

```
{  
    FILE* sfile;  
    int count = 0;  
  
    sfile = fopen("file1", "r");  
  
    if( sfile == NULL )  
    {  
        return -1;  
    }  
  
    while (1)  
    {  
        char c;  
        c = fgetc(sfile);  
        if(c == EOF)  
        {  
            break;  
        }  
        else  
        {  
            count++;  
        }  
    }  
  
    return count;  
}
```

TriCore v2.0

C Compiler, Assembler, Linker Reference Guide

A publication of
Altium BV
Documentation Department
Copyright © 2002–2003 Altium BV

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

TASKING is a brand name of Altium Limited.

The following trademarks are acknowledged:

FLEXlm is a registered trademark of Globetrotter Software, Inc.

Intel is a trademark of Intel Corporation.

Motorola is a registered trademark of Motorola, Inc.

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

SUN is a trademark of Sun Microsystems, Inc.

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

All other trademarks are property of their respective owners.

Data subject to alteration without notice.

<http://www.tasking.com>

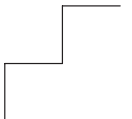
<http://www.altium.com>

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

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

CONTENTS

TABLE OF CONTENTS



CONTENTS

TRICORE C LANGUAGE **1-1**

| | | |
|-------|---|------|
| 1.1 | Introduction | 1-3 |
| 1.2 | Data Types | 1-4 |
| 1.3 | Keywords | 1-6 |
| 1.4 | Function Qualifiers | 1-9 |
| 1.5 | Intrinsic Functions | 1-12 |
| 1.5.1 | Minimum and maximum of (Short) Integers | 1-13 |
| 1.5.2 | Fractional Arithmetic Support | 1-14 |
| 1.5.3 | Packed Data Type Support | 1-15 |
| 1.5.4 | Interrupt Handling | 1-18 |
| 1.5.5 | Insert Single Assembly Instruction | 1-19 |
| 1.5.6 | Register Handling | 1-20 |
| 1.5.7 | Insert / Extract Bit-fields and Bits | 1-21 |
| 1.5.8 | Miscellaneous Intrinsic Functions | 1-23 |
| 1.6 | Pragmas | 1-24 |
| 1.7 | Predefined Macros | 1-25 |

LIBRARIES **2-1**

| | | |
|-------|----------------------------|------|
| 2.1 | Introduction | 2-3 |
| 2.1.1 | Header Files | 2-4 |
| 2.1.2 | C Library Functions | 2-9 |
| 2.1.3 | C Library Reentrancy | 2-63 |

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 | 3-18 |
| 3.3.1 | Overview of Assembler Directives | 3-18 |
| 3.3.2 | Detailed Description of Assembler Directives | 3-20 |
| 3.3.3 | Overview of Assembler Controls | 3-66 |
| 3.3.4 | Detailed Description of Assembler Controls | 3-67 |

TOOL OPTIONS **4-1**

| | | |
|-----|-------------------------------|-------|
| 4.1 | Compiler Options | 4-3 |
| 4.2 | Assembler Options | 4-57 |
| 4.3 | Linker Options | 4-94 |
| 4.4 | Control Program Options | 4-137 |
| 4.5 | Make Utility Options | 4-162 |
| 4.6 | Archiver Options | 4-191 |

LIST FILE FORMATS **5-1**

| | | |
|-----|----------------------------------|-----|
| 5.1 | Assembler List File Format | 5-3 |
| 5.2 | Linker Map File Format | 5-5 |

OBJECT FILE FORMATS **6-1**

| | | |
|-----|--------------------------------|-----|
| 6.1 | ELF/DWARF Object Format | 6-3 |
| 6.2 | Motorola S-Record Format | 6-4 |
| 6.3 | Intel Hex Record Format | 6-8 |

LINKER SCRIPT LANGUAGE **7-1**

| | | |
|-------|--|------|
| 7.1 | Introduction | 7-3 |
| 7.2 | Structure of a Linker Script File | 7-3 |
| 7.3 | Syntax of the Linker Script Language | 7-6 |
| 7.3.1 | Identifiers | 7-7 |
| 7.3.2 | Expressions | 7-7 |
| 7.3.3 | Built-in Functions | 7-8 |
| 7.3.4 | LSL Definitions in the Linker Script File | 7-10 |
| 7.3.5 | Memory and Bus Definitions | 7-10 |
| 7.3.6 | Architecture Definition | 7-12 |
| 7.3.7 | Derivative Definition | 7-14 |
| 7.3.8 | Processor Definition and Board Specification | 7-15 |
| 7.3.9 | Section Placement Definition | 7-15 |
| 7.4 | Expression Evaluation | 7-18 |

| | | |
|-------|--|------|
| 7.5 | Semantics of the Architecture Definition | 7-19 |
| 7.5.1 | Defining an Architecture | 7-20 |
| 7.5.2 | Defining Internal Busses | 7-20 |
| 7.5.3 | Defining Address Spaces | 7-21 |
| 7.5.4 | Mappings | 7-23 |
| 7.6 | Semantics of the Derivative Definition | 7-26 |
| 7.6.1 | Defining a Derivative | 7-26 |
| 7.6.2 | Instantiating Core Architectures | 7-27 |
| 7.6.3 | Defining Internal Memory and Busses | 7-27 |
| 7.7 | Semantics of the Board Specification | 7-29 |
| 7.7.1 | Defining a Processor | 7-29 |
| 7.7.2 | Instantiating Derivatives | 7-30 |
| 7.7.3 | Defining External Memory and Busses | 7-30 |
| 7.8 | Semantics of the Section Layout Definition | 7-32 |
| 7.8.1 | Defining a Section Layout | 7-32 |
| 7.8.2 | Creating and Locating Groups of Sections | 7-33 |
| 7.8.3 | Creating or Modifying Special Sections | 7-39 |
| 7.8.4 | Conditional Group Statements | 7-41 |

CPU FUNCTIONAL PROBLEMS **8-1**

| | | |
|-----|--|------|
| 8.1 | Introduction | 8-3 |
| 8.2 | CPU Functional Problem bypasses TC1 V1.2 | 8-5 |
| 8.3 | CPU Functional Problem bypasses TC1 V1.3 | 8-16 |

MISRA C RULES **9-1**

INDEX



CONTENTS

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 Guide which includes a Getting Started section and a separate Reference Guide (this manual).

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

After installation you are ready to follow the *Getting Started* in Chapter 2 of the *User's Guide*.

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

Once you are familiar with these tools, you can use the Reference Guide to lookup specific options and details to make 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. First libraries are listed per header file that contains the prototypes. These tables also show the level of implementation per function. Second, all library functions are listed and discussed into detail.

Chapter 3: TriCore Assembly Language

Contains an overview of all assembly functions that you can use in your assembly source code.

Chapter 4: Tool Options

Contains a description of all tool options:

- Compiler options
- Assembler options
- Linker options
- Control program options
- Make utility options
- Archiver options

Chapter 5: List File Formats

Contains a description of the following list file formats:

- Assembler List File Format
- Linker Map File Format

Chapter 6: 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 7: Linker Script Language

Contains a description of the linker script language (LSL).

Chapter 8: 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 9: 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 losing information.



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



Command line: type your input on the command line.



Reference: follow this reference to find related topics.

RELATED PUBLICATIONS

C Standards

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

MISRA C

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

TASKING Tools

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

TriCore

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

CHAPTER

1

TRICORE C LANGUAGE



1

CHAPTER

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:

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

| Type | Keyword | Size (bit) | Align (bit) | Ranges |
|--------|-----------------------------|------------|-------------|------------------------------|
| Accum | __laccum | 64 | 64 | $[-131072, 131072>$ |
| Packed | __packb signed __packb | 32 | 16 | $4x: -2^7 \dots 2^7-1$ |
| | unsigned __packb | 32 | 16 | $4x: 0 \dots 2^8-1$ |
| | __packhw signed __packhw | 32 | 16 | $2x: -2^{15} \dots 2^{15}-1$ |
| | unsigned __packhw | 32 | 16 | $2x: 0 \dots 2^{16}-1$ |

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.

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

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 Guide*.

`__at()`

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

```
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:

```
__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 Guide*.

`__near` **`__far`**

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.

`__sfrbit16` **`__sfrbit32`**

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 Guide*.

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:

```
__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:

```
__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 Guide*.

__indirect

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

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

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 Guide*.

__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 (PIPn) 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 Guide*.

__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 Guide*.

__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 to 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 minimum or maximum of a signed integer, unsigned integer or short integer.

| Intrinsic Function | Description |
|--|---|
| <code>int __min(int,int)</code> | Return minimum of two integers |
| <code>short __mins(short,short)</code> | Return minimum of two short integers |
| <code>unsigned int __minu(unsigned int, unsigned int)</code> | Return minimum of two unsigned integers |
| <code>int __max(int,int)</code> | Return maximum of two integers |
| <code>short __maxs(short,short)</code> | Return maximum of two short integers |
| <code>unsigned int __maxu(unsigned int, unsigned int)</code> | Return maximum of two unsigned integers |

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

| Intrinsic Function | Description |
|--|---|
| <code>long</code> <code>__mulfraclong(__fract,long)</code> | Integer part of <code>__fract x long</code> |
| <code>__sfract</code> <code>__roundl6(__fract)</code> | Convert <code>__fract</code> to <code>__sfract</code> |
| <code>__fract</code> <code>__getfract(__accum)</code> | Convert <code>__accum</code> to <code>__fract</code> |
| <code>short</code> <code>__clssf(__sfract)</code> | Count the consecutive number of bits that have the same value as bit 15 of an <code>__sfract</code> |
| <code>__sfract</code> <code>__shasfracts(__sfract,int)</code> | Left/right shift of an <code>__sfract</code> |
| <code>__fract</code> <code>__shafracts(__fract,int)</code> | Left/right shift of an <code>__fract</code> |
| <code>__laccum</code> <code>__shaaccum(__laccum,int)</code> | Left/right shift of an <code>__laccum</code> |

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

| Intrinsic Function | Description |
|---|--|
| <code>__packb __initpackbl(long)</code> | Initialize <code>__packb</code> with a long integer |
| <code>__packb __initpackb(int,int,int,int)</code> | Initialize <code>__packb</code> with four integers |
| <code>__packhw __initpackhw1(long)</code> | Initialize <code>__packhw</code> with a long integer |
| <code>__packhw __initpackhw(int,int)</code> | Initialize <code>__packhw</code> with two integers |

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.

| Intrinsic Function | Description |
|---|---|
| <code>char __extractbyte1(__packb)</code> | Extract first byte from a <code>__packb</code> |
| <code>char __extractbyte2(__packb)</code> | Extract second byte from a <code>__packb</code> |
| <code>char __extractbyte3(__packb)</code> | Extract third byte from a <code>__packb</code> |
| <code>char __extractbyte4(__packb)</code> | Extract fourth byte from a <code>__packb</code> |
| <code>short __extracthw1(__packhw)</code> | Extract first short from a <code>__packhw</code> |
| <code>short __extracthw2(__packhw)</code> | Extract second short from a <code>__packhw</code> |
| <code>char __getbyte1(__packb *)</code> | Extract first byte from a <code>__packb</code> |
| <code>char __getbyte2(__packb *)</code> | Extract second byte from a <code>__packb</code> |
| <code>char __getbyte3(__packb *)</code> | Extract third byte from a <code>__packb</code> |
| <code>char __getbyte4(__packb *)</code> | Extract fourth byte from a <code>__packb</code> |

| Intrinsic Function | Description |
|------------------------------|---------------------------------------|
| short __gethw1(__packhw *) | Extract first integer from a __packhw |
| short __gethw2(__packhw *) | Extract short integer from a __packhw |

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.

| Intrinsic Function | Description |
|---|---|
| __packb __insertbyte1(__packb, char) | Insert char into first byte of a __packb |
| __packb __insertbyte2(__packb, char) | Insert char into second byte of a __packb |
| __packb __insertbyte3(__packb, char) | Insert char into third byte of a __packb |
| __packb __insertbyte4(__packb, char) | Insert char into fourth byte of a __packb |
| __packhw __inserthw1(__packhw, short) | Insert short into first halfword of a __packhw |
| __packhw __inserthw2(__packhw, short) | Insert short into second halfword of a __packhw |
| void __setbyte1(__packb *, char) | Insert first byte into a __packb |
| void __setbyte2(__packb *, char) | Insert second byte into a __packb |
| void __setbyte3(__packb *, char) | Insert third byte into a __packb |
| void __setbyte4(__packb *, char) | Insert fourth byte into a __packb |
| void __sethw1(__packhw *, short) | Insert first integer into a __packhw |
| void __sethw2(__packhw *, short) | Insert short integer into a __packhw |

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**).

| Intrinsic Function | Description |
|---|-----------------------------------|
| <code>unsigned long long __transpose_byte(__packb, __packb)</code> | Combine two <code>__packb</code> |
| <code>unsigned long long __transpose_hword(__packhw, __packhw)</code> | Combine two <code>__packhw</code> |

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.

| Intrinsic Function | Description |
|---|---|
| <code>__packb __absb(__packb)</code> | Absolute value of <code>__packb</code> |
| <code>__packhw __absh(__packhw)</code> | Absolute value of <code>__packhw</code> |
| <code>__sat __packhw __abssh(__sat __packhw)</code> | Absolute value of <code>__packhw</code> using saturation |

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.

| Intrinsic Function | Description |
|--|--|
| <code>__packb __minb(__packb, __packb)</code> | Minimum of two <code>__packb</code> values |
| <code>unsigned __packb __minbu(unsigned __packb, unsigned __packb)</code> | Minimum of two unsigned <code>__packb</code> values |
| <code>__packhw __minh(__packhw, __packhw)</code> | Minimum of two <code>__packhw</code> values |
| <code>unsigned __packhw __minhu(unsigned __packhw, unsigned __packhw)</code> | Minimum of two unsigned <code>__packhw</code> values |

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:

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

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 Guide*

| Intrinsic Function | Description |
|----------------------|--------------------------|
| void __debug(void) | Insert DEBUG instruction |
| void __dsync(void) | Insert DSYNC instruction |
| void __isync(void) | Insert ISYNC instruction |
| void __svlcx(void) | Insert SVLCX instruction |
| void __rslcx(void) | Insert RSLCX instruction |
| void __nop(void) | Insert NOP instruction |

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.

| Intrinsic Function | Description |
|--------------------------------------|---|
| <code>int __mfcrr(int)</code> | move contents of the addressed core SFR into a data register |
| <code>void __mtcr (int,int)</code> | move contents of a data register (second int) to the addressed core SFR (first int) |

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.

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

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.

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

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.

| Intrinsic Function |
|---|
| <code>void __imaskldmst(int* address,int value,int bitoffset,int bits)</code> |

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:

```
#define __putbit ( value, address, bitoffset ) __imaskldmst  
    ( address, value, bitoffset, 1 )
```

Intrinsic Macro

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

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 *bitoffset* and returned as an unsigned integer value.

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

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

Intrinsic Macro

```
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*.

| Intrinsic Function |
|--|
| <code>int __mulsc(int a, int b, int offset)</code> |

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`).

| Intrinsic Function |
|---|
| <code>void __swapmsk (int value, int mask, int * memory)</code> |

Initialize Circular Pointer

With the next intrinsic you can initialize a circular pointer with a dynamically allocated buffer at run-time.

| Intrinsic Function |
|---|
| <code>__circ void * __initcirc(void * buf, unsigned short bufsize, unsigned short byteindex)</code> |



See also Section 3.4.1, *Circular Buffers*, in Chapter *TriCore C Language* of the *User's Guide*.

1.6 PRAGMAS

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

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

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

| Pragma name | Description |
|---|---|
| align { <i>n</i> restore} | Specifies object alignment |
| clear noclear | Specifies 'clearing' of non-initialized static/public variables |
| default_a0_size <i>value</i> | Threshold for '__a0' allocation |
| default_near_size <i>value</i> | Threshold for '__near' allocation |
| inline noinline smartinline | Specifies function inlining |
| optimize <i>flags</i> endoptimize | Controls compiler optimizations |
| pack {2 0} | Specifies packing of structures |
| section <i>type</i> [="name"] | Changes section names |
| section code_init section data_overlay | At startup copy code to RAM Allow overlaying data sections |
| source nosource | Specifies which C source lines must be shown in assembly output |
| switch {auto jumptab linear lookup restore} | Specifies switch statement |

Table 1-16: Pragmas



For more information see section 3.7, *Pragmas to Control the Compiler*, in Chapter *TriCore C Language* of the *User's Guide*.

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 1-17. The macros are useful to make conditional C code.

| Macro | Description |
|-------------------------------|--|
| <code>__DOUBLE_FP__</code> | Defined when you do not use compiler option -F (Treat double as float) |
| <code>__SINGLE_FP__</code> | Defined when you use compiler option -F (Treat double as float) |
| <code>__FPU__</code> | Defined when you use compiler option --fpu-present (Use hardware floating point instructions) |
| <code>__CTC__</code> | 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. |
| <code>__TASKING__</code> | Identifies the compiler as the TASKING TriCore compiler. It expands to 1. |
| <code>__DSPC__</code> | Indicates conformation to the DSP-C standard. It expands to 1. |
| <code>__DSPC_VERSION__</code> | Expands to the decimal constant 200001L. |

Table 1-17: Predefined macros

C LANGUAGE

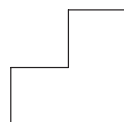
CHAPTER

2

LIBRARIES



TASKING



2

CHAPTER

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 2.1.1, *Header Files*, gives an overview of relevant header files and shows which header file you must include for the functions and/or macros that you use in your C source.

Section 2.1.2, *C Library Functions*, alphabetically lists all library functions you can use in detail. All listed functions reside in the standard C library (`libc.a`) unless stated otherwise.

Section 2.1.3, *C Library Reentrancy*, gives an overview of which functions are reentrant and which are not.

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

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

Table 2-1: Overview of libraries

2.1.1 HEADER FILES

In the table below you can find which header file you must include for the library functions or macros you use in your C source.

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, CrossView Pro's *file system simulation* is implemented which enables you to debug your application.

Explanation:

Yes – Fully implemented

FSS – Implemented via CrossView Pro's *file system simulation*

Empty – Delivered as a skeleton

[illegible]

| Header file | Function or macro name | Implemented | Comments |
|-------------|---|---|--------------------------------|
| limits.h | | Yes | Only Macros |
| locale.h | localeconv setlocale | Empty Empty | No OS present No OS present |
| math.h | acos asin atan atan2 ceil cos cosh exp fabs floor fmod frexp ldexp log log10 modf pow sin sinh sqrt tan tanh | Yes | |
| setjmp.h | longjmp setjmp | Yes Yes | |
| signal.h | raise signal | Yes Yes | |
| stdarg.h | va_arg va_end va_start | Yes Yes Yes | |
| stddef.h | | Yes | Only Macros |

| Header file | Function or macro name | Implemented | Comments |
|-------------|------------------------|-------------|---|
| stdio.h | clearerr | Yes | Delivered as a random name generator, but should use some process ID. |
| | fclose | FSS | |
| | feof | Yes | |
| | ferror | Yes | |
| | fflush | FSS | |
| | fgetc | FSS | |
| | fgetpos | FSS | |
| | fgets | FSS | |
| | fopen | FSS | |
| | fprintf | FSS | |
| | fputc | FSS | |
| | fputs | FSS | |
| | fread | FSS | |
| | freopen | FSS | |
| | fscanf | FSS | |
| | fseek | FSS | |
| | fsetpos | FSS | |
| | ftell | FSS | |
| | fwrite | FSS | |
| | getc | FSS | |
| | getchar | FSS | |
| | gets | FSS | |
| | perror | Yes | |
| | printf | FSS | |
| | putc | FSS | |
| | putchar | FSS | |
| | puts | FSS | |
| | remove | Empty | |
| | rename | Empty | |
| | rewind | FSS | |
| | scanf | FSS | |
| | setbuf | Yes | |
| | setvbuf | Yes | |
| | sprintf | Yes | |
| | sscanf | Yes | |
| | tmpfile | Empty | |
| | tmpnam | Empty | |
| | ungetc | Yes | |
| | vfprintf | FSS | |
| | vprintf | FSS | |
| | vsprintf | Yes | |

| Header file | Function or macro name | Implemented | Comments |
|-------------|------------------------|-------------|--------------------------|
| | _close | FSS | Defined in fss__close.c |
| | _open | FSS | Defined in fss__open.c |
| | _lseek | FSS | Defined in fss__lseek.c |
| | _read | FSS | Defined in fss__read.c |
| | _unlink | FSS | Defined in fss__unlink.c |
| | _write | FSS | Defined in fss__write.c |
| stdlib.h | abort | Yes | Calls _exit() in cstart |
| | abs | Yes | |
| | atexit | Yes | Calls _exit() in cstart |
| | atoac | Yes | |
| | atof | Yes | No OS present |
| | atofr | Yes | |
| | atoi | Yes | No OS present |
| | atol | Yes | |
| | atolac | Yes | No OS present |
| | atolfr | Yes | |
| | bsearch | Yes | No OS present |
| | calloc | Yes | |
| | div | Yes | No OS present |
| | exit | Yes | |
| | free | Yes | No OS present |
| | getenv | Empty | |
| | labs | Yes | No OS present |
| | ldiv | Yes | |
| | malloc | Yes | No OS present |
| | qsort | Yes | |
| | rand | Yes | No OS present |
| | realloc | Yes | |
| | strtoac | Yes | No OS present |
| | strtod | Yes | |
| | strtofr | Yes | No OS present |
| | strtol | Yes | |
| | strtolac | Yes | No OS present |
| | strtolfr | Yes | |
| | strtoul | Yes | No OS present |
| | srand | Yes | |
| | system | Empty | No OS present |
| | mblen | Empty | |
| | mbstowcs | Empty | wide chars not supported |
| | mbtowc | Empty | |
| | wcstombs | Empty | wide chars not supported |
| | wctomb | Empty | |

Table 2-2: Overview of header files

2.1.2 C LIBRARY FUNCTIONS

_close

