equivalent options in EDE.
Defining Macros

Command line syntax

\textit{macro} = \textit{definition}

Description

With this argument you can define a macro and specify it to the make utility.

A macro definition remains in existence during the execution of the makefile, even when the makefile recursively calls the make utility again. In the recursive call, the macro acts as an environment variable. This means that it is overruled by definitions in the recursive call. Use the option \texttt{-e} to prevent this.

You can specify as many macros as you like. If the command line exceeds the limit of the operating system, you can define the macros in an \textit{option file} which you then must specify to the compiler with the option \texttt{-m file}.

Defining macros on the command line is, for example, useful in combination with conditional processing as shown in the example below.

Example

Consider the following makefile with conditional rules to build a demo program and a real program:

```
ifdef DEMO       # the value of DEMO is of no importance
    real.out : demo.o
    ltc demo.o main.o -lc -lfp -lrt
else
    real.out : real.o
    ltc real.o main.o -lc -lfp -lrt
endif

real.elf : real.out
    ltc -FELF -oreal.elf real.elf
```

You can now use a macro definition to set the DEMO flag:

```
mktc real.elf DEMO=1
```

In both cases the absolute object file \texttt{real.elf} is created but depending on the DEMO flag it is linked with \texttt{demo.o} or with \texttt{real.o}. 

Related information

- Make utility option \(-\text{e}\) (Environment variables override macro definitions)
- Make utility option \(-\text{m}\) (Name of invocation file)
-?

Command line syntax
-?

Description
Displays an overview of all command line options.

Example
The following invocation displays a list of the available command line options:

```
mkto -?
```

Related information
-a

Command line syntax

-a

Description

Normally the make utility rebuilds only those files that are out of date. With this option you tell the make utility to rebuild all files, without checking whether they are out of date.

Example

mktc -a

Rebuilds all your files, regardless of whether they are out of date or not.

Related information
\textbf{\texttt{-c}}

\textbf{Command line syntax}

\texttt{-c}

\textbf{Description}

EDE uses this option for the graphical version of make when you create sub-projects. In this case make calls another instance of make for the sub-project. With the option \texttt{-c}, the make utility runs as a child process of the current make.

The option \texttt{-c} overrules the option \texttt{-err}.

\textbf{Example}

The following command runs the make utility as a child process:

\texttt{mkte -c}

\textbf{Related information}

Make utility option \texttt{-err} (Redirect error message to file)
**-D/-DD**

**Command line syntax**

- `D`
- `-DD`

**Description**

With the option `-D` the make utility prints every line of the makefile to standard output as it is read by `mktc`.

With the option `-DD` not only the lines of the makefile are printed but also the lines of the `mktc.mk` file (implicit rules).

**Example**

`mktc -D`

Each line of the makefile that is read by the make utility is printed to standard output (usually your screen).

**Related information**

-
-d/-dd

Command line syntax

- d
- dd

Description

With the option -d the make utility shows which files are out of date and thus need to be rebuilt. The option -dd gives more detail than the option -d.

Example

mktc -d

Shows which files are out of date and rebuilds them.

Related information
-e

Command line syntax
  -e

Description
  If you use macro definitions, they may overrule the settings of the environment variables.

  With the option -e, the settings of the environment variables are used even if macros define otherwise.

Example
  mktc -e

  The make utility uses the settings of the environment variables regardless of macro definitions.

Related information
**-err**

**Command line syntax**

`-err file`

**Description**

With this option the make utility redirects error messages and verbose messages to a specified file.

With the option `-s` the make utility only displays error messages.

**Example**

```
mktc -err error.txt
```

The make utility writes messages to the file `error.txt`.

**Related information**

* Make utility option `-s` (Do not print commands before execution)
-f

Command line syntax
   -f my_makefile

Description
   Default the make utility uses the file makefile to build your files.

   With this option you tell the make utility to use the specified file instead of
   the file makefile. Multiple -f options act as if all the makefiles were
   concatenated in a left-to-right order.

Example
   mktc mymake

   The make utility uses the file mymake to build your files.

Related information
Command line syntax

$-G \text{path}$

Description

Normally you must call the make utility `mktc` from the directory where your makefile and other files are stored.

With the option `$-G` you can call the make utility from within another directory. The `path` is the path to the directory where your makefile and other files are stored and can be absolute or relative to your current directory.

Example

Suppose your makefile and other files are stored in the directory `\curdir\myfiles`. When your current directory is `\curdir`, you can call the make utility as follows:

```
mktc -G myfiles
```

Related information

-
−i

Command line syntax
−i

Description
When an error occurs during the make process, the make utility exits with a certain exit code.

With the option −i, the make utility exits without an error code, even when errors occurred.

Example
mkic −i

The make utility exits without an error code, even when an error occurs.

Related information
-K

Command line syntax

- K

Description

With this option the make utility keeps temporary files it creates during the make process. The make utility stores temporary files in the directory that you have specified with the environment variable TMPDIR or in the default 'temp' directory of your system when the TMPDIR variable is not specified.

Example

mktc -K

The make utility preserves all temporary files.

Related information

Section 1.3.2, Configuring the Command Line Environment, in Chapter Software Installation of the User’s Manual.
-k

Command line syntax
-k

Description
When during the make process the make utility encounters an error, it stops rebuilding your files.

With the option -k, the make utility only stops building the target that produced the error. All other targets defined in the makefile are built.

Example
mktc -k

If the make utility encounters an error, it stops building the current target but proceeds with the other targets that are defined in the makefile.

Related information
Make utility option -s (Undo the effect of -k)
\texttt{\textbf{-m}}

\textbf{Command line syntax}

\texttt{-m \textit{file}}

\textbf{Description}

Instead of typing all options on the command line, you can create an option file which contains all options and flags you want to specify. With this option you specify the option file to the make utility.

Use an option file when the length of the command line would exceed the limits of the operating system, or just to store options and save typing.

You can specify the option \texttt{-m} multiple times.

\textbf{Format of an option file}

- Multiple command line arguments on one line in the option file are allowed.
- To include whitespace in an argument, surround the argument with single or double quotes.
- If you want to use single quotes as part of the argument, surround the argument by double quotes and vice versa:
  
  "This has a single quote ' embedded"

  'This has a double quote " embedded'

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

  Note that adjacent strings are concatenated.

- When a text line reaches its length limit, use a \textbackslash \ to continue the line. Whitespace between quotes is preserved.

  "This is a continuation \ line"

  \rightarrow "This is a continuation line"

- It is possible to nest command line files up to 25 levels.
Example
Suppose the file `myoptions` contains the following lines:

```
-k
-err errors.txt
test.elf
```

Specify the option file to the make utility:

```
mktc -m myoptions
```

This is equivalent to the following command line:

```
mktc -k -err errors.txt test.elf
```
-n

Command line syntax
-`n`

Description
With this option you tell the make utility to perform a *dry run*. The make utility shows what it would do but does not actually perform these tasks.

This option is for example useful to quickly inspect what would happen if you call the make utility.

Example
`mktc -n`

The make utility does not perform any tasks but displays what it would do if called without the option `-n`.

Related information
Make utility option `-s` (Do not print commands before execution)
-p

Command line syntax

-p

Description
Normally, if a command in a target rule in a makefile returns an error or when the target construction is interrupted, the make utility removes that target file. With this option you tell the make utility to make all target files precious. This means that all dependency files are never removed.

Example

mktc -p

The make utility never removes target dependency files.

Related information

Special target .PRECIOUS in section 8.3.2, Writing a Makefile in Chapter Using the Utilities of the Reference Manual.
-q

Command line syntax

- q

Description
With this option the make utility does not perform any tasks but only returns an error code. A zero status indicates that all target files are up to date, a non–zero status indicates that some or all target files are out of date.

Example

mktc -q

The make utility only returns an error code that indicates whether all target files are up to date or not. It does not rebuild any files.

Related information
-r

Command line syntax

- r

Description

When you call the make utility, it first reads the implicit rules from the file mktc.mk, then it reads the makefile with the rules to build your files. (The file mktc.mk is located in the \etc directory of the TriCore toolchain.)

With this option you tell the make utility not to read mktc.mk and to rely fully on the make rules in the makefile.

Example

    mktc -r

The make utility does not read the implicit make rules in mktc.mk.

Related information
Command line syntax

-S

Description
With this option you cancel the effect of the option -k. This is never necessary except in a recursive make where the option -k might be inherited from the top-level make via MAKEFLAGS or if you set the option -k in the environment variable MAKEFLAGS.

Example

mktc -S

The effect of the option -k is cancelled so the make utility stops with the make process after it encounters an error.

The option -k in this example may have been set with the environment variable MAKEFLAGS or in a recursive call to mktc in the makefile.

Related information

Make utility option -k (On error, abandon the work for the current target only)
\texttt{\textasciitilde s}

**Command line syntax**

\texttt{\textasciitilde s}

**Description**

With this option you tell the make utility to perform its tasks without printing the commands it executes. Error messages are normally printed.

**Example**

\texttt{mktc \textasciitilde s}

The make utility rebuilds your files but does not print the commands it executes during the make process.

**Related information**

Make utility option \texttt{\textasciitilde n} (Perform a dry run)
-t

Command line syntax

- t

Description

With this option you tell the make utility to *touch* the target files, bringing them up to date, rather than performing the rules to rebuild them.

Example

`mktc -t`

The make utility updates out-of-date files by giving them a new date and time stamp. The files are not actually rebuild.

Related information
-time

Command line syntax
- time

Description
With this option you tell the make utility to display the current date and time on standard output.

Example
mktc -time

The make utility displays the current date and time and updates out-of-date files.

Related information
-v

Command line syntax

- v

Description

Display version information. The make utility ignores all other options or input files.

Example

mktc -v

The make utility does not perform any tasks but displays the following version information:

    TASKING TriCore VX-toolset program builder    vxx.yrz Build nnn
    Copyright year Altium BV                  Serial# 00000000

Related information
-W

Command line syntax

- W target

Description

With this option the make utility considers the specified target file always as up to date and will not rebuild it.

Example

mktc -W test.elf

The make utility rebuilds out of date targets in the makefile except the file test.elf which is considered now as up to date.

Related information

-
\textbf{\texttt{-W}}

\textbf{Command line syntax}

\texttt{-w}

\textbf{Description}

With this option the make utility sends error messages and verbose messages to standard out. Without this option, the make utility sends these messages to standard error.

This option is only useful on UNIX systems.

\textbf{Example}

\texttt{mktc -w}

The make utility sends messages to standard out instead of standard error.

\textbf{Related information}

\texttt{-}
-x

Command line syntax

- x

Description

With this option the make utility shows extended error messages. Extended error messages give more detailed information about the exit status of the make utility after errors. EDE uses this option for the graphical version of make.

Example

mktc -x

If errors occur, the make utility gives extended information.

Related information
5.6 ARCHIVER OPTIONS

The archiver and library maintainer `artc` is a tool to build library files and it offers the possibility to replace, extract and remove modules from an existing library.

The invocation syntax is:

```
artc key_option [sub_option...] library [object_file]
```

This section describes all options for the archiver. Some suboptions can only be used in combination with certain key options. They are described together. Suboptions that can always be used are described separately.

The archiver is a command line tool so there are no equivalent options in EDE.

<table>
<thead>
<tr>
<th>Description</th>
<th>Option</th>
<th>Suboption</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display an overview of all options</td>
<td>–?</td>
<td>–?</td>
</tr>
<tr>
<td>Display version information</td>
<td>–V</td>
<td>–V</td>
</tr>
<tr>
<td>Print object module to standard output</td>
<td>–p</td>
<td>–p</td>
</tr>
</tbody>
</table>

**Main functions**

<table>
<thead>
<tr>
<th>Description</th>
<th>Option</th>
<th>Suboption</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delete object module from library</td>
<td>–d</td>
<td>–v</td>
</tr>
<tr>
<td>Move object module to another position</td>
<td>–m</td>
<td>–a –b –v</td>
</tr>
<tr>
<td>Replace or add an object module</td>
<td>–r</td>
<td>–a –b –c –u –v</td>
</tr>
<tr>
<td>Print a table of contents of the library</td>
<td>–t</td>
<td>–s0 –s1</td>
</tr>
<tr>
<td>Extract an object module from the library</td>
<td>–x</td>
<td>–v</td>
</tr>
</tbody>
</table>

*Table 5–1: Overview of archiver options and suboptions*
There is a line that seems to be cut off or incomplete. It appears to be a command line syntax or a related information item, but it's not clear from the image.

```
-?

Command line syntax
  -?

Description
  Displays an overview of all command line options.

Example
  The following invocations display a list of the available command line options:

  artc -?
  artc

Related information
```

-d

Command line syntax
-\textit{d} [-v]

Description
Delete the specified object modules from a library. With the suboption \textit{-v} the archiver shows which files are removed.

\textit{-v} Verbose: the archiver shows which files are removed.

Example
\texttt{artc -d lib.a obj1.o obj2.o}

The archiver deletes \texttt{obj1.o} and \texttt{obj2.o} from the library \texttt{lib.a}.

\texttt{artc -d -v lib.a obj1.o obj2.o}

The archiver deletes \texttt{obj1.o} and \texttt{obj2.o} from the library \texttt{lib.a} and displays which files are removed.

Related information
-m

Command line syntax

    -m [-a posname] [-b posname]

Description

Move the specified object modules to another position in the library.

The ordering of members in a library can make a difference in how
programs are linked if a symbol is defined in more than one member.

Default, the specified members are moved to the end of the archive. Use
the suboptions -a or -b to move them to a specified place instead.

    -a posname    Move the specified object module(s) after
                  the existing module posname.

    -b posname    Move the specified object module(s) before
                  the existing module posname.

Example

Suppose the library lib.a contains the following objects (see option -t):

    obj1.o
    obj2.o
    obj3.o

To move obj1.o to the end of lib.a:

    artc -m lib.a obj1.o

To move obj3.o just before obj2.o:

    artc -m -b obj3.o lib.a obj2.o

The library lib.a after these two invocations now looks like:

    obj3.o
    obj2.o
    obj1.o

Related information

    Archiver option -t (Print library contents)
-p

Command line syntax

-p

Description

Print the specified object module(s) in the library to standard output.

This option is only useful when you redirect or pipe the output to other files or tools that serve your own purposes. Normally you do not need this option.

Example

artc -p lib.a obj1.o > file.o

The archiver prints the file obj1.o to standard output where it is redirected to the file file.o. The effect of this example is very similar to extracting a file from the library but in this case the 'extracted' file gets another name.

Related information

-
-r

Command line syntax

```bash
-r [-a posname] [-b posname] [-c] [-u] [-v]
```

Description

You can use the option `-r` for several purposes:

- Adding new objects to the library
- Replacing objects in the library with the same object of a newer date
- Creating a new library

The option `-r` normally adds a new module to the library. However, if the library already contains a module with the specified name, the existing module is replaced. If you specify a library that does not exist, the archiver creates a new library with the specified name.

If you add a module to the library without specifying the suboption `-a` or `-b`, the specified module is added at the end of the archive. Use the suboptions `-a` or `-b` to insert them to a specified place instead.

- `-a posname` Add the specified object module(s) after the existing module `posname`.
- `-b posname` Add the specified object module(s) before the existing module `posname`.
- `-c` Create a new library without checking whether it already exists. If the library already exists, it is overwritten.
- `-u` Insert the specified object module only if it is newer than the module in the library.
- `-v` Verbose: the archiver shows which files are removed.

The suboptions `-a` or `-b` have no effect when an object is added to the library.
Examples
Suppose the library lib.a contains the following objects (see option -t):

    obj1.o

To add obj2.o to the end of lib.a:

    artc -r lib.a obj2.o

To insert obj3.o just before obj2.o:

    artc -r -b obj2.o lib.a obj3.o

The library lib.a after these two invocations now looks like:

    obj1.o
    obj3.o
    obj2.o

Creating a new library
To create a new library file, add an object file and specify a library that does not yet exist:

    artc -r obj1.o newlib.a

The archiver creates the library newlib.a and adds the object obj1.o to it.

To create a new library file and overwrite an existing library, add an object file and specify an existing library with the supoption -c:

    artc -r -c obj1.o lib.a

The archiver overwrites the library lib.a and adds the object obj1.o to it. The new library lib.a only contains obj1.o.

Related information

Archiver option -t (Print library contents)
-t

Command line syntax

- t [-s0 | -s1]

Description

Print a table of contents of the library to standard out. With the suboption -s you the archiver displays all symbols per object file.

- s0 Displays per object the library in which it resides, the name of the object itself and all symbols in the object.

- s1 Displays only the symbols of all object files in the library.

Example

artc -t lib.a

The archiver prints a list of all object modules in the library lib.a.

artc -t -s0 lib.a

The archiver prints per object all symbols in the library. This looks like:

prolog.o
  symbols:
  lib.a:prolog.o:__Qabi_callee_save
  lib.a:prolog.o:__Qabi_callee_restore
div16.o
  symbols:
  lib.a:div16.o:__udiv16
  lib.a:div16.o:__div16
  lib.a:div16.o:__urem16
  lib.a:div16.o:__rem16

Related information
-V

Command line syntax

- V

Description

Display version information. The archiver ignores all other options or input files.

Example

artc -V

The archiver does not perform any tasks but displays the following version information:

  TASKING TriCore VX-toolset ELF archiver  vxx.yrz Build nnn
  Copyright year Altium BV               Serial# 00000000

Related information

-
extract an existing module from the library.

- **-o** Give the extracted object module the same date as the last-modified date that was recorded in the library.

  Without this suboption it receives the last-modified date of the moment it is extracted.

- **-v** Verbose: the archiver shows which files are extracted.

**Example**

To extract the file `obj.o` from the library `lib.a`:

```
artc -x lib.a obj1.o
```

If you do not specify an object module, all object modules are extracted:

```
artc -x lib.a
```
\texttt{\texttt{-w}}

\textbf{Command line syntax}

\texttt{-wlevel}

\textbf{Description}

With this suboption you tell the archiver to suppress all warnings above the specified level. The level is a number between 0 – 9.

The level of a message is printed between parentheses after the warning number. If you do not use the \texttt{-w} option, the default warning level is 8.

\textbf{Example}

To suppress warnings above level 5:

\texttt{artc -x -w5 lib.a obj1.o}

\textbf{Related information}
CHAPTER 6

LIST FILE FORMATS

TASKING
6.1 ASSEMBLER LIST FILE FORMAT

The assembler list file is an additional output file of the assembler that contains information about the generated code.

The list file consists of a page header and a source listing.

Page header

The page header consists of four lines:

TASKING TriCore VX-toolset Assembler vx.yrz Build nnn SN 00000000
This is the page header title

<table>
<thead>
<tr>
<th>ADDR</th>
<th>CODE</th>
<th>CYCLES</th>
<th>LINE</th>
<th>SOURCE</th>
<th>LINE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The first line contains information about the assembler name, version number and serial number. The second line contains a title specified by the TITLE (first page) assembler directive and a page number. The third line is empty. The fourth line contains the heading of the source listing.

Source listing

The following is a sample part of a listing. An explanation of the different columns follows below.

ADDR CODE   CYCLES LINE SOURCE LINE
          .                      .
0002 85Fr   1  2  27   ld.a  a15,world
0006 F4AF   1  3  28   st16.a [a10],a15
0008 91rrr4r 1  4  29   movh.a a4,#@his(_2_ini)
000C D944rrr 1  5  30   lea a4,[a4]@los(_2_ini)
0010 1Drrrrrr 1  6  31   j   printf
          .                      .
0000              44 buf: .space 4
0003

The meaning of the different columns is:

ADDR This column contains the memory address. The address is a hexadecimal number that represents the offset from the beginning of a relocatable section or the absolute address for an absolute section. The address only appears on lines that generate object code.
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. In this case the letter 'r' is printed for the relocatable code part in the listing. For lines that allocate space, the code field contains the text "RESERVED".

CYCLES

The first number in this column is the number of instruction cycles needed to execute the instruction(s) as generated in the CODE field. The second number is the accumulated cycle count of this section.