```
#include <stdio.h>
int _close( int fd );
```

Low level file close function. `_close` is used by the functions `close` and `fclose`. The given file descriptor should be properly closed, any buffer is already flushed. This function interfaces to CrossView Pro's file system simulation.

_lseek

```
#include <stdio.h>
off_t _lseek( int fd, off_t offset, int whence );
```

Low level file positioning function. `_lseek` is used by all file positioning functions (`fgetpos`, `fseek`, `fsetpos`, `ftell`, `rewind`). This function interfaces to CrossView Pro's file system simulation.

_open

```
#include <stdio.h>
int _open( int fd, int flags );
```

Low level file open function. `_open` is used by the functions `fopen` and `freopen`. The given file descriptor should be properly opened. This function interfaces to CrossView Pro's file system simulation.

_read

```
#include <stdio.h>
size_t _read( int fd, char *buffer, size_t count );
```

Low level input function. It reads a sequence of characters from a file. This function interfaces to CrossView Pro's file system simulation.

Returns the number of characters read.

_tolower

```
#include <ctype.h>
int _tolower( int c );
```

Converts *c* to a lowercase character, does not check if *c* really is an uppercase character. This is a non-ANSI function.

Returns the converted character.

_toupper

```
#include <ctype.h>
int _toupper( int c );
```

Converts *c* to an uppercase character, does not check if *c* really is a lowercase character. This is a non-ANSI function.

Returns the converted character.

_unlink

```
#include <stdio.h>
int _unlink( const char *name );
```

Low level file remove function. *_unlink* is used by the function *remove*. This function interfaces to CrossView Pro's file system simulation.

_write

```
#include <stdio.h>
size_t
_write( int fd, char *buffer, size_t count );
```

Low level output function. It writes a sequence of characters to a file. This function interfaces to CrossView Pro's file system simulation.

Returns the number of characters correctly written.

abort

```
#include <stdlib.h>
void abort( void );
```

Terminates the program abnormally. It calls the function `_exit`, which is defined in the start-up module.

Returns nothing.

abs

```
#include <stdlib.h>
int abs( int n );
```

Returns the absolute value of the signed int argument.

access

```
#include <unistd.h>
int access( const char * name, int mode );
```

Use the file system simulation feature 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. |

Returns zero if successful,
 -1 on error.

acos

```
#include <math.h>
double acos( double x );
```

Returns the arccosine $\cos^{-1}(x)$ of x in the range $[0, \pi]$,
 $x \in [-1, 1]$.

asctime

```
#include <time.h>
char *asctime( const struct tm *tp );
```

Converts the time in the structure **tp* into a string of the form:

```
Mon Jan 21 16:15:14 1989\n\0
```

Returns the time in string form.

asin

```
#include <math.h>
double asin( double x );
```

Returns the arcsine $\sin^{-1}(x)$ of *x* in the range $[-\pi/2, \pi/2]$, $x \in [-1, 1]$.

assert

```
#include <assert.h>
void assert( int expr );
```

When compiled with NDEBUG, this is an empty macro. When compiled without NDEBUG defined, it checks if *expr* is true. If it is true, then a line like:

```
"Assertion failed: expression, file filename, line num"
```

is printed.

Returns nothing.

atan

```
#include <math.h>
double atan( double x );
```

Returns the arctangent $\tan^{-1}(x)$ of *x* in the range $[-\pi/2, \pi/2]$, $x \in [-1, 1]$.

atan2

```
#include <math.h>
double atan2( double y, double x );
```

Returns the result of: $\tan^{-1}(y/x)$ in the range $[-\pi, \pi]$.

atexit

```
#include <stdlib.h>
int atexit( void (*fcn)( void ) );
```

Registers the function `fcn` to be called when the program terminates normally.

Returns zero, if program terminates normally.
non-zero, if the registration cannot be made.

atoac

```
#include <stdlib.h>
__accum atoac( const char *s );
```

Converts the initial portion of the string pointed to by `s` to a accumulator value. Initial white spaces are skipped

Returns accumulator value of the converted string,
zero when the conversion failed.



See also "strtoac"

atof

```
#include <stdlib.h>
double atof( const char *s );
```

Converts the given string to a double value. White space is skipped, conversion is terminated at the first unrecognized character.

Returns the double value.

atoi

```
#include <stdlib.h>
int atoi( const char *s );
```

Converts the given string to an integer value. White space is skipped, conversion is terminated at the first unrecognized character.

Returns the integer value.

atofr

```
#include <stdlib.h>
__fract atofr( const char * s )
```

Converts the initial portion of the string pointed to by *s* to a fractional value. Initial white spaces are skipped.

Returns fractional value of the converted string,
zero when the conversion failed.



See also "strtofr"

atol

```
#include <stdlib.h>
long atol( const char *s );
```

Converts the given string to a long value. White space is skipped, conversion is terminated at the first unrecognized character.

Returns the long value.

atolac

```
#include <stdlib.h>
__laccum atolac( const char *s );
```

Converts the initial portion of the string pointed to by *s* to a long accumulator value. Initial white spaces are skipped

Returns long accumulator value of the converted string,
zero when the conversion failed.



See also "strtolac"

atolfr

```
#include <stdlib.h>
__lfract atolfr( const char *restrict s )
```

Converts the initial portion of the string pointed to by *s* to a long fractional value. Initial white spaces are skipped.

Returns long fractional value of the converted string,
zero when the conversion failed.



See also "strtolfr"

bsearch

```
#include <stdlib.h>
void *bsearch( const void *key,
               const void *base, size_t n, size_t size, int (* cmp)
               (const void *, const void *) );
```

This function searches in an array of *n* members, for the object pointed to by *ptr*. The initial base of the array is given by *base*. The size of each member is specified by *size*. The given array must be sorted in ascending order, according to the results of the function pointed to by *cmp*.

Returns a pointer to the matching member in the array, or NULL
when not found.

calloc

```
#include <stdlib.h>
void *calloc( size_t nobj, size_t size );
```

The allocated space is filled with zeros. The maximum space that can be allocated can be changed by customizing the heap size. By default no heap is allocated. When "calloc()" is used while no heap is defined, the linker gives an error.

Returns a pointer to space in external memory for nobj items of size bytes length.
 NULL if there is not enough space left.

ceil

```
#include <math.h>
double ceil( double x );
```

Returns the smallest integer not less than x, as a double.

chdir

```
#include <unistd.h>
int chdir( const char *path );
```

Use the file system simulation feature of CrossView Pro to change the current directory on the host to the directory indicated by path.

Returns zero if successful,
 -1 on error.

clearerr

```
#include <stdio.h>
void clearerr( FILE *stream );
```

Clears the end of file and error indicators for stream.

Returns nothing.

clock

```
#include <time.h>
clock_t clock( void );
```

Determines the processor time used.

Returns number of microseconds since the last reset, assuming a 100 MHz cpu.

close

```
#include <unistd.h>
int close( int fd );
```

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

Returns zero if successful,
 -1 on error.

copysign

```
#include <float.h>
double copysign( double d, double sign );
```

IEEE-754-1985 recommended function. Copy the sign of the second argument to the value of the first argument and return that as result.

Returns the first argument with the sign of the second argument.

cos

```
#include <math.h>
double cos( double x );
```

Returns the cosine of `x`.

cosh

```
#include <math.h>
double cosh( double x );
```

Returns the hyperbolic cosine of x.

ctime

```
#include <time.h>
char *ctime( const time_t *tp );
```

Converts the calendar time *tp into local time, in string form. This function is the same as:

```
asctime( localtime( tp ) );
```

Returns the local time in string form.

difftime

```
#include <time.h>
double
difftime( time_t time2, time_t time1 );
```

Returns the result of time2 - time1 in seconds.

div

```
#include <stdlib.h>
div_t div( int num, int denom );
```

Both arguments are integers. The returned quotient and remainder are also integers.

Returns a structure containing the quotient and remainder of num divided by denom.

exit

```
#include <stdlib.h>
void exit( int status );
```

Terminates the program normally. Acts as if 'main()' returns with `status` as the return value.

Returns zero, on successful termination.

exp

```
#include <math.h>
double exp( double x );
```

Returns the result of the exponential function e^x .

fabs

```
#include <math.h>
double fabs( double x );
```

Returns the absolute double value of `x`. $|x|$

fclose

```
#include <stdio.h>
int fclose( FILE *stream );
```

Flushes any unwritten data for stream, discards any unread buffered input, frees any automatically allocated buffer, then closes the `stream`.

Returns zero if the `stream` is successfully closed, or EOF on error.

feof

```
#include <stdio.h>
int feof( FILE *stream );
```

Returns a non-zero value if the end-of-file indicator for `stream` is set.

ferror

```
#include <stdio.h>
int ferror( FILE *stream );
```

Returns a non-zero value if the error indicator for `stream` is set.

fflush

```
#include <stdio.h>
int fflush( FILE *stream );
```

Writes any buffered but unwritten data, if `stream` is an output stream. If `stream` is an input stream, the effect is undefined.

Returns zero if successful, or EOF on a write error.

fgetc

```
#include <stdio.h>
int fgetc( FILE *stream );
```

Reads one character from the given `stream`.

Returns the read character, or EOF on error.

fgetpos

```
#include <stdio.h>
int fgetpos( FILE *stream, fpos_t *ptr );
```

Stores the current value of the file position indicator for the stream pointed to by `stream` in the object pointed to by `ptr`. The type `fpos_t` is suitable for recording such values.

Returns zero if successful,
a non-zero value on error.

fgets

```
#include <stdio.h>
char *fgets( char *s, int n, FILE *stream );
```

Reads at most the next $n-1$ characters from the given `stream` into the array `s` until a newline is found.

Returns `s`, or NULL on EOF or error.

floor

```
#include <math.h>
double floor( double x );
```

Returns the largest integer not greater than x , as a double.

fmod

```
#include <math.h>
double fmod( double x, double y );
```

Returns the floating-point remainder of x/y , with the same sign as x . If y is zero, the result is implementation-defined.

fopen

```
#include <stdio.h>
FILE *fopen( const char *filename, const char *mode );
```

Opens a file for a given mode.

Returns a stream. If the file cannot not be opened, NULL is returned.

You can specify the following values for mode:

| | |
|-----|---|
| "r" | read; open text file for reading |
| "w" | write; create text file for writing; if the file already exists its contents is discarded |
| "a" | append; open existing text file or create new text file for writing at end of file |

| | |
|------|--|
| "r+" | open text file for update; reading and writing |
| "w+" | create text file for update; previous contents if any is discarded |
| "a+" | append; open or create text file for update, writes at end of file |

The update mode (with a '+') allows reading and writing of the same file. In this mode the function `fflush` must be called between a read and a write or vice versa. By including the letter `b` after the initial letter, you can indicate that the file is a binary file. E.g. "rb" means read binary, "w+b" means create binary file for update. The filename is limited to `FILENAME_MAX` characters. At most `FOPEN_MAX` files may be open at once.

fprintf

```
#include <stdio.h>
int fprintf( FILE *stream, const char *format, ... );
```

Performs a formatted write to the given stream.



See also "printf()", "_write()".

fputc

```
#include <stdio.h>
int fputc( int c, FILE *stream );
```

Puts one character onto the given stream.



See also "_write()".

Returns EOF on error.

fputs

```
#include <stdio.h>
int fputs( const char *s, FILE *stream );
```

Writes the string to a `stream`. The terminating NULL character is not written.



See also "`_write()`".

Returns NULL if successful, or EOF on error.

fread

```
#include <stdio.h>
size_t fread( void *ptr, size_t size,
              size_t nobj, FILE *stream );
```

Reads `nobj` members of `size` bytes from the given `stream` into the array pointed to by `ptr`.



See also "`_read()`".

Returns the number of successfully read objects.

free

```
#include <stdlib.h>
void free( void *p );
```

Deallocates the space pointed to by `p`. `p` Must point to space earlier allocated by a call to "`calloc()`", "`malloc()`" or "`realloc()`". Otherwise the behavior is undefined.



See also "`calloc()`", "`malloc()`" and "`realloc()`".

Returns nothing

freopen

```
#include <stdio.h>
FILE * freopen( const char *filename,
               const char *mode, FILE *stream );
```

Opens a file for a given mode associates the `stream` with it. This function is normally used to change the files associated with `stdin`, `stdout`, or `stderr`.



See also "fopen".

Returns `stream`, or `NULL` on error.

frexp

```
#include <math.h>
double frexp( double x, int *exp );
```

Splits `x` into a normalized fraction in the interval $[1/2, 1>$, which is returned, and a power of 2, which is stored in `*exp`. If `x` is zero, both parts of the result are zero. For example: `frexp(4.0, &var)` results in $0.5 \cdot 2^3$. The function returns 0.5, and 3 is stored in `var`.

Returns the normalized fraction.

fscanf

```
#include <stdio.h>
int fscanf( FILE *stream, const char *format, ... );
```

Performs a formatted read from the given `stream`.



See also "scanf()", "_read()".

Returns the number of items converted successfully.

fseek

```
#include <stdio.h>
int fseek( FILE *stream, long offset, int origin );
```

Sets the file position indicator for `stream`. A subsequent read or write will access data beginning at the new position. For a binary file, the position is set to `offset` characters from `origin`, which may be `SEEK_SET` for the beginning of the file, `SEEK_CUR` for the current position in the file, or `SEEK_END` for the end-of-file. For a text stream, `offset` must be zero, or a value returned by `ftell`. In this case `origin` must be `SEEK_SET`.

Returns zero if successful,
 a non-zero value on error.

fsetpos

```
#include <stdio.h>
int fsetpos( FILE *stream, const fpos_t *ptr );
```

Positions `stream` at the position recorded by `fgetpos` in `*ptr`.

Returns zero if successful,
 a non-zero value on error.

ftell

```
#include <stdio.h>
long ftell( FILE *stream );
```

Returns the current file position for `stream`, or
 `-1L` on error.

fwrite

```
#include <stdio.h>
size_t fwrite( const void *ptr, size_t size,
               size_t nobj, FILE *stream );
```

Writes `nobj` members of `size` bytes to the given `stream` from the array pointed to by `ptr`.

Returns the number of successfully written objects.

getc

```
#include <stdio.h>
int getc( FILE *stream );
```

Reads one character out of the given `stream`. Currently #defined as `getchar()`, because FILE I/O is not supported.



See also `”_read()”`.

Returns the character read or EOF on error.

getchar

```
#include <stdio.h>
int getchar( void );
```

Reads one character from standard input.



See also `”_read()”`.

Returns the character read or EOF on error.

getcwd

```
#include <unistd.h>
char * getcwd( char * buf, size_t size );
```

Use the file system simulation feature of CrossView Pro to retrieve the current directory on the host.

Returns the directory name if successful,
 NULL on error.

getenv

```
#include <stdlib.h>
char *getenv( const char *name );
```

Returns the environment string associated with name, or NULL if no string exists.

gets

```
#include <stdio.h>
char *gets( char *s );
```

Reads all characters from standard input until a newline is found. The newline is replaced by a NULL-character.



See also ”_read()”.

Returns a pointer to the read string or NULL on error.

gmtime

```
#include <time.h>
struct tm *gmtime( const time_t *tp );
```

Converts the calendar time *tp into Coordinated Universal Time (UTC).

Returns a structure representing the UTC, or NULL if UTC is not available.

isalnum

```
#include <ctype.h>
int isalnum( int c );
```

Returns a non-zero value when *c* is an alphabetic character or a number ([A–Z][a–z][0–9]).

isalpha

```
#include <ctype.h>
int isalpha( int c );
```

Returns a non-zero value when *c* is an alphabetic character ([A–Z][a–z]).

isascii

```
#include <ctype.h>
int isascii( int c );
```

Returns a non-zero value when *c* is in the range of 0 and 127. This is a non-ANSI function.

isctrl

```
#include <ctype.h>
int isctrl( int c );
```

Returns a non-zero value when *c* is a control character.

isdigit

```
#include <ctype.h>
int isdigit( int c );
```

Returns a non-zero value when *c* is a numeric character ([0–9]).

isfinite

```
#include <float.h>
int isfinite( double d );
```

IEEE-754-1985 recommended function. Test the given variable on being a finite (IEEE-754) value.

Returns zero if the variable is not finite, else non-zero.

isgraph

```
#include <ctype.h>
int isgraph( int c );
```

Returns a non-zero value when *c* is printable, but not a space.

isinf

```
#include <float.h>
int isinf( double d );
```

IEEE-754-1985 recommended function. Test the given variable on being an infinite (IEEE-754) value.

Returns zero if the variable is not \pm -infinite, else non-zero.

islower

```
#include <ctype.h>
int islower( int c );
```

Returns a non-zero value when *c* is a lowercase character ([a-z]).

isnan

```
#include <float.h>
int isnan( double d );
```

IEEE-754-1985 recommended function. Test the given variable on being a NaN (Not a Number, IEEE-754) value.

Returns zero if the variable is not NaN, else non-zero.

isprint

```
#include <ctype.h>
int isprint( int c );
```

Returns a non-zero value when *c* is printable, including spaces.

ispunct

```
#include <ctype.h>
int ispunct( int c );
```

Returns a non-zero value when *c* is a punctuation character (such as '!', ',', ';', etc.).

isspace

```
#include <ctype.h>
int isspace( int c );
```

Returns a non-zero value when *c* is a space type character (space, tab, vertical tab, formfeed, linefeed, carriage return).

isupper

```
#include <ctype.h>
int isupper( int c );
```

Returns a non-zero value when *c* is an uppercase character ([A-Z]).

isxdigit

```
#include <ctype.h>
int isxdigit( int c );
```

Returns a non-zero value when *c* is a hexadecimal digit ([0-9][A-F][a-f]).

labs

```
#include <stdlib.h>
long labs( long n );
```

Returns the absolute value of the signed long argument.

ldexp

```
#include <math.h>
double ldexp( double x, int n );
```

Returns the result of: $x \cdot 2^n$.

ldiv

```
#include <stdlib.h>
ldiv_t ldiv( long num, long denom );
```

Both arguments are long integers. The returned quotient and remainder are also long integers.

Returns a structure containing the quotient and remainder of *num* divided by *denom*.

localeconv

```
#include <locale.h>
struct lconv *localeconv( void );
```

Sets the components of an object with type `struct lconv` with values appropriate for the formatting of numeric quantities according to the rules of the current locale.

Returns a pointer to the filled-in object.

localtime

```
#include <time.h>
struct tm *localtime( const time_t *tp );
```

Converts the calendar time `*tp` into local time.

Returns a structure representing the local time.

log

```
#include <math.h>
double log( double x );
```

Returns the natural logarithm $\ln(x)$, $x > 0$.

log10

```
#include <math.h>
double log10( double x );
```

Returns the base 10 logarithm $\log_{10}(x)$, $x > 0$.

longjmp

```
#include <setjmp.h>
void longjmp( jmp_buf env, int val);
```

Restores the environment previously saved with a call to `setjmp()`. The function calling the corresponding call to `setjmp()` may not be terminated yet. The value of `val` may not be zero.

Returns nothing.

lseek

```
#include <unistd.h>
off_t lseek( int fd, off_t offset, int whence );
```

Moves read-write file offset. This function calls `_lseek`.

Returns the resulting pointer location if successful,
 -1 on error.

malloc

```
#include <stdlib.h>
void *malloc( size_t size );
```

The allocated space is not initialized. The maximum space that can be allocated can be changed by customizing the heap size. By default no heap is allocated. When "malloc()" is used while no heap is defined, the linker gives an error.

Returns a pointer to space in external memory of `size` bytes length.
 NULL if there is not enough space left.

mblen

```
#include <stdlib.h>
int mblen( const char *s, size_t n );
```

Determines the number of bytes comprising the multi-byte character pointed to by *s*, if *s* is not a null pointer. Except that the shift state is not affected. At most *n* characters will be examined, starting at the character pointed to by *s*.

Returns the number of bytes, or 0 if *s* points to the null character, or -1 if the bytes do not form a valid multi-byte character.

mbstowcs

```
#include <stdlib.h>
size_t mbstowcs( wchar_t *pwcs,
                 const char *s, size_t n );
```

Converts a sequence of multi-byte characters that begins in the initial shift state from the array pointed to by *s*, into a sequence of corresponding codes and stores these codes into the array pointed to by *pwcs*, stopping after *n* codes are stored or a code with value zero is stored.

Returns the number of array elements modified (not including a terminating zero code, if any), or (size_t)-1 if an invalid multi-byte character is encountered.

mbtowc

```
#include <stdlib.h>
int mbtowc( wchar_t *pwc,
            const char *s, size_t n );
```

Determines the number of bytes that comprise the multi-byte character pointed to by *s*. It then determines the code for value of type *wchar_t* that corresponds to that multi-byte character. If the multi-byte character is valid and *pwc* is not a null pointer, the *mbtowc* function stores the code in the object pointed to by *pwc*. At most *n* characters will be examined, starting at the character pointed to by *s*.

Returns the number of bytes, or 0 if *s* points to the null character, or -1 if the bytes do not form a valid multi-byte character.

memchr

```
#include <string.h>
void *memchr( const void *cs, int c, size_t n );
```

Checks the first *n* bytes of *cs* on the occurrence of character *c*.

Returns NULL when not found, otherwise a pointer to the found character is returned.

memcmp

```
#include <string.h>
int memcmp( const void *cs, const void *ct,
            size_t n );
```

Compares the first *n* bytes of *cs* with the contents of *ct*.

Returns a value < 0 if *cs* < *ct*,
 0 if *cs* == *ct*,
 or a value > 0 if *cs* > *ct*.

memcpy

```
#include <string.h>
void *memcpy( void *s, const void *ct, size_t n );
```

Copies *n* characters from *ct* to *s*. No care is taken if the two objects overlap.

Returns *s*

memmove

```
#include <string.h>
void *memmove( void *s, const void *ct, size_t n );
```

Copies *n* characters from *ct* to *s*. Overlapping objects will be handled correctly.

Returns *s*

memset

```
#include <string.h>
void *memset( void *s, int c, size_t n );
```

Fills the first *n* bytes of *s* with character *c*.

Returns *s*

mktime

```
#include <time.h>
time_t mktime( struct tm *tp );
```

Converts the local time in the structure **tp* into calendar time.

Returns the calendar time, or -1 if it cannot be represented.

modf

```
#include <math.h>
double modf( double x, double *ip );
```

Splits *x* into integral and fractional parts, each with the same sign as *x*. It stores the integral part in **ip*.

Returns the fractional part.

offsetof

```
#include <stddef.h>
int offsetof( type, member );
```

Returns the offset for the given member in an object of type.

open

```
#include <fcntl.h>
int open( const char * name, int flags );
```

Opens a file a file for reading or writing. This function calls `_open`.



See also "fopen()".

Returns the file descriptor if successful (a non-negative integer), or -1 on error.

perror

```
#include <stdio.h>
void perror( const char *s );
```

Prints `s` and an implementation-defined error message corresponding to the integer `errno`, as if by:

```
fprintf( stderr, "%s: %s\n", s, "error message" );
```

The contents of the error message are the same as those returned by the `strerror` function with the argument `errno`.



See also the "strerror()" function.

Returns nothing.

pow

```
#include <math.h>
double pow( double x, double y );
```

A domain error occurs if $x=0$ and $y \leq 0$, or if $x < 0$ and y is not an integer.

Returns the result of x raised to the power of y : x^y .

printf

```
#include <stdio.h>
int printf( const char *format,...);
```

Performs a formatted write to the standard output stream.



See also `”_write()”`.

Returns the number of characters written to the output stream.

The `format` string may contain plain text mixed with conversion specifiers. Each conversion specifier should be preceded by a `'%'` character. The conversion specifier should be build in order:

- Flags (in any order):
 - specifies left adjustment of the converted argument.
 - + a number is always preceded with a sign character.
+ has higher precedence as space.
 - space a negative number is preceded with a sign, positive numbers with a space.
 - 0 specifies padding to the field width with zeros (only for numbers).
 - # specifies an alternate output form. For `o`, the first digit will be zero. For `x` or `X`, `"0x"` and `"0X"` will be prefixed to the number. For `e`, `E`, `f`, `g`, `G`, the output always contains a decimal point, trailing zeros are not removed.
- A number specifying a minimum field width. The converted argument is printed in a field with at least the length specified here. If the converted argument has fewer characters than specified, it will be padded at the left side (or at the right when the flag `'-'` was specified) with spaces. Padding to numeric fields will be done with zeros when the flag `'0'` is also specified (only when padding left). Instead of a numeric value, also `'*'` may be specified, the value is then taken from the next argument, which is assumed to be of type `int`.
- A period. This separates the minimum field width from the precision.

- A number specifying the maximum length of a string to be printed. Or the number of digits printed after the decimal point (only for floating point conversions). Or the minimum number of digits to be printed for an integer conversion. Instead of a numeric value, also '*' may be specified, the value is then taken from the next argument, which is assumed to be of type int.
- A length modifier 'h', 'l' or 'L'. 'h' indicates that the argument is to be treated as a short or unsigned short number. 'l' should be used if the argument is a long integer. 'L' indicates that the argument is a long double.

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

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

Table 2-3: Printf conversion characters

putc

```
#include <stdio.h>
int putc( int c, FILE *stream );
```

Puts one character onto the given stream.



See also ”_write()”.

Returns EOF on error.

putchar

```
#include <stdio.h>
int putchar( int c );
```

Puts one character onto standard output.



See also ”_write()”.

Returns the character written or EOF on error.

puts

```
#include <stdio.h>
int puts( const char *s );
```

Writes the string to stdout, the string is terminated by a newline.



See also ”_write()”.

Returns NULL if successful, or EOF on error.

qsort

```
#include <stdlib.h>
void qsort(
    const void *base, size_t n, size_t size,
    int (* cmp)(const void *, const void *) );
```

This function sorts an array of *n* members. The initial base of the array is given by *base*. The size of each member is specified by *size*. The given array is sorted in ascending order, according to the results of the function pointed to by *cmp*.

Returns nothing.

raise

```
#include <signal.h>
int raise( int sig );
```

Sends the signal *sig* to the program.



See also "signal()".

Returns zero if successful, or a non-zero value if unsuccessful.

rand

```
#include <stdlib.h>
int rand( void );
```

Returns a sequence of pseudo-random integers, in the range 0 to RAND_MAX.

read

```
#include <unistd.h>
size_t read( int fd, char * buffer, size_t count );
```

Reads a sequence of characters from a file. This function calls `_read`.



See also "`_read()`".

realloc

```
#include <stdlib.h>
void *realloc( void *p, size_t size );
```

Reallocates the space for the object pointed to by *p*. The contents of the object will be the same as before calling `realloc()`. The maximum space that can be allocated can be changed by customizing the heap size. By default no heap is allocated. When "`realloc()`" is used while no heap is defined, the linker gives an error.



See also "`malloc()`".

Returns NULL and **p* is not changed, if there is not enough space for the new allocation. Otherwise a pointer to the newly allocated space for the object is returned.

remove

```
#include <stdio.h>
int remove( const char *filename );
```

Removes the named file, so that a subsequent attempt to open it fails.

Returns zero if file is successfully removed, or
a non-zero value, if the attempt fails.

rename

```
#include <stdio.h>
int rename( const char *oldname,
            const char *newname );
```

Changes the name of the file.

Returns zero if file is successfully renamed, or
a non-zero value, if the attempt fails.

rewind

```
#include <stdio.h>
void rewind( FILE *stream );
```

Sets the file position indicator for the stream pointed to by `stream` to the beginning of the file. This function is equivalent to:

```
(void) fseek( stream, 0L, SEEK_SET );
clearerr( stream );
```

Returns nothing.

scalb

```
#include <float.h>
double scalb( double d, int power );
```

IEEE-754-1985 Recommended function.

Returns $d * 2^{\text{power}}$ for integral values power without computing 2^N .

scanf

```
#include <stdio.h>
int scanf( const char *format, ... );
```

Performs a formatted read from the standard input stream.



See also "`_read()`".

Returns the number of items converted successfully.

All arguments to this function should be pointers to variables (in default memory) of the type which is specified in the format string.

The format string may contain :

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

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

- A `*`, meaning that no assignment is done for this field.
- A number specifying the maximum field width.
- The conversion characters `d`, `i`, `n`, `o`, `u` and `x` can be preceded 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` can be preceded by `'l'` if a pointer double rather than `float` is in the argument list, and by `'L'` if a pointer to a `long double`.
- A conversion specifier. `*`, maximum field width and length modifier are optional, the conversion character is not. The conversion character must be one of the following, if a character following `'%'` is not in the list, the behavior is undefined.

Length specifier and length modifier are optional, the conversion character is not. The conversion character must be one of the following, if a character following `'%'` is not in the list, the behavior is undefined.

| Character | Scanned as |
|---------------------------------|--|
| <code>d</code> | <code>int</code> , signed decimal. |
| <code>i</code> | <code>int</code> , the integer can be octal (i.e. with a leading 0) or hexadecimal (leading <code>"0x"</code> or <code>"0X"</code>), or just decimal. |
| <code>o</code> | <code>int</code> , unsigned octal. |
| <code>u</code> | <code>int</code> , unsigned decimal. |
| <code>x</code> | <code>int</code> , unsigned hexadecimal in lowercase or uppercase. |
| <code>c</code> | single character (converted to unsigned char). |
| <code>s</code> | <code>char *</code> , 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. |
| <code>f</code> | <code>float</code> |
| <code>e</code> , <code>E</code> | <code>float</code> |
| <code>g</code> , <code>G</code> | <code>float</code> |
| <code>n</code> | <code>int *</code> , the number of characters written so far is written into the argument. No scanning is done. |
| <code>p</code> | pointer; hexadecimal 24-bit value which must be entered without <code>0x-</code> prefix. |
| <code>r</code> | <code>__fract</code> , <code>__sfract</code> |

| Character | Scanned as |
|-----------|---|
| R | __laccum |
| [...] | Matches a string of input characters from the set between the brackets. A NULL character is added to terminate the string. Specifying [...] includes the ']' character in the set of scanning characters. |
| [^...] | Matches a string of input characters not in the set between the brackets. A NULL character is added to terminate the string. Specifying [^...] includes the ']' character in the set. |
| % | Literal '%', no assignment is done. |

Table 2-4: *Scanf conversion characters*


setbuf

```
#include <stdio.h>
void setbuf( FILE *stream, char *buf );
```

Buffering is turned off for the stream, if buf is NULL. Otherwise, setbuf is equivalent to:

```
(void) setvbuf( stream, buf, _IOFBF, BUFSIZ )
```

Returns nothing.


 See also "setvbuf()".

setjmp

```
#include <setjmp.h>
int setjmp( jmp_buf env );
```

Saves the current environment for a subsequent call to longjmp.

Returns 0 after a direct call to setjmp(). Calling the function "longjmp()" using the saved env restores the current environment and jumps to this place with a non-zero return value.

 See also "longjmp()".

setlocale

```
#include <locale.h>
char *setlocale( int category, const char *locale );
```

Selects the appropriate portion of the program's locale as specified by the category and locale arguments.

Returns the string associated with the specified category for the new locale if the selection can be honored.
 null pointer if the selection cannot be honored.

setvbuf

```
#include <stdio.h>
int setvbuf( FILE *stream, char *buf,
             int mode, size_t size );
```

Controls buffering for the `stream`; this function must be called before reading or writing. `mode` can have the following values:

`_IOFBF` causes full buffering
`_IOLBF` causes line buffering of text files
`_IONBF` causes no buffering

If `buf` is not `NULL`, it will be used as a buffer; otherwise a buffer will be allocated. `size` determines the buffer size.

Returns zero if successful
 a non-zero value for an error.



See also "setbuf()".

signal

```
#include <signal.h>
void (*signal( int sig, void (*handler)(int)))(int);
```

Determines how subsequent signals will be handled. If `handler` is `SIG_DFL`, the default behavior is used; if `handler` is `SIG_IGN`, the signal is ignored; otherwise, the function pointed to by `handler` will be called, with the argument of the type of signal. Valid signals are:

| | |
|----------------------|---|
| <code>SIGABRT</code> | abnormal termination, e.g. from <code>abort</code> |
| <code>SIGFPE</code> | arithmetic error, e.g. zero divide or overflow |
| <code>SIGILL</code> | illegal function image, e.g. illegal instruction |
| <code>SIGINT</code> | interactive attention, e.g. interrupt |
| <code>SIGSEGV</code> | illegal storage access, e.g. access outside memory limits |
| <code>SIGTERM</code> | termination request sent to this program |

When a signal `sig` subsequently occurs, the signal is restored to its default behavior; then the signal-handler function is called, as if by `(*handler)(sig)`. If the handler returns, the execution will resume where it was when the signal occurred.

Returns the previous value of `handler` for the specific signal, or `SIG_ERR` if an error occurs.

sin

```
#include <math.h>
double sin( double x );
```

Returns the sine of `x`.

sinh

```
#include <math.h>
double sinh( double x );
```

Returns the hyperbolic sine of `x`.

sprintf

```
#include <stdio.h>
int sprintf( char *s, const char *format, ... );
```

Performs a formatted write to a string.



See also "printf".

sqrt

```
#include <math.h>
double sqrt( double x );
```

Returns the square root of x . \sqrt{x} , where $x \geq 0$.

srand

```
#include <stdlib.h>
void srand( unsigned int seed );
```

This function uses `seed` as the start of a new sequence of pseudo-random numbers to be returned by subsequent calls to `srand()`. When `srand` is called with the same seed value, the sequence of pseudo-random numbers generated by `rand()` will be repeated.

Returns pseudo random numbers.

sscanf

```
#include <stdio.h>
int sscanf( char *s, const char *format, ... );
```

Performs a formatted read from a string.



See also "scanf".

stat

```
#include <unistd.h>
int stat( const char * name, struct stat * buf );
```

Use the file system simulation feature of CrossView Pro to stat() a file on the host platform.

Returns zero if successful,
 -1 on error.

strcat

```
#include <string.h>
char *strcat( char *s, const char *ct );
```

Concatenates string ct to string s, including the trailing NULL character.

Returns s

strchr

```
#include <string.h>
char *strchr( const char *cs, int c );
```

Returns a pointer to the first occurrence of character c in the string cs. If not found, NULL is returned.

strcmp

```
#include <string.h>
int strcmp( const char *cs, const char *ct );
```

Compares string cs to string ct.

Returns <0 if cs < ct,
 0 if cs == ct,
 >0 if cs > ct.

strcoll

```
#include <string.h>
int strcoll( const char *cs, const char *ct );
```

Compares string *cs* to string *ct*. The comparison is based on strings interpreted as appropriate to the program's locale.

Returns <0 if *cs* < *ct*,
 0 if *cs* == *ct*,
 >0 if *cs* > *ct*.

strcpy

```
#include <string.h>
char *strcpy( char *s, const char *ct );
```

Copies string *ct* into the string *s*, including the trailing NULL character.

Returns *s*

strcspn

```
#include <string.h>
size_t strcspn( const char *cs, const char *ct );
```

Returns the length of the prefix in string *cs*, consisting of characters not in the string *ct*.

strerror

```
#include <string.h>
char *strerror( size_t n );
```

Returns pointer to implementation-defined string corresponding to error *n*.

strftime

```
#include <time.h>
size_t strftime( char *s, size_t smax,
                 const char *fmt,
                 const struct tm *tp );
```

Formats date and time information from the structure **tp* into *s* according to the specified format *fmt*. *fmt* is analogous to a *printf* format. Each *%c* is replaced as described below:

| | |
|----|--|
| %a | abbreviated weekday name |
| %A | full weekday name |
| %b | abbreviated month name |
| %B | full month name |
| %c | local date and time representation |
| %d | day of the month (01-31) |
| %H | hour, 24-hour clock (00-23) |
| %I | hour, 12-hour clock (01-12) |
| %j | day of the year (001-366) |
| %m | month (01-12) |
| %M | minute (00-59) |
| %p | local equivalent of AM or PM |
| %S | second (00-59) |
| %U | week number of the year, Sunday as first day of the week (00-53) |
| %w | weekday (0-6, Sunday is 0) |
| %W | week number of the year, Monday as first day of the week (00-53) |
| %x | local date representation |
| %X | local time representation |
| %y | year without century (00-99) |
| %Y | year with century |
| %Z | time zone name, if any |
| %% | % |

Ordinary characters (including the terminating *'\0'*) are copied into *s*. No more than *smax* characters are placed into *s*.

Returns the number of characters (*'\0'* not included), or zero if more than *smax* characters were produced.

strlen

```
#include <string.h>
size_t strlen( const char *cs );
```

Returns the length of the string in *cs*, not counting the NULL character.

strncat

```
#include <string.h>
char *strncat( char *s, const char *ct, size_t n );
```

Concatenates string *ct* to string *s*, at most *n* characters are copied. Add a trailing NULL character.

Returns *s*

strncmp

```
#include <string.h>
int strncmp( const char *cs, const char *ct,
             size_t n );
```

Compares at most *n* bytes of string *cs* to string *ct*.

Returns <0 if *cs* < *ct*,
 0 if *cs* == *ct*,
 >0 if *cs* > *ct*.

strncpy

```
#include <string.h>
char *strncpy( char *s, const char *ct, size_t n );
```

Copies string *ct* onto the string *s*, at most *n* characters are copied. Add a trailing NULL character if the string is smaller than *n* characters.

Returns *s*

strpbrk

```
#include <string.h>
char *strpbrk( const char *cs, const char *ct );
```

Returns a pointer to the first occurrence in *cs* of any character out of string *ct*. If none are found, NULL is returned.

strrchr

```
#include <string.h>
char *strrchr( const char *cs, int c );
```

Returns a pointer to the last occurrence of *c* in the string *cs*. If not found, NULL is returned.

strspn

```
#include <string.h>
size_t strspn( const char *cs, const char *ct );
```

Returns the length of the prefix in string *cs*, consisting of characters in the string *ct*.

strstr