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.

SOURCE LINE

This column contains the source text. This is a copy of the source line from the assembly source file.

For the .SET and .EQU directives the ADDR and CODE columns do not apply. The symbol value is listed instead.

Related information

See section 6.6, Generating a List File, in Chapter Using the Assembler of the User’s Manual for more information on how to generate a list file and specify the amount of list file information.
6.2 LINKER MAP FILE FORMAT

The linker map file is an additional output file of the linker that shows how the link phase has mapped the sections and symbols from the various object files (.o) to output sections. The locate part shows the absolute position of each section. External symbols are listed per space with their absolute address, both sorted on symbol and sorted on address.

With the linker option `-m` (map file formatting) you can specify which parts of the map file you want to see.

Example (part of) linker map file

Tricore VX-toolset object linker - mapfile (task: task1)
Options: -o hello.elf -m -mq -ms --map-file

****************************************************************************** File Part ******************************************************************************

* Processed files:  

<table>
<thead>
<tr>
<th>File</th>
<th>From archive</th>
<th>Symbol causing the extraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>cstart.o</td>
<td>libc.a</td>
<td>_START</td>
</tr>
<tr>
<td>hello.o</td>
<td></td>
<td></td>
</tr>
<tr>
<td>printf.o</td>
<td>libc.a</td>
<td>printf</td>
</tr>
</tbody>
</table>

****************************************************************************** Call Graph Part ******************************************************************************

****************************************************************************** Link Part ******************************************************************************

* Section translation:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>hello.o</td>
<td>.text.hello.main</td>
<td>0x00000014</td>
<td>0x00000000</td>
<td>.text.hello.main</td>
</tr>
<tr>
<td>cstart.o</td>
<td>.text.libc</td>
<td>0x000001fc</td>
<td>0x00000000</td>
<td>.text.libc</td>
</tr>
<tr>
<td>strcpy.o</td>
<td>.text.libc</td>
<td>0x00000024</td>
<td>0x000001fc</td>
<td></td>
</tr>
<tr>
<td>printf.o</td>
<td>.text.libc</td>
<td>0x00000024</td>
<td>0x00000070</td>
<td></td>
</tr>
<tr>
<td>cinit.o</td>
<td>.text.libc</td>
<td>0x0000004e</td>
<td>0x00000d68</td>
<td></td>
</tr>
<tr>
<td>cstart.o</td>
<td>.text.libc.reset</td>
<td>0x00000008</td>
<td>0x00000000</td>
<td>.text.libc.reset</td>
</tr>
</tbody>
</table>
* Local symbol translation (sorted on symbol):

+ File "hello.o"
+ Scope "/.hello.o"

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Address</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>hello.src</td>
<td>0x00000000</td>
<td></td>
</tr>
<tr>
<td>.zdata.hello.world</td>
<td>0xd0000000</td>
<td>spe:tc:abs18</td>
</tr>
<tr>
<td>.zrodata.hello..1.ini</td>
<td>0xa0000008</td>
<td></td>
</tr>
<tr>
<td>_1.ini</td>
<td>0xa0000008</td>
<td></td>
</tr>
<tr>
<td>.rodata.hello..2.ini</td>
<td>0xa00000dc</td>
<td>spe:tc:linear</td>
</tr>
<tr>
<td>_2.ini</td>
<td>0xa00000df</td>
<td></td>
</tr>
<tr>
<td>main_function_end</td>
<td>0xa0000108</td>
<td></td>
</tr>
</tbody>
</table>

* Defined symbols:

<table>
<thead>
<tr>
<th>Definition file</th>
<th>Definition section</th>
<th>Symbol</th>
<th>Referenced in</th>
</tr>
</thead>
<tbody>
<tr>
<td>_dprintf_int.o</td>
<td>.text.libc</td>
<td>__printf_int2</td>
<td>printf_int.o</td>
</tr>
<tr>
<td>_dprintf_int.o</td>
<td>.text.libc</td>
<td>_dprintf</td>
<td>printf.o</td>
</tr>
<tr>
<td>cstart.o</td>
<td>.text.libc</td>
<td>_Exit</td>
<td>exit.o</td>
</tr>
<tr>
<td>cstart.o</td>
<td>.text.libc</td>
<td>_start</td>
<td>hello.o</td>
</tr>
<tr>
<td>hello.o</td>
<td>.text.hello.main</td>
<td>_start</td>
<td>cstart.o</td>
</tr>
</tbody>
</table>

* Undefined symbols:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Referenced in</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>LITERAL_DATA</em></td>
<td>cstart.o</td>
</tr>
<tr>
<td><em>SMALL_DATA</em></td>
<td>cstart.o</td>
</tr>
<tr>
<td>_lc_cp</td>
<td>cinit.o</td>
</tr>
</tbody>
</table>

* Overlay Part

**********************************************************
Module Local Symbols Part
**********************************************************
### Locate Part

* Task entry address:

```
symbol:  _START
absolute: 0xa0000000
```

* Section translation:

```
+ Space spe:tc:abs18

<table>
<thead>
<tr>
<th>Chip</th>
<th>Group</th>
<th>Section</th>
<th>Size (MAU)</th>
<th>Space addr</th>
<th>Chip addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>ext_c</td>
<td></td>
<td>.zrodata_hello..1.ini</td>
<td>0x00000006</td>
<td>0xa0000008</td>
<td>0x00000008</td>
</tr>
<tr>
<td>spe:dsram</td>
<td></td>
<td>.zdata_hello.world</td>
<td>0x00000004</td>
<td>0xd000000</td>
<td>0x00000000</td>
</tr>
</tbody>
</table>

+ Space spe:tc:csa

```
<table>
<thead>
<tr>
<th>Chip</th>
<th>Group</th>
<th>Section</th>
<th>Size (MAU)</th>
<th>Space addr</th>
<th>Chip addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>spe:dsram</td>
<td></td>
<td>csa.01</td>
<td>0x00001000</td>
<td>0xd0000040</td>
<td>0x00000040</td>
</tr>
</tbody>
</table>

+ Space spe:tc:linear

```
<table>
<thead>
<tr>
<th>Chip</th>
<th>Group</th>
<th>Section</th>
<th>Size (MAU)</th>
<th>Space addr</th>
<th>Chip addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>ext_c</td>
<td></td>
<td>.text.libc.reset</td>
<td>0x00000008</td>
<td>0xa000000</td>
<td>0x00000000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[.data.libc]</td>
<td>0x000000c8</td>
<td>0xa000010</td>
<td>0x00000010</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[.zdata_hello.world]</td>
<td>0x00000004</td>
<td>0xa0000d8</td>
<td>0x000000d8</td>
</tr>
<tr>
<td></td>
<td>libraries</td>
<td>.text.libc</td>
<td>0x00000db6</td>
<td>0xa000400</td>
<td>0x00004000</td>
</tr>
</tbody>
</table>
```

* Symbol translation (sorted on symbol):

```
Symbol | Address | Space |
--------|---------|-------|
world   | 0xd0000000 | spe:tc:abs18 |

_A8_DATA_ | 0x00000000 | spe:tc:linear |
_A9_DATA_ | 0x00000000 |
_exit    | 0xa000416a |
```

* Symbol translation (sorted on address):

```
Address | Symbol            | Space |
--------|-------------------|-------|
0xd0000000 | world            | spe:tc:abs18 |
0x00000000 | _LITERAL_DATA_   | spe:tc:linear |
0x00000000 | _A9_DATA_        | |
```
Call Graph Part

By default this part is not shown in the map file. You have to turn this part on manually with linker option -mc (call graph info).
**Link Part: Section translation**

This part of the map file shows per object file how the link phase has mapped the sections from the various object files (.o) to output sections.

- **[in] File** The name of an input object file.
- **[in] Section** A section name from the input object file.
- **[in] Size** The size of the input section.
- **[out] Offset** The offset relative to the start of the output section.
- **[out] Section** The resulting output section name.

The input sections `.text.libc` in the object modules `cstart.o`, `strncpy.o`, `printf.o` and `cinit.o` in the example above are all mapped on the output section `.text.libc` on succeeding offsets.

**Module Local Symbols Part**

This part of the map file shows a table for each local scope within an object file. Each table has three columns, 1 the symbol name, 2 the address of the symbol and 3 the space where the symbol resides in. The table is sorted on symbol name within each space.

By default this part is not shown in the map file. You have to turn this part on manually with linker option `-mq` (module local symbols).

**Cross Reference Part**

This part of the map file lists all symbols defined in the object modules and for each symbol the object modules that contain a reference to the symbol are shown. Also, symbols that remain undefined are shown.

By default this part is not shown in the map file. You have to turn this part on manually with linker option `-mr` (cross references info).

**Overlay Part**

This part of the map file shows how the static stack is organized. This part is empty for the TriCore. This part also shows the locate overlay information if you used overlay groups in the linker script file.

By default this part is not shown in the map file. You have to turn this part on manually with linker option `-mo` (overlay info).
**Locate Part: Section translation**

This part of the map file shows the absolute position of each section in the absolute object file. It is organized per address space, memory chip and group.

- **Space** The names of the address spaces as defined in the linker script file (tc*.lsl). The names are constructed of the derivative name followed by a colon ‘:’, the core name, another colon ‘:’ and the space name. For example: `spe:tc:linear`

- **Chip** The names of the memory chips as defined in the linker script file (*.lsl) in the memory definitions.

- **Group** Sections can be ordered in groups. These are the names of the groups as defined in the linker script file (*.lsl) with the keyword `group` in the `section_layout` definition. The name that is displayed is the name of the deepest nested group.

- **Section** The name of the section. Names within square brackets `[]` will be copied during initialization from ROM to the corresponding section name in RAM.

- **Size (MAU)** The size of the section in minimum addressable units.

- **Space addr** The absolute address of the section in the address space.

- **Chip addr** The absolute offset of the section from the start of a memory chip.

**Locator Part: Symbol translation**

This part of the map file lists all external symbols per address space name, both sorted on symbol name and sorted on address.

- **Symbol** The name of the symbol.

- **Address** The absolute address of the symbol in the address space.
Space

The names of the address spaces as defined in the linker script file (tc*.lsl). The names are constructed of the derivative name followed by a colon ':', the core name, another colon ':' and the space name. For example: spe:tc:linear

Memory Part

This part of the map file shows the memory usage in totals and percentages for spaces and chips. The largest free block of memory per space and per chip is also shown.

By default this part is not shown in the map file. You have to turn this part on manually with linker option -mm (memory usage info).

Linker Script File Part

This part of the map file shows the processor and memory information of the linker script file.

By default this part is not shown in the map file. You have to turn this part on manually with linker option -ms (processor and memory info). You can print this information to a separate file with linker option --lsl-dump.

Locate Rule Part

This part of the map file shows the rules the linker uses to locate sections.

Address space

The names of the address spaces as defined in the linker script file (*.lsl). The names are constructed of the derivative name followed by a colon ':', the core name, another colon ':' and the space name. For example: spe:tc:linear

Type

The rule type:

ordered/contiguous/clustersed Specifies how sections are grouped.

absolute address The section must be located at the address shown in the Properties column

address range The section must be located in the union of the address ranges shown in the Properties column; end addresses are not included in the range.
address range size  The sections must be located in some address range with size not larger than shown in the Properties column; the second number in that field is the alignment requirement for the address range.

Properties  The contents depends on the Type column.

Sections  The sections to which the rule applies; restrictions between sections are shown in this column:

< ordered
| contiguous
+ clustered

For contiguous sections, the linker uses the section order as shown here. Clustered sections can be located in any relative order.

By default this part is not shown in the map file. You have to turn this part on manually with linker option `-mu` (locate rules).

Related information

Section 7.9, *Generating a Map File*, in Chapter *Using the Linker* of the *User’s Manual*.

Linker option `-M` (Generate map file)
CHAPTER 7

OBJECT FILE FORMATS

TASKING
7.1 ELF/DWARF OBJECT FORMAT

The TriCore toolchain by default produces objects in the ELF/DWARF 2 (.elf) format.

The ELF/DWARF 2 Object Format for the TriCore toolchain follows the convention as described in the *TriCore Embedded Application Binary Interface* [2000, Infineon].

For a complete description of the ELF and DWARF formats, please refer to the *Tool Interface Standard (TIS)*.
7.2 MOTOROLA S-RECORD FORMAT

With the linker option `-o filename:SREC` option the linker produces output in Motorola S-record format with three types of S-records: S0, S2 and S8. With the options `-o filename:SREC:2` or `-o filename:SREC:4` option you can force other types of S-records. They have the following layout:

**S0 - record**

'S' '0' `<length_byte>` `<2 bytes 0>` `<comment>` `<checksum_byte>`

A linker generated S-record file starts with a S0 record with the following contents:

- `length_byte` : 0x6
- `comment` : ltc (TriCore linker)
- `checksum` : 0xB6

```
ltc
S006000006C7463B6
```

The S0 record is a comment record and does not contain relevant information for program execution.

The `length_byte` represents the number of bytes in the record, not including the record type and length byte.

The checksum is calculated by first adding the binary representation of the bytes following the record type (starting with the `length_byte`) to just before the checksum. Then the one’s complement is calculated of this sum. The least significant byte of the result is the checksum. The sum of all bytes following the record type is 0xFF.

**S1 - record**

With the linker option `-o filename:SREC:2`, the actual program code and data is supplied with S1 records, with the following layout:

'S' '1' `<length_byte>` `<address>` `<code bytes>` `<checksum_byte>`

This record is used for 2-byte addresses.
Example:

```
S1130250F03EF04DF0ACE8A408A2A013EDFCDB00E6
```

The linker 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 linker option `-ofilename:SREC:3`, 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 linker generates 3-byte addresses.

Example:

```
S213FF0020023222754E00754F04AF4FAE4E22BF
```

The linker 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 linker option `-ofilename:SREC:4`, 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:

```
S3070000FE6E6825
   |   |     checksum
   |   |     code
   |   |     address
   |     length
```

The linker 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 linker option `-ofilename:4`, at the end of an S-record file, the linker 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:

```
S7050006E6824
   |   |     checksum
   |   |     address
   |     length
```

The checksum calculation of S7 records is identical to S0.

**S8 - record**

With the linker option `-ofilename:3`, which is the default, at the end of an S-record file, the linker generates an S8 record, which contains the program start address.

Layout:

```
'S' '8' <length_byte> <address> <checksum_byte>
```

Example:

```
S8040003F9
   |   |     checksum
   |   |     address
   |     length
```
The checksum calculation of S8 records is identical to S0.

**S9 - record**

With the linker option `–ofilename:SREC:4`, at the end of an S-record file, the linker 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.
7.3 INTEL HEX RECORD FORMAT

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 linker generates records in the 32-bit format (4-byte addresses with linker option –ofilename:HEX).

General Record Format

In the output file, the record format is:

<table>
<thead>
<tr>
<th>:</th>
<th>length</th>
<th>offset</th>
<th>type</th>
<th>content</th>
<th>checksum</th>
</tr>
</thead>
</table>

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 linker outputs records of 255 bytes (32 hexadecimal digits) or less; that is, length is never greater than 0xFF.

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:
content is the information contained in the record. This depends on the record type.

checksum is the record checksum. The linker computes the checksum by first adding the binary representation of the previous bytes (from length to content). The linker 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 upper_address checksum
```

The 32-bit absolute address of a byte in a Data Record is calculated as:

\[
(\text{address} + \text{offset} + \text{index}) \mod 4G
\]

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:

:0200004000FFB
  | | | | _ checksum
  | | | | _ upper_address
  | | | _ type
  | _ offset
  | _ length

**Data Record**

The Data Record specifies the actual program code and data.

<table>
<thead>
<tr>
<th>:</th>
<th>length</th>
<th>offset</th>
<th>00</th>
<th>data</th>
<th>checksum</th>
</tr>
</thead>
</table>

The *length* byte specifies the number of *data* bytes. The linker 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:

:0F002000023222754E00754F04AF4FAE4E22C3
  | | | | _ data
  | | | _ type
  | | _ offset
  | _ length
  | _ checksum
**Start Linear Address Record**

The Start Linear Address Record contains the 32-bit program execution start address.

Layout:

<table>
<thead>
<tr>
<th></th>
<th>04</th>
<th>0000</th>
<th>05</th>
<th>address</th>
<th>checksum</th>
</tr>
</thead>
</table>

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 |
CHAPTER 8

LINKER SCRIPT LANGUAGE
8.1 INTRODUCTION

To make full use of the linker, you can write a script with information about the architecture of the target processor and locating information. The language for the script is called the **Linker Script Language** (LSL). This chapter first describes the structure of an LSL file. The next section contains a summary of the LSL syntax. Finally, in the remaining sections, the semantics of the Linker Script Language is explained.

The TASKING linker is a target independent linker/locator that can simultaneously link and locate all programs for all cores available on a target board. The target board may be of arbitrary complexity. A simple target board may contain one standard processor with some external memory that executes one task. A complex target board may contain multiple standard processors and DSPs combined with configurable IP–cores loaded in an FPGA. Each core may execute a different program, and external memory may be shared by multiple cores.

LSL serves two purposes. First it enables you to specify the characteristics (that are of interest to the linker) of your specific target board and of the cores installed on the board. Second it enables you to specify how sections should be located in memory.

8.2 STRUCTURE OF A LINKER SCRIPT FILE

A script file consists of several definitions. The definitions can appear in any order.

The architecture definition *(required)*

In essence an **architecture definition** describes how the linker should convert logical addresses into physical addresses for a given type of core. If the core supports multiple address spaces, then for each space the linker must know how to perform this conversion. In this context a physical address is an offset on a given internal or external bus. Additionally the architecture definition contains information about items such as the (hardware) stack and the interrupt vector table.

This specification is normally written by Altium. The architecture definition of the LSL file should not be changed by you unless you also modify the core’s hardware architecture. If the LSL file describes a multi-core system an architecture definition must be available for each different type of core.
See section 8.5, *Semantics of the Architecture Definition* for detailed descriptions of LSL in the architecture definition.

**The derivative definition (required)**

The *derivative definition* describes the configuration of the internal (on-chip) bus and memory system. Basically it tells the linker how to convert offsets on the buses specified in the architecture definition into offsets in internal memory. A derivative definition must be present in an LSL file. Microcontrollers and DSPs often have internal memory and I/O sub-systems apart from one or more cores. The design of such a chip is called a *derivative*.

Altium provides LSL descriptions of supported derivatives, along with "SFR files", which provide easy access to registers in I/O sub-systems from C and assembly programs. When you build an ASIC or use a derivative that is not (yet) supported by the TASKING tools, you may have to write a derivative definition.

When you want to use multiple cores of the same type, you must instantiate the cores in a derivative definition, since the linker automatically instantiates only a single core for an unused architecture.

See section 8.6, *Semantics of the Derivative Definition* for a detailed description of LSL in the derivative definition.

**The processor definition**

The *processor definition* describes an instance of a derivative. A processor definition is only needed in a multi-processor embedded system. It allows you to define multiple processors of the same type.


**The memory and bus definitions (optional)**

Memory and bus definition are used within the context of a derivative definition to specify internal memory and on-chip buses. In the context of a board specification the memory and bus definitions are used to define external (off-chip) memory and buses. Given the above definitions the linker can convert a logical address into an offset into an on-chip or off-chip memory device.
See section 8.7.3, *Defining External Memory and Buses*, for more information on how to specify the external physical memory layout. *Internal* memory for a processor should be defined in the derivative definition for that processor.

**The board specification**

The processor definition and memory and bus definitions together form a *board specification*. LSL provides language constructs to easily describe single-core and heterogeneous or homogeneous multi-core systems. The board specification describes all characteristics of your target board's system buses, memory devices, I/O sub-systems, and cores that are of interest to the linker. Based on the information provided in the board specification the linker can for each core:

- convert a logical address to a physical addresses (offsets within a memory device)
- locate sections in physical memory
- maintain an overall view of the used and free physical memory within the whole system while locating

**The section layout definition (optional)**

The optional *section layout definition* enables you to exactly control where input sections are located. Features are provided such as: the ability to place sections at a given load-address or run-time address, to place sections in a given order, and to overlay code and/or data sections.

Which object files (sections) constitute the task that will run on a given core is specified on the command line when you invoke the linker. The linker will link and locate all sections of all tasks simultaneously. From the section layout definition the linker can deduce where a given section may be located in memory, form the board specification the linker can deduce which physical memory is (still) available while locating the section.

See section 8.8, *Semantics of the Section Layout Definition*, for more information on how to locate a section at a specific place in memory.

**Skeleton of a Linker Script File**

The skeleton of a linker script file now looks as follows:

```plaintext
architecture architecture_name
{
    architecture definition
}
```
derivative derivative_name
{
    derivative definition
}

processor processor_name
{
    processor definition
}

memory definitions and/or bus definitions

section_layout space_name
{
    section placement statements
}

8.3 SYNTAX OF THE LINKER SCRIPT LANGUAGE

8.3.1 PREPROCESSING

When the linker loads an LSL file, the linker processes it with a C-style preprocessor. As such, it strips C and C++ comments. You can use the standard ISO C preprocessor directives, such as `#include`, `#define`, `#if/#else/#endif`.

For example:

```
#include "arch.lsl"
```

Preprocess and include the file `arch.lsl` at this point in the LSL file.
8.3.2 LEXICAL SYNTAX

The following lexicon is used to describe the syntax of the Linker Script Language:

\[
\begin{align*}
A ::= B &= A \text{ is defined as } B \\
A ::= B \ C &= A \text{ is defined as } B \text{ and } C; \ B \text{ is followed by } C \\
A ::= B | C &= A \text{ is defined as } B \text{ or } C \\
<B>^0 | 1 &= \text{ zero or one occurrence of } B \\
<B>^0 | 0 &= \text{ zero of more occurrences of } B \\
<B>^1 | 1 &= \text{ one of more occurrences of } B
\end{align*}
\]

**IDENTIFIER** = a character sequence starting with 'a'–'z', 'A'–'Z' or '.
Following characters may also be digits and dots '.'

**STRING** = sequence of characters not starting with \n, \r or \t

**DQSTRING** = " STRING " (double quoted string)

**OCT_NUM** = decimal number, starting with a zero (06, 045)

**DEC_NUM** = decimal number, not starting with a zero (14, 1024)

**HEX_NUM** = hexadecimal number, starting with '0x' (0x0023, 0xFF00)

**OCT_NUM**, **DEC_NUM** and **HEX_NUM** can be followed by a **k** (kilo), **M**
(mega), or **G** (giga).

Characters in **bold** are characters that occur literally. Words in **italics** are
higher order terms that are defined in the same or in one of the other
sections.

To write comments in LSL file, you can use the C style ‘/* */’ or C++
style ‘//’.

8.3.3 IDENTIFIERS

\[
\begin{align*}
\text{arch_name} &= ::= \text{ IDENTIFIER} \\
\text{bus_name} &= ::= \text{ IDENTIFIER} \\
\text{core_name} &= ::= \text{ IDENTIFIER} \\
\text{derivative_name} &= ::= \text{ IDENTIFIER} \\
\text{file_name} &= ::= \text{ DQSTRING} \\
\text{group_name} &= ::= \text{ IDENTIFIER} \\
\text{mem_name} &= ::= \text{ IDENTIFIER} \\
\text{proc_name} &= ::= \text{ IDENTIFIER} \\
\text{section_name} &= ::= \text{ DQSTRING} \\
\text{space_name} &= ::= \text{ IDENTIFIER} \\
\text{stack_name} &= ::= \text{ section_name} \\
\text{symbol_name} &= ::= \text{ DQSTRING}
\end{align*}
\]
8.3.4 EXPRESSIONS

The expressions and operators in this section work the same as in ANSI C.

```
number ::= OCT_NUM
  | DEC_NUM
  | HEX_NUM

expr ::= number
  | symbol_name
  | unary_op expr
  | expr binary_op expr
  | expr ? expr : expr
  | ( expr )
  | function_call

unary_op ::= ! // logical NOT
  | ~ // bitwise complement
  | - // negative value

binary_op ::= ^ // exclusive OR
  | * // multiplication
  | / // division
  | % // modulus
  | + // addition
  | - // subtraction
  | >> // right shift
  | << // left shift
  | == // equal to
  | != // not equal to
  | > // greater than
  | < // less than
  | >= // greater than or equal to
  | <= // less than or equal to
  | & // bitwise AND
  | | // bitwise OR
  | && // logical AND
  | || // logical OR
```
8.3.5 BUILT-IN FUNCTIONS

`function_call` ::= `absolute ( expr )`
  | `addressof ( addr_id )`
  | `exists ( section_name )`
  | `max ( expr , expr )`
  | `min ( expr , expr )`
  | `sizeof ( size_id )`

`addr_id` ::= `sect : section_name`
  | `group : group_name`

`size_id` ::= `group : group_name`
  | `mem : mem_name`
  | `sect : section_name`

- Every space, bus, memory, section or group your refer to, must be
defined in the LSL file.
- The `addressof()` and `sizeof()` functions with the `group` or
  `sect` argument can only be used in the right hand side of an
  assignment. The `sizeof()` function with the `mem` argument can be
  used anywhere in section layouts.

You can use the following built-in functions in expressions. All functions
return a numerical value. This value is a 64-bit signed integer.

**absolute()**

```c
int absolute( expr )
```

Converts the value of `expr` to a positive integer.

```c
absolute( "labelA"-"labelB" )
```

**addressof()**

```c
int addressof( addr_id )
```

Returns the address of `addr_id`, which is a named section or group. To get
the offset of the section with the name `sect`:

```c
addressof( sect: "asect")
```

This function only works in assignments.
exists()

```c
int exists( section_name )
```

The function returns 1 if the section `section_name` exists in one or more object file, 0 otherwise. If the section is not present in input object files, but generated from LSL, the result of this function is undefined.

To check whether the section `myssection` exists in one of the object files that is specified to the linker:

```c
exists( "myssection" )
```

max()

```c
int max( expr, expr )
```

Returns the value of the expression that has the largest value. To get the highest value of two symbols:

```c
max( "sym1", "sym2" )
```

min()

```c
int min( expr, expr )
```

Returns the value of the expression hat has the smallest value. To get the lowest value of two symbols:

```c
min( "sym1", "sym2" )
```

sizeof()

```c
int sizeof( size_id )
```

Returns the size of the object (group, section or memory) the identifier refers to. To get the size of the section "asection":

```c
sizeof( sect: "asection" )
```

The `group` and `sect` arguments only works in assignments. The `mem` argument can be used anywhere in section layouts.
8.3.6 LSL DEFINITIONS IN THE LINKER SCRIPT FILE

description ::= <definition>\n
definition ::= architecture_definition
    | derivative_definition
    | board_spec
    | section_definition

- At least one architecture_definition must be present in the LSL file.

8.3.7 MEMORY AND BUS DEFINITIONS

mem_def ::= memory mem_name { <mem_descr ;>\n
- A mem_def defines a memory with the mem_name as a unique name.

mem_descr ::= type = <reserved>\n    | mau = expr
    | size = expr
    | speed = number
    | mapping

- A mem_def contains exactly one type statement.
- A mem_def contains exactly one mau statement (non-zero size).
- A mem_def contains exactly one size statement.
- A mem_def contains zero or one speed statement (default value is 1).
- A mem_def contains at least one mapping.

mem_type ::= rom    // attrs = rx
    | ram    // attrs = rw
    | nram   // attrs = rwx

bus_def ::= bus bus_name { <bus_descr ;>\n
- A bus_def statement defines a bus with the given bus_name as a unique name within a core architecture.

bus_descr ::= mau = expr
    | width = expr   // bus width, nr
    |                  // of data bits
    | mapping         // legal destination
    |                  // 'bus' only
• The **mau** and **width** statements appear exactly once in a **bus_descr**. The default value for **width** is the **mau** size.

• The bus width must be an integer times the bus MAU size.

• The MAU size must be non-zero.

• A bus can only have a **mapping** on a destination **bus** (through **dest = bus**).

\[\text{mapping} \quad ::= \quad \text{map} \ ( \ \text{map_descr} \ <, \ \text{map_descr} \ >=0 \ )\]

\[\text{map_descr} \quad ::= \quad \text{dest} = \text{destination} \]

\[\quad | \quad \text{dest\_dbits} = \text{range} \]

\[\quad | \quad \text{dest\_offset} = \text{expr} \]

\[\quad | \quad \text{size} = \text{expr} \]

\[\quad | \quad \text{src\_dbits} = \text{range} \]

\[\quad | \quad \text{src\_offset} = \text{expr} \]

• A **mapping** requires at least the **size** and **dest** statements.

• Each **map\_descr** can occur only once.

• You can define multiple mappings from a single source.

• Overlap between source ranges or destination ranges is not allowed.

• If the **src\_dbits** or **dest\_dbits** statement is not present, its value defaults to the **width** value if the source/destination is a bus, and to the **mau** size otherwise.

\[\text{destination} \quad ::= \quad \text{space} : \text{space\_name} \]

\[\quad | \quad \text{bus} : (<\text{proc\_name} \ |
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad <\text{core\_name} : 0|1 \ \text{bus\_name})\]

• A **space\_name** refers to a defined address space.

• A **proc\_name** refers to a defined processor.

• A **core\_name** refers to a defined core.

• A **bus\_name** refers to a defined bus.

• The following mappings are allowed (source to destination)

  - space => space
  - space => bus
  - bus => bus
  - memory => bus

\[\text{range} \quad ::= \quad \text{number} \ \ldots \ \text{number}\]
8.3.8 ARCHITECTURE DEFINITION

architecture_definition
  ::= architecture arch_name
     ⟨( parameter_list )⟩⁺|₁
     <extends arch_name
           ⟨( argument_list )⟩⁺|₁ >⁺|₁
     { arch_spec=⁺ }

- An architecture_definition defines a core architecture with the given arch_name as a unique name.
- At least one space_def and at least one bus_def have to be present in an architecture_definition.
- An architecture_definition that uses the extends construct defines an architecture that inherits all elements of the architecture defined by the second arch_name. The parent architecture must be defined in the LSL file as well.

parameter_list ::= parameter <, parameter>⁺

parameter ::= IDENTIFIER <= expr⁺|₁

argument_list ::= expr <, expr>⁺

arch_spec ::= bus_def
              | space_def
              | endianness_def

space_def ::= space space_name { <space_descr;>⁺ }

- A space_def defines an address space with the given space_name as a unique name within an architecture.

space_descr ::= space_property ;
               | section_definition //no space ref

space_property ::= id = number // as used in object
                 | mau = expr
                 | align = expr
                 | page_size = expr
                 | stack_def
                 | heap_def
                 | copy_table_def
                 | start_address
                 | mapping

- A space_def contains exactly one id and one mau statement.
• A `space_def` contains at most one `align` statement.
• A `space_def` contains at most one `page_size` statement.
• A `space_def` contains at least one mapping.

```
stack_def ::= stack stack_name ( stack_heap_descr <, stack_heap_descr >=0 )
```

• A `stack_def` defines a stack with the `stack_name` as a unique name.

```
heap_def ::= heap heap_name ( stack_heap_descr <, stack_heap_descr >=0 )
```

• A `heap_def` defines a heap with the `heap_name` as a unique name.

```
copy_table_def ::= copytable ( copy_table_descr <, copy_table_descr>=0 )
```

• A `space_def` contains at most one `copytable` statement.
• If the architecture definition contains more than one address space, exactly one copy table must be defined in one of the spaces. If the architecture definition contains only one address space, a copy table definition is optional (it will be generated in the space).

```
stack_heap_descr ::= min_size = expr
            | grows = direction
            | align = expr
            | fixed
```

• The `min_size` statement must be present.
• You can specify at most one `align` statement and one `grows` statement.

```
direction ::= low_to_high
            | high_to_low
```

• If you do not specify the `grows` statement, the stack and grow `low-to-high`.

```
copy_table_descr ::= align = expr
            | copy_unit = expr
            | dest <space_name>0|1 = space_name
```

• The `copy_unit` is defined by the size in MAUs in which the startup code moves data.
• The `dest` statement is only required when the startup code initializes memory used by another processor that has no access to ROM.
- A *space_name* refers to a defined address space.

\[
\text{start_addr} ::= \text{start_address} ( \text{start_addr_descr} \\
\quad , \text{start_addr_descr} >>=0 ) 
\]

\[
\text{start_addr_descr} ::= \text{run_addr} = \text{expr} \\
\quad | \text{symbol} = \text{symbol_name} 
\]

- A *symbol_name* refers to the section that contains the startup code.

\[
\text{endianness_def} ::= \text{endianness} \{ <\text{endianness_type};>=1 \} 
\]

\[
\text{endianness_type} ::= \text{big} \\
\quad | \text{little} 
\]

### 8.3.9 DERIVATIVE DEFINITION

\[
\text{derivative_definition} ::= \text{derivative} \text{derivative_name} \\
\quad <( \text{parameter_list } )>^0|^1 \\
\quad <\text{extends} \text{derivative_name} \\
\quad \quad <( \text{argument_list } )>^0|^1 >^0|^1 \\
\quad \quad \{ <\text{derivative_spec}>=0 \} 
\]

- A *derivative_definition* defines a derivative with the given *derivative_name* as a unique name.

- At least one *core_def* must be present in a *derivative_definition*.

\[
\text{derivative_spec} ::= \text{core_def} \\
\quad | \text{bus_def} \\
\quad | \text{mem_def} \\
\quad | \text{section_definition} // \text{no processor} \\
\quad // \text{name} 
\]

\[
\text{core_def} ::= \text{core} \text{core_name} \{ <\text{core_descr }>=0 \} 
\]

- A *core_def* defines a *core* with the given *core_name* as a unique name.

\[
\text{core_descr} ::= \text{architecture} = \text{arch_name} \\
\quad <( \text{argument_list } )>^0|^1 \\
\quad | \text{endianness} = ( \text{endianness_type} \\
\quad \quad , \text{endianness_type} >>=0 ) 
\]

- An *arch_name* refers to a defined core architecture.
• Exactly one architecture statement must be present in a core_def.

8.3.10 PROCESSOR DEFINITION AND BOARD SPECIFICATION

board_spec ::= proc_def
  | bus_def
  | mem_def

proc_def ::= processor proc_name
  { proc_descr ; }

proc_descr ::= derivative = derivative_name
  <( argument_list )>0|1

• A proc_def defines a processor with the proc_name as a unique name.
• If you do not explicitly define a processor for a derivative in an LSL file, the linker defines a processor with the same name as that derivative.
• A derivative_name refers to a defined derivative.
• A proc_def contains exactly one derivative statement.

8.3.11 SECTION PLACEMENT DEFINITION

section_definition ::= section_layout <space_ref>0|1
  <( locate_direction )>0|1
  { <section_statement> >=0 }

• A section definition inside a space definition does not have a space_ref.
• All global section definitions have a space_ref.

space_ref ::= <proc_name>0|1 : <core_name>0|1
  : space_name

• If more than one processor is present, the proc_name must be given for a global section layout.
• If the section layout refers to a processor that has more than one core, the core_name must be given in the space_ref.
• A proc_name refers to a defined processor.
- A core_name refers to a defined core.
- A space_name refers to a defined address space.

locate_direction ::= direction = direction

direction ::= low_to_high
             | high_to_low

- A section layout contains at most one direction statement.
- If you do not specify the direction statement, the locate direction of the section layout is low-to-high.

section_statement ::= simple_section_statement ;
                     | aggregate_section_statement

simple_section_statement ::= assignment
                          | select_section_statement
                          | special_section_statement

assignment ::= symbol_name assign_op expr

assign_op ::= =
           | :=

select_section_statement ::= select <section_name>0|1
                           <section_selections>0|1

- Either a section_name or at least one section_selection must be defined.

section_selections ::= ( section_selection
                        <, section_selection>=0 )

section_selection ::= attributes = < <+|-> attribute>0

- +attribute means: select all sections that have this attribute.
- -attribute means: select all sections that do not have this attribute.
special_section_statement
 ::= heap stack_name <size_spec>0|1
    | stack stack_name <size_spec>0|1
    | copytable
    | reserved <section_name>0|1
    | <reserved_specs>0|1

• Special sections cannot be selected in load-time groups.

size_spec ::= ( size = expr )

reserved_specs ::= ( reserved_spec
                  , reserved_spec >=0 )

reserved_spec ::= attributes
                | fill_spec
                | size = expr
                | alloc_allowed = absolute

• If a reserved section has attributes r, rw, x, rx or rwx, and no fill pattern is defined, the section is filled with zeros. If no attributes are set, the section is created as a scratch section (attributes ws, no image).

aggregate_section_statement
 ::= { <section_statement>=0 }
    | group_descr
    | if_statement
    | section_creation_statement

group_descr ::= group <group_name>0|1
               | <( group_specs )>=0|1
               | section_statement

• No two groups for an address space can have the same
  group_name.

group_specs ::= group_spec , group_spec >=0

group_spec ::= group_alignment
              | attributes
              | group_load_address
              | fill <= fill_values>=0|1
              | group_page
              | group_run_address
              | group_type
              | allow_cross_references
The allow-cross-references property is only allowed for overlay groups.
- Sub groups inherit all properties from a parent group.

```
group_alignment ::= align = expr
attributes ::= attributes = <attribute>=1

group_load_address ::= load_addr <= load_or_run_addr>0|1
fill_spec ::= fill = fill_values
fill_values ::= expr
| [ expr <, expr>=0 ]
group_page ::= page <= expr>0|1


group_run_address ::= run_addr <= load_or_run_addr>0|1


group_type ::= clustered
| contiguous
| ordered
| overlay
```

- For non-contiguous groups, you can only specify group_alignment and attributes.
- The overlay keyword also sets the contiguous property.
- The clustered property cannot be set together with contiguous or ordered on a single group.

```
attribute ::= r // read-only sections
| w // read/write sections
| x // executable code sections
| i // initialized sections
| s // scratch sections
| b // blanked (cleared) sections

load_or_run_addr ::= addr_absolute
| addr_range < addr_range>=0

addr_absolute ::= expr
| memory_reference [ expr ]


```

- An absolute address can only be set on ordered groups.

```
addr_range ::= [ expr .. expr ]
| memory_reference
| memory_reference [ expr .. expr ]
```
• The parent of a group with an \texttt{addr\_range} or \texttt{page} restriction cannot be \texttt{ordered}, \texttt{contiguous} or \texttt{clustered}.

\begin{verbatim}
memory\_reference ::= mem : <proc\_name :>\textsuperscript{0|1} \\
                    <core\_name :>\textsuperscript{0|1} mem\_name
\end{verbatim}

• A \texttt{proc\_name} refers to a defined processor.
• A \texttt{core\_name} refers to a defined core.
• A \texttt{mem\_name} refers to a defined memory.

\begin{verbatim}
if\_statement ::= if ( expr ) section\_statement \\
                 <else section\_statement>\textsuperscript{0|1}
\end{verbatim}

\begin{verbatim}
section\_creation\_statement \\
 ::= section section\_name \\
   ( <section\_spec>\textsuperscript{0|1} ) \\
   { <select\_section\_statement ;>\textsuperscript{0} } 
\end{verbatim}

\begin{verbatim}
section\_spec ::= attributes \\
               | fill\_spec \\
               | size = expr
\end{verbatim}

### 8.4 EXPRESSION EVALUATION

Only \textit{constant} expressions are allowed, including sizes, but not addresses, of sections in object files.

All expressions are evaluated with 64-bit precision integer arithmetic. The result of an expression can be absolute or relocatable. A symbol you assign is created as an absolute symbol.
8.5 SEMANTICS OF THE ARCHITECTURE DEFINITION

Keywords in the architecture definition

architecture
extends
endianness big little
bus
mau
width
map
space
id
mau
align
page_size
stack
    min_size
grows low_to_high high_to_low
align
fixed
heap
    min_size
grows low_to_high high_to_low
align
fixed
copytable
    align
    copy_unit
dest
start_address
    run_addr
symbol
map
    map
dest bus space
    dest_dbits
dest_offset
    size
    src_dbits
    src_offset

\[\text{••••••••}8/C00465\text{••••••••}8\]
8.5.1 DEFINING AN ARCHITECTURE

With the keyword architecture you define an architecture and assign a unique name to it. The name is used to refer to it at other places in the LSL file:

```lsl
architecture name
{
    definitions
}
```

If you are defining multiple core architectures that show great resemblance, you can define the common features in a parent core architecture and extend this with a child core architecture that contains specific features. The child inherits all features of the parent. With the keyword extends you create a child core architecture:

```lsl
architecture name_child_arch extends name_parent_arch
{
    definitions
}
```

A core architecture can have any number of parameters. These are identifiers which get values assigned on instantiation or extension of the architecture. You can use them in any expression within the core architecture. Parameters can have default values, which are used when the core architecture is instantiated with less arguments than there are parameters defined for it. When you extend a core architecture you can pass arguments to the parent architecture. Arguments are expressions that set the value of the parameters of the sub-architecture.