```
#include <string.h>
char *strstr( const char *cs, const char *ct );
```

Returns a pointer to the first occurrence of string *ct* in the string *cs*. Returns NULL if not found.

strtoac

```
#include <stdlib.h>
__accum strtoc( const char *restrict s,
                char **restrict endp )
```

Converts the initial portion of the string pointed to by *s* to a accumulator value. Initial white spaces are skipped. When *endp* is not a NULL pointer, after this function is called, **endp* will point to the first character not used by the conversion.

Returns accumulator value of the converted string,
zero when the conversion failed.



See also "atoc"

strtod

```
#include <stdlib.h>
double strtod( const char *s, char **endp );
```

Converts the initial portion of the string pointed to by *s* to a double value. Initial white spaces are skipped. When *endp* is not a NULL pointer, after this function is called, **endp* will point to the first character not used by the conversion.

Returns the read value.

strtofr

```
#include <stdlib.h>
__fract strtufr( const char *restrict s,
                 char **restrict endp )
```

Converts the initial portion of the string pointed to by *s* to a fractional value. Initial white spaces are skipped. When *endp* is not a NULL pointer, after this function is called, **endp* will point to the first character not used by the conversion.

Returns fractional value of the converted string,
zero when the conversion failed.



See also "atofr"

strtok

```
#include <string.h>
char *strtok( char *s, const char *ct );
```

Search the string *s* for tokens delimited by characters from string *ct*. It terminates the token with a NULL character.

Returns a pointer to the token. A subsequent call with *s* == NULL will return the next token in the string.

strtol

```
#include <stdlib.h>
long strtol( const char *s, char **endp, int base );
```

Converts the initial portion of the string pointed to by *s* to a long integer. Initial white spaces are skipped. Then a value is read using the given base. When base is zero, the base is taken as defined for integer constants. I.e. numbers starting with an '0' are taken octal, numbers starting with '0x' or '0X' are taken hexadecimal. Other numbers are taken decimal. When *endp* is not a NULL pointer, after this function is called, **endp* will point to the first character not used by the conversion.

Returns the read value.

strtoul

```
#include <stdlib.h>
__laccum strtoul( const char *restrict s,
                  char **restrict endp )
```

Converts the initial portion of the string pointed to by *s* to a long accumulator value. Initial white spaces are skipped. When *endp* is not a NULL pointer, after this function is called, **endp* will point to the first character not used by the conversion.

Returns long accumulator value of the converted string, zero when the conversion failed.



See also "atolac"

strtolfr

```
#include <stdlib.h>
__lfract strtolfr( const char *restrict s,
                  char **restrict endp )
```

Converts the initial portion of the string pointed to by *s* to a long fractional value. Initial white spaces are skipped. When *endp* is not a NULL pointer, after this function is called, **endp* will point to the first character not used by the conversion.

Returns long fractional value of the converted string,
zero when the conversion failed.



See also "atolfr"

strtoul

```
#include <stdlib.h>
unsigned long strtoul(
    const char *s, char **endp, int base );
```

Converts the initial portion of the string pointed to by *s* to an unsigned long integer. Initial white spaces are skipped. Then a value is read using the given base. When *base* is zero, the base is taken as defined for integer constants. I.e. numbers starting with an '0' are taken octal, numbers starting with '0x' or '0X' are taken hexadecimal. Other numbers are taken decimal. When *endp* is not a NULL pointer, after this function is called, **endp* will point to the first character not used by the conversion.

Returns the read value.

strxfrm

```
#include <string.h>
size_t
strncmp( char *ct, const char *cs, size_t n );
```

Transforms the string pointed to by *cs* and places the resulting string into the array pointed to by *ct*. No more than *n* characters are placed into the resulting string pointed to by *ct*, including the terminating null character.

Returns the length of the transformed string.

system

```
#include <stdlib.h>
int system( const char *s );
```

Passes the string *s* to the environment for execution.

Returns a non-zero value if there is a command processor, if *s* is NULL; or an implementation-dependent value, if *s* is not NULL.

tan

```
#include <math.h>
double tan( double x );
```

Returns the tangent of *x*.

tanh

```
#include <math.h>
double tanh( double x );
```

Returns the hyperbolic tangent of *x*.

time

```
#include <time.h>
time_t time( time_t *tp );
```

The return value is also assigned to **tp*, if *tp* is not NULL.

Returns the current calendar time, or -1 if the time is not available.

tmpfile

```
#include <stdio.h>
FILE *tmpfile( void );
```

Creates a temporary file of the mode "wb+" that will be automatically removed when closed or when the program terminates normally.

Returns a stream if successful, or NULL if the file could not be created.

tmpnam

```
#include <stdio.h>
char *tmpnam( char s[L_tmpnam] );
```

Creates a temporary name (not a file). Each time tmpnam is called a different name is created.

tmpnam(NULL) creates a string that is not the name of an existing file, and returns a pointer to an internal static array. tmpnam(s) creates a string and stores it in s and also returns it as the function value. s must have room for at least L_tmpnam characters. At most TMP_MAX different names are guaranteed during execution of the program.

Returns a pointer to the temporary name, as described above.

toascii

```
#include <ctype.h>
int toascii( int c );
```

Converts c to an ascii value (strip highest bit). This is a non-ANSI function.

Returns the converted value.

tolower

```
#include <ctype.h>
int tolower( int c );
```

Returns *c* converted to a lowercase character if it is an uppercase character, otherwise *c* is returned.

toupper

```
#include <ctype.h>
int toupper( int c );
```

Returns *c* converted to an uppercase character if it is a lowercase character, otherwise *c* is returned.

ungetc

```
#include <stdio.h>
int ungetc( int c, FILE *fin );
```

Pushes at the most one character back onto the input buffer.

Returns EOF on error.

unlink

```
#include <unistd.h>
int unlink( const char * name );
```

Removes the named file, so that a subsequent attempt to open it fails. This function calls `_unlink`.

Returns zero if file is successfully removed, or
a non-zero value, if the attempt fails.

va_arg

```
#include <stdarg.h>
va_arg( va_list ap, type );
```

Returns the value of the next argument in the variable argument list. Its return type has the type of the given argument `type`. A next call to this macro will return the value of the next argument.

va_end

```
#include <stdarg.h>
va_end( va_list ap );
```

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

va_start

```
#include <stdarg.h>
va_start( va_list ap, lastarg );
```

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

vfprintf

```
#include <stdio.h>
int vfprintf( FILE *stream,
              const char *format, va_list arg );
```

Is equivalent to `vprintf`, but writes to the given stream.



See also "vprintf()", "_write()".

vprintf

```
#include <stdio.h>
int vprintf( const char *format, va_list arg );
```

Does a formatted write to standard output. Instead of a variable argument list as for `printf()`, this function expects a pointer to the list.



See also "`printf()`", "`_write()`".

vsprintf

```
#include <stdio.h>
int vsprintf( char *s, const char *format,
              va_list arg );
```

Does a formatted write a string. Instead of a variable argument list as for `printf()`, this function expects a pointer to the list.



See also "`printf()`", "`_write()`".

wcstombs

```
#include <stdlib.h>
size_t wcstombs( char *s, const wchar_t *pwcs,
                 size_t n );
```

Converts a sequence of codes that correspond to multi-byte characters from the array pointed to by `pwcs`, into a sequence of multi-byte characters that begins in the initial shift state and stores these multi-byte characters into the array pointed to by `s`, stopping if a multi-byte character would exceed the limit of `n` total bytes or if a null character is stored.

Returns the number of bytes modified (not including a terminating null character, if any), or `(size_t)-1` if a code is encountered that does not correspond to a valid multi-byte character.

wctomb

```
#include <stdlib.h>
int wctomb( char *s, wchar_t wchar );
```

Determines the number of bytes needed to represent the multi-byte corresponding to the code whose value is `wchar` (including any change in the shift state). It stores the multi-byte character representation in the array pointed to by `s` (if `s` is not a null pointer). At most `MB_CUR_MAX` characters are stored. If the value of `wchar` is zero, the `wctomb` function is left in the initial shift state.

Returns the number of bytes, or -1 if the value of `wchar` does not correspond to a valid multi-byte character.

write

```
#include <unistd.h>
size_t write( int fd, char * buffer, size_t count );
```

Write a sequence of characters to a file. This function calls `_write`.



See also "`_write()`".

2.1.3 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, whether they are reentrant and, if not, the reason why. Note that some of the functions are not reentrant because they set the global variable '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.

| Function | Reentrant | Cause |
|----------|-----------|---|
| abort | no | Calls exit |
| abs | yes | — |
| access | no | Uses global File System Simulation buffer, _fss_buffer |
| acos | no | Function sets errno when error occurs. If errno not used, acos is reentrant. |
| asctime | no | asctime defines static array for broken-down time string. |
| asin | no | Function sets errno when error occurs. If errno not used, asin is reentrant. |
| atan | yes | — |
| atan2 | yes | — |
| atexit | no | atexit defines static array with function pointers to execute at exit of program. |
| atof | yes | — |
| atoi | yes | — |
| atol | yes | — |
| bsearch | yes | — |
| calloc | no | calloc uses static buffer management structures. See malloc (5). |
| ceil | yes | — |
| chdir | no | Uses global File System Simulation buffer, _fss_buffer |
| cleanup | no | Calls fclose. See (1) |
| clearerr | no | Modifies iob[]. See (1) |

| Function | Reentrant | Cause |
|----------|-----------|--|
| clock | yes | – |
| close | no | Calls _close |
| _close | no | Uses global File System Simulation buffer, _fss_buffer |
| cos | yes | – |
| cosh | no | cosh calls exp(), which sets errno. If errno is discarded, cosh is reentrant. |
| ctime | no | Calls asctime |
| difftime | yes | – |
| div | yes | – |
| _doflt | no | Uses I/O functions which modify iob[]. See (1). |
| _doprint | no | Uses indirect access to static iob[] array. See (1). |
| _doscan | no | Uses indirect access to iob[] and calls ungetc (access to local static ungetc[] buffer). See (1). |
| exit | no | Calls fclose indirectly which uses iob[] calls functions in _atexit array. See (1). To make exit reentrant kernel support is required. |
| exp | no | Sets errno. If errno not used, exp is reentrant. |
| fabs | yes | – |
| fclose | no | Uses values in iob[]. See (1). |
| feof | no | Uses values in iob[]. See (1). |
| ferror | no | Uses values in iob[]. See (1). |
| fflush | no | Modifies iob[]. See (1). |
| fgetc | no | Uses pointer to iob[]. See (1). |
| fgetpos | no | Sets the variable errno and uses pointer to iob[]. See (1) / (2). |
| fgets | no | Uses iob[]. See (1). |
| _filbuf | no | Uses iob[]. See (1). |
| floor | yes | – |
| _flsbuf | no | Uses iob[]. See (1). |
| fmod | yes | – |

| Function | Reentrant | Cause |
|----------|-----------|---|
| fopen | no | Uses iob[] and calls malloc when file open for buffered IO. See (1) |
| fprintf | no | Uses iob[]. See (1). |
| fputc | no | Uses iob[]. See (1). |
| fputs | no | Uses iob[]. See (1). |
| fread | no | Calls fgetc. See (1). |
| free | no | free uses static buffer management structures. See malloc (5). |
| freopen | no | Modifies iob[]. See (1). |
| frexp | yes | — |
| fscanf | no | Uses iob[]. See (1) |
| fseek | no | Uses iob[] and calls _doscan. Accesses ungetc[] array. See (1). |
| fsetpos | no | Uses iob[] and sets errno. See (1) / (2). |
| ftell | no | Uses iob[] and sets errno. Calls _lseek. See (1) / (2). |
| fwrite | no | Uses iob[]. See (1). |
| getc | no | Uses iob[]. See (1). |
| getchar | no | Uses iob[]. See (1). |
| getcwd | no | Uses global File System Simulation buffer, _fss_buffer |
| getenv | yes | Skeleton only. |
| _getflt | no | Uses iob[]. See (1). |
| gets | no | Uses iob[]. See (1). |
| gmtime | no | gmtime defines static structure |
| halloc | no | Needs kernel support. See malloc (5). |
| hcalloc | no | hcalloc uses static buffer management structures. See malloc (5). |
| hfree | no | hfree uses static buffer management structures. See malloc (5). |
| hrealloc | no | See malloc (5). |
| _iob | no | Defines static iob[]. See (1). |
| _ioread | no | Depends on low level I/O implementation. Uses iob[]. See (1). |

| Function | Reentrant | Cause |
|------------|-----------|--|
| _iowrite | no | Depends on low level I/O implementation. Uses iob[]. See (1). |
| isalnum | yes | – |
| isalpha | yes | – |
| isascii | yes | – |
| iscntrl | yes | – |
| isdigit | yes | – |
| isgraph | yes | – |
| islower | yes | – |
| isprint | yes | – |
| ispunct | yes | – |
| isspace | yes | – |
| isupper | yes | – |
| isxdigit | yes | – |
| _itoa | yes | – |
| labs | yes | – |
| ldexp | no | Sets errno. See (2). |
| ldiv | yes | – |
| localeconv | – | Skeleton function |
| localtime | yes | |
| log | no | Sets errno. See (2). |
| log10 | no | Calls log. See (2). |
| longjmp | yes | – |
| lseek | no | Calls _lseek |
| _lseek | no | Uses global File System Simulation buffer, _fss_buffer |
| ltoa | yes | – |
| malloc | no | Needs kernel support. See (5). |
| mblen | – | Skeleton function |
| mbstowcs | – | Skeleton function |
| mbtowc | – | Skeleton function |
| memchr | yes | – |

| Function | Reentrant | Cause |
|--------------------|-----------|---|
| memcmp | yes | — |
| memcpy | yes | — |
| memmove | yes | — |
| memset | yes | — |
| mktime | yes | — |
| modf | yes | — |
| open | no | Calls <code>_open</code> |
| <code>_open</code> | no | Uses global File System Simulation buffer, <code>_fss_buffer</code> |
| perror | no | Uses <code>errno</code> . See (2) |
| pow | no | Sets <code>errno</code> . See (2) |
| printf | no | Uses <code>iob[]</code> . See (1) |
| putc | no | Uses <code>iob[]</code> . See (1) |
| putchar | no | Uses <code>iob[]</code> . See (1) |
| puts | no | Uses <code>iob[]</code> . See (1) |
| qsort | yes | — |
| raise | no | Updates the signal handler table |
| rand | no | Uses static variable to remember latest random number. Must diverge from ANSI standard to define reentrant <code>rand</code> . See (4). |
| read | no | Calls <code>_read</code> |
| <code>_read</code> | no | Uses global File System Simulation buffer, <code>_fss_buffer</code> |
| realloc | no | See <code>malloc</code> (5). |
| remove | — | Skeleton only. |
| rename | — | Skeleton only. |
| rewind | — | Skeleton only. |
| sbrk | no | Allocates memory which is assigned at locate time. Needs kernel for memory management. |
| scanf | no | Uses <code>iob[]</code> , calls <code>_doscan</code> . See (1). |
| setbuf | no | Sets <code>iob[]</code> . See (1). |
| setjmp | yes | — |
| setlocale | — | Skeleton function |

| Function | Reentrant | Cause |
|----------|-----------|---|
| setvbuf | no | Sets iob and calls malloc. See (1) / (5). |
| signal | no | Updates the signal handler table |
| sin | yes | – |
| sinh | no | Sinh calls exp() which sets errno. If errno is discarded sinh is reentrant. |
| sprintf | no | Calls dprintf. See (1). |
| sqrt | no | Sets errno. See (2). |
| srand | no | See rand |
| sscanf | no | Calls _doscan |
| stat | no | Uses global File System Simulation buffer, _fss_buffer |
| strcat | yes | – |
| strchr | yes | – |
| strcmp | yes | – |
| strcoll | – | Skeleton function |
| strcpy | yes | – |
| strcspn | yes | – |
| strerror | yes | – |
| strftime | yes | – |
| strlen | yes | – |
| strncat | yes | – |
| strncmp | yes | – |
| strncpy | yes | – |
| strpbrk | yes | – |
| strrchr | yes | – |
| strspn | yes | – |
| strstr | yes | – |
| strtod | yes | – |
| strtok | no | Strtok saves last position in string in local static variable. This function is not reentrant by design. See (4). |
| strtol | no | Sets errno. See (2). |
| strtoul | no | Sets errno. See (2). |

| Function | Reentrant | Cause |
|----------|-----------|---|
| strxfrm | – | Skeleton function |
| system | – | Skeleton function |
| tan | no | Sets errno. See (2). |
| tanh | no | Uses sinh for calculation. |
| time | no | Uses static variable which defines initial start time |
| tmpfile | no | Uses iob[]. See (1). |
| tmpnam | no | Uses local buffer to build filename. Function can be adapted to use user buffer. This makes the function non ANSI. See (4). |
| toascii | yes | – |
| tolower | yes | – |
| toupper | yes | – |
| ungetc | no | Uses static buffer to hold ungetted characters for each file. Can be moved into iob structure. See (1). |
| unlink | no | Calls _unlink |
| _unlink | no | Uses global File System Simulation buffer, _fss_buffer |
| vfprintf | no | Uses iob[], calls dprintf. See (1). |
| vprintf | no | Uses iob[], calls dprintf. See (1). |
| vsprintf | no | Calls dprintf. |
| wcstombs | – | Skeleton function |
| wctomb | – | Skeleton function |
| write | no | Calls _write |
| _write | no | Uses global File System Simulation buffer, _fss_buffer |

Table 2-5: C library reentrancy

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. The following functions read or modify `errno`:
`acos, asin, _doprint, _doscan, exp, fgetpos, fsetpos, ftell, log, log10, perror, pow, rewind, sqrt, strerror, strtol, strtoul, tan`
- `_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` can be set to `ERR_NOFLOAT` by the scan functions if you use floating point formatting while using the `SMALL` formatting routines. See also the next section *Printf and Scanf Formatting Routines*.

`errno` will never be set to `ERR_NOLONG` or `ERR_NOPOINT` 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 `ungetc` 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 that the `sbrk()` function can do 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.

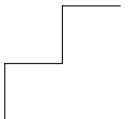


This paragraph on reentrancy applies to multi-process environments only. If reentrancy is required for calling library functions from an exception handler, another approach is required. For such a situation it is of no use to allocate e.g. multiple `ioB[]` structures. In such a situation several pieces 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

CHAPTER

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.

Overview of mathematical functions

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

Overview of conversion functions

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

Overview of string functions

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

Overview of macro functions

| Function | Description |
|----------------------------|-----------------------------------|
| @ARG(<i>symbol expr</i>) | Test if macro argument is present |
| @CNT() | Return number of macro arguments |
| @MAC(<i>symbol</i>) | Test if macro is defined |
| @MXP() | Test if macro expansion is active |

Overview of address calculation functions

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

Overview of assembler mode functions

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

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 dummy argument name. If the argument is an *expression* it refers to the ordinal position of the argument in the macro dummy 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?
```

@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 π . *Expr1* and *expr2* must be separated by a comma.

Example:

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


@ATAN(*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*)

Returns an integer 1 if *symbol* has been defined, 0 otherwise. *symbol* may be any 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 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 $((\text{expression} \gg 16) \& 0\text{xffff})$.

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 $((\text{expression} + 0\text{x800}) \gg 16) \& 0\text{xffff}$.

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:

```
LOG      .EQU    @L10(100.0)    ;LOG = 2
```

@LEN(*string*)

Returns the length of *string* as an integer.

Example:

```
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:

```
        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:

```
        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:

```
        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) & 0xffff) - 0x8000).

Example:

```
        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:

```
        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:

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

@LUN(expression)

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

Example:

```
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?
```

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

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

Example:

```
ID      .EQU  @POS('TriCore','Core')    ;ID = 3
ID2     .EQU  @POS('ABCDABCD','B',2)     ;ID2 = 5
```

@POW(*expr1*,*expr2*)

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

Example:

```
BUF     .EQU  @CVI(@POW(2.0,3.0))        ;BUF = 8
```

@RND()

Returns a random value in the range 0.0 to 1.0.

Example:

```
SEED    .EQU  @RND( )                  ;save initial SEED value
```

@RVB(*expr1*,*expr2*)

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

Example:

```
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
```

@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
```

@SQT(*expression*)

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

Example:

```
SQRT1 .EQU @SQT(3.5)           ;SQRT1 = 1.870829
SQRT2 .EQU @SQT(16)           ;SQRT2 = 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:

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

@TAN(*expression*)

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

Example:

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

@TNH(*expression*)

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

Example:

```
HTAN .SET @TNH(1)             ;HTAN = 0.76159415595
```

@UNF(*expression*)

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

Example:

```
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

| Directive | Description |
|-----------|--|
| .COMMENT | Start comment lines |
| .DEFINE | Define substitution string |
| .END | End of source program |
| .FAIL | Programmer generated error message |
| .INCLUDE | Include secondary file |
| .MESSAGE | Programmer generated message |
| .NAME | Identification for object file (instead of file name) |
| .ORG | Initialize memory space and location counters to create a nameless section |
| .SDECL | Declare a section with name, type and attributes |
| .SECT | Activate a declared section |
| .UNDEF | Undefine DEFINE symbol |
| .WARNING | Programmer generated warning |

Overview of symbol definition directives

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

Overview of data definition / storage allocation directives

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

Overview of macro and conditional assembly directives

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



Overview of debug directives

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

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** *argument*[,*argument*]...

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 *argument* can be:

- a numeric constant
- a single or multiple character string constant
- a symbol
- an expression
- 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. 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 on that address. The assembler fills the 'gap' with NOP instructions. If the location counter is already aligned on the specified alignment, it remains unchanged.

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 with this directive.

Example

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

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

Related information



—

.ASCII/.ASCIIZ

Syntax

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

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

Description

With the **.ASCII** or **.ASCIIZ** directive the assembler allocates and initializes memory each *string*.

The **.ASCII** directive does *not* add a NULL byte to the end of the string. The **.ASCIIZ** directive does add a NULL byte to the end of the string. Use commas to separate multiple strings.

Example

```
STRING: .ASCII  "Hello world"
```

```
STRING: .ASCIIZ "Hello world"
```



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

```
STRING: .BYTE  "Hello world"      ; without a NULL byte
```

```
STRING: .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 numeric constant
- a single or multiple character string constant
- a symbol
- an 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:

```
.BYTE 'R'           ; = 0x52
.BYTE 'AB' , , 'D'  ; = 0x41420043
```

Example

```
TABLE .BYTE 'two',0,'strings',0
CHARS .BYTE 'A','B','C','D'
```

Related information



.SPACE (Define storage)

.ASCII / **.ASCHZ** (Define ASCII string without/with ending NULL)

.WORD / **.HALF** (Define a word / halfword)

.CALLS

Syntax

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

Description

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

When you compile with the option **-g** (include debugging information), the compiler inserts these directives to pass call tree information. Normally it is not necessary to use the **.CALLS** directive in hand coded assembly.

A label is not allowed with this directive.

Example

To indicate that the function `main` calls the function `nfunc`:

```
.CALLS 'main', 'nfunc'
```

Related information



.COMMENT

Syntax

```
.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 with this directive.

Example

```
.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

.DEFINE *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*.

This directive is useful for providing better documentation in the source program. *Symbol* must adhere to the restrictions for labels. That is, 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 with this directive.

Example

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

```
.DEFINE  SIZE  '32'
```

then the source line below:

```
.SPACE  SIZE
```

would be transformed by the assembler to the following:

```
.SPACE  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 ; NOP BY COUNT  
      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  dummy,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 *dummy* 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
.BYTE  32
.BYTE  34
```

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 dummy,string  
.  
.  
.ENDM
```

Description

With the `.DUPC` and `.ENDM` directives (Duplicate Sequence with Characters) you can repeat a block of source statements for each character of *string*. For each repetition, every occurrence of the *dummy* parameter within the block is replaced with each succeeding character in the *string*. If the *string* is 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),
- .DUPA** (Duplicate Sequence with Arguments),
- .DUPF** (Duplicate Sequence in Loop),
- .MACRO** (Define Macro)

.DUPF / .ENDM

Syntax

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

Description

With the `.DUPF` and `.ENDM` directives (Duplicate Sequence in Loop) you can repeat a block of source statements $(end - start) + 1$ times when *increment* is 1. *Start* is the starting value for the loop index; *end* represents the final value. *Increment* is the increment for the loop index; it defaults to 1 if omitted (as does the *start* value). The *dummy* parameter holds the loop index value and may be used within the body of instructions.

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 with this directive.

Example

```
.END                ;End of source program
```

Related information



—

.EQU

Syntax

symbol **.EQU** *expression*

Description

With the **.EQU** directive you assign the value of *expression* to *symbol* permanently. Once defined, you cannot redefine the *symbol*.

The *expression* can be relative or absolute and forward references are allowed.

Example

To assign the value 0x4000 permanently to the symbol A_D_PORT:

```
A_D_PORT    .EQU    0x4000
```

You cannot redefine the symbol A_D_PORT after this.

Related information



.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 with 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*[,*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 with this directive.

Example

```
.EXTERN AA,CC,DD           ;defined elsewhere
```

Related information



.GLOBAL (Global symbol declaration)

.LOCAL (Local symbol declaration)

.FAIL

Syntax

.FAIL [{*str* | *exp*},{*str* | *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. The assembler outputs a space between each argument.

A label is not allowed with this directive.

Example

```
.FAIL 'Parameter out of range'
```

Related information



.MESSAGE (Programmer Generated Message),
.WARNING (Programmer Generated Warning)

.FLOAT/.DOUBLE

Syntax

[*label*] **.FLOAT** *argument*[,*argument*]...

[*label*] **.DOUBLE** *argument*[,*argument*]...

Description

With the **.FLOAT** or **.DOUBLE** directive the assembler allocates and initializes one word (32 bits) or a double-word (64 bits) of memory for each *argument*.

An *argument* can be:

- a numeric constant
- a single or multiple character string constant
- a symbol
- an 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.

Examples

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

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

Related information



.SPACE (Define storage)

.FRACT/.SFRACT

Syntax

[*label*:] **.FRACT** *argument*[,*argument*]...

[*label*:] **.SFRACT** *argument*[,*argument*]...

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 *argument* can be:

- a numeric constant
- a single or multiple character string constant
- a symbol
- an expression
- 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

With the **.GLOBAL** directive (Global Section Symbol Declaration) you declare one or 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.

Only symbols that are defined with the **.EQU** directive 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 with 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 it reaches.

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 with 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
.ELIF DEMO
... ; code for the demo version
.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 -DDEMO -DTEST=0 test.src
```

Related information



.INCLUDE

Syntax

.INCLUDE *'string'* | *<string>*

Description

With the **.INCLUDE** directive you direct the assembler to include another file before the resulting file is assembled. The **.INCLUDE** directive works similarly to the `#include` statement in C.

The *string* specifies the filename of the secondary file. The filename must be compatible with the operating system and can include a directory specification.

The order in which the assembler searches for include files is:

1. The current directory if only a filename is given, unless the *<string>* syntax is used, or in the directory specified in *string* if you specify both a pathname and a filename.
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 default directory `...\ctc\include`.

A label is not allowed with 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 4.2, *Assembler Options*, of Chapter *Tool Options*.

.LOCAL

Syntax

.LOCAL *symbol* [, *symbol*] ...

Description

By default, labels in a module are defined "global". With the **.LOCAL** directive (Local Section Symbol Declaration) you declare one or more symbols as local. This means that the specified symbols are explicitly local to the section or 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 with this directive.

Example

```
.SDECL  ".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

```
macro_name .MACRO [dumarg[,dumarg]...]
           .
           macro_definition_statements
           .
           .
           .ENDM
```

Description

With the `.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 `.ENDM` directive indicates the end of the macro.

The definition of a macro consists of three parts:

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

The dummy arguments are symbolic names that the macro processor replaces with the literal arguments when the macro is expanded (called). Each dummy argument must follow the same rules as symbol names. Dummy 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.

Example

The macro definition:

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

The macro call:

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

CONSTD    d4,0x12345678

.END
```

The macro expands as follows:

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

Related information



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

Section 4.9, *Macro Operations*, in Chapter *Assembly Language* of the *User's Guide*.

.MESSAGE

Syntax

```
.MESSAGE [{str | exp},{str | exp}...]
```

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. The assembler outputs a space between each argument.

A label is not allowed with this directive.

Example

```
.DEFINE LONG "SHORT"  
.MESSAGE 'This is a LONG string'  
.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:

```
This is a LONG string  
This is a SHORT string
```

Related information



.FAIL (Programmer Generated Error)

.WARNING (Programmer Generated Warning)

.NAME

Syntax

```
.NAME  "str"
```

Description

With the `.NAME` directive you give an identification to the generated object file. The linker and or debugger uses this identification (instead of the file name) to refer to the file.

When you use the control program **cctc**, this name may become a random name.

Example

Suppose the assembler assembles the file `test.src` and generates `test.o`. To change the identification (used by the linker and debugger) from the name "test" into "strcat":

```
.NAME  "strcat"
```

.ORG

Syntax

.ORG [*abs-loc*][,*sect_type*][,*attribute*]...

Description

With the **.ORG** directive you can specify an absolute location (*abs_loc*) in memory of a section. This is the same as a **.SDECL/.SECT** without a section name.

This directive uses the following arguments:

abs-loc Initial value to assign to the run-time location counter.
abs-loc must be an absolute expression. If *abs_loc* is not specified, then the value is zero.

sect_type An optional 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 |
| noexec | 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 |

A label is not allowed with this directive.

Example

```
; define a section on location 100 decimal
.org 100

; define a relocatable nameless section
.org

; define a relocatable data section
.org ,data
```

```
; define a data section on 0x8000  
.org    0x8000,data
```

Related information



.SDECL (Declare section name and attributes)

.SECT (Activate a declared section)

.PMACRO

Syntax

.PMACRO *symbol[,symbol]...*

Description

With the **.PMACRO** directive you tell the assembler to purge the specified macro from the macro table, reclaiming the macro table space.

A label is not allowed with this directive.

Example

```
.PMACRO  MAC1,MAC2
```

This statement causes the macros named **MAC1** and **MAC2** to be purged.

Related information



.MACRO (Define Macro)

.SDECL

Syntax

.SDECL "*name*", *type* [, *attr*]... [**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.

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

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

symbol **.SET** *expression*

.SET *symbol expression*

Description

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

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

Example

```
COUNT .SET 0 ; Initialize COUNT. Later on you can
           ; assign other values to the symbol COUNT.
```

Related information



..EQU (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:
    .          ; function main
    .
    ret
    .SIZE main, (*-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 bytes in memory. The reserved block of memory is not initialized to any value.

With *expression* you specify the number of bytes 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).

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

Example

```
S_BUF .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)

.SYMB

Syntax

.SYMB *string*, *expression* [, *abs_expr*] [, *abs_expr*]

Description

When you compile with the option **-g** (include debugging information), the compiler inserts the **.SYMB** directives to pass high-level language symbolic debug information via the assembler (and linker) to the debugger.

Expression can be any expression. *Abs_expr* can be any expression resulting in an absolute value.

The **.SYMB** directive is not meant for hand coded assembly and is documented here for completeness only.

Example

If you compile a C file `test.c` with the **-g** option, you might find the following **.SYMB** directives in your assembly listing file:

```
.symb TOOL, "TASKING TriCore C compiler vx.y", 1
.symb TYPE, 256, "bit", 'g', 0, 1
.symb FILE, "test.c"
```

Related information



.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. |

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 with this directive.

Example

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

Related information



.DEFINE (Define Substitution String)

.WARNING

Syntax

.WARNING [{*str* | *exp*},{*str* | *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. The assembler outputs a space between each argument.

A label is not allowed with this directive.

Example

```
.WARNING 'parameter too large'
```

Related information



.FAIL (Programmer Generated Error),
.MESSAGE (Programmer Generated Message)

.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 numeric constant
- a single or multiple character string constant
- a symbol
- an 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.

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 represent in a word / halfword, the assembler issues an error.

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

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

Overview of miscellaneous assembler controls

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

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. Default the assembler operates in case sensitive mode.

Example

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

Related option



Assembler option `-c` (Switch to case insensitive mode) in section 4.2, *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 `-gl`).

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 4.2, *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. 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 4.2, *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 4.2, *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**.

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 4.2, *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  
$NOLIST OFF        ; Counter value = 2  
$NOLIST OFF        ; Counter value = 1
```

The listing still would not be disabled until another **NOLIST** control was issued.

A label is not allowed with this control.

Example

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

```
.SDECL ".text",CODE  
.SECT ".text"  
... ; source line in list file  
$LIST ON  
... ; source line not in list file  
$LIST  
... ; 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 4.2, *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 |
| s/S | Lines with symbolic debug information (.SYMB) |
| 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 **-lflag**, the assembler uses the default: **-LcDEGiLMnPqsVWXy**.

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 4.2, *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 4.2, *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 4.2, *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.

| | |
|------------------|--|
| <i>width</i> | Number of characters on a line (1–255). Default is 132. |
| <i>length</i> | 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. |
| <i>blanktop</i> | Number of blank lines at the top of the page. Default = 0. Specify a value so that $blanktop + blankbtm \leq length - 10$. |
| <i>blankbtm</i> | Number of blank lines at the bottom of the page. Default = 0. Specify a value so that $blanktop + blankbtm \leq length - 10$. |
| <i>blankleft</i> | Number of blank columns at the left of the page. Default = 0. Specify a value smaller than <i>width</i> . |

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 **\$STITLE** (Set program subtitle in header of list file)

Assembler control **\$TITLE** (Set program title in header of list file)

Assembler option **-I** (Generate list file) in Section 4.2, *Assembler Options*, of Chapter *Tool Options*.

Assembler option **-L** (List file formatting options) in Section 4.2, *Assembler Options*, of Chapter *Tool Options*.

\$PRCTL

Syntax

\$PRCTL *exp* | *string* [, *exp* | *string*] ...

Description

If you generate a list file with the assembler option **-l**, you can use the **\$PRCTL** control to send control strings to the printer.

The **\$PRCTL** 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:

| | |
|---------------|--|
| <i>exp</i> | a byte expression which may be used to encode non-printing control characters, such as ESC. |
| <i>string</i> | an assembler string, which may be of arbitrary length, up to the maximum assembler-defined limits. |

The **\$PRCTL** 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 **\$PRCTL** 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 **PRCTL** control to restore a printer to a previous mode after printing is done.

Similarly, if the **\$PRCTL** control appears as the first line in the first input file, the assembler sends out the control string before page headings or titles.

Example

```
$PRCTL $lB, 'E' ; Reset HP LaserJet printer
```

Related option



–

Related information



Assembler option **-l** (Generate list file) in Section 4.2, *Assembler Options*, of Chapter *Tool Options*.

\$PRINT / \$NOPRINT

Syntax

\$PRINT(*file*)
\$NOPRINT

Description

If you generate a list file with the assembler option **-l**, you can use the **\$PRINT** control to specify an alternative name for the assembly list file.

With the **\$NOPRINT** control, you override the assembler option **-l** and no list file is generated at all.

Example

```
$PRINT(mylist.lst)    ; generate an assembler list file  
                      with the name 'mylist.lst'.
```

Related option



–

Related information



Assembler option **-l** (Generate list file) in Section 4.2, *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. 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

```
$TITLE    'This is the title'
$STITLE   'This is the subtitle'
```

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                                     Page 2
This is the subtitle
```

Related option



Related information



Assembler control **\$TITLE** (Set program title in header of list file)

Assembler option **-l** (Generate list file) in Section 4.2, *Assembler Options*, of Chapter *Tool Options*.

\$TC

Syntax

```
$TCdefect ON
$TCdefect OFF
```

Description

With this control you can enable or disable specific CPU functional problem checks.

To enable the assembler checks for *all* TriCore CPU TC112 problems (respectively TC113 problems) at once, use the control \$TC112_DEFECTS (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 4.2, *Assembler Options*, of Chapter *Tool Options*.

Related information



See Chapter 8, *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 4.2, *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.

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                                     Page 1
```

Related option



Related information



\$TITLE (Set program subtitle in header of assembly list file)

\$WARNING OFF

Syntax

\$WARNING OFF
\$WARNING OFF *number*

Description

With the `$WARNING OFF` control you can suppresses all warning messages or specific warning messages.

- 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

```
$WARNING OFF      ; all warning messages are suppressed  
  
$WARNING OFF 135 ; suppress warning message 135
```

Related option



Assembler option `-w` (Suppress some or all warnings) in section 4.2, *Assembler Options*, of Chapter *Tool Options*.

Related information



—

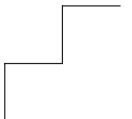


ASSEMBLY LANGUAGE

CHAPTER

4

TOOL OPTIONS



4

CHAPTER

4.1 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

Description

Displays an overview of all command line options.

Example

The following invocations all display a list of the available command line options:

```
ctc -?  
ctc --help  
ctc
```

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 language extension keywords** and **Allow C++ style comments in ISO C90 mode**.

Command line syntax

-A[flags]

--language=[flags]

You can set the following flags:

| | | |
|------------|------------------------|-------------------------------------|
| k/K | (+/- keywords) | Allow language extension keywords |
| p/P | (+/- comments) | Allow C++ style comments in ISO C90 |

Default

-Akp

Description

With this option you control the language extensions the compiler can accept. Default the TriCore C compiler allows all language extensions.

-A is the equivalent of **-AKP** and disables all language extensions.

With **-Ak** you tell the compiler to allow language extension keywords, such as `__fract`. Use **-AK** to tell the compiler to generate a syntax error when it finds a language extension keyword in your C source.

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.

Example

```
ctc -AkP -c90 test.c  
ctc --language=+keywords,-comments --iso=90 test.c
```

The compiler compiles in ISO C90 mode, accepts keywords 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 **Miscellaneous**.
3. Add the option **--align** to the **Additional options** 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: Pragas*, in Chapter *TriCore C Language* of the *TriCore User's Guide*.

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

Command line syntax

-Ccpu

--cpu=cpu

Description

With this option you define the target processor for which you create your application.

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 automatically includes the register file `regcpu.sfr`, unless you specify compiler option **--no-tasking-sfr**.

Example

In EDE, the target CPU has the following settings:

- Target processor: TC10GP

To define this on the command line:

```
ctc -Ctc10gp test.c
ctc --cpu=tc10gp test.c
```

The compiler compiles `test.c` for the TC10GP processor and includes the register file `regtc10gp.sfr`.



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 5.5, *Specifying a Target Processor*, in Chapter *Using the Compiler* of the *User's Guide*.

-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 C90 standard on the command line:

```
ctc -c90 test.c
ctc --iso=90 test.c
```

Related information



Compiler option **-A** (Language extensions)

--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)

-D (--define)

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 a macro name and/or definition in the **Define user macros** field.

Use commas to separate multiple macro definitions.

Command line syntax

-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. In EDE, use commas to separate multiple macro definitions. On the command line, 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 compiler with the option **-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:

```
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:

```
ctc -DDEMO test.c
ctc -DDEMO=1 test.c

ctc --define=DEMO test.c
ctc --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.

```
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.

If you want the output in a file, you have to use output redirection.

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:

```
ctc --diag=html:all > errors.html
```

Related information



Section 5.9, *C Compiler Error Messages*, in Chapter *Using the Compiler* of the *User's Guide*.

-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)**.
4. Enable or disable the options **Keep comments** and **Strip #line source position info**.

Command line syntax

-E[/flags]

--preprocess[=/flags]

You can set the following flags (when you specify **-E** without flags, the default is **-ECP**):

| | | |
|------------|------------------------|----------------------------------|
| c/C | (+/- comments) | Keep comments |
| p/P | (+/- noline) | Strip #line source position info |

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 **-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

```
ctc -EcP test.c -o test.pre
```

```
ctc --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

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 **Miscellaneous**.
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 (--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:

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 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 *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 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:

```
-Ctcl0gp  
-s  
test.c
```

Specify the option file to the compiler:

```
ctc -f myoptions
```

This is equivalent to the following command line:

```
ctc -Ctcl0gp -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 a **(user defined TriCore)** option.
4. Enable the option **FPU present (on user defined CPU)**.
5. Expand the **C Compiler** entry and select **Miscellaneous**.
6. Enable the option **Use hardware single precision floating point instructions**.

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.

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 (--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:

```
ctc -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 5.6, *How the Compiler Searches Include Files*, in Chapter *Using the Compiler* of the *User's Guide*.