```lsl
architecture name_child_arch (parm1, parm2=1)
    extends name_parent_arch (arguments)
{
    definitions
}
```
8.5.2 DEFINING INTERNAL BUSES

With the `bus` keyword you define a bus (the combination of data and corresponding address bus). The bus name is used to identify a bus and does not conflict with other identifiers. Bus descriptions in an architecture definition or derivative definition define internal buses. Some internal buses are used to communicate with the components outside the core or processor. Such buses on a processor have physical pins reserved for the number of bits specified with the `width` statements.

- The `mau` field specifies the MAU size (Minimum Addressable Unit) of the data bus. This field is required.
- The `width` field specifies the width (number of address lines) of the data bus. The default value is the MAU size.
- The `map` keyword specifies how this bus maps onto another bus (if so). Mappings are described in section 8.5.4, Mappings.

```c
bus bus_name
{
    mau = 8;
    width = 8;
    map ( map_description );
}
```

8.5.3 DEFINING ADDRESS SPACES

With the `space` keyword you define a logical address space. The space name is used to identify the address space and does not conflict with other identifiers.

- The `id` field defines how the addressing space is identified in object files. In general, each address space has a unique ID. The linker locates sections with a certain ID in the address space with the same ID. This field is required. In IEEE this ID is specified explicitly for sections and symbols, ELF sections map by default to the address space with ID 1. Sections with one of the special names defined in the ABI (Application Binary Interface) may map to different address spaces.
- The `mau` field specifies the MAU size (Minimum Addressable Unit) of the space. This field is required.
- The `align` value must be a power of two. The linker uses this value to compute the start addresses when sections are concatenated. An align value of \( n \) means that objects in the address space have to be aligned on \( n \) MAUs.
The **page_size** field sets the page size in MAUs for the address space. It must be a power of 2. The default page size is 1. See also the **page** keyword in subsection *Locating a group* in section 8.8.2, *Creating and Locating Groups of Sections*.

The **map** keyword specifies how this address space maps onto an internal bus or onto another address space. Mappings are described in section 8.5.4, *Mappings*.

### Stacks and heaps

- The **stack** keyword defines a stack in the address space and assigns a name to it. The architecture definition must contain at least one stack definition. Each stack of a core architecture must have a unique name. See also the **stack** keyword in section 8.8.3, *Creating or Modifying Special Sections*.

The stack is described in terms of a minimum size (**min_size**) and the direction in which the stack grows (**grows**). This can be either from **low_to_high** addresses (stack grows upwards, this is the default) or from **high_to_low** addresses (stack grows downwards). The **min_size** is required.

By default, the linker tries to maximize the size of the stacks and heaps. After locating all sections, the largest remaining gap in the space is used completely for the stacks and heaps. If you specify the keyword **fixed**, you can disable this so-called 'balloon behavior'. The size is also fixed if you used a stack or heap in the software layout definition in a restricted way. For example when you override a stack with another size or select a stack in an ordered group with other sections.

Optionally you can specify an alignment for the stack with the argument **align**. This alignment must be equal or larger than the alignment that you specify for the address space itself.

- The **heap** keyword defines a heap in the address space and assigns a name to it. The definition of a heap is similar to the definition of a stack. See also the **heap** keyword in section 8.8.3, *Creating or Modifying Special Sections*.

See section 8.8, *Semantics of the Section Layout Definition* for information on creating and placing stack sections.
Copy tables
- The copytable keyword defines a copy table in the address space. The content of the copy table is created by the linker and contains the start address and size of all sections that should be initialized by the startup code. If the architecture definition contains more than one address space, you must define exactly one copy table in one of the address spaces. If the architecture definition contains only one address space, the copy table definition is optional.

Optionally you can specify an alignment for the copy table with the argument align. This alignment must be equal or larger than the alignment that you specify for the address space itself. If smaller, the alignment for the address space is used.

The copy_unit argument specifies the size in MAUs of information chunks that are copied. If you do not specify the copy unit, the MAU size of the address space itself is used.

The dest argument specifies the destination address space that the code uses for the copy table. The linker uses this information to generate the correct addresses in the copy table. The memory into where the sections must be copied at run-time, must be accessible from this destination space.

Start address
- The start_address keyword specifies the start address for the position where the C startup code is located. When a processor is reset, it initializes its program counter to a certain start address, sometimes called the reset vector. In the architecture definition, you must specify this start address in the correct address space in combination with the name of the label in the application code which must be located here.

The run_addr argument specifies the start address (reset vector). If the core starts executing using an entry from a vector table, and directly jumps to the start label, you should omit this argument.

The symbol argument specifies the name of the label in the application code that should be located at the specified start address. The symbol argument is required. The linker will resolve the start symbol and use its value after locating for the start address field in IEEE-695 files and Intel Hex files. If you also specified the run_addr argument, the start symbol (label) must point to a section. The linker locates this section such that the start symbol ends up on the start address.
space space_name
{
    id = 1;
    mau = 8;
    align = 8;
    page_size = 1;
    stack name (min_size = 1k, grows = low_to_high);
    start_address (run_addr = 0x0000,
        symbol = "start_label")
    map (map_description);
}

8.5.4 MAPPINGS

You can use a mapping when you define a space, bus or memory. With the map field you specify how addresses from the source (space, bus or memory) are translated to addresses of a destination (space, bus). The following mappings are possible:

- space => space
- space => bus
- bus => bus
- memory => bus

With a mapping you specify a range of source addresses you want to map (specified by a source offset and a size), the destination to which you want to map them (a bus or another address space), and the offset address in the destination.

- The dest argument specifies the destination. This can be a bus or another address space (only for a space to space mapping). This argument is required.
- The src_offset argument specifies the offset of the source addresses. In combination with size, this specifies the range of address that are mapped. Default the source offset is 0x0000.
- The size argument specifies the number of addresses that are mapped. This argument is required.
- The dest_offset argument specifies the position in the destination to which the specified range of addresses is mapped. Default the destination offset is 0x0000.
If you are mapping a bus to another bus, the number of data lines of each bus may differ. In this case you have to specify a range of source data lines you want to map (`src_dbits = begin..end`) and the range of destination data lines you want to map them to (`dest_dbits = first..last`).

- The `src_dbits` argument specifies a range of data lines of the source bus. Default all data lines are mapped.
- The `dest_dbits` argument specifies a range of data lines of the destination bus. Default, all data lines from the source bus are mapped on the data lines of the destination bus (starting with line 0).

**From space to space**

If you map an address space to another address space (nesting), you can do this by mapping the subspace to the containing larger space. In this example a small space of 64k is mapped on a large space of 16M.

```plaintext
space small
{
  id = 2;
  mau = 4;
  map (src_offset = 0, dest_offset = 0,
       dest = space : large, size = 64k);
}
```

**From space to bus**

All spaces that are not mapped to another space must map to a bus in the architecture:

```plaintext
space large
{
  id = 1;
  mau = 4;
  map (src_offset = 0, dest_offset = 0,
       dest = bus:bus_name, size = 16M);
}
```

**From bus to bus**

The next example maps an external bus called `e_bus` to an internal bus called `i_bus`. This internal bus resides on a core called `mycore`. The source bus has 16 data lines whereas the destination bus has only 8 data lines. Therefore, the keywords `src_dbits` and `dest_dbits` specify which source data lines are mapped on which destination data lines.
architecture mycore
{
  bus i_bus
  {
    mau = 4;
  }

  space i_space
  {
    map (dest=bus:i_bus, size=256);
  }
}

bus e_bus
{
  mau = 16;
  width = 16;
  map (dest = bus:mycore:i_bus, 
       src_dbits = 0..7, dest_dbits = 0..7 )
}

It is not possible to map an internal bus to an external bus.
8.6 SEMANTICS OF THE DERIVATIVE DEFINITION

Keywords in the derivative definition

derivative
extends
core
  architecture
bus
  mau
  width
  map
memory
  type  rom  ram  n ram
  mau
  size
  speed
  map

map
  dest  bus  space
  dest_dbits
  dest_offset
  size
  src_dbits
  src_offset

8.6.1 DEFINING A DERIVATIVE

With the keyword `derivative` you define a derivative and assign a unique name to it. The name is used to refer to it at other places in the LSL file:

```markdown
derivative name
{
  definitions
}
```
If you are defining multiple derivatives that show great resemblance, you can define the common features in a parent derivative and extend this with a child derivative that contains specific features. The child inherits all features of the parent (cores and memories). With the keyword `extends` you create a child derivative:

```
    derivative name_child_deriv extends name_parent_deriv
     {
       definitions
     }
```

As with a core architecture, a derivative can have any number of parameters. These are identifiers which get values assigned on instantiation or extension of the derivative. You can use them in any expression within the derivative definition.

```
    derivative name_child_deriv (parm1, parm2=1)
       extends name_parent_derivh (arguments)
     {
       definitions
     }
```

### 8.6.2 Instantiating Core Architectures

With the keyword `core` you instantiate a core architecture in a derivative.

- With the keyword `architecture` you tell the linker that the given core has a certain architecture. The architecture name refers to an existing architecture definition in the same LSL file.

For example, if you have two cores (called `mycore_1` and `mycore_2`) that have the same architecture (called `mycorearch`), you must instantiate both cores as follows:

```
    core mycore_1
    {
      architecture = mycorearch;
    }
    
    core mycore_2
    {
      architecture = mycorearch;
    }
```
If the architecture definition has parameters you must specify the arguments that correspond with the parameters. For example `mycorearch1` expects two parameters which are used in the architecture definition:

```c
core mycore
{
    architecture = mycorearch1 (1,2);
}
```

### 8.6.3 DEFINING INTERNAL MEMORY AND BUSES

With the `memory` keyword you define physical memory that is present on the target board. The memory name is used to identify the memory and does not conflict with other identifiers. It is common to define internal memory (on–chip) in the derivative definition. External memory (off–chip memory) is usually defined in the board specification (See section 8.7.3, Defining External Memory and Buses).

- The `type` field specifies a memory type:
  - `rom`: read only memory
  - `ram`: random access memory
  - `nvram`: non volatile ram

  The optional `reserved` qualifier before the memory type, tells the linker not to locate any section in the memory by default. You can locate sections in such memories using an absolute address or range restriction (see subsection Locating a group in section 8.8.2, Creating and Locating Groups of Sections).

- The `mau` field specifies the MAU size (Minimum Addressable Unit) of the memory. This field is required.
- The `size` field specifies the size in MAU of the memory. This field is required.
- The `speed` field specifies a symbolic speed for the memory (0..4): 0 is the fastest, 4 the slowest. The linker uses the relative speed of the memories in such a way, that optimal speed is achieved. The default speed is 1.
- The `map` field specifies how this memory maps onto an (internal) bus. Mappings are described in section 8.5.4, Mappings.
memory mem_name
{
    type = rom;
    mau = 8;
    size = 64k;
    speed = 2;
    map ( map_description );
}

With the `bus` keyword you define a bus in a derivative definition. Buses are described in section 8.5.2, *Defining Internal Buses*. 
8.7 SEMANTICS OF THE BOARD SPECIFICATION

Keywords in the board specification

- processor
- derivative
- bus
- mau
- width
- map
- memory
- type (reserved, rom, ram, nvram)
- mau
- size
- speed
- map
- map
- dest (bus, space)
- dest_dbits
- dest_offset
- size
- src_dbits
- src_offset

8.7.1 DEFINING A PROCESSOR

If you have a target board with multiple processors that have the same derivative, you need to instantiate each individual processor in a processor definition. This information tells the linker which processor has which derivative and enables the linker to distinguish between the present processors.

If you use processors that all have a unique derivative, you may omit the processor definitions. In this case the linker assumes that for each derivative definition in the LSL file there is one processor. The linker uses the derivative name also for the processor.
With the keyword `processor` you define a processor. You can freely choose the processor name. The name is used to refer to it at other places in the LSL file:

```plaintext
processor proc_name
{
    processor definition
}
```

### 8.7.2 Instantiating Derivatives

With the keyword `derivative` you tell the linker that the given processor has a certain derivative. The derivative name refers to an existing derivative definition in the same LSL file.

For examples, if you have two processors on your target board (called `myproc_1` and `myproc_2`) that have the same derivative (called `myderiv`), you must instantiate both processors as follows:

```plaintext
processor myproc_1
{
    derivative = myderiv;
}

processor myproc_2
{
    derivative = myderiv;
}
```

If the derivative definition has parameters you must specify the arguments that correspond with the parameters. For example `myderiv1` expects two parameters which are used in the derivative definition:

```plaintext
processor myproc
{
    derivative = myderiv1 (2,4);
}
```
8.7.3 DEFINING EXTERNAL MEMORY AND BUSSES

It is common to define external memory (off-chip) and external buses at the global scope (outside any enclosing definition). Internal memory (on-chip memory) is usually defined in the scope of a derivative definition.

With the keyword memory you define physical memory that is present on the target board. The memory name is used to identify the memory and does not conflict with other identifiers. If you define memory parts in the LSL file, only the memory defined in these parts is used for placing sections.

If no external memory is defined in the LSL file and if the linker option to allocate memory on demand is set then the linker will assume that all virtual addresses are mapped on physical memory. You can override this behavior by specifying one or more memory definitions.

```plaintext
memory mem_name
{
    type = rom;
    mau = 8;
    size = 64k;
    speed = 2;
    map ( map_description );
}
```

For a description of the keywords, see section 8.6.3, Defining Internal Memory and Buses.

With the keyword bus you define a bus (the combination of data and corresponding address bus). The bus name is used to identify a bus and does not conflict with other identifiers. Bus descriptions at the global scope (outside any definition) define external buses. These are buses that are present on the target board.

```plaintext
bus bus_name
{
    mau = 8;
    width = 8;
    map ( map_description );
}
```

For a description of the keywords, see section 8.5.2, Defining Internal Buses.
You can connect off-chip memory to any derivative: you need to map the off-chip memory to a bus and map that bus on the internal bus of the derivative you want to connect it to.
8.8 SEMANTICS OF THE SECTION LAYOUT DEFINITION

Keywords in the section layout definition

- section_layout
- direction: low_to_high, high_to_low
- group
  - align
  - attributes: + -, r w x b i s
- fill
- ordered
- clustered
- contiguous
- overlay
- allow_cross_references
- load_addr
  - mem
- run_addr
  - mem
- page
- select
- heap
  - size
- stack
  - size
- reserved
  - size
  - attributes: r w x
- fill
  - alloc_allowed
  - absolute
- copytable
- section
  - size
  - attributes: r w x
  - fill

if
else
8.8.1 DEFINING A SECTION LAYOUT

With the keyword `section_layout` you define a section layout for exactly one address space. In the section layout you can specify how input sections are placed in the address space, relative to each other, and what the absolute run and load addresses of each section will be.

You can define one or more section definitions. Each section definition arranges the sections in one address space. You can precede the address space name with a processor name and/or core name, separated by colons. You can omit the processor name and/or the core name if only one processor is defined and/or only one core is present in the processor. A reference to a space in the only core of the only processor in the system would look like `::my_space`. A reference to a space of the only core on a specific processor in the system could be `my_chip::my_space`. The next example shows a section definition for sections in the `my_space` address space of the processor called `my_chip`:

```plaintext
section_layout my_chip::my_space ( locate_direction )
{
    section statements
}
```

With the optional keyword `direction` you specify whether the linker starts locating sections from `low_to_high` (default) or from `high_to_low`. In the second case the linker starts locating sections at the highest addresses in the address space but preserves the order of sections when necessary (one processor and core in this example).

```plaintext
section_layout ::my_space ( direction = high_to_low )
{
    section statements
}
```

If you do not explicitly tell the linker how to locate a section, the linker decides on the basis of the section attributes in the object file and the information in the architecture definition and memory parts where to locate the section.
8.8.2 CREATING AND LOCATING GROUPS OF SECTIONS

Sections are located per group. A group can contain one or more (sets of) input sections as well as other groups. Per group you can assign a mutual order to the sets of sections and locate them into a specific memory part.