-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

-I*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 c:\ctc\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 -Iinclude test.c
```

First the compiler looks in the directory where `test.c` is located for the file `myinc.h`.

Then the compiler looks in the `include` subdirectory relative to the current directory for the `stdio.h` file and, if it was not found yet, also for the `myinc.h` file (this option).

If the file(s) are 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)

Section 5.6, *How the Compiler Searches Include Files*, in Chapter *Using the Compiler* of the *User's Guide*.

Section 1.3.2, *Configuring the Command Line Environment*, in Chapter *Software Installation* of the *User's Guide*.

--indirect

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 **Call functions indirect**.

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.

Related information



See also section 3.9.3, *Function Calling Modes: __indirect*, in Chapter *TriCore C Language* of the *User's Guide*.

--inline-max-incr / --inline-max-size

EDE

–

Command line syntax

--inline-max-incr=*percentage*

--inline-max-size=*threshold*

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 *threshold*. 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 Guide*.

--integer-enumeration

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 **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 uses the smallest suitable integer type.

Example

```
ctc --integer-enumeration test.c
```

The compiler uses 32-bit integers for enumerations.

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

--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.

Example

To allow the use of TriCore 2 instructions in the assembly code, enter:

```
ctc --is-tricore2 test.c
```

Related information



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 9 *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. Enter a threshold value in the **Threshold for '___near' allocation** field.

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` if you specified

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 **-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 Guide*.

-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

```
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

With this option the compiler does not include the register file `regcpu.sfr` as based on the compiler option **-C**.

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

-O[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 |
| 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: -OACEFGIKLMOPSWY |
| -O1 | (--optimize=1) | Few optimizations Alias for: -OaCefgIKLMOPSWy |
| -O2 | (--optimize=2) | Medium optimization (default) Alias for: -OacefgIklMopswy |

-O3 (**--optimize=3**) Full optimization
Alias for: **-Oacefgiklmopswy**

Default

-O2

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 **-O2** or **-O** or **-Oacefgiklmopswy**).

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` and `#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 -O2 test.c
ctc --optimize=2 test.c

ctc -O test.c
ctc --optimize test.c

ctc -Oacefgiklpwy test.c
ctc --optimize=+coalesce,+cse,+expression,+flow,
    +glo,-inline,+schedule,+loop,+propagate,
    +subscript,+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 Guide*.

-o (--output)

EDE

–

Command line syntax

-o*file*

--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



-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 memory type abbreviation, the module name and a symbol name, for example `.text.module_name.symbol_name` for code sections. In case a module must be loaded at a fixed address or a data section needs a special place in memory, you can use the **-R** option to generate a different section name (*section_type.name*). You can now use this 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.NEW* instead of the default section name *section_type.mod_name*, enter:

```
ctc -RNEW test.c
```

To generate the section name *section_type* instead of the default section name *section_type.mod_name*, enter:

```
ctc -R"" test.c
```

Related information



Section 3.10, *Compiler Generated Sections*, in Chapter *TriCore C Language* of the *User's Guide*.

-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

-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

```
ctc -s test.c
```

The output file `test.src` contains the original C source lines as comments, besides the generated assembly code.

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:

| | |
|------------------|--|
| all-tc112 | All TriCore 1 v1.2 (TC112) workarounds |
| all-tc113 | All TriCore 1 v1.3 (TC113) workarounds |
| cor1 | workaround for TC112 COR1 |
| cor4 | workaround for TC112 COR4 |
| cor7 | workaround for TC112 COR7 |
| cor10 | workaround for TC112 COR10 |
| cor13 | workaround for TC112 COR13 |
| cor14 | workaround for TC112 COR14 |
| cor16 | workaround for TC112 COR16 |
| cor17 | workaround for TC112 COR17 |
| cpu5 | workaround for TC113 CPU5 |
| cpu9 | workaround for TC113 CPU9 |
| cpu11 | workaround for TC113 CPU11 |
| cpu14 | workaround for TC113 CPU14 |
| cpu15 | workaround for TC113 CPU15 |
| cpu16 | workaround for TC113 CPU16 |
| dmu1 | workaround for TC113 DMU1 |
| lfi2 | workaround for TC113 LFI2 |
| lfi3 | workaround for TC113 LFI3 |

Description

With this option you tell the compiler to use software workarounds for some CPU functional problems.

Example

```
ctc --silicon-bug=cpu5,cpu9 test.c
```

The compiler uses workarounds for TC113 problems CPU5 and CPU9.

Related information

See Chapter 8, *CPU Functional Problems*, for more information about the individual problems and workarounds.

--switch

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 **--switch** to the **Additional options** field.

Command line syntax

--switch=arg

You can give one of the following arguments:

| | |
|----------------|--|
| auto | Compiler determines the best switch method |
| linear | Use linear jump chain code |
| jumptab | Generate jump tables |
| lookup | Generate lookup tables |

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 Guide*.

-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**.
3. Select a trade-off level in the **Size/speed trade-off** box.

Command line syntax

-t{0 | 1 | 2 | 3 | 4}

--tradeoff= {0 | 1 | 2 | 3 | 4}

Default

-t0

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 optimizes the selected optimizations for more speed (**-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

`-Umacro_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:

```
#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:

```
__FILE__ current source filename
__LINE__ current source line number (int type)
__TIME__ hh:mm:ss
__DATE__ Mmm dd yyyy
```

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 *Users Guide*.

-u (--uchar)

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 **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 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:

```
ctc test.c -w
ctc test.c --no-warnings
```

To suppress warnings 135 and 136:

```
ctc test.c -w135 -w136
ctc 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)

-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. Enter a threshold value in the **Threshold for '___a0' allocation** field.

Command line syntax

-Z[*threshold*]

--default-a0-size[=*threshold*]

Default

-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 **-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 Guide*.

4.2 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

Description

Displays an overview of all command line options.

Example

The following invocations all display a list of the available command line options:

```
astc -?  
astc --help  
astc
```

Related information



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

Command line syntax

`-Ccpu`

`--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 -Ctc11ib test.src
astc --cpu=tc11ib test.src
```

The assembler assembles `test.src` for the TC11IB processor and includes the register file `regtc11ib.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 6.5, *Specifying a Target Processor*, in Chapter *Using the Assembler* of the *User's Guide*.

-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. Default the assembler considers upper and lower case characters as different characters.



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)

-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

-D*macro_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 **.DEFINE**, **.SET**, and **.EQU** directives. (similar to **#define** in the C language). With the **.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 4.9.5, *Conditional Assembly*, in Chapter *TriCore Assembly Language* of the *User's Guide*.

--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 assembler displays a description and explanation of the specified error message(s) on stdout (usually the screen). The assembler does not assemble any files.

If you want the output in a file, you have to use output redirection.

Example

To display an explanation of message number 240, enter:

```
astc --diag=240
```

This results in the following message and explanation:

```
W240: 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



Section 6.8, *Assembler Error Messages*, in Chapter *Using the Assembler* of the *User's Guide*.

--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 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:

```
-Ctcl0gp  
test.src
```

Specify the option file to the assembler:

```
astc -f myoptions
```

This is equivalent to the following command line:

```
astc -Ctcl0gp 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 (on user defined CPU)**.
5. Expand the **Assembler** entry and select **Miscellaneous**.
6. Enable the option **Allow hardware floating point instructions**.

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. 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**).

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

-H*file*,...

--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
```

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 6.6, *How the Assembler Searches Include Files*, in Chapter *Using the Assembler* of the *User's Guide*.

-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

-I*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 pathname in the assembly file and the directory of the assembly source.
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 directory `c:\ctc\include`.

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
```

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



Section 6.6, *How the Assembler Searches Include Files*, in Chapter *Using the Assembler* of the *User's Guide*.

Section 1.3.2, *Configuring the Command Line Environment*, in Chapter *Software Installation* of the *User's Guide*.

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.

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

--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:

```
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 **mkc**. 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

```
astc -k test.src
```

When an error occurs during assembly, the generated output file test.o will *not* be removed.

Related information



Assembler option **--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

```
-Lflags  
  
--list-format=flags
```

You can set the following flags:

| | | |
|------------|------------------------|------------------------------------|
| 0 | | same as -LCDEGILMNPQSVWXY |
| 1 | | same as -Lcdegilmnpqsvwxy |
| 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 |
| s/S | (+/-hll) | HLL symbolic debug information |
| 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

```
-LcDEGiLmNpQsVWXy
```

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:

-LcDEGIJmNPqsVWXy.

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)

Section 5.1, *Assembler List File Format*, in Chapter *List File Formats*.

-l (--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 5.1, *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. 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 **Assembler** entry and select **Miscellaneous**.
3. Enable the option **Allow memory management instructions**.



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

With this option the assembler does not include the register file `regcpu.def` as based on the assembler option **-C**.

Use this option if you want to use your own set of SFR '.def' files.

Example

```
astc -Ctcllib --no-tasking-sfr test.src
```

The register file `regtcllib.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



Section 6.3, *Assembler Optimizations*, in Chapter *Using the Assembler* of the *User's Guide*.

-o (--output)

EDE

–

Command line syntax

-o*file*

--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 |
| all-tc113 | All TriCore 1 v1.3 (TC113) checks |
| 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 |
| 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 assembler 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 8, *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

-tflags

--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 (REP/DO). 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

```
astc -V
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.

Related information



Assembler option **-w** (suppress some or all warnings)

4.3 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 4.4, *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 **-Wlk** and **-Wlc** to pass the option via the control program directly to the link and locate phase of the linker.*

*For example, if you enter the option **-DDEMO** in the **Additional options** field, EDE generates the options **-Wlk-DDEMO** **-Wlc-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:

```
ltd -mfl test.o
ltd --map-file-format=+files,+link,+locate test.o
```

When you do not specify an option, a default value may become active.

-?/-H (--help)

EDE

-

Command line syntax

-?
-H
--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 -H  
ltc --help  
ltc
```

Related information



--case-insensitive

EDE

1. From the **Project** menu, select **Project Options...**
The Project Options dialog appears.
2. Expand the **Linker/Locator** entry and select **Linker**.
3. Disable the option **Link case sensitive**.

Command line syntax

--case-insensitive

Description

With this option you tell the linker to consider upper and lower case characters the same. 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:

```
ltdc --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`.

Using the control program to pass the option directly to the linker:

```
cctc -Wlk--case-insensitive test.o
```

Related information



Assembler option **-c** (Assemble case insensitive)

-c (--chip-format)

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.

Command line syntax

`-cformat[:addr_size][,format[:addr_size]]...`

`--chip-format=format[:addr_size][,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 memory chip.

Use option **-F (--format)** to specify an absolute (debugging) output format.

Examples

Generate Intel Hex output files for each chip:

```
ltc -cIHEX test1.o test2.out
ltc --chip-format=IHEX test1.o test2.out
```

Related information



Linker option **-F** (output format),
Section 6.2, *Motorola S-Record Format*,
Section 6.3, *Intel Hex Record Format*, in Chapter *Object File Formats*.

-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

-D*macro_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 **-f***file*.

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 `tc1v1_3.lsl`, enter:

```
ltc test.o -otest.elf -dtc1v1_3.lsl -DRESET=0xa0000000
-DINTTAB=0xa00f0000 -DTRAPTAB=0xa00f2000
```

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 standard description for selected derivative** or select **Use project specific processor description** 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 does not use a 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 partition describes the core's hardware architecture.
- the memory partition describes the physical memory available in the system.
- the section partition 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
```

Using the control program:

```
cctc -dte.lsl test.o
```

Related information

Chapter 7, *Linker Script Language*.

--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 linker displays a description and explanation of the specified error message(s) on stdout (usually the screen). The linker does not process any files.

If you want the output in a file, you have to use output redirection.

Example

To display an explanation of message number 104, enter:

```
ltc --diag=104
```

This results in the following message and explanation:

```
E104: unresolved external(s)
```

```
The linker could not resolve all external symbols. This is an
error when the incremental linking option is disabled.
```

To write an explanation of all errors and warnings in HTML format to file `lerrors.html`, enter:

```
ltc --diag=html:all > lerrors.html
```


Related information

Section 7.9, *Linker Error Messages*, in Chapter *Using the Linker* of the *User's Guide*.

-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 `_START` in the library and (if found) extracts the object that contains `_START`, the startup code. If this module contains new unresolved symbols, the linker looks again in `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 Guide*.

--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 `task1.elk`.

Example

```
ltdc --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 (--format)

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.

Command line syntax

-F*format*

--format=*format*

You can specify the following formats:

| | |
|-------------|-----------|
| ELF | ELF/DWARF |
| IEEE | IEEE-695 |

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.

Use option **-c (--chip-format)** to create Intel Hex or Motorola S-record output files for loading into a PROM-programmer.

Examples

Generate ELF/DWARF output file:

```
ltc -FELF test1.o test2.out -otest.elf
ltc --format=ELF test1.o test2.out --output=test.elf
```

Related information



Linker option **--chip-format** (Hex files per chip)
Section 6.1, *ELF/DWARF Object Format*, in Chapter *Object File Formats*.

-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 at 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:

```
ltdc --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)

-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 **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 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

```
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 (--library-directory / --ignore-default-library-path)

EDE

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

The Directories dialog appears.

2. Add a pathname in the **Library Files Path** field.
3. In the **Library Files Path** field, add the pathnames of the directories where the linker should look for library files.

If you enter multiple paths, separate them with a semicolon (;).

Command line syntax

```
-Lpath,...  
--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.

By default path this is \$(PRODDIR)\ctc\lib directory.

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 variable LIBTC1V1_2, LIBTC1V1_3 or 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 **-l** option is:

1. The path that is specified with the **-L** option.
2. The path that is specified in the environment variable LIBTC1V1_2, LIBTC1V1_3 or LIBTC2 when the product was installed.

3. The default directory `c:\ctc\lib`.
4. The processor specific directory, for example `c:\ctc\lib\tc1`.

Example

Suppose you call the linker as follows:

```
ltd 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 `LIBTC1V1_2`, `LIBTC1V1_3` or `LIBTC2`.

Then the linker looks in the default directory `c:\ctc\lib` for libraries.

Related information



Linker option **-l** (Link system library)

Section 7.4.2, *How the Linker Searches Libraries*, in Chapter *Using the Linker* of the *User's Guide*.

Section 1.3.2, *Configuring the Command Line Environment*, in Chapter *Software Installation* of the *User's Guide*.

-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 Guide*.

--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 `hello.out`.

Using the control program:

```
cctc -cl hello.o
```

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:

```
ltdc --lsl-check -dmylslfile.lsl
```

Related information



Linker option **-d** (Linker script file)

Linker option **--lsl-dump** (Dump LSL info)

--lsl-dump

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 **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`):

```
ltd -Mtest.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)

Section 5.2, *Linker Map File Format*, in Chapter *List File Formats*.

-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:

| | | | |
|------------|----------------|---------------------------|---------------|
| 0 | | same as -mcfkLMorS | (link info) |
| 1 | | same as -mCfklmoRs | (locate info) |
| 2 | | same as -mcfkImors | (all) |
| c/C | (+/-callgraph) | Call graph info | |
| f/F | (+/-files) | Processed files info | |
| k/K | (+/-link) | Link result info | |
| l/L | (+/-locate) | Locate result info | |
| m/M | (+/-memory) | Memory usage info | |
| o/O | (+/-overlay) | Overlay info | |
| r/R | (+/-crossref) | Cross references info | |
| s/S | (+/-lsI) | Processor and memory info | |

Default

-mCfklMORS

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: **-mCfklMORS**.

Example

```
ltd -Mtest.map -mFr test.o
ltd --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)

Section 5.2, *Linker Map File Format*, in Chapter *List File Formats*.

--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 **--misrac**

-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

-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 section. 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

```
ltdc -N test.o  
ltdc --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 at 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 at 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:

```
ltdc --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

-O*flags*

--optimize=*flags*

You can set the following flags:

- l/L** **(+/-first-fit-decreasing)**
Use a 'first fit decreasing' algorithm to locate unrestricted sections in memory.
- t/T** **(+/-copytable-compression)**
Locate (unrestricted) sections in such a way that the size of the copy table is as small as possible.

Use the following options for predefined sets of flags:

- | | |
|----------------------------------|--|
| -O0 (--optimize=0) | No optimization. Alias for: -OLT |
| -O1 (--optimize=1) | Normal optimization (default). Alias for: -OLt |
| -O2 (--optimize=2) | All optimizations. Alias for: -Olt |

Default

-O1

Description

With this option you can control the level of optimization. If you do not use this option, **-OLt** is the default.

Example

The following invocations are equivalent and result all in the default optimizations.

```
ltc test.o
ltc -OLt test.o
ltc --optimize=-first-fit-decreasing,
    +copytable-compression test.o
```

Related information



Section 7.2.3, *Linker Optimizations*, in Chapter *Using the Linker* of the *User's Guide*.

-o (--output-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 **-o** in the **Additional options** field.

Command line syntax

`-ofile`
`--output-file=file`

Description

By default, the linker generates the file `task1.elf`.

With this option you specify another name for the linker output file.

EDE and the control program name the output file always after the first input file with the extension `.elf`.

Example

To create the output file `test.elf` instead of `task1.elf`:

```
ltc test.o -otest.elf
```

Related information



-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 -otest.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



Section 7.5, *Incremental Linking*, in Chapter *Using the Linker* of the *User's Guide*.

-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

```
ltdc -S test.o -otest.elf
ltdc --strip-debug test.o --output=test.elf
```

The linker generates the object file `test.elf` without symbolic debug information.

Related information



Linker option **-M** (Generate map file)

-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 "/etc/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)

--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 ($\neq 0$) 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)

4.4 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 4.3, *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.

—?

Command line syntax

—?

Description

Displays an overview of all command line options.

Example

The following invocations all display a list of the available command line options:

```
cctc -?  
cctc
```

Related information



-C

Command line syntax

`-Ccpu`

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 TC10GP.

Based on the target processor, the compiler includes the register file `regcpu.sfr` and the assembler includes the file `regcpu.def`.

Example

In EDE, the target CPU has the following settings:

- Target processor: TC10GP

To define this on the command line:

```
cctc -Ctc10gp test.c
```

The control program generates an absolute object file `test.elf` for the TC10GP processor. The compiler includes the register file `regtc10gp.sfr` and the assembler includes the file `regtc10gp.def`.

Related information



Compiler option **-C** (Use SFR definitions for CPU)

Assembler option **-C** (Select CPU)

Section 5.5, *Specifying a Target Processor*, in Chapter *Using the Compiler* of the *User's Guide*.

-C++

Command line syntax

-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

```
cctc -c++ test.c
```

The file `test.c` is considered to be a C++ file.

Related information



Control program option **-noc++** (Force C++ files to C mode)

-cc/-cs/-c/-cl

Command line syntax

-cc
-cs
-c
-cl

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 or C files are compiled to assembly (.src) |
| -c | Stop after the files are assembled to objects (.o) |
| -cl | Stop after the files are linked to a linker object file (.out) |

Example

To generate the object file `test.o`:

```
cctc -c test.c
```

The control program stops after the file is assembled. It does not link nor locate the generated output.

Related information



—

-cm

Command line syntax

-cm

Description

With this option you force the control program to invoke the C++ muncher.

Example

```
cctc -cm test.cc
```

The control program always invokes the C++ muncher when generating `test.elf`.

Related information



-

-cp

Command line syntax

-cp

Description

With this option you force the control program to invoke the C++ pre-linker.

Example

```
cctc -cp test.cc
```

The control program always invokes the C++ pre-linker when generating `test.elf`.

Related information



-elf

Command line syntax

-elf

Description

With this option you tell the control program to generate an ELF/DWARF object file.

Example

```
cctc -elf test.c
```

The control program generates the ELF/DWARF object file `test.elf` from `test.c`.

Related information



Linker option **-Fformat** (Output format)

Section 6.1, *ELF/DWARF Object Format*, in Chapter *Object File Formats*.

-f

Command line syntax

-f *file*

Description

Instead of typing all options on the command line, you can create a 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 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:

```
-nomap  
-elf  
test.c
```

Specify the option file to the control program:

```
cctc -m myoptions
```

This is equivalent to the following command line:

```
cctc -nomap -elf test.c
```

-fptrap

Command line syntax

-fptrap

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 -fptrap test.c
```

Link the trapping floating point library when generating the object file `test.elf`.

Related information



–

-ieee

Command line syntax

-ieee

Description

With this option you tell the control program to generate an IEEE-695 object file.

Example

```
cctc -ieee test.c
```

The control program generates the IEEE-695 object file `test.abs` from `test.c`.

Related information



Linker option **-Fformat** (Output format)

-ihex

Command line syntax

-ihex

Description

With this option you tell the control program to generate an Intel hex object file.

Example

```
cctc -ihex test.c
```

The control program generates the Intel hex object file `test.hex` from `test.c`.

Related information



Linker option **-Fformat** (Output format)

Section 6.3, *Intel Hex Record Format*, in Chapter *Object File Formats*.

-noc++

Command line syntax

-noc++

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

```
cctc -noc++ test.cc
```

The C++ file `test.cc` is considered to be a normal C file.

Related information



Control program option **-c++** (Force C files to C++ mode)

-nolib

Command line syntax

-nolib

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 **-l***library_name*. The control program recognizes the option **-l** as an option for the linker.

Example

```
cctc -nolib test.c
```

The control program does not specify any libraries to the linker. In normal cases this would result in unresolved externals.

To specify your own libraries (libmy.a) and avoid unresolved externals:

```
cctc -nolib -lmy test.c
```

Related information



Linker option **-l** (Search also in system library libx.a)

-nomap

Command line syntax

-nomap

Description

Default the control program generates a linker map file (.map). With this option you tell the control program to skip the generation of a linker map file.

Example

```
cctc -nomap test.c
```

The control program does not generate the linker map file test.map.

Related information



Linker option **-M** (Generate map file)

—O

Command line syntax

-o*file*

Description

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

```
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`:

```
cctc -oresult.elf test.c prog.c
```

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:

| | |
|------------------|--|
| all-tc112 | All TriCore 1 v1.2 (TC112) workarounds |
| all-tc113 | All TriCore 1 v1.3 (TC113) workarounds |
| cor1 | workaround for TC112 COR1 |
| cor3 | workaround for TC112 COR3 |
| cor4 | workaround for TC112 COR4 |
| cor6 | workaround for TC112 COR6 |
| cor7 | workaround for TC112 COR7 |
| cor10 | workaround for TC112 COR10 |
| cor13 | workaround for TC112 COR13 |
| cor14 | workaround for TC112 COR14 |
| cor15 | workaround for TC112 COR15 |
| cor16 | workaround for TC112 COR16 |
| cor17 | workaround for TC112 COR17 |
| cpu5 | workaround for TC113 CPU5 |
| cpu9 | workaround for TC113 CPU9 |
| cpu11 | workaround for TC113 CPU11 |
| cpu13 | workaround for TC113 CPU13 |
| cpu14 | workaround for TC113 CPU14 |
| cpu15 | workaround for TC113 CPU15 |
| cpu16 | workaround for TC113 CPU16 |
| dmu1 | workaround for TC113 DMU1 |
| lfi2 | workaround for TC113 LFI2 |
| lfi3 | workaround for TC113 LFI3 |
| pmu1 | workaround for TC113 PMU1 |
| pmu3 | workaround for TC113 PMU3 |

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 8, *CPU Functional Problems*, for more information about the individual problems and workarounds.

-srec

Command line syntax

-srec

Description

With this option you tell the control program to generate a Motorola S-record object file.

Example

```
cctc -srec test.c
```

The control program generates the Motorola S-record object file `test.sre` from `test.c`.

Related information



Linker option **-Fformat** (Output format)

Section 6.2, *Motorola S-Record Format*, in Chapter *Object File Formats*.

-tmp

Command line syntax

-tmp

Description

Default, the control program removes intermediate files like the `.src` file (result of the compiler phase) and the `.out` file (result of the linking phase).

With this option you tell the control program to keep temporary files it generates while creating the absolute object file.

Example

```
cctc -tmp test.c
```

The control program keeps all intermediate files it generates while creating the absolute object file `test.elf`.

Related information



–

-Wtool

Command line syntax

| | |
|---------------------------|--|
| -Wp <i>option</i> | Pass option directly to the C++ prelinker |
| -Wc <i>option</i> | Pass option directly to the C++ compiler |
| -W <i>option</i> | Pass option directly to the C compiler |
| -Wa <i>option</i> | Pass option directly to the assembler |
| -Wpc <i>option</i> | Pass option directly to the second assembler |
| -Wlk <i>option</i> | Pass option directly to the linker |
| -Wl <i>option</i> | Pass option directly to the locating phase of 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

```
cctc -Wlk-r test.c
```

The control program does not use the option **-r** but calls the linker with the option **-r** (`ltc -r`).

Related information



-V

Command line syntax

-V

Description

Display version information. The control program ignores all other options or input files.

Example

```
cctc -V
```

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 years Altium BV                     Serial# 00000000
```

Related information



-v/-v0

Command line syntax

-v
-v0

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. With the option **-v0** the control program only prints the steps it would do without actually performing these steps.

Example

```
ctc -v test.c
```

The control program processes the file `test.c` and displays the steps it performs:

```
+ ctc/bin/ctc -o /tmp/cc12440b.src test.c
+ ctc/bin/astc /tmp/cc12440b.src -o test.o
+ ctc/bin/ltc -r test.o -lc -lfp -lrt -L/ctc/lib/tcl -o/tmp/cc12440c.out
+ ctc/bin/ltc -FELF -M -d/ctc/etc/tc.lsl /tmp/cc12440c.out -otest.elf
```

Related information



–

-WC++

Command line syntax

-WC++

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 -wc++ 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



—

4.5 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

macro=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 **-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 *option file* which you then must specify to the compiler with the option **-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.out
```

You can now use a macro definition to set the DEMO flag:

```
mktc real.elf DEMO=1
```

In both cases the absolute object file `real.elf` is created but depending on the DEMO flag it is linked with `demo.o` or with `real.o`.

Related information



- Make utility option **-e** (Environment variables override macro definitions)
- Make utility option **-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:

```
mktc -?
```

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



-

-c

Command line syntax

-c

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 **-c**, the make utility runs as a child process of the current make.

The option **-c** overrules the option **-err**.

Example

The following command runs the make utility as a child process:

```
mktc -c
```

Related information



Make utility option **-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 rebuild. 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



–



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



-G

Command line syntax

-G *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 `\currdir\myfiles`. When your current directory is `\currdir`, you can call the make utility as follows:

```
mktc -G myfiles
```

Related information



–



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

```
mktool -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

```
mktemp -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 Guide*.

-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

```
mktool -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**)

-m

Command line syntax

-m *file*

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 **-m** 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:

```
-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
```

Related information



-

-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

```
mkrc -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 Guide*.

-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

```
make -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

-S

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)

-s

Command line syntax

-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

```
mktc -s
```

The make utility rebuilds your files but does not print the commands it executes during the make process.

Related information



Make utility option **-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

```
mktdc -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



-W

Command line syntax

-w

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.

Example

```
mktc -w
```

The make utility sends messages to standard out instead of standard error.

Related information



—

-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



4.6 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.

| Description | Option | Suboption |
|---|--------|----------------|
| Display an overview of all options | -? | |
| Display version information | -V | |
| Print object module to standard output | -p | |
| Main functions | | |
| Delete object module from library | -d | -v |
| Move object module to another position | -m | -a -b -v |
| Replace or add an object module | -r | -a -b -c -u -v |
| Print a table of contents of the library | -t | -s0 -s1 |
| Extract an object module from the library | -x | -v |

Table 4-1: Overview of archiver options and suboptions

—?

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

-d [-v]

Description

Delete the specified object modules from a library. With the suboption **-v** the archiver shows which files are removed.

-v Verbose: the archiver shows which files are removed.

Example

```
artc -d lib.a obj1.o obj2.o
```

The archiver deletes `obj1.o` and `obj2.o` from the library `lib.a`.

```
artc -d -v lib.a obj1.o obj2.o
```

The archiver deletes `obj1.o` and `obj2.o` from the library `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

-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 <i>posname</i> | Add the specified object module(s) after the existing module <i>posname</i> . |
| -b <i>posname</i> | Add the specified object module(s) before the existing module <i>posname</i> . |
| -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`:

```
arvc -r lib.a obj2.o
```

To insert `obj3.o` just before `obj2.o`:

```
arvc -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:

```
arvc -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**:

```
arvc -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



-X

Command line syntax

-x [-o] [-v]

Description

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`:

```
arxc -x lib.a obj1.o
```

If you do not specify an object module, all object modules are extracted:

```
arxc -x lib.a
```

Related information



-W

Command line syntax

-wlevel

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 **-w** option, the default warning level is 8.

Example

To suppress warnings above level 5:

```
arvc -x -w5 lib.a obj1.o
```

Related information



—



TOOL OPTIONS

CHAPTER

5

LIST FILE FORMATS



5

CHAPTER

5.1 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                                     Page 1
```

```
ADDR CODE          CYCLES  LINE SOURCE LINE
```

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 850F0008      26      ld.a   a15,world
0006 F4AF          27      stl6.a  [a10],a15
0008 91000040      28      movh.a  a4,#@his(_2_ini)
000C D9440000      29      lea     a4,[a4]@los(_2_ini)
0010 1D000000      30      jg      printf
.
.
0000              44 buf:    .space  4
| RESERVED
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. 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 6.7, *Generating a List File*, in Chapter *Using the Assembler* of the *User's Guide* for more information on how to generate a list file and specify the amount of list file information.

5.2 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)
-----

***** File Part *****

* Processed files:
=====

File          | From archive | Symbol causing the extraction
-----
cstart.o      | libc.a       | _START
hello.o       |              |
printf.o      | libc.a       | printf

***** Call Graph Part *****

***** Link Part *****

* Section translation:
=====

[in] File | [in] Section | [in] Size | [out] Offset | [out] Section
-----
hello.o   | .text.hello.main | 0x00000014 | 0x00000000 | .text.hello.main
-----
cstart.o  | .text.libc      | 0x000001ce | 0x00000000 | .text.libc
strcpy.o  | .text.libc      | 0x00000024 | 0x000001d0 |
cinit.o   | .text.libc      | 0x0000004e | 0x000001f4 |
-----
printf.o  | .text.printf.printf | 0x0000002a | 0x00000000 | .text.printf.printf
```


***** Cross Reference Part *****

* Defined symbols:

=====

| Definition file | Definition section | Symbol | Referenced in |
|-----------------|-------------------------|----------|---------------|
| ----- | ----- | ----- | ----- |
| _doflt.o | .text._doflt._doflt | _doflt | _doprint.o |
| _doprint.o | .text._doprint._doprint | _doprint | printf.o |
| cstart.o | .text.libc.reset | _START | hello.o |
| hello.o | .text.hello.main | main | cstart.o |
| hello.o | .zdata.hello.world | world | |

* Undefined symbols:

=====

| Symbol | Referenced in |
|------------------|---------------|
| ----- | ----- |
| __LITERAL_DATA__ | cstart.o |
| __SMALL_DATA__ | cstart.o |
| __lc_cp | cinit.o |

***** Overlay Part *****

***** Locate Part *****

* Section translation:

=====

| Space | Chip | Group | Section | Size | Space addr | Chip addr |
|------------------|--------|--------|--------------|------------|------------|------------|
| ----- | ----- | ----- | ----- | ----- | ----- | ----- |
| TC1920:tc:csa | dsram | | csa | 0x00001000 | 0xd0000000 | 0x00000000 |
| ----- | ----- | ----- | ----- | ----- | ----- | ----- |
| TC1920:tc:linear | ext_c | cstart | .text.libc | 0x00000242 | 0xa0000000 | 0x00000000 |
| | ext_c2 | | [.data.libc] | 0x000000f8 | 0xb0000000 | 0x00000000 |

* Symbol translation:

=====

| Space | Symbol | Address | Space | Address | Symbol |
|------------------|------------|------------|------------------|------------|------------|
| ----- | ----- | ----- | ----- | ----- | ----- |
| TC1920:tc:csa | _lc_ub_csa | 0xd0000000 | TC1920:tc:csa | 0xd0000000 | _lc_ub_csa |
| | _lc_ue_csa | 0xd0001000 | | 0xd0001000 | _lc_ue_csa |
| ----- | ----- | ----- | ----- | ----- | ----- |
| TC1920:tc:linear | _START | 0xa0000242 | TC1920:tc:linear | 0xa000013c | _exit |
| | _exit | 0xa000013c | | 0xa0000242 | _START |
| | printf | 0xa0001f90 | | 0xa0001f90 | printf |

***** Linker Script File Part *****

***** Memory Part *****

The meaning of the different parts is:

File Part

This part of the map file shows all processed files. This also includes object files that are extracted from a library, with the symbol that led to the extraction

Call Graph Part

This part of the map file contains a schematic overview that shows how (library) functions call each other. To obtain call graph information, the assembly file must contain `.CALLS` directives which you must manually add to the assembly source.

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`, `strcpy.o` and `cinit.o` in the example above are all mapped on the output section `.text.libc` on succeeding offsets.

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 (<code>tc*.lsl</code>). The names are constructed of the derivative name followed by a colon ':', the core name, another colon ':' and the space name. For example: <code>TC1920:tc:linear</code> |
| Chip | The names of the memory chips as defined in the linker control file (<code>tc.i</code>) and the <code>memory_layout</code> part of the linker script file (<code>tc*.lsl</code>). |
| Group | Sections can be ordered in groups. These are the names of the groups as defined in the linker script file (<code>tc*.lsl</code>) with the keyword <code>group</code> in the <code>section_layout</code> part. |
| Section | The name of the section. Names within square brackets <code>[]</code> will be copied during initialization from ROM to the corresponding section name in RAM. |
| Size | The size of the section. |
| 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 address and sorted on symbol name.

| | |
|---------|--|
| Space | The names of the address spaces as defined in the linker script file (<code>tc*.lsl</code>). The names are constructed of the derivative name followed by a colon ':', the core name, another colon ':' and the space name. For example: <code>TC1920:tc:linear</code> |
| Symbol | The name of the symbol. |
| Address | The absolute address of the symbol in the address space. |

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**.

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).

Related information

Section 7.8, *Generating a Map File*, in Chapter *Using the Linker* of the *User's Guide*.

Linker option **-M** (Generate map file)

LIST FILE FORMATS

CHAPTER

6

OBJECT FILE FORMATS



6

CHAPTER

6.1 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 *Tools Interface Standards* on Intel's website for developers:
<http://developer.intel.com/vtune/tis.htm>

6.2 MOTOROLA S-RECORD FORMAT

With the **-cSREC** option the linker produces output in Motorola S-record format with three types of S-records: S0, S2 and S8. With the **-cSREC:2** or **-cSREC4** 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
```

```
l t c
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 **-cSREC:2** option of the linker, 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:

```

S3070000FFFE6E6825
| | | |
| | | | _ 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 **-cSREC:4** option of the linker, 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:

```

S70500006E6824
| | | |
| | | | _checksum
| | | | _ address
| | | | _ length

```

The checksum calculation of S7 records is identical to S0.

S8 - record

With the **-cSREC:3** option of the linker, 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:

```

S804FF0003F9
| | | |
| | | | _checksum
| | | | _ address
| | | | _ length

```

The checksum calculation of S8 records is identical to S0.

S9 - record

With the **-cSREC:2** option of the linker, 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.

6.3 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, linker option **-ciHEX**).

General Record Format

In the output file, the record format is:

| | | | | | |
|---|---------------|---------------|-------------|----------------|-----------------|
| : | <i>length</i> | <i>offset</i> | <i>type</i> | <i>content</i> | <i>checksum</i> |
|---|---------------|---------------|-------------|----------------|-----------------|

Where:

- :

is the record header.
- length*

is the record length which specifies the number of bytes of the *content* field. This value occupies one byte (two hexadecimal digits). The 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:

| Byte Type | Record type |
|-----------|-------------------------------------|
| 00 | Data |
| 01 | End of File |
| 02 | Extended segment address (not used) |
| 03 | Start segment address (not used) |
| 04 | Extended linear address (32-bit) |
| 05 | Start linear address (32-bit) |

content is the information contained in the record. This depends on the record type.

checksum is the record checksum. The 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 | <i>upper_address</i> | <i>checksum</i> |
|---|----|------|----|----------------------|-----------------|

The 32-bit absolute address of a byte in a Data Record is calculated as:

$(address + offset + index) \text{ modulo } 4G$

where:

address is the base address, where the two most significant bytes are the *upper_address* and the two least significant bytes are zero.

offset is the 16-bit offset from the Data Record.

index is the index of the data byte within the Data Record (0 for the first byte).

Example:

```
:0200000400FFFB
| | | | | _ checksum
| | | | | _ upper_address
| | | | | _ type
| | | | | _ offset
| | | | | _ length
```

Data Record

The Data Record specifies the actual program code and data.

| | | | | | |
|---|---------------|---------------|----|-------------|-----------------|
| : | <i>length</i> | <i>offset</i> | 00 | <i>data</i> | <i>checksum</i> |
|---|---------------|---------------|----|-------------|-----------------|

The *length* byte specifies the number of *data* bytes. The 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:

```
:0F00200000232222754E00754F04AF4FAE4E22C3
| | | | | _ data
| | | | | _ type
| | | | | _ offset
| | | | | _ length
| | | | | _ checksum
```

Start Linear Address Record

The Start Linear Address Record contains the 32-bit program execution start address.

Layout:

| | | | | | |
|---|----|------|----|----------------|-----------------|
| : | 04 | 0000 | 05 | <i>address</i> | <i>checksum</i> |
|---|----|------|----|----------------|-----------------|

Example:

```
:0400000500FF0003F5
| | | | |
| | | | | _ checksum
| | | | | _ address
| | | | | _ type
| | | | | _ offset
| | | | | _ length
```

End of File Record

The hexadecimal file always ends with the following end-of-file record:

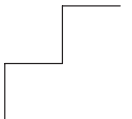
```
:00000001FF
| | | | |
| | | | | _ checksum
| | | | | _ type
| | | | | _ offset
| | | | | _ length
```




OBJECT FORMATS

CHAPTER 7

LINKER SCRIPT LANGUAGE



7 | CHAPTER

7.1 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.

7.2 STRUCTURE OF A LINKER SCRIPT FILE

A script file generally consists of the following parts:

The architecture definition (required)

In essence an *architecture definition* describes how the linker should convert virtual addresses into physical addresses for a given type of core. If the core supports multiple address spaces, then for each space the linker must know how to perform this conversion. In this context a physical address is an offset on a given internal or external bus. Additionally the architecture definition contains information about items such as the (hardware) stack and the interrupt vector table.

Typically an architecture definition is written by Altium and should not be changed by you unless you also modify a 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 7.5, *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 busses specified in the architecture definition into offsets in internal memory. A derivative definition must be present in an LSL file. Microcontrollers and DSPs often have internal memory and I/O sub-systems apart from one or more cores. The design of such a chip is called a *derivative*. Altium provides LSL descriptions of supported derivatives, along with "SFR files", which provide easy access to registers in I/O sub-systems from C and assembly programs. When you build an ASIC or use a derivative that is not (yet) supported by the TASKING tools, you may have to write a derivative definition.



See section 7.6, *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. Typically the processor definition instantiates one derivative only (single-core processor). A processor that contains multiple cores having the same (homogeneous) or different (heterogeneous) architecture can also be described by instantiating multiple derivatives of the same or different types in separate processor definitions.



See section 7.7, *Semantics of the Board Specification* for a detailed description of LSL in the processor definition.

The memory and bus definitions (optional)

Memory and bus definition are used within the context of a derivative definition to specify internal memory and on-chip busses. In the context of a board specification the memory and bus definitions are used to define external (off-chip) memory and busses. Given the above definitions the linker can convert a logical address into an offset into an on-chip or off-chip memory device.



See section 7.7.3, *Defining External Memory and Busses*, 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 busses, memory devices, I/O sub-systems, and cores that are of interest to the linker. Based on the information provided in the board specification the linker can for each core:

- convert a virtual address to a physical addresses (offsets within a memory device)
- locate sections in physical memory
- maintain an overall view of the used and free physical memory within the whole system while locating

The section layout definition (optional)

The optional *section layout definition* enables you to exactly control where input sections are located. Features are provided such as: the ability to place sections at a given load-address or run-time address, to place sections in a given order, and to overlay code and/or data sections.

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, from the board specification the linker can deduce which physical memory is (still) available while locating the section.



See section 7.8, *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:

```
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
}

```

7.3 SYNTAX OF THE LINKER SCRIPT LANGUAGE

The following lexicon is used to describe the syntax of the Linker Script Language:

| | |
|------------------------------|--|
| $A ::= B$ | = A is defined as B |
| $A ::= B\ C$ | = A is defined as B and C ; B is followed by C |
| $A ::= B\ \ C$ | = A is defined as B or C |
| $\langle B \rangle^0 1$ | = zero or one occurrence of B |
| $\langle B \rangle^{\geq 0}$ | = zero or more occurrences of B |
| $\langle B \rangle^{\geq 1}$ | = one or more occurrences of B |
| | |
| <i>IDENTIFIER</i> | = a character sequence starting with 'a'-'z', 'A'-'Z', '_', . or @ |
| <i>STRING</i> | = sequence of characters not starting with \n, \r or \t |
| <i>DQSTRING</i> | = " <i>STRING</i> " (double quoted string) |
| <i>OCT_NUM</i> | = octal number, starting with a zero (06, 045) |
| <i>DEC_NUM</i> | = decimal number, not starting with a zero (14, 1024) |
| <i>HEX_NUM</i> | = 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.

7.3.1 IDENTIFIERS

```
arch_name      ::= IDENTIFIER
bus_name       ::= IDENTIFIER
core_name      ::= IDENTIFIER
derivative_name ::= IDENTIFIER
file_name      ::= DQSTRING
group_name     ::= IDENTIFIER
mem_name       ::= IDENTIFIER
proc_name      ::= IDENTIFIER
section_name   ::= DQSTRING
space_name     ::= IDENTIFIER
stack_name     ::= section_name
symbol_name    ::= DQSTRING
```

7.3.2 EXPRESSIONS

The expressions and operators in this section work the same as in ANSI C.