```c
    group ( group_specifications )
    {
        section_statements
    }
```

With the `section_statements` you generally select sets of sections to form the group. This is described in subsection Selecting sections for a group.

Instead of selecting sections, you can also modify special sections like stack and heap or create a reserved section. This is described in section 8.8.3, Creating or Modifying Special Sections.

With the `group_specifications` you actually locate the sections in the group. This is described in subsection Locating a group.

Selecting sections for a group

With the `select` keyword you can select one or more sections for the group. You can select a section by name or by attributes. If you select a section by name, you can use a wildcard pattern:

<table>
<thead>
<tr>
<th>Wildcard</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>*</code></td>
<td>matches with all section names</td>
</tr>
<tr>
<td><code>?</code></td>
<td>matches with a single character in the section name</td>
</tr>
<tr>
<td><code>\</code></td>
<td>takes the next character literally</td>
</tr>
<tr>
<td><code>[abc]</code></td>
<td>matches with a single ‘a’, ‘b’ or ‘c’ character</td>
</tr>
<tr>
<td><code>[a–z]</code></td>
<td>matches with any single character in the range ‘a’ to ‘z’</td>
</tr>
</tbody>
</table>

```c
    group ( ... )
    {
        select ".mysection";
        select "*";
    }
```

The first `select` statement selects the section with the name ".mysection". The second `select` statement selects all sections that were not selected yet.
A section is selected by the first `select` statement that matches, in the union of all section layouts for the address space. Global section layouts are processed in the order in which they appear in the LSL file. Internal core architecture section layouts always take precedence over global section layouts.

- The `attributes` field selects all sections that carry (or do not carry) the given attribute. With `+attribute` you select sections that have the specified attribute set. With `-attribute` you select sections that do not have the specified attribute set. You can specify one or more of the following attributes:
  - `r` readable sections
  - `w` readable/writable sections
  - `x` executable sections
  - `i` initialized sections
  - `b` sections that should be cleared at program startup
  - `s` scratch sections (not cleared and not initialized)

To select all read-only sections:

```plaintext
{ select (attributes = +r); }
```

Keep in mind that all section selections are restricted to the address space of the section layout in which this group definition occurs.

**Locating a group**

```plaintext
group group_name ( group Specifications )
{
  section_statements
}
```

With the `group Specifications` you actually define how the linker must locate the group. You can roughly define three things: 1) assign properties to the group like alignment and read/write attributes, 2) define the mutual order in the address space for sections in the group and 3) restrict the possible addresses for the sections in a group.
The linker creates labels that allow you to refer to the begin and end address of a group from within the application software. Labels \_lc_gb\_group\_name and \_lc_ge\_group\_name mark the begin and end of the group respectively, where the begin is the lowest address used within this group and the end is the highest address used. Notice that a group not necessarily occupies all memory between begin and end address. The given label refers to where the section is located at run-time (versus load-time).

1. Assign properties to the group like alignment and read/write attributes. These properties are assigned to all sections in the group (and subgroups) and override the attributes of the input sections.
   - The align field tells the linker to align all sections in the group and the group as a whole according to the align value. Default the linker uses the largest alignment constraint of either the input sections or the alignment of the address space.
   - The attributes field tells the linker to assign one or more attributes to the sections in the group. Default the linker uses the attributes of the input sections. The list of available attributes is the same as described above for the selection of sections.

2. Define the mutual order of the sections in the group.
   By default, a group is unrestricted which means that the linker has total freedom to place the sections of the group in the address space.
   - The ordered keyword tells the linker to locate the sections in the same order in the address space as they appear in the group (but not necessarily adjacent).

Suppose you have an ordered group that contains the sections 'A', 'B' and 'C'. Default the linker places the sections in the address space like 'A' – 'B' – 'C', where section 'A' gets the lowest possible address. With direction=high_to_low in the section_layout space properties, the linker places the sections in the address space like 'C' – 'B' – 'A', where section 'A' gets the highest possible address.

- The contiguous keyword tells the linker to locate the sections in the group in a single address range. Within a contiguous group the input sections are located in arbitrary order, however the group occupies one contiguous range of memory. Due to alignment of sections there can be 'alignment gaps' between the sections.
When you define a group that is both ordered and contiguous, this is called a sequential group. In a sequential group the linker places sections in the same order in the address space as they appear in the group and it occupies a contiguous range of memory.

- The clustered keyword tells the linker to locate the sections in the group in a number of contiguous blocks. It tries to keep the number of these blocks to a minimum. If enough memory is available, the group will be located as if it was specified as contiguous. Otherwise, it gets split into two or more blocks.

If a contiguous or clustered group contains alignment gaps, the linker can locate sections that are not part of the group in these gaps. To prevent this, you can use the fill keyword. If the group is located in RAM, the gaps are treated as reserved (scratch) space. If the group is located in ROM, the alignment gaps are filled with zeros by default. You can however change the fill pattern by specifying a bit pattern. The result of the expression, or list of expressions, is used as values to write to memory, each in MAU.

- The overlay keyword tells the linker to overlay the sections in the group. The linker places all sections in the address space using a contiguous range of addresses. (Thus an overlay group is automatically also a contiguous group.) To overlay the sections, all sections in the overlay group share the same run–time address.

For each input section within the overlay the linker automatically defines two symbols. The symbol _lc_cb_section_name is defined as the load–time start address of the section. The symbol _lc_ce_section_name is defined as the load–time end address of the section. C (or assembly) code may be used to copy the overlaid sections.

If sections in the overlay group contain references between groups, the linker reports an error. The keyword allow_cross_references tells the linker to accept cross–references. Normally, it does not make sense to have references between sections that are overlaid.
group ovl (overlay)
{
    group a
    {
        select "my_ovl_p1";
        select "my_ovl_p2";
    }
    group b
    {
        select "my_ovl_q1";
    }
}

It may be possible that one of the sections in the overlay group already has been defined in another group where it received a load–time address. In this case the linker does not overrule this load–time address and excludes the section from the overlay group.

3. Restrict the possible addresses for the sections in a group. The load–time address specifies where the group’s elements are loaded in at download time. The run–time address specifies where sections are located at run–time, that is when the program is executing. If you do not explicitly restrict the address in the LSL file, the linker assigns addresses to the sections based on the restrictions relative to other sections in the LSL file and section alignments. The program is responsible for copying overlay sections at appropriate moment from its load–time location to its run–time location (this is typically done by the startup code).

- The `run_addr` keyword defines the run–time address. If the run–time location of a group is set explicitly, the given order between groups specify whether the run–time address propagates to the parent group or not. The location of the sections a group can be restricted either to a single absolute address, or to a number of address ranges. With an expression you can specify that the group should be located at the absolute address specified by the expression:

  ```
  group (run_addr = 0xa00f0000)
  ```

  You can use the `[offset]` variant to locate the group at the given absolute offset in memory:

  ```
  group (run_addr = mem:A[0x1000])
  ```

  A range can be an absolute space address range, written as `expr .. expr`, a complete memory device, written as `mem:mem_name`, or a memory address range, `mem:mem_name[ expr .. expr ]`
group (run_addr = mem:my_dram)

You can use the '|' to specify an address range of more than one physical memory device:

group (run_addr = mem:A | mem:B)

- The load_addr keyword changes the meaning of the section selection in the group: the linker selects the load-time ROM copy of the named section(s) instead of the regular sections. Just like run_addr you can specify an absolute address or an address range.

The load_addr keyword itself (without an assignment) specifies that the group’s position in the LSL file defines its load-time address.

```plaintext
group (load_addr)
select "mydata"; // select ROM copy of mydata:
// "[mydata]"
```

The load-time and run-time addresses of a group cannot be set at the same time. If the load-time property is set for a group, the group (only) restricts the positioning at load-time of the group’s sections. It is not possible to set the address of a group that has a not-unrestricted parent group.

The properties of the load-time and run-time start address are:

- At run-time, before using an element in an overlay group, the application copies the sections from their load location to their run-time location, but only if these two addresses are different. For non-overlay sections this happens at program start-up.
- The start addresses cannot be set to absolute values for unrestricted groups.
- For non-overlay groups that do not have an overlay parent, the load-time start address equals the run-time start address.
- For any group, if the run-time start address is not set, the linker selects an appropriate address.

For overlays, the linker reserves memory at the run-time start address as large as the largest element in the overlay group.

- The page keyword tells the linker to place the group in one page. Instead of specifying a run-time address, you can specify a page and optional a page number. Page numbers start from zero. If you omit the page number, the linker chooses a page.
The **page** keyword refers to pages in the address space as defined in the architecture definition. See also the **page** keyword in section 8.5.3, *Defining Address Spaces*.

```c
  group (page, ...)  
  group (page = 3, ...)
```

### 8.8.3 CREATING OR MODIFYING SPECIAL SECTIONS

Instead of selecting sections, you can also create a reserved section or an output section or modify special sections like a stack or a heap. Because you cannot define these sections in the input files, you must use the linker to create them.

**Stack**

- The **stack** keyword tells the linker to reserve memory for the stack. The name for the stack section refers to the stack as defined in the architecture definition. If no name was specified in the architecture definition, the default name is **stack**.

  With the keyword **size** you can specify the size for the stack. If the **size** is not specified, the linker uses the size given by the **min_size** argument as defined for the stack in the architecture definition. Normally the linker automatically tries to maximize the size, unless you specified the **fixed** keyword.

```c
  group ( ...)  
  {  
    stack "mystack" ( size = 2k );  
  }
```

The linker creates two labels to mark the begin and end of the stack, **_lc_ub_stack_name** for the begin of the stack and **_lc_ue_stack_name** for the end of the stack. The linker allocates space for the stack when there is a reference to either of the labels.

See also the **stack** keyword in section 8.5.3, *Defining Address Spaces*. 
**Heap**

- The `heap` keyword tells the linker to reserve a dynamic memory range for the `malloc()` function. Optionally you can assign a name to the heap section. With the keyword `size` you can change the size for the heap. If the `size` is not specified, the linker uses the size given by the `min_size` argument as defined for the heap in the architecture definition. Normally the linker automatically tries to maximize the size, unless you specified the `fixed` keyword.

```c
{ 
  heap "myheap" ( size = 2k );
}
```

The linker creates two labels to mark the begin and end of the heap, `_lc_ub_heap_name` for the begin of the heap and `_lc_ue_heap_name` for the end of the heap. The linker allocates space for the heap when a reference to either of the section labels exists in one of the input object files.

**Reserved section**

- The `reserved` keyword tells the linker to create an area or section of a given size. The linker will not locate any other sections in the memory occupied by a reserved section, with some exceptions. Optionally you can assign a name to a reserved section. With the keyword `size` you can specify a size for a given reserved area or section.

```c
{ 
  reserved "myreserved" ( size = 2k );
}
```

The optional `fill` field contains a bit pattern that the linker writes to all memory addresses that remain unoccupied during the locate process. The result of the expression, or list of expressions, is used as values to write to memory, each in MAU. The first MAU of the fill pattern is always the first MAU in the section.

By default, no sections can overlap with a reserved section. With `alloc_allowed=absolute` sections that are located at an absolute address due to an absolute group restriction can overlap a reserved section.
With the **attributes** field you can set the access type of the reserved section. The linker locates the reserved section in its space with the restrictions that follow from the used attributes, r, w or x or a valid combination of them. The allowed attributes are shown in the following table. A value between < and > in the table means this value is set automatically by the linker.

<table>
<thead>
<tr>
<th>attributes</th>
<th>filled</th>
<th>access</th>
<th>memory</th>
<th>content</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>yes</td>
<td></td>
<td>&lt;rom&gt;</td>
<td>executable</td>
</tr>
<tr>
<td>r</td>
<td>yes</td>
<td>r</td>
<td>&lt;rom&gt;</td>
<td>data</td>
</tr>
<tr>
<td>r</td>
<td>no</td>
<td>r</td>
<td>&lt;rom&gt;</td>
<td>scratch</td>
</tr>
<tr>
<td>rx</td>
<td>yes</td>
<td>r</td>
<td>&lt;rom&gt;</td>
<td>executable</td>
</tr>
<tr>
<td>rw</td>
<td>yes</td>
<td>rw</td>
<td>&lt;ram&gt;</td>
<td>data</td>
</tr>
<tr>
<td>rw</td>
<td>no</td>
<td>rw</td>
<td>&lt;ram&gt;</td>
<td>scratch</td>
</tr>
<tr>
<td>rwx</td>
<td>yes</td>
<td>rw</td>
<td>&lt;ram&gt;</td>
<td>executable</td>
</tr>
</tbody>
</table>

If you do not specify any attributes, the linker will reserve the given number of maus, no matter what type of memory lies beneath. If you do not specify a fill pattern, no section is generated.

The linker creates two labels to mark the begin and end of the section, `__lc_ub_name` for the start, and `__lc_ue_name` for the end of the reserved section.

**Output sections**

- The **section** keyword tells the linker to accumulate sections obtained from object files (“input sections”) into an output section of a fixed size in the locate phase. You can select the input sections with **select** statements. With the keyword **size** you specify the size of the output section.
The **fill** field contains a bit pattern that the linker writes to all unused space in the output section. When all input sections have an image (code/data) you must specify a fill pattern. If you do not specify a fill pattern, all input sections must be scratch sections. The fill pattern is aligned at the start of the output section.

As with a reserved section you can use the **attributes** field to set the access type of the output section.

```plaintext
group ( ... )
{
  section "myoutput" ( size = 4k, attributes = rw,
                         fill = 0xaa )
  {
    select "myinput1";
    select "myinput2";
  }
}
```

The linker creates two labels to mark the begin and end of the section, `__lc_ub_name` for the start, and `__lc_ue_name` for the end of the output section.

**Copy table**

- The **copytable** keyword tells the linker to select a section that is used as **copy-table**. The content of the copy-table is created by the linker. It contains the start address and length of all sections that should be initialized by the startup code.

The linker creates two labels to mark the begin and end of the section, `__lc_ub_table` for the start, and `__lc_ue_table` for the end of the copy table. The linker generates a copy table when a reference to either of the section labels exists in one of the input object files.

### 8.8.4 CREATING SYMBOLS

You can tell the linker to create symbols before locating by putting assignments in the section layout definition. Symbol names are represented by double-quoted strings. Any string is allowed, but object files may not support all characters for symbol names. You can use two different assignment operators. With the simple assignment operator `=`, the symbol is created unconditionally. With the `:=` operator, the symbol is only created if it already exists as an undefined reference in an object file.
The expression that represents the value to assign to the symbol may contain references to other symbols. If such a referred symbol is a special section symbol, creation of the symbol in the left hand side of the assignment will cause creation of the special section.

```c
section_layout
{
    "_lc_bs" := "_lc_ub_stack";
    // when the symbol _lc_bs occurs in the object
    // file, the linker allocates space for the stack
}
```

### 8.8.5 CONDITIONAL GROUP STATEMENTS

Within a group, you can conditionally select sections or create special sections.

- With the **if** keyword you can specify a condition. The succeeding section statement is executed if the condition evaluates to **TRUE** (1).
- The optional **else** keyword is followed by a section statement which is executed in case the if-condition evaluates to **FALSE** (0).

```c
group ( ... )
{
    if ( size_of ( sect:.mysection ) < 2k )
        select "..mysection";
    else
        select "..othersection";
}
```
9.1 INTRODUCTION

Infineon Technologies regularly publishes microcontroller errata sheets reporting functional problems and deviations from the electrical specifications and timing specifications.

The TASKING TriCore software development tools provide solutions for a number of these functional problems in the TriCore architecture.

Support to deal with CPU functional problem is provided in three areas:

- Whenever possible and relevant, compiler bypasses will modify the code in order to avoid the identified erroneous code sequences;
- The TriCore assembler gives warnings for suspicious or erroneous code sequences;
- Ready-built, 'protected' standard C libraries with bypasses for all identified TriCore CPU functional problems are included in the toolchain.

This chapter lists a summary of identified functional problems which can be bypassed by the TASKING TriCore tool kit.

Please refer to the Infineon errata sheets for the TriCore architecture revision-step of your particular device, to check the need for applying any of these bypasses. Also refer to the Infineon errata sheets for a complete description of the CPU functional problems, as the workarounds listed below do not describe the functional problem itself.

With the TASKING C compiler and assembler command line options, pragmas and macro definitions you can enable or disable specific CPU functional problem bypasses.

To enable the compiler bypasses and assembler checks for all TriCore CPU TC112 problems (respectively TC113 problems) at once, use the command line option --silicon-bug=all-tc112 (respectively --silicon-bug=all-tc113)

To enable the bypasses from the embedded development environment (EDE):

1. From the Projects menu select Project Options...
2. Expand the Processor entry.
3. Select Bypasses. Depending on the target processor you have selected, this shows the bypasses for the TC1 v1.2 or TC1 v1.3.
9.2 CPU FUNCTIONAL PROBLEM BYPASSES

CPU_TC.013

Compiler and assembler option:
--silicon-bug=cpu-tc013

Assembler control:
$CPU_TC013 {on | off}

Assembler macro:
The assembler macro __CPU_TC013__ is defined if you specify option --silicon-bug=cpu-tc013.

Protected libraries to link:
lib\p\tc112\*.a

Compiler bypass:
To bypass this CPU functional problem, the C compiler generates a NOP16 instruction if a 16-bit load/store address register instruction (instructions: LD16.A en ST16.A) is followed by a lower context load/store instruction (instructions: LDLCX and STLCX).

Assembler check:
The assembler gives a warning if a 16-bit load/store address register instruction (instructions: LD16.A en ST16.A) is followed by a lower context load/store instruction (instructions: LDLCX and STLCX).

\textit{Wnum:} suspicious instruction concerning CPU functional defect TC013

You can suppress this warning with the option \textit{–wnum}. 
**CPU_TC.030**

**Compiler and assembler option:**

```sh
--silicon-bug=cpu-tc030
```

**Assembler control:**

```sh
$CPU_TC030 {on | off}
```

**Assembler macro:**

The assembler macro `__CPU_TC030__` is defined if you specify option
```sh
--silicon-bug=cpu-tc030.
```

**Protected libraries to link:**

```sh
lib\p\tc112\*.a
```

**Compiler bypass:**

To bypass this CPU functional problem, the C compiler generates an
```sh
ISYNC instruction prior to the LOOP instruction if the last instruction in the
loop is a DVSTEP or a DVSTEP.U.
```

**Assembler check:**

The assembler gives a warning for loops where the last instruction is a
```sh
DVSTEP or a DVSTEP.U:
```

```
Wnum: suspicious instruction concerning CPU functional
defect TC030
```

You can suppress this warning with the option `--wnum`. 
CPU_TC.031

Compiler and assembler option:

--silicon--bug=cpu-tc031

Assembler control:

$CPU_TC031 {on | off}

Assembler macro:

The assembler macro __CPU_TC031__ is defined if you specify option
--silicon--bug=cpu-tc031.

Protected libraries to link:

lib\p\tc112\*.a

Compiler bypass:

To bypass this CPU functional problem, the C compiler generates an ISYNC instruction prior to the LOOP instruction.

Assembler check:

The assembler gives a warning if the LOOP instruction is not preceded by an ISYNC instruction:

 Wnum: suspicious instruction concerning CPU functional defect TC031

You can suppress this warning with the option --wnum.
**CPU_TC.034**

**Compiler and assembler option:**

`--silicon-bug=cpu-tc034`

**Assembler control:**

`$CPU_TC034 {on | off}`

**Assembler macro:**

The assembler macro `__CPU_TC034__` is defined if you specify option `--silicon-bug=cpu-tc034`.

**Protected libraries to link:**

`lib\p\tc112\*.a`

**Compiler bypass:**

To bypass this CPU functional problem, the C compiler generates an ISYNC instruction after each DSYNC instruction.

**Assembler check:**

The assembler gives a warning if a DSYNC instruction is not followed by an ISYNC instruction:

`Wnum: suspicious instruction concerning CPU functional defect TC034`

You can suppress this warning with the option `--wnum`. 
CPU_TC.043

Linker option:

-D_CPU_TC043-

Linker bypass:

To bypass this CPU functional problem, a preprocessor define is used in the tc*.ls1 linker script files. The linker will not use the last 16 bytes of a segment.
CPU_TC.048

Compiler and assembler option:
---silicon-bug=cpu-tc048

Assembler control:
$CPU_TC048 {on | off}

Assembler macro:
The assembler macro __CPU_TC048__ is defined if you specify option
---silicon-bug=cpu-tc048.

Protected libraries to link:
lib\p\tc112\*.a

Compiler bypass:
To bypass this CPU functional problem, the C compiler generates a NOP
instruction before a JI or CALLI instruction when this instruction is not
directly preceded by either a NOP instruction or an integer instruction or a
MAC instruction. The compiler also generates a NOP instruction before a
RET and RET16 instruction if there is no or just one instruction before RET,
starting from the function entry point.

Assembler check:
The assembler gives a warning when a JI or CALLI instruction is not
directly preceded by a NOP instruction. The assembler also gives a
warning when there is no or just one instruction (not a NOP instruction)
between label and RET or RET16:

\textit{Wnum: suspicious instruction concerning CPU functional
defect TC048}

You can suppress this warning with the option \textit{--wnum}. 
CPU_TC.050

Compiler and assembler option:
--silicon-bug=cpu-tc050

Assembler control:
$CPU_TC050 {on | off}

Assembler macro:
The assembler macro __CPU_TC050__ is defined if you specify option
--silicon-bug=cpu-tc050.

Protected libraries to link:
lib\p\tc112\*.a

Compiler bypass:
To bypass this CPU functional problem, the C compiler generates a NOP
instruction between a multi-cycle integer instruction and a load
instruction.

Assembler check:
The assembler gives a warning if a multi-cycle integer instruction is
directly followed by a load instruction.:

\texttt{Wnum: suspicious instruction concerning CPU functional
defect TC050}

You can suppress this warning with the option \texttt{-wnum}. 

CPU_TC051

**Linker option:**

-D_CPU_TC051-

**Linker bypass:**

To bypass this CPU functional problem, a preprocessor define is used in the tc*.ls1 linker script files. The linker will use more than one section for context stores if the required CSA area exceeds the 4k. Each section will have a maximum size of 4k and will start on an 8k boundary.
CPU_TC.060

Compiler and assembler option:
--silicon-­bug=cpu-tc060

Assembler control:
$CPU_TC060 {on | off}

Assembler macro:
The assembler macro __CPU_TC060__ is defined if you specify option
--silicon-­bug=cpu-­tc060.

Protected libraries to link:
lib\p\tc112\*.a or lib\p\tc113\*.a

Compiler bypass:
To bypass this CPU functional problem, the C compiler generates a NOP
instruction between an LD.A / LD.DA instruction and a following LD.W /
LD.D instruction, even if an integer instruction occurs in between.

Assembler check:
The assembler gives a warning when an LD.A / LD.DA instruction is
directly followed by an LD.W / LD.D instruction, or when only an integer
instruction is in between.

\texttt{Wnum}: suspicious instruction concerning CPU functional
defect TC060

You can suppress this warning with the option \texttt{-wnum}. 
**TC112_COR1**

**Compiler and assembler option:**

--silicon-bug=cor1

**Assembler control:**

$TC112_COR1 {on | off}

**Assembler macro:**

The assembler macro __TC112_COR1__ is defined if you specify the option --silicon-bug=cor1.

**Protected libraries to link:**

lib\p\tc112\*.a

**Compiler bypass:**

To bypass this CPU functional problem, the C compiler generates an ISYNC instruction before each LOOP, LOOP16 and LOOPU instruction.

**Assembler check:**

The assembler gives a warning when the preceding instruction of a LOOP, LOOP16 or LOOPU instruction is not an ISYNC instruction:

`Wnum: suspicious instruction concerning CPU functional defect TC112_COR1`

You can suppress this warning with the option --wnum.
TC112_COR3

Linker option:

-D_TC112_COR3-

Linker bypass:
To bypass this CPU functional problem, a preprocessor define is used in the tc*.ls1 linker script files to restrict the size in the CSA absolute address mapping to 32Kb scratch pad RAM on the DMU.
**TC112_COR4**

**Compiler and assembler option:**

--silicon-bug=cor4

**Assembler control:**

$TC112_COR4 \{on | off\}

**Assembler macro:**

The assembler macro __TC112_COR4__ is defined if you specify the option --silicon-bug=cor4.

**Protected libraries to link:**

lib\p\tc112\*.a

**Compiler bypass:**

To bypass this CPU functional problem, the C compiler generates a NOP instruction between a (target) label and the instruction following it. This is done when the instruction directly uses an An register for either an effective address calculation or as the target of an indirect branch. Optionally an integer instruction may directly follow the label.

For example, a NOP will be inserted after the following labels:

```
A_label:
   ji a4
```

```
B_label:
   add d0, d1 ; integer instruction
   ji a4
```

**Assembler check:**

The assembler gives a warning for an instruction using 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, optionally with an integer instruction in between:

```
Wnum: suspicious instruction concerning CPU functional defect TC112_COR4
```

You can suppress this warning with the option -wnum.
TC112_COR6

Assembler option:
   --silicon-bug=cor6

Assembler control:
   $TC112_COR6 {on|off}

Assembler macro:
   The assembler macro __TC112_COR6__ is defined if you specify the option --silicon-bug=cor6.

Protected libraries to link:
   lib\p\tc112\*.a

Compiler bypass:
   There is no C compiler workaround required for this CPU functional problem, because the compiler does not generate CALLI instructions with a target address in register A11.

Assembler check:
   The assembler generates an error for instruction CALLI A11.
**TC112_COR7**

**Compiler and assembler option:**

```
--silicon-bug=cor7
```

**Assembler control:**

```
$TC112_COR7 {on | off}
```

**Assembler macro:**

The assembler macro `__TC112_COR7__` is defined if you specify the option `--silicon-bug=cor7`.

**Protected libraries to link:**

```
lib\p\tc112\*.a
```

**Compiler bypass:**

To bypass this CPU functional problem, the C compiler generates a NOP instruction at the very top of any subroutine that starts with a CALL instruction or that starts with an integer instruction or MAC instruction directly followed by a CALL instruction.

**Assembler check:**

The assembler gives a warning when the first instruction of a subroutine is a CALL instruction or an integer instruction or MAC instruction directly followed by a CALL instruction.

```
Wnum: suspicious instruction concerning CPU functional defect TC112_COR7
```

You can suppress this warning with the option `-wnum`. 
TC112_COR10

Compiler and assembler option:

`--silicon- bug=cor10`

Assembler control:

`$TC112_COR10 {on | off}`

Assembler macro:

The assembler macro `__TC112_COR10__` is defined if you specify the option `--silicon- bug=cor10`.

Protected libraries to link:

`lib\p\tc112\*.*a`

Compiler bypass:

To bypass this CPU functional problem, the C compiler avoids generation of store instructions that use a circular addressing mode with an offset value not equal to zero. An additional circular load instruction is generated with the required offset to post-increment the circular buffer pointer.

For example:

```
    st.w  [a6/a7+c]0,d15
    ld.w  d15,[a6/a7+c]4
```

Instead of:

```
    st.w  [a6/a7+c]4,d15
```

Assembler check:

The assembler gives a warning for store operations that use a circular addressing mode with an offset not equal to zero:

`Wnum: suspicious instruction concerning CPU functional defect TC112_COR10`

You can suppress this warning with the option `-Wnum`. 
TC112_COR13

Compiler and assembler option:
   --silicon-bug=cor13

Assembler control:
   $TC112_COR13 {on | off}

Assembler macro:
   The assembler macro __TC112_COR13__ is defined if you specify option
   --silicon-bug=cor13.

Protected libraries to link:
   lib\p\tc112\*.a

Compiler bypass:
   To bypass this CPU functional problem, the C compiler generates an
   ISYNC instruction prior to the LOOP instruction if the loop contains a
   single integer instruction that is a DVSTEP or a DVSTEP.U.

Assembler check:
   The assembler gives a warning for loops that contain a single integer
   instruction that is a DVSTEP or a DVSTEP.U:

   Wnum: suspicious instruction concerning CPU functional
defect TC112_COR13

   You can suppress this warning with the option -wnum.
TC112_COR14

Compiler option:

--silicon-bug=cor14

Protected libraries to link:

lib\p\tc112\*.a

Compiler bypass:

To bypass this CPU functional problem, the C compiler uses code that protects a divide instruction sequence against interrupts. Instead of generating inline divide code, the C compiler generates calls to run-time library functions that support divide operations with interrupt protection. Next skeleton code demonstrates the protective code used in these run-time library functions:

```c
;;
;; Save interrupt state and disable interrupts
;;
mfcr d0,#0xfe2c ; save ICR in d0
disable ; disable interrupts
```

divide instructions:

```c
;;
;; Restore interrupt state
;;
jz.t d0:8,disabled ; do not enable interrupts
enable ; when they were disabled

disabled:
```

The C run–time library modules involved are acircint.asm, dfrfr.asm, sdivmod.asm and udivmod.asm.

Assembler check:

An assembler check for this CPU functional problem is not available, because global interrupt enable state cannot be checked at assembly level.
TC112_COR15

Assembler option:
--silicon-bug=cor15

Assembler control:
$TC112_COR15 {on | off}

Assembler macro:
The assembler macro __TC112_COR15__ is if you specify the option
--silicon-bug=cor15.

Protected libraries to link:
lib\p\tc112\*.a (or add lib\src\cstart.asm to your project).

Compiler bypass:
There is no compiler bypass for this problem.

Assembler bypass:
To bypass this CPU functional problem, the assembler adds a macro to the
C startup code to disable the starvation protection by resetting the
BCUCON.SPE bit.
TC112_COR16

Compiler and assembler option:
   --silicon-bug=cor16

Linker option:
   -D_TC112_COR16_

Assembler control:
   TC112_COR16 {on | off}

Assembler macro:
   The assembler macro __TC112_COR16__ is defined if you specify the option --silicon-bug=cor16.

Protected libraries to link:
   lib\p\tc112\*.a (or add lib\src\cstart.asm to your project).

Compiler bypass:
   To bypass this CPU functional problem, the C compiler aligns circular qualified buffers to a quad-word boundary, and the compiler sizes all stack frames to an integral number of quad-words. See section 3.4.1, Circular Buffers in the User's Manual, for a description on how to declare a circular buffer.

Assembler bypass:
   To bypass this CPU functional problem, the assembler adds a macro to the C startup code to enable initialization of the stack pointers to a quad-word boundary.

Linker bypass:
   A preprocessor define is used in the tc*.ls1 linker script files to set the alignment of the user stack and the interrupt stack to a quad-word alignment.
TC112_COR17

Compiler and assembler option:

```
--silicon-bug=cor17
```

Assembler control:

```
$TC112_COR17 {on | off}
```

Assembler macro:

The assembler macro `__TC112_COR17__` is defined if you specify the option ``--silicon-bug=cor17``.

Protected libraries to link:

```
lib\p\tc112\*.a
```

Compiler bypass:

To bypass this CPU functional problem, the C compiler generates an ISYNC instruction after a DSYNC instruction. The C compiler only generates a DSYNC instruction when bypass TC113_CPU14 is enabled.

Assembler check:

The assembler gives a warning if a DSYNC is not followed by an ISYNC instruction:

```
Wnum: suspicious instruction concerning CPU functional defect TC112_COR17
```

You can suppress this warning with the option `--wnum`. 
TC113_CPU5

Compiler option:

--silicon-bug=cpu5

Protected libraries to link:

lib\p\tc113\*.a

Compiler bypass:

To bypass this CPU functional problem, the C compiler generates an ISYNC instruction before a loop body.

Example:

    isync
    _loop_start:
    ..
    ..
    loop a8, _loop_start

Assembler check:

This CPU functional problem does not cause a run–time problem, it is only a performance issue. Therefore no assembler checking is required to warn you for possible run–time problems.
TC113_CPU9

**Compiler and assembler option:**

\--silicon-\text{\texttt{-bug=cpu9}}

**Assembler control:**

\$TC113\_CPU9 \{on|off\}

**Assembler macro:**

The assembler macro \texttt{__TC113\_CPU9__} is defined if you specify the option \texttt{--silicon-\text{\texttt{-bug=cpu9}}}.

**Protected libraries to link:**

\texttt{lib\p\tc113\*.a}

**Compiler bypass:**

To bypass this CPU functional problem, the C compiler generates two NOP instructions after a DSYNC instruction. The C compiler only generates a DSYNC instruction when CPU functional problem bypass TC113\_CPU14 is enabled.

**Assembler check:**

The assembler gives a warning if a DSYNC is not followed by two NOP instructions:

\texttt{\textit{Wnum: suspicious instruction concerning CPU functional defect TC113\_CPU9}}

You can suppress this warning with the option \texttt{-wnum}.
TC113_CPU11

Compiler and assembler option:
--silicon-bug=cpu11

Pragma:
#pragma TC113_CPU11 [on | off | restore]

Assembler control:
$TC113_CPU11 {on | off}

Assembler macro:
The assembler macro __TC113_CPU11__ is defined if you specify the option --silicon-bug=cpu11.

Protected libraries to link:
lib\p\tc113\*.a

Compiler bypass:
To bypass this CPU functional problem, the C compiler generates a NOP instruction before a JI instruction. The compiler also generates a NOP before a RET and RET16 instruction if there is no or just one instruction before RET, starting from the function entry point.

Assembler check:
The assembler gives a warning when a JI instruction is not directly preceded by a NOP instruction. The assembler also gives a warning when there is no or just one instruction (not a NOP instruction) between label and RET or RET16:

Wnum: suspicious instruction concerning CPU functional defect TC113_CPU11

You can suppress this warning with the option -wnum.
**TC113_CPU13**

**Assembler option:**

--silicon-bug=cpu13

**Assembler macro:**

The assembler macro __TC113_CPU13__ is defined if you specify the option --silicon-bug=cpu13.

**Protected libraries to link:**

lib\p\tc113\*.a (or add lib\src\cstart.asm to your project).

**Compiler bypass:**

There is no compiler bypass for this problem.

**Assembler bypass:**

To bypass this CPU functional problem, the assembler adds a macro to the C startup code to enable the 16Kb D–Cache. The DCSIZ bits are set to 16Kb in the SFR register DMU_CON.
TC113_CPU14

Compiler and assembler option:
   --silicon-bug=cpu14

Assembler control:
   \$TC113_CPU14 \{on | off\}

Assembler macro:
   The assembler macro \_TC113_CPU14\_ is defined if you specify the option --silicon-bug=cpu14.

Protected libraries to link:
   lib\p\tc113\*.a

Compiler bypass:
   To bypass this CPU functional problem, the C compiler generates a DSYNC instruction directly after a (interrupt) function entry point label. Also an assembler macro is added to the run-time library functions for optionally adding a DSYNC instruction after a function entry point label.

Assembler check:
   The assembler gives a warning when the first label in a code section is not followed by a DSYNC instruction:

   \textit{Wnum}: suspicious instruction concerning CPU functional defect TC113_CPU14

   You can suppress this warning with the option \textit{-wnum}. 
TC113_CPU15

Compiler and assembler option:
   --silicon-bug=cpu15

Assembler control:
   $TC113_CPU15 {on | off}

Assembler macro:
   The assembler macro __TC113_CPU15__ is defined if you specify the
   option --silicon-bug=cpu15.

Protected libraries to link:
   lib\p\tc113\*.a

Compiler bypass:
   To bypass this CPU functional problem, the C compiler avoids generation
   of the ST.T, SWAP and LDMST instructions. For immediate __bit__ and
   bit-field operations alternative instructions are used.

Assembler check:
   The assembler gives a warning for ST.T, SWAP and LDMST instructions:

   Wnum: suspicious instruction concerning CPU functional
defect TC113_CPU15

   You can suppress this warning with the option --wnum.
TC113_CPU16

Compiler and assembler option:

--silicon~bug=cpu16

Assembler control:

$TC113_CPU16 {on | off}

Assembler macro:

The assembler macro __TC113_CPU16__ is defined if you specify the option --silicon~bug=cpu16.

Protected libraries to link:

lib\p\tc113\*.a

Compiler bypass:

To bypass this CPU functional problem, the C compiler generates a NOP instruction before a JI, JLI or CALLI instruction when there is not one already. The compiler also generates a NOP instruction before a RET and RET16 instruction if there is no or just one instruction before RET, starting from the function entry point.

Assembler check:

The assembler gives a warning when a JI, JLI or CALLI instruction is not directly preceded by a NOP instruction. The assembler also gives a warning when there is no or just one instruction (not a NOP instruction) between label and RET or RET16:

\textit{Wnum}: suspicious instruction concerning CPU functional defect TC113_CPU16

You can suppress this warning with the option \texttt{-wnum}.
TC113_DMU1

Compiler and assembler option:
   --silicon-bug=dmu1

Assembler control:
   $TC113_DMU1 {on | off}

Assembler macro:
   The assembler macro __TC113_DMU1__ is defined if you specify the option --silicon-bug=dmu1.

Protected libraries to link:
   lib\p\tc113\*.a

Compiler bypass:
   To bypass this CPU functional problem, the C compiler avoids generation of the ST.T, SWAP and LDMST instructions. For direct __bit and bit-field operations, alternative instructions are used.

Assembler check:
   The assembler gives a warning for SWAP, LDMST and ST.T instructions:

   Wnum: suspicious instruction concerning CPU functional defect TC113_DMU1

   You can suppress this warning with the option -wnum.
TC113_LFI2

Compiler and assembler option:

--silicon-bug=LFI2

Assembler control:

$TC113_LFI2 \{on | off\}

Assembler macro:

The assembler macro __TC113_LFI2__ is defined if you specify the option --silicon-bug=LFI2.

Protected libraries to link:

lib\p\tc113\*.a

Compiler bypass:

To bypass this CPU functional problem, the C compiler avoids generation of ST.T, SWAP and LDMST instructions. For immediate __bit and bit-field operations alternative instructions are used.

Assembler check:

The assembler gives a warning for SWAP, LDMST and ST.T instructions:

Wnum: suspicious instruction concerning CPU functional defect TC113_LFI2

You can suppress this warning with the option -wnum.
TC113_LFI3

Compiler and assembler option:
   --silicon-bug=lfi3

Assembler control:
   $TC113_LFI3 {on | off}

Assembler macro:
   The assembler macro __TC113_LFI3__ is defined if you specify the option --silicon-bug=lfi3.

Protected libraries to link:
   lib\p\tc113\*.a

Compiler bypass:
   To bypass this CPU functional problem, the compiler avoids generation of the ST.T, SWAP and LDMST instructions. For direct __bit__ and bit-field operations alternative instructions are used.

Assembler check:
   The assembler gives a warning for SWAP, LDMST and ST.T instructions:

   **Wnum:** suspicious instruction concerning CPU functional defect TC113_LFI3

   You can suppress this warning with the option -wnum.
TC113_PMU1

Assembler option:
   --silicon-bug=pmu1

Protected libraries to link:
   lib\p\tc113\*.a, or add lib\src\cstart.asm to your project.

Assembler macro:
   The assembler macro __TC113_PMU1__ is defined if you specify the option --silicon-bug=pmu1.

Compiler bypass:
   There is no compiler bypass for this problem.

Assembler bypass:
   To bypass this CPU functional problem, the assembler adds a macro to the C startup code to disable the split mode on the LMB bus. The SPLT bit of the SFR register LFI_CON is set to zero.
TC113_PMU3

Assembler option:
   --silicon-bug=pmu3

Assembler macro:
   The assembler macro __TC113_PMU3__ is defined if you specify the option --silicon-bug=pmu3.

Protected libraries to link:
   lib\p\tc113\*.a (or add lib\src\cstart.asm to your project).

Compiler bypass:
   There is no compiler bypass for this problem.

Assembler bypass:
   To bypass this CPU functional problem, the assembler adds a macro to the C startup code to set the TLB–A and TLB–B mappings to a page size of 16 Kb. The SZA and SZB in the MMU_CON are set to 16 Kb.
CHAPTER 10

MISRA C RULES
**Supported and unsupported MISRA C rules**

A number of MISRA C rules leave room for interpretation. Other rules can only be checked in a limited way. In such cases the implementation decisions and possible restrictions for these rules are listed.

1. The code shall conform to standard C, without language extensions
2. Other languages should only be used with an interface standard
3. Inline assembly is only allowed in dedicated C functions
4. Provision should be made for appropriate run–time checking
5. Only use characters and escape sequences defined by ISO C
6. Character values shall be restricted to a subset of ISO 10646–1
7. Trigraphs shall not be used
8. Multibyte characters and wide string literals shall not be used
9. Comments shall not be nested
10. Sections of code should not be ”commented out”

In general, it is not possible to decide whether a piece of comment is C code that is commented out, or just some pseudo code. Instead, the following heuristics are used to detect possible C code inside a comment:

- a line ends with `;`, or
- a line starts with `;`, possibly preceded by white space

11. Identifiers shall not rely on significance of more than 31 characters
12. The same identifier shall not be used in multiple namespaces
13. Specific–length typedefs should be used instead of the basic types
14. Use `unsigned char’ or `signed char’ instead of plain `char’
15. Floating point implementations should comply with a standard
16. The bit representation of floating point numbers shall not be used

A violation is reported when a pointer to a floating point type is converted to a pointer to an integer type.
17. "typedef" names shall not be reused

18. Numeric constants should be suffixed to indicate type

   A violation is reported when the value of the constant is outside the range indicated by the suffixes, if any.

19. Octal constants (other than zero) shall not be used

20. All object and function identifiers shall be declared before use

21. Identifiers shall not hide identifiers in an outer scope

22. Declarations should be at function scope where possible

* 23. All declarations at file scope should be static where possible

24. Identifiers shall not have both internal and external linkage

* 25. Identifiers with external linkage shall have exactly one definition

26. Multiple declarations for objects or functions shall be compatible

* 27. External objects should not be declared in more than one file

28. The "register" storage class specifier should not be used

29. The use of a tag shall agree with its declaration

30. All automatics shall be initialized before being used

   This rule is checked using worst-case assumptions. This means that violations are reported not only for variables that are guaranteed to be uninitialized, but also for variables that are uninitialized on some execution paths.

31. Braces shall be used in the initialization of arrays and structures

32. Only the first, or all enumeration constants may be initialized

33. The right hand operand of && or || shall not contain side effects

34. The operands of a logical && or || shall be primary expressions

35. Assignment operators shall not be used in Boolean expressions

36. Logical operators should not be confused with bitwise operators

37. Bitwise operations shall not be performed on signed integers
38. A shift count shall be between 0 and the operand width minus 1
   This violation will only be checked when the shift count evaluates
to a constant value at compile time.
39. The unary minus shall not be applied to an unsigned expression
40. "sizeof" should not be used on expressions with side effects
* 41. The implementation of integer division should be documented
42. The comma operator shall only be used in a "for" condition
43. Don’t use implicit conversions which may result in information loss
44. Redundant explicit casts should not be used
45. Type casting from any type to or from pointers shall not be used
46. The value of an expression shall be evaluation order independent
   This rule is checked using worst-case assumptions. This means that
   a violation will be reported when a possible alias may cause the
   result of an expression to be evaluation order dependent.
47. No dependence should be placed on operator precedence rules
48. Mixed arithmetic should use explicit casting
49. Tests of a (non-Boolean) value against 0 should be made explicit
50. F.P. variables shall not be tested for exact equality or inequality
51. Constant unsigned integer expressions should not wrap-around
52. There shall be no unreachable code
53. All non-null statements shall have a side-effect
54. A null statement shall only occur on a line by itself
55. Labels should not be used
56. The "goto" statement shall not be used
57. The "continue" statement shall not be used
58. The "break" statement shall not be used (except in a "switch")
59. An "if" or loop body shall always be enclosed in braces
60. All "if", "else if" constructs should contain a final "else"
61. Every non-empty "case" clause shall be terminated with a "break"
62. All "switch" statements should contain a final "default" case
63. A "switch" expression should not represent a Boolean case
64. Every "switch" shall have at least one "case"
65. Floating point variables shall not be used as loop counters
66. A "for" should only contain expressions concerning loop control

   A violation is reported when the loop initialization or loop update
   expression modifies an object that is not referenced in the loop test.

67. Iterator variables should not be modified in a "for" loop
68. Functions shall always be declared at file scope
69. Functions with variable number of arguments shall not be used
70. Functions shall not call themselves, either directly or indirectly

   A violation will be reported for direct or indirect recursive function
   calls in the source file being checked. Recursion via functions in
   other source files, or recursion via function pointers is not detected.

71. Function prototypes shall be visible at the definition and call
72. The function prototype of the declaration shall match the definition
73. Identifiers shall be given for all prototype parameters or for none
74. Parameter identifiers shall be identical for declaration/definition
75. Every function shall have an explicit return type
76. Functions with no parameters shall have a "void" parameter list
77. An actual parameter type shall be compatible with the prototype
78. The number of actual parameters shall match the prototype
79. The values returned by "void" functions shall not be used
80. Void expressions shall not be passed as function parameters
81. "const" should be used for reference parameters not modified
82. A function should have a single point of exit
83. Every exit point shall have a "return" of the declared return type
84. For "void" functions, "return" shall not have an expression
85. Function calls with no parameters should have empty parentheses
86. If a function returns error information, it should be tested
   A violation is reported when a the return value of a function is ignored.
87. #include shall only be preceded by another directives or comments
88. Non-standard characters shall not occur in #include directives
89. #include shall be followed by either <filename> or "filename"
90. Plain macros shall only be used for constants/qualifiers/specifiers
91. Macros shall not be #define’d and #undef’d within a block
92. #undef should not be used
93. A function should be used in preference to a function–like macro
94. A function–like macro shall not be used without all arguments
95. Macro arguments shall not contain pre-preprocessing directives
   A violation is reported when the first token of an actual macro argument is '#'.
96. Macro definitions/parameters should be enclosed in parentheses
97. Don’t use undefined identifiers in pre-processing directives
98. A macro definition shall contain at most one # or ## operator
99. All uses of the #pragma directive shall be documented
   This rule is really a documentation issue. The compiler will flag all #pragma directives as violations.
100. "defined" shall only be used in one of the two standard forms
101. Pointer arithmetic should not be used
102. No more than 2 levels of pointer indirection should be used

   A violation is reported when a pointer with three or more levels of
   indirection is declared.

103. No relational operators between pointers to different objects

   In general, checking whether two pointers point to the same object
   is impossible. The compiler will only report a violation for a
   relational operation with incompatible pointer types.

104. Non-constant pointers to functions shall not be used
105. Functions assigned to the same pointer shall be of identical type
106. Automatic address may not be assigned to a longer lived object
107. The null pointer shall not be de-referenced

   A violation is reported for every pointer dereference that is not
   guarded by a NULL pointer test.

108. All struct/union members shall be fully specified
109. Overlapping variable storage shall not be used

   A violation is reported for every 'union' declaration.

110. Unions shall not be used to access the sub-parts of larger types

   A violation is reported for a 'union' containing a 'struct' member.

111. Bit fields shall have type "unsigned int" or "signed int"
112. Bit fields of type "signed int" shall be at least 2 bits long
113. All struct/union members shall be named
114. Reserved and standard library names shall not be redefined
115. Standard library function names shall not be reused

   * 116. Production libraries shall comply with the MISRA C restrictions

   * 117. The validity of library function parameters shall be checked
118. Dynamic heap memory allocation shall not be used
119. The error indicator "errno" shall not be used
120. The macro "offsetof" shall not be used
121. `<locale.h>` and the "setlocale" function shall not be used
122. The "setjmp" and "longjmp" functions shall not be used
123. The signal handling facilities of `<signal.h>` shall not be used
124. The `<stdio.h>` library shall not be used in production code
125. The functions `atof/atoi/atol` shall not be used
126. The functions `abort/exit/getenv/system` shall not be used
127. The time handling functions of library `<time.h>` shall not be used

* = Not supported by the TASKING TriCore C compiler

See also section 5.7, C Code Checking: MISRA C, in Chapter Using the Compiler of the User’s Manual.
## Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>#define</td>
<td>5–13, 5–112</td>
</tr>
<tr>
<td>#include</td>
<td>5–26</td>
</tr>
<tr>
<td>#undef</td>
<td>5–56</td>
</tr>
<tr>
<td><strong>BUILD</strong></td>
<td>1–31</td>
</tr>
<tr>
<td><strong>REVISION</strong></td>
<td>1–31</td>
</tr>
<tr>
<td><strong>VERSION</strong></td>
<td>1–31</td>
</tr>
<tr>
<td>_close</td>
<td>2–24</td>
</tr>
<tr>
<td>_Complex</td>
<td>2–4</td>
</tr>
<tr>
<td>_Exit</td>
<td>2–36</td>
</tr>
<tr>
<td>_fp_get_exception_mask</td>
<td>4–15</td>
</tr>
<tr>
<td>_fp_get_exception_status</td>
<td>4–16</td>
</tr>
<tr>
<td>_fp_install_trap_handler</td>
<td>4–16</td>
</tr>
<tr>
<td>_fp_set_exception_mask</td>
<td>4–15</td>
</tr>
<tr>
<td>_fp_set_exception_status</td>
<td>4–16</td>
</tr>
<tr>
<td>_fss_break</td>
<td>2–10</td>
</tr>
<tr>
<td>_fss_init</td>
<td>2–10</td>
</tr>
<tr>
<td>_Imaginary</td>
<td>2–4</td>
</tr>
<tr>
<td>_IOFBF</td>
<td>2–25</td>
</tr>
<tr>
<td>_IOLBF</td>
<td>2–25</td>
</tr>
<tr>
<td>_IONBF</td>
<td>2–25</td>
</tr>
<tr>
<td>_lseek</td>
<td>2–24</td>
</tr>
<tr>
<td>_open</td>
<td>2–24</td>
</tr>
<tr>
<td>_read</td>
<td>2–24</td>
</tr>
<tr>
<td>_START</td>
<td>4–3</td>
</tr>
<tr>
<td>_tolower</td>
<td>2–7</td>
</tr>
<tr>
<td>_unlink</td>
<td>2–24</td>
</tr>
<tr>
<td>_write</td>
<td>2–24</td>
</tr>
</tbody>
</table>

## A

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>abort</td>
<td>2–36</td>
</tr>
<tr>
<td>abs</td>
<td>2–37, 3–6</td>
</tr>
<tr>
<td>access</td>
<td>2–44</td>
</tr>
<tr>
<td>accum</td>
<td>3–22</td>
</tr>
<tr>
<td>acos functions</td>
<td>2–13</td>
</tr>
<tr>
<td>acosh functions</td>
<td>2–14</td>
</tr>
<tr>
<td>acs</td>
<td>3–7</td>
</tr>
<tr>
<td>address spaces</td>
<td>8–23</td>
</tr>
<tr>
<td>alias</td>
<td>1–26</td>
</tr>
</tbody>
</table>

align, 1–26, 3–23  
Alignment gaps, 8–42  
architecture definition, 8–3, 8–21  
arborer options  
-?, 5–244  
-d, 5–245  
-p, 5–247  
-m, 5–246  
-r, 5–248  
-t, 5–250  
-V, 5–251  
-w, 5–253  
-x, 5–252  
add module, 5–248  
create library, 5–248  
delete module, 5–245  
extract module, 5–252  
move module, 5–246  
print list of objects, 5–250  
print list of symbols, 5–250  
print module, 5–247  
replace module, 5–248  
arg, 3–7  
Argument, 2–5  
asci, 3–24  
asciiz, 3–24  
asctime, 2–42  
asin functions, 2–13  
asinh functions, 2–13  
asn, 3–7  
aspcp, 3–7  
avassembler controls  
$case, 3–66  
$debug, 3–67  
$fpu, 3–68  
$hw_only, 3–69  
$ident, 3–70  
$list, 3–73  
$list on/off, 3–71  
$mmu, 3–75  
$object, 3–76  
$page, 3–77
Index

$prctl, 3–79
$title, 3–80
$tc, 3–81
$tc2, 3–82
$title, 3–83
$warning off, 3–84
detailed description, 3–65
listing controls (overview), 3–64
miscellaneous (overview), 3–64
overview, 3–64
assembler directives
accum, 3–22
align, 3–23
ascii, 3–24
asciiz, 3–24
assembly control (overview), 3–19
byte, 3–25
calls, 3–26
comment, 3–27
conditional assembly (overview), 3–21
data definition (overview), 3–20
debug information (overview), 3–21
define, 3–28
detailed description, 3–21
double, 3–38
dup/endm, 3–29
dupa/endm, 3–30
dupc/endm, 3–31
dupf/endm, 3–32
end, 3–33
equ, 3–34
exitm, 3–35
extern, 3–36
fail, 3–37
float, 3–38
fract, 3–39
global, 3–40
half, 3–62
if, 3–41
include, 3–43
local, 3–44
macro/endm, 3–45
macros (overview), 3–21
message, 3–47
org, 3–48
overview, 3–19
pmacro, 3–50
sdecl, 3–51
sect, 3–54
set, 3–55
sfrac, 3–39
size, 3–56
space, 3–57
storage allocation (overview), 3–20
symbol definitions (overview), 3–20
type, 3–58
undef, 3–59
warning, 3–60
weak, 3–61
word, 3–62
assembler list file, 5–90
assembler options
–?, 5–68
--case-sensitive, 5–71
--check, 5–72
--cpu, 5–69
--debug-info, 5–82
--define, 5–73
--diag, 5–75
--emit-locals, 5–77
--error-file, 5–78
--fpu-present, 5–81
--help, 5–68
--include-directory, 5–85
--include-file, 5–84
--is-tricore2, 5–88
--keep-output-files, 5–89
--list-file, 5–92
--list-format, 5–90
--mmu-present, 5–94
--no-tasking-sfr, 5–95
--no-warnings, 5–103
--optimize, 5–96
--option-file, 5–79
--output, 5–97
--preprocessor-type, 5–93
--section-info, 5–100
--silicon-bug, 5–98
--symbol-scope, 5–87
--version, 5–102
--warnings-as-errors, 5–105
-C, 5–69
-c, 5–71
-D, 5–73
-f, 5–79
-g, 5–82
-H, 5–84
-l, 5–85
-i, 5–87
-k, 5–89
-L, 5–90
-l, 5–92
-m, 5–93
-O, 5–96
-o, 5–97
-t, 5–100
-V, 5–102
-w, 5–103
assembly functions
abs, 3–6
acs, 3–7
address calculation (overview), 3–6
arg, 3–7
asn, 3–7
aspcp, 3–7
assembler mode (overview), 3–6
astc, 3–7
at2, 3–8
atn, 3–8
fract, 3–8, 3–11
cel, 3–8
cnt, 3–8
cob, 3–9
conversions (overview), 3–5
cos, 3–9
cpu, 3–9
cvf, 3–9
cvi, 3–9

def, 3–10
exp, 3–10
fld, 3–10
flr, 3–11
bi, 3–11
bis, 3–11
int, 3–11
l10, 3–12
len, 3–12
lng, 3–12
to, 3–12
log, 3–12
los, 3–13
lsb, 3–13
fun, 3–13
mac, 3–14
macros (overview), 3–5
mathematical (overview), 3–4
max, 3–14
min, 3–14
msb, 3–14
mxp, 3–14
pos, 3–15
pow, 3–15
rnd, 3–15
rub, 3–15
scp, 3–16
sfract, 3–16
sgn, 3–16
sin, 3–16
snb, 3–16
sqi, 3–17
strings (overview), 3–5
sub, 3–17
syntax, 3–3
tan, 3–17
tnb, 3–17
xpn, 3–18
astc, 3–7
at2, 3–8
atan functions, 2–13
atan2 functions, 2–13
atanh functions, 2–14
atexit, 2–36
atn, 3–8
atof, 2–34
atoi, 2–34
atol, 2–34
atoll, 2–34
cexp, 2–5
char type, treat as unsigned, 5–58
chdir, 2–44
check source code, 5–11, 5–72, 5–160
cimag, 2–5
clear/noclear, 1–26
clearerr, 2–33
clock, 2–42
clock_t, 2–41
CLOCKS_PER_SEC, 2–42
clog, 2–5
close, 2–44
cnt, 3–8
coh, 3–9
command file, 5–21, 5–79, 5–120,
5–169, 5–229
comment, 3–27
comments, 8–7
common subexpression elimination, 5–12
compiler options
--?, 5–4
--align, 5–7
--check, 5–11
--cpu, 5–8
--cse--all--addresses, 5–12
--debug--info, 5–24
--default--a0-size, 5–65
--default--a1-size, 5–63
--default--near-size, 5–37
--define, 5–13
--diag, 5–15
--error--file, 5–19
--fpu--present, 5–23
--help, 5–4
--include--directory, 5–26
--include--file, 5–25
--indirect, 5–28
--indirect--runtime, 5–29
--inline, 5–30
--inline--max--incr, 5–31
--inline--max--size, 5–31
--integer--enumeration, 5–33
Index

--is-tricore2, 5–34
--iso, 5–10
--keep-output-files, 5–35
--language, 5–5
--misrac, 5–36
--no-double, 5–20
--no-tasking-sfr, 5–40
--no-warnings, 5–60
--object-comments, 5–45
--option-file, 5–21
--output, 5–44
--preprocess, 5–17
--rename-sections, 5–46
--section-per-data-object, 5–49
--silicon-bug, 5–50, 5–53
--source, 5–48
--static, 5–52
--stdout, 5–39
--tradeoff, 5–55
--uchar, 5–58
--undefined, 5–56
--version, 5–59
--warnings-as-errors, 5–62
-A, 5–5
-C, 5–8
-c, 5–10
-D, 5–13
-E, 5–17
-F, 5–20
-f, 5–21
-g, 5–24
-H, 5–25
-I, 5–26
-k, 5–35
-N, 5–37
-n, 5–39
--optimize, 5–41
-O, 5–41
-o, 5–44
-R, 5–46
-s, 5–48
-t, 5–55
-U, 5–56
-u, 5–58
-V, 5–59
-w, 5–60
-Y, 5–63
-Z, 5–65

complex, 2–4
conditional make rules, 5–215
conj, 2–5
Conjugate value, 2–5
control program options
-?, 5–153
--address-size, 5–156
--case-sensitive, 5–158
--check, 5–160
--cpu, 5–157
--create, 5–159
--d, 5–163
--debug-info, 5–178
--define, 5–161
--diag, 5–164
--dry-run, 5–192
--error-file, 5–166
--exceptions, 5–167
--force-c, 5–171
--force-c++, 5–172
--force-munch, 5–173
--force-prelink, 5–174
--format, 5–175
--fptrap, 5–176
--fpupresent, 5–177
--help, 5–153
--ignore-default-library-path,
5–187
--include-directory, 5–179
--instantiate, 5–180
--instantiation-dir, 5–182
--instantiation-file, 5–183
--is-tricore2, 5–184
--iso, 5–185
--keep-output-files, 5–186
--keep-temporary-files, 5–207
--language, 5–154
--library, 5–189
<table>
<thead>
<tr>
<th>Command</th>
<th>Page (or Section)</th>
</tr>
</thead>
<tbody>
<tr>
<td>--library-directory</td>
<td>5-187</td>
</tr>
<tr>
<td>--list-object-files</td>
<td>5-190</td>
</tr>
<tr>
<td>--lsl-file</td>
<td>5-163</td>
</tr>
<tr>
<td>--mmu-present</td>
<td>5-191</td>
</tr>
<tr>
<td>--no-auto-instantiation</td>
<td>5-193</td>
</tr>
<tr>
<td>--no-default-libraries</td>
<td>5-194</td>
</tr>
<tr>
<td>--no-double</td>
<td>5-168</td>
</tr>
<tr>
<td>--no-map-file</td>
<td>5-195</td>
</tr>
<tr>
<td>--no-one-instantiation-per-object</td>
<td>5-196</td>
</tr>
<tr>
<td>--no-tasking-sfr</td>
<td>5-197</td>
</tr>
<tr>
<td>--no-warnings</td>
<td>5-212</td>
</tr>
<tr>
<td>--option-file</td>
<td>5-169</td>
</tr>
<tr>
<td>--output</td>
<td>5-198</td>
</tr>
<tr>
<td>--pass</td>
<td>5-211</td>
</tr>
<tr>
<td>--pass-assembler</td>
<td>5-211</td>
</tr>
<tr>
<td>--pass-c</td>
<td>5-211</td>
</tr>
<tr>
<td>--pass-linker</td>
<td>5-211</td>
</tr>
<tr>
<td>--pass-c++</td>
<td>5-211</td>
</tr>
<tr>
<td>--pass-prelinker</td>
<td>5-211</td>
</tr>
<tr>
<td>--prelink-copy-if-non-local</td>
<td>5-199</td>
</tr>
<tr>
<td>--prelink-local-only</td>
<td>5-200</td>
</tr>
<tr>
<td>--prelink-remove-instantiation-flags</td>
<td>5-201</td>
</tr>
<tr>
<td>--preprocess</td>
<td>5-165</td>
</tr>
<tr>
<td>--show-c++-warnings</td>
<td>5-202</td>
</tr>
<tr>
<td>--silicon-bug</td>
<td>5-203</td>
</tr>
<tr>
<td>--space</td>
<td>5-205</td>
</tr>
<tr>
<td>--static</td>
<td>5-206</td>
</tr>
<tr>
<td>--undefined</td>
<td>5-208</td>
</tr>
<tr>
<td>--verbose</td>
<td>5-210</td>
</tr>
<tr>
<td>--version</td>
<td>5-209</td>
</tr>
<tr>
<td>--warnings-as-errors</td>
<td>5-213</td>
</tr>
</tbody>
</table>

---

See also assembler directives
detailed description, 3–65

copy table, 5–137, 8–25, 8–48

copysign functions, 2–17

core type, 5–69

cos, 3–9

cos functions, 2–13

cosh functions, 2–13

cpow, 2–5

cproj, 2–6

cpu, 3–9

CPU type, 5–8, 5–69, 5–157

creal, 2–6

CSE, 5–12

csin, 2–5

csinh, 2–5

csqrt, 2–5

cstart.asm, 4–3

ctan, 2–5

cctanh, 2–5

cftime, 2–42

cvf, 3–9

cvi, 3–9

cycle count, 5–100
Index

D

data types, 1–4
debug, 3–67
debug information, 5–24, 5–82, 5–146
def, 3–10
default_a0_size, 1–27
default_a1_size, 1–27
default_near_size, 1–27
define, 3–28
derivative definition, 8–4, 8–29
difftime, 2–42
directives
  See also assembler directives
detailed description, 3–21
div, 2–37
double, 3–38
dup, 3–29
dupa, 3–30
dupc, 3–31
dupf, 3–32

E

ELF/DWARF object format, 7–3
elif, 3–41
double, 3–41
end, 3–33
endif, 3–41
enum, 5–33
EOF, 2–23
equ, 3–34
erf functions, 2–18
erfc functions, 2–18
errno, 2–7
errno declaration, 2–59
errno.h, 2–59
exceptions, floating-point, 4–13
exit, 2–36
exit macro, 3–35
EXIT_FAILURE, 2–34
EXIT_SUCCESS, 2–34
exitm, 3–35
exp, 3–10
exp functions, 2–14
exp2 functions, 2–14
expm1 functions, 2–14
extension isuffix, 1–27
extern, 1–27, 3–36
f

fabs functions, 2–17
fail, 3–37
fclose, 2–24
fdim functions, 2–24
FE_ALL_EXCEPT, 2–10
FE_DIVBYZERO, 2–10
FE_INEXACT, 2–10
FE_INVALID, 2–10
FE_OVERFLOW, 2–10
FE_UNDERFLOW, 2–10
fclearexcept, 2–9
fgetenv, 2–9
fgetexceptflag, 2–9
fesetenv, 2–9
fesetexceptflag, 2–9
fetestexcept, 2–9, 2–10
feupdateenv, 2–9
fflush, 2–24
fgetc, 2–29
fgetpos, 2–32
fgets, 2–29
fgetwc, 2–29
fgetws, 2–29
File system simulation, 2–4
FILENAME_MAX, 2–23
fld, 3–10
float, 3–38
floating-point, 4–10
   libraries, 4–13
   single precision, 5–23, 5–81, 5–177
   special values, 4–13
   trap handler, 4–14
   trap handling api, 4–15
   trapping, 4–13
floor functions, 2–15
flr, 3–11
fma functions, 2–17
fmax functions, 2–18
fmin functions, 2–18
fmod functions, 2–16
fopen, 2–24
FOPEN_MAX, 2–23
for_constant_data_use_memory, 1–28
forExtern_data_use_memory, 1–28
forInitialized_data_use_memory, 1–28
for uninitialized_data_use_memory, 1–28
fpcrclassify, 2–19
fprintf, 2–31
fputc, 2–30
fputs, 2–30
fputwc, 2–30
fputws, 2–30
fract, 3–11, 3–39
fractional arithmetic support, 1–14
fread, 2–32
free, 2–35
freopen, 2–25
frexp functions, 2–16
fscanf, 2–29
fseek, 2–32
fsetpos, 2–32
FSS, 2–4
ftell, 2–32
functional problems, 9–3
functions, assembly, 3–3
fwprintf, 2–31
fwrite, 2–32
fswscanf, 2–29

G
getc, 2–29
getchar, 2–29
cgetcwd, 2–44
cgetenv, 2–36
gets, 2–29
cgetwc, 2–29
cgetwchar, 2–29
cglobal, 3–40
gmtime, 2–42

H
half, 3–62
Header files, 2–4
   alert.b, 2–4
   complex.b, 2–4
   ctype.b, 2–6
   errno.b, 2–7
   fcntl.b, 2–9
   fenv.b, 2–9
   float.b, 2–10
   fss.b, 2–10
   inttypes.b, 2–11
   iso646.b, 2–12
   limits.b, 2–12
   locale.b, 2–12
   math.b, 2–13
   setjmp.b, 2–20
   signal.b, 2–20
   stdarg.b, 2–21
   stdbool.b, 2–21
   stddef.b, 2–22
   stdint.b, 2–11
   stdio.b, 2–22
   stdlib.b, 2–33
   string.b, 2–37
   tgmath.b, 2–13
   time.b, 2–41
   unistd.b, 2–44
ucbar.b, 2–22, 2–37, 2–41, 2–45
wctype.b, 2–6, 2–46
heap, 4–10, 8–24
begin of, 4–10
end of, 4–10
hi, 3–11
his, 3–11
hw_only, 3–69
hypot functions, 2–17

ident, 3–70
if, 3–41
ilogb functions, 2–14
imaginary, 2–4
imaxabs, 2–11
imaxdiv, 2–11
include, 3–43
indirect, 1–28
indirect function calling, 5–28, 5–29
indirect_runtime, 1–28
inline functions, 5–31
inline/noinline, 1–28
insert assembly instruction, 1–21
int, 3–11
Intel hex, record type, 7–8
interrupt handling, 1–19
intrinsic functions, 1–12
bit handling, 1–23
fractional data type, 1–14
insert assembly instruction, 1–21
interrupt handling, 1–19
min/max of integers, 1–13
miscellaneous, 1–25
packed data type, 1–15
register handling, 1–22
iob structures, 2–59
isalnum, 2–6
isalpha, 2–6
isblank, 2–6
iscntrl, 2–6
isdigit, 2–6
isfinite, 2–19
isgreater, 2–19
isgreaterequal, 2–19
isinf, 2–19
isless, 2–19
islessequal, 2–19
islessgreater, 2–19
islower, 2–6
isnan, 2–19
isnormal, 2–19
ISO C standard, 5–10
isprint, 2–6
ispunct, 2–6
isspace, 2–6
isunordered, 2–19
isupper, 2–6
iswalnum, 2–6, 2–46
iswalpha, 2–6, 2–46
iswblank, 2–6
iswcntrl, 2–6, 2–46
iswctype, 2–46
iswdigit, 2–6, 2–46
iswgraph, 2–6, 2–46
iswlower, 2–6, 2–47
iswprint, 2–6, 2–47
iswpunct, 2–6, 2–47
iswspace, 2–6, 2–47
iswupper, 2–6, 2–47
iswxdigit, 2–6
iswxdigit, 2–47
isxdigit, 2–6

L
L_tmpnam, 2–23
l10, 3–12
labs, 2–37
language extensions, intrinsic
functions, 1–12
ldexp functions, 2–16
ldiv, 2–37
len, 3–12
lgamma functions, 2–18
linker map file, 5–132
linker options
–?, 5–108
--case-insensitive, 5–111
--chip-output, 5–109
--define, 5–112
--diag, 5–115
--error-file, 5–119
--extern, 5–117
--first-library-first, 5–122
--help, 5–108
--ignore-default-library-path, 5–126
--include-directory, 5–123
--incremental, 5–145
--keep-output-files, 5–125
--library, 5–128
--library-directory, 5–126
--link-only, 5–129
--lsl-check, 5–130
--lsl-dump, 5–131
--map-file, 5–132
--map-file-format, 5–133
--misra-c-report, 5–135
--munch, 5–136
--no-rescan, 5–138
--no-rom-copy, 5–137
--no-warnings, 5–149
--non-romable, 5–140
--optimize, 5–141
--option-file, 5–120
--output-file, 5–143
--strip-debug, 5–146
--user-provided-initialization-code, 5–124
--verbose, 5–148
--version, 5–147
--warnings-as-errors, 5–151
-c, 5–109
-D, 5–112
-d, 5–113
-e, 5–117
-f, 5–120
-I, 5–123
-i, 5–124
-k, 5–125
-L, 5–126
-l, 5–128
-M, 5–132
-m, 5–133
-N, 5–137
-O, 5–141
-o, 5–143
-r, 5–145
-S, 5–146
-t, 5–148
-V, 5–147
-v, 5–148
-w, 5–149

linker script file
architecture definition, 8–3
board specification, 8–5
bus definition, 8–4
derivative definition, 8–4
memory definition, 8–4
preprocessing, 8–6
processor definition, 8–4
section layout definition, 8–5
structure, 8–3
list, 3–73
list file, 5–92
assembler, 5–90
linker, 5–132
list on/off, 3–71
llabs, 2–37
lldiv, 2–37
llrint functions, 2–15
llround functions, 2–15
lng, 3–12
lo, 3–44
localeconv, 2–13
localtime, 2–42
log, 3–12
log functions, 2–14
log10 functions, 2–14
log1p functions, 2–14
log2 functions, 2–14
logb functions, 2–14
longjmp, 2–20
los, 3–13
lrint functions, 2–20
lround functions, 2–20
lsb, 3–13
lseek, 2–44
LSL expression evaluation, 8–20
LSL functions
absolute(), 8–9
addressof(), 8–9
exists(), 8–10
max(), 8–10
min(), 8–10
sizeof(), 8–10
LSL keywords
align, 8–23, 8–24, 8–25, 8–41
alloc_allowed, 8–46
allow_cross_references, 8–42
architecture, 8–22, 8–30
attributes, 8–40, 8–41
bus, 8–23, 8–26, 8–35
clustered, 8–42
contiguous, 8–41
copy_unit, 8–25
copytable, 8–25, 8–48
core, 8–30
derivative, 8–29, 8–34
dest, 8–25, 8–26
dest_dbits, 8–27
dest_offset, 8–26
direction, 8–38, 8–41
down, 8–22, 8–29
fill, 8–42, 8–46, 8–48
fixed, 8–24, 8–45, 8–46
group, 8–39, 8–40
grows, 8–24

heap, 8–24, 8–46
high_to_low, 8–24, 8–38
id, 8–23
if, 8–49
load_addr, 8–44
low_to_high, 8–24, 8–38
map, 8–23, 8–24, 8–26, 8–31
mau, 8–23, 8–31, 8–35
mem, 8–43
memory, 8–31, 8–35
min_size, 8–24, 8–45, 8–46
nram, 8–31
ordered, 8–41
overlay, 8–42
page, 8–44
page_size, 8–24
processor, 8–33
ram, 8–31
reserved, 8–31, 8–46
rom, 8–31
run_addr, 8–25, 8–43
section, 8–47
section_layout, 8–38
select, 8–39
size, 8–26, 8–31, 8–35, 8–45, 8–46, 8–47
space, 8–23, 8–26
speed, 8–31, 8–35
src_dbits, 8–27
src_offset, 8–26
stack, 8–24, 8–45
start_address, 8–25
symbol, 8–25
type, 8–31, 8–35
width, 8–23
LSL syntax, 8–6
lst, 3–13
lun, 3–13
mac, 3–14
macro, 3–45
define, 5–161
definition, 3–45
undefined, 3–50, 5–208
macro/nomacro, 1–28
macros, 1–31
make utility, 5–215
macros, predefined
_DATE__, 5–56
_FILE__, 5–56
_LINE__, 5–56
_STDC__, 5–56
_TIME__, 5–56
Magnitude, 2–5
make utility options
-V, 5–239
-W, 5–240
-w, 5–241
-x, 5–242
defining a macro, 5–215
malloc, 2–35
map file
control program option, 5–195
format, 5–133
linker, 5–132
mappings, 8–26
max, 3–14
MB_CUR_MAX, 2–34, 2–45
MB_LEN_MAX, 2–45
mblen, 2–37
mbrlen, 2–46
mbrtowc, 2–45
mbsinit, 2–45
mbsrtowcs, 2–45
memchr, 2–40
memcmp, 2–39
memcpy, 2–38
memmove, 2–38
memory definition, 8–4
memory management instructions,
      5–94, 5–191
memset, 2–41
message, 1–28, 3–47
min, 3–14
min/max of integers, 1–13
MISRA C, 5–36
 supported rules, 10–3
mktime, 2–42
fpu, 3–68
mmu, 3–75
modf functions, 2–16
Modulus, 2–5
msb, 3–14
mxp, 3–14

N
nan functions, 2–17
nearbyint functions, 2–15
nextafter functions, 2–17
nexttoward functions, 2–17
Norm, 2–5
NULL, 2–22

O
object, 3–76
object_comment, 1–29
offsetof, 2–22
open, 2–9
optimization, 5–41, 5–96, 5–141
optimize/endoptimize, 1–29
option file, 5–21, 5–79, 5–120, 5–169, 5–229
org, 3–48
output file, 5–44, 5–97, 5–143, 5–198
output format, 5–109, 5–175

P
pack, 1–29
packed data type support, 1–15
page, 3–77
pass option to tool, 5–211
perror, 2–33
Phase angle, 2–5
pmacro, 3–50
pos, 3–15
pow, 3–15
pow functions, 2–17
power-on vector, 4–3
Pragma
extern, 1–27
indirect, 1–28
indirect_runtime, 1–28
macro, 1–28
message, 1–28
object_comment, 1–29
tradeoff, 1–30
warning, 1–30
weak, 1–30
Pragma
alias, 1–26, 1–30
align, 1–26
clear/noclear, 1–26
default_a0_size, 1–27
default_a1_size, 1–27
default_near_size, 1–27
extension isuffix, 1–27
for_constant_data_use_memory, 1–28
for_external_data_use_memory, 1–28
for_initialized_data_use_memory, 1–28
for_uninitialized_data_use_memory, 1–28
inline/nINLINE, 1–28
optimize/endoptimize, 1–29
pack, 1–29
section, 1–29
section all, 1–29
section code_init, 1–29
section const_init, 1–29
section data_overlay, 1–29
section vector_init, 1–29
smartinline, 1–28
source/nosource, 1–29
pragmas, 1–26
prctl, 3–79
predefined macros, 1–31
predefined macros in C
__CTC__, 1–31
__DOUBLE_FP__, 1–31
__DSPC__, 1–31
__DSPC_VERSION__, 1–31
__FPU__, 1–31
__SINGLE_FP__, 1–31
__TASKING__, 1–31
preprocessing, 8–6
preprocessor, 5–93
printf, 2–25, 2–31
conversion characters, 2–27
processor definition, 8–4, 8–33
ptrdiff_t, 2–22
putc, 2–30
putchar, 2–30
puts, 2–31
putwc, 2–30
putwchar, 2–30

S
scalbln functions, 2–16
scalbn functions, 2–16
scanf, 2–27, 2–30
conversion characters, 2–28
scp, 3–16
sdecl, 3–51
sect, 3–54
section, 1–29
summary, 5–100
section activation, 3–54
section all, 1–29
section attributes, 3–51
section code_init, 1–29
section data_overlay, 1–29
section declaration, 3–51
section layout definition, 8–5, 8–37
section names, 3–52
sections
grouping, 8–39
rename, 5–46
SEEK_CUR, 2–32
SEEK_END, 2–32
SEEK_SET, 2–32
set, 3–55
setbuf, 2–25
setjmp, 2–20
setlocale, 2–12
setvbuf, 2–25
sfrac, 3–16, 3–39
sgn, 3–16
SIGABRT, 2–20
SIGFPE, 2–20
SIGFPE signal handler, 4–14
SIGILL, 2–20
SIGINT, 2–20
signal, 2–20
signbit, 2–19
SIGSEGV, 2–20
SIGTERM, 2–20
silicon bug workaround, 5–50, 5–98, 5–203

Q
qsort, 2–36

R
raise, 2–20
rand, 2–35
RAND_MAX, 2–34
read, 2–44
realloc, 2–35
reentrancy, 2–48
register handling, 1–22
remainder functions, 2–16
remove, 2–33
remquo functions, 2–16
rename, 2–33
rename sections, 5–46
reset vector, 8–25
rewind, 2–32
Riemann sphere, 2–6
rint functions, 2–15
rnd, 3–15
round functions, 2–15
rvb, 3–15
sin, 3–16
sin functions, 2–13
sinh functions, 2–13
size, 3–56
size_t, 2–22
smartline, 1–28
snh, 3–16
snprintf, 2–31
source/nosource, 1–29
space, 3–57
sqrt functions, 2–17
sqrt, 3–17
srand, 2–35
sscanf, 2–30
stack, 4–9, 8–24
begin of, 4–9
start address, 8–25
startup code, 4–3
stat, 2–44
stderr, 2–23
stdin, 2–23
stdout, 2–23
stitle, 3–80
strcat, 2–38
strchr, 2–40
strcmp, 2–39
strcoll, 2–39
strcpy, 2–38
strcspn, 2–40
strerror, 2–41
strftime, 2–4
strptime, 2–43
strncat, 2–38
strncpy, 2–39
strncpy, 2–38
strncpy, 2–39
strncpy, 2–38
strpbrk, 2–40
strrchr, 2–40
strspn, 2–40
strstr, 2–40
strtof, 2–35
strtol, 2–35
strtol, 2–34
strtol, 2–35
strtol, 2–35
strtol, 2–35
strtol, 2–11
strxfrm, 2–39
sub, 3–17
switch auto, 1–30
switch jumptab, 1–30
switch linear, 1–30
switch lookup, 1–30
switch restore, 1–30
switch statement, 5–53
swprintf, 2–31
swscanf, 2–30
syntax error checking, 5–11, 5–72,
5–160
system, 2–36
system libraries, 5–126, 5–128

T

tan, 3–17
tan functions, 2–13
tanh functions, 2–13
tc, 3–81
tc2, 3–82
temporary files, 5–207
tgamma functions, 2–18
time, 2–42
time_t, 2–41
tm (struct), 2–41
TMP_MAX, 2–23
tmpfile, 2–33
tmpnam, 2–33
tnh, 3–17
tolower, 2–7
toupper, 2–7
towctrans, 2–47
towlower, 2–7, 2–47
INDEX

W

warning, 1–30, 3–60
title, 3–83, 3–84
warnings, 5–213
  suppress, 5–103
warnings as errors, 5–62, 5–105, 5–151
warnings, suppress, 5–60, 5–149
wchar_t, 2–22
wctomb, 2–46
wcscat, 2–38
wcschr, 2–40
wcscmp, 2–39
wcscoll, 2–39
wcscpy, 2–38
wcsncpy, 2–40
wcsncat, 2–38
wcsncmp, 2–39
wcsncpy, 2–38
wcsncpy, 2–40
wcsrchr, 2–40
wcsrtombs, 2–45
wcsstr, 2–40
wcstod, 2–34
wcstof, 2–34
wcstol, 2–34
wcstold, 2–34
wcstoll, 2–34
wcstombs, 2–37
wcstoul, 2–35
wcstoull, 2–35
wcstoumax, 2–11
wcsxfrm, 2–39
wctob, 2–46
wctomb, 2–37
wctrans, 2–47
wctype, 2–46
weak, 1–30, 3–61

towupper, 2–7, 2–47
tradeoff, 1–30
trap, 4–18
trap handler, 4–14
trap handling, 5–176
trap handling api, 4–15
TriCore 2 instructions, 5–34, 5–88, 5–184
trunc functions, 2–15
type, 3–58

U

undef, 3–59
unf, 3–17
ungetc, 2–29
ungetwc, 2–29
unlink, 2–44

V

va_arg, 2–21
va_end, 2–21
va_start, 2–21
verbose, 5–148, 5–210
version information, 5–59, 5–102,
  5–147, 5–209, 5–239, 5–240, 5–251
vfprintf, 2–31
vfscanf, 2–30
vfscanf, 2–31
vfscanf, 2–30
vprintf, 2–31
scanf, 2–30
printf, 2–31
scanf, 2–30
scanf, 2–31
scanf, 2–30
scanf, 2–31
scanf, 2–30
scanf, 2–31
scanf, 2–30
<table>
<thead>
<tr>
<th>Function</th>
<th>Page Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>WEOF</td>
<td>2–23</td>
</tr>
<tr>
<td>wmemchr</td>
<td>2–40</td>
</tr>
<tr>
<td>wcncmp</td>
<td>2–39</td>
</tr>
<tr>
<td>wmemcpy</td>
<td>2–38</td>
</tr>
<tr>
<td>wmemmove</td>
<td>2–38</td>
</tr>
<tr>
<td>wmemset</td>
<td>2–41</td>
</tr>
<tr>
<td>word</td>
<td>3–62</td>
</tr>
<tr>
<td>wprintf</td>
<td>2–31</td>
</tr>
<tr>
<td>write</td>
<td>2–44</td>
</tr>
<tr>
<td>wscanf</td>
<td>2–30</td>
</tr>
<tr>
<td>wstrftime</td>
<td>2–43</td>
</tr>
<tr>
<td>xpn</td>
<td>3–18</td>
</tr>
</tbody>
</table>