```
number          ::= OCT_NUM
                  | DEC_NUM
                  | HEX_NUM

assignment      ::= symbol_name assign_op expr ;

assign_op       ::= =
                  | :=

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

```

7.3.3 BUILT-IN FUNCTIONS

```

function_call ::= absolute ( expr )
                  | addressof ( addr_id )
                  | max ( expr , expr )
                  | min ( expr , expr )
                  | sizeof ( size_id )

```

```

addr_id ::= sect : section_name
            | mem : mem_name
            | group : group_name

```

```

size_id ::= sect : section_name
            | group : group_name

```

- Every space, bus, memory, section or group your refer to, must be defined in the LSL file.
- The `addressof()` and `sizeof()` functions can only be used in the right hand side of an assignment.

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()

```
int absolute( expr )
```

Converts the value of *expr* to a positive integer.

```
absolute( "labelA"-"labelB" )
```

addressof()

```
int addressof( addr_id )
```

Returns the address of *addr_id*, which is a named section or memory. To get the offset of the section with the name *asect*:

```
addressof( sect: "asect" )
```



This function only works in assignments.

max()

```
int max( expr, expr )
```

Returns the value of the expression that has the largest value. To get the highest value of two symbols:

```
max( "sym1" , "sym2" )
```

min()

```
int min( expr, expr )
```

Returns the value of the expression that has the smallest value. To get the lowest value of two symbols:

```
min( "sym1" , "sym2" )
```

sizeof()

```
int sizeof( size_id )
```

Returns the size of a section or group in an object file. To get the size of the section "asection":

```
sizeof( sect: "asection" )
```



This function only works in assignments.

7.3.4 LSL DEFINITIONS IN THE LINKER SCRIPT FILE

```
description      ::= <definition>>=1
definition      ::= architecture_definition
                    | derivative_definition
                    | board_spec
                    | section_definition
```

- At least one *architecture_definition* must be present in the LSL file.

7.3.5 MEMORY AND BUS DEFINITIONS

```
mem_def          ::= memory mem_name { <mem_descr ;>=0 }
```

- A *mem_def* defines a *memory* with the *mem_name* as a unique name.

```
mem_descr       ::= type = mem_type
                    | 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
                   | nvram         // attrs = rwx

```

```

bus_def           ::= bus bus_name { <bus_descr ;>*=0 }

```

- 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:**).

```

mapping           ::= map ( map_descr <, map_descr>*=0 )

```

```

map_descr         ::= dest = destination
                   | dest_dbits = range
                   | dest_offset = expr
                   | size = expr
                   | src_dbits = range
                   | src_offset = 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.

```

destination       ::= space : space_name
                   | bus : <proc_name |
                               core_name :>0|1 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

range ::= *number* .. *number*

7.3.6 ARCHITECTURE DEFINITION

architecture_definition

```
 ::= architecture arch_name
    <extends arch_name>0|1
    { arch_spec=0 }
```

- 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.

```
arch_spec ::= bus_def
              | space_def
              | endianness_def
```

```
space_def ::= space space_name { <space_descr; >=0 }
```

- 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 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

```

- 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
```

- 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.

```
start_addr ::= start_address ( start_addr_descr
                              <, start_addr_descr>=>0 )
```

```
start_addr_descr ::= run_addr = expr
                  | section = section_name
```

- A *section_name* refers to the section that contains the startup code.

```
endianness_def ::= endianness { <endianness_type;>=>1 }
```

```
endianness_type ::= big
                  | little
```

7.3.7 DERIVATIVE DEFINITION

```
derivative_definition
    ::= derivative derivative_name
       <extends derivative_name>0|1
       { <derivative_spec>=>0 }
```

- A *derivative_definition* defines a derivative with the given *derivative_name* as a unique name.
- At least one *core_def* and at least one *bus_def* have to be present in a *derivative_definition*.

```
derivative_spec ::= core_def
                  | bus_def
                  | mem_def
```

```
core_def ::= core core_name { <core_descr ;>=>0 }
```

- A *core_def* defines a *core* with the given *core_name* as a unique name.

```

core_descr      ::= architecture = arch_name
                  | endianness = ( endianness_type
                                <, endianness_type>>=0 )

```

- An *arch_name* refers to a defined core architecture.
- Exactly one **architecture** statement must be present in a *core_def*.

7.3.8 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

```

- 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.

7.3.9 SECTION PLACEMENT DEFINITION

```

section_definition ::= section_layout <space_ref>^0|1
                      <( space_props )>^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.

```
space_props ::= space_property <, space_property>*=0
```

```
space_property ::= locate_direction
```

```
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  
    | if_statement  
    | select_section_statement  
    | special_section_statement
```

```
aggregate_section_statement  
    ::= { <section_statement>*=0 }  
    | group_descr
```

```
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.

```
if_statement ::= if ( expr ) section_statement
                  <else section_statement>0|1
```

```
special_section_statement
    ::= heap stack_name <size_spec>0|1
       | stack stack_name <size_spec>0|1
       | copytable
       | reserved <section_name>0|1
                  <size_spec>0|1
```

```
size_spec ::= ( size = expr )
```

```
group_descr ::= group <group_name>0|1
                  <( group_specs )>0|1
                  section_statement
```

```
group_specs ::= group_spec <, group_spec >=0
```

```
group_spec ::= group_alignment
               | attributes
               | group_load_address
               | 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_assignment
```

```
group_page ::= page <= expr>0|1
```

```
group_run_address ::= run_addr load_or_run_addr_assignment
```

```
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_assignment
                    ::= <= load_or_run_addr>0|1

```

```

load_or_run_addr   ::= expr
                   | memory_reference
                   < | memory_reference >>=0

```

```

memory_reference   ::= mem : <proc_name :>0|1
                    <core_name :>0|1 mem_name

```

- A *proc_name* refers to a defined processor.
- A *core_name* refers to a defined core.
- A *mem_name* refers to a defined memory.

7.4 EXPRESSION EVALUATION

Only *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.

7.5 SEMANTICS OF THE ARCHITECTURE DEFINITION

Keywords in the architecture definition

```
architecture
  extends
bus
  mau
  width
  map
space
  id
  mau
  align
  page_size
  stack
    min_size
    grows          low_to_high  high_to_low
    align
  heap
    min_size
    grows          low_to_high  high_to_low
    align
  copytable
    align
    copy_unit
    dest
  start_address
    run_addr
    section
  map

  map
    dest          bus  space
    dest_dbits
    dest_offset
    size
    src_dbits
    src_offset
```

7.5.1 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:

```
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:

```
architecture name_child_arch extends name_parent_arch
{
    definitions
}
```

7.5.2 DEFINING INTERNAL BUSSES

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* busses. Some internal busses are used to communicate with the components outside the core or processor. Such busses 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 7.5.4, *Mappings*.

```
bus bus_name
{
    mau = 8;
    width = 8;
    map ( map_description );
}
```

7.5.3 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 7.8.2, *Creating and Locating Groups of Sections*.
- 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 7.8.3, *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.

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 7.8.3, *Creating or Modifying Special Sections*.



See section 7.8, *Semantics of the Section Layout Definition* for information on creating and placing stack sections.

- 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.

- 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 start section which must be located here.

The **run_addr** argument specifies the start address (reset vector). The **section** argument specifies the name of the start section that should be located at the specified start address. The **section** argument is required.

- The **map** keyword specifies how this address space maps onto an internal bus or onto another address space. Mappings are described in section 7.5.4, *Mappings*.

```
space space_name
{
    id = Y1;
    mau = 8;
    align = 8;
    page_size = 1;
    stack name (min_size = 1k, grows = low_to_high);
    start_address ( run_addr = 0x0000,
                  section = "start_section_name" )
    map ( map_description );
}
```

7.5.4 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 bus

All spaces that are not mapped to another space must map to a bus in the architecture:

```
space large
{
    id = Y1;
    mau = 4;
    map (src_offset = 0, dest_offset = 0,
        dest = bus:bus_name, size = 16M );
}
```

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.

```
space small
{
    id = Y2;
    mau = 4;
    map (src_offset = 0, dest_offset = 0,
        dest = space : large, size = 64k);
}
```

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.

7.6 SEMANTICS OF THE DERIVATIVE DEFINITION

Keywords in the derivative definition

```

derivative
    extends
core
    architecture
bus
    mau
    width
    map
memory
    type                rom  ram  nvram
    mau
    size
    speed
    map

    map
        dest            bus  space
        dest_dbits
        dest_offset
        size
        src_dbits
        src_offset

```

7.6.1 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:

```

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
}
```

7.6.2 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;
}
```

7.6.3 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 7.7.3, *Defining External Memory and Busses*).

- The **type** field specifies a memory type:
 - **rom**: read only memory
 - **ram**: random access memory
 - **nvr****am**: non volatile ram
- 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 address space maps onto an (internal) bus. Mappings are described in section 7.5.4, *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. Busses are described in section 7.5.2, *Defining Internal Busses*.

7.7 SEMANTICS OF THE BOARD SPECIFICATION

Keywords in the board specification

```

processor
    derivative
bus
    mau
    width
    map
memory
    type                rom  ram  nvram
    mau
    size
    speed
    map

    map
        dest            bus  space
        dest_dbits
        dest_offset
        size
        src_dbits
        src_offset

```

7.7.1 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:

```
processor proc_name
{
    processor definition
}
```

7.7.2 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:

```
processor myproc_1
{
    derivative = myderiv;
}

processor myproc_2
{
    derivative = myderiv;
}
```

7.7.3 DEFINING EXTERNAL MEMORY AND BUSES

It is common to define external memory (off-chip) and external busses 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.

```
memory mem_name
{
    type = rom;
    mau = 8;
    size = 64k;
    speed = 2;
    map ( map_description );
}
```



For a description of the keywords, see section 7.6.3, *Defining Internal Memory and Busses*.

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* busses. These are busses that are present on the target board.

```
bus bus_name
{
    mau = 8;
    width = 8;
    map ( map_description );
}
```



For a description of the keywords, see section 7.5.2, *Defining Internal Busses*.

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.

7.8 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
    ordered
    clustered
    contiguous
    overlay
    allow_cross_references
    load_addr
        mem
    run_addr
        mem
    page
select
heap
    size
stack
    size
reserved
    size
copytable

if
else

```

7.8.1 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`:

```
section_layout my_chip::my_space ( space_props )
{
    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).

```
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.

7.8.2 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.

```
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 7.8.3, *Creating or Mofifying 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:

| | |
|---------|---|
| "*" | matches with all section names |
| "?" | matches with a single character in the section name |
| "\" | takes the next character literally |
| "[abc]" | matches with a single 'a', 'b' or 'c' character |
| "[a-z]" | matches with any single character in the range 'a' to 'z' |

```
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 (BSS)
- **s** scratch sections (not cleared and not initialized)

To select all read-only sections:

```
group ( ... )
{
    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

```
group group_name ( group_specifications )
{
    section_statements
}
```

With the *group_specifications* you actually define how the locator 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) assign a load-time address or run-time address to the 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, thus without '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 leaves no 'gaps' between them.

- 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.
- 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. Assign a load-time address or run-time address to the group.
By default, the position and properties of a group in an LSL file specify the run-time addresses of its sections. The linker uses the addresses defined by the run-time addresses when it patches symbol references in the object code. It may be possible that you want to assign a different load-time and run-time address to 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. The program is responsible for copying overlay sections at appropriate moment from its load-time location to its run-time location.

- The **run_addr** keyword defines the run-time start 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. With an *expression* you can specify that the group should be located at the absolute address specified by the expression:

```
group (run_addr = 0xa00f0000)
```

With the **mem** keyword you can specify that the group should be located within a physical memory device, instead of an address value:

```
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 **run_addr** keyword itself (without an assignment) specifies that the group's position in the section layout defines its run-time address, which is the default.

```
group (run_addr)
```

- The **load_addr** keyword defines the load-time start address. With an *expression* you can specify that the first element of the group should be loaded at the absolute address specified by the expression:

```
group (load_addr = 0x00000000)
```

With the **mem** keyword you can specify that the elements of the group should be loaded within a physical memory (or use `'|'` to specify an address range):

```
group (load_addr = mem:my_ram, ...)
```

The **load_addr** keyword itself (without an assignment) specifies that the group's location defines its load-time address.

```
group (load_addr)
```

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 allowed 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. Default the linker chooses a page number by itself.

The **page** keyword refers to pages in the address space as defined in the architecture definition. See also the **page** keyword in section 7.5.3, *Defining Address Spaces*.

```
group ( page, ... )  
group ( page = 3, ... )
```

7.8.3 CREATING OR MODIFYING SPECIAL SECTIONS

Instead of selecting sections, you can also create a reserved section or modify special sections like a stack or a heap, a reserved section. Because you cannot define these sections in the input files, you must use the linker to create them.

- 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 given then the size given in the architecture definition is used.


```
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 7.5.3, *Defining Address Spaces*.

- 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 given then the size given in the architecture definition is used.

```
group ( ... )
{
    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.

- The **reserved** keyword tells the linker to create a section of a given size. The linker will not locate any other sections in the memory occupied by a reserved section. Optionally you can assign a name to a reserved section. With the keyword **size** you can specify a size for a given reserved section.

```
group ( ... )
{
    reserved "myreserved" ( size = 2k );
}
```

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.

- 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.

7.8.4 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).

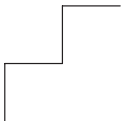
```
group ( ... )
{
    if ( size_of ( sect:.mysection ) < 2k )
        select ".mysection";
    else
        select ".othersection";
}
```

LINKER SCRIPT LANGUAGE

CHAPTER

8

CPU FUNCTIONAL PROBLEMS



8

CHAPTER

8.1 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.

The syntax used by Infineon to identify a CPU functional problem is:

TC<architecture_nr><version>_<module_name><problem_nr>

For example: **TC113_CPU5** (TC1, version 1.3, module "CPU", problem #5)

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.

The table below shows an overview of all CPU functional problems.

| TC Version | Functional Problem |
|------------|--------------------|
| 112 | COR1 |
| 112 | COR3 |
| 112 | COR4 |
| 112 | COR6 |
| 112 | COR7 |
| 112 | COR10 |
| 112 | COR13 |
| 112 | COR14 |
| 112 | COR15 |
| 112 | COR16 |
| 112 | COR17 |
| 113 | CPU5 |
| 113 | CPU9 |
| 113 | CPU11 |
| 113 | CPU13 |
| 113 | CPU14 |
| 113 | CPU15 |
| 113 | CPU16 |
| 113 | DMU1 |
| 113 | LFI2 |
| 113 | LFI3 |
| 113 | PMU1 |
| 113 | PMU3 |

Table A-2: Overview of supported TriCore CPU functional problems

8.2 CPU FUNCTIONAL PROBLEM BYPASSES TC1 V1.2

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:

```
W253: suspicious instruction concerning CPU functional
defect TC112_COR1
```

You can suppress this warning with the option **-w253**.

TC112_COR3

Locator option:

-D__TC112_COR3__

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.

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:

```
w254: suspicious instruction concerning CPU functional
defect TC112_COR4
```

You can suppress this warning with the option **-w254**.

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 if it was previously beginning with a CALL instruction and in a "multiple DMU master" situation.

Assembler check:

The assembler gives a warning when the register A11 is incorrectly updated if there is a status "DMU not ready" on the second micro op of a 2 cycle call variant. This is due to an error in the fetch unit block.

```
W255: suspicious instruction concerning CPU functional
defect TC112_COR7
```

You can suppress this warning with the option **-w255**.

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:

```
W256: suspicious instruction concerning CPU functional
defect TC112_COR10
```

You can suppress this warning with the option **-w256**.

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 a NOP 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:

```
W257: suspicious instruction concerning CPU functional
defect TC112_COR13
```

You can suppress this warning with the option **-w257**.

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:

```
;;
;; Save interrupt state and disable interrupts
;;
mfcr d0,#0xfe2c          ; save ICR in d0
disable                  ; disable interrupts
```

divide instructions:

```
;;
;; 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 Guide*, 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*.lsl` 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 a NOP instruction after a DSYNC instruction. The C compiler only generates a DSYNC instruction when bypass TC113_CPU17 is enabled.

Assembler check:

The assembler gives a warning if a DSYNC is not followed by a NOP instruction:

```
W258: suspicious instruction concerning CPU functional
defect TC112_COR17
```

You can suppress this warning with the option **-w258**.

8.3 CPU FUNCTIONAL PROBLEM BYPASSES TC1 V1.3

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-bug=cpu9

Assembler control:

\$TC113_CPU9 {on | off}

Assembler macro:

The assembler macro `__TC113_CPU9__` is defined if you specify the option **--silicon-bug=cpu9**.

Protected libraries to link:

`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:

```
w259: suspicious instruction concerning CPU functional
defect TC113_CPU9
```

You can suppress this warning with the option **-w259**.

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 between a LDA, LDDA, LD16A and the 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 an LDA, LDDA, or LD16A instruction is directly followed by a JI 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:

```
W260: suspicious instruction concerning CPU functional
defect TC113_CPU11
```

You can suppress this warning with the option **-w260**.

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:

```
W261: suspicious instruction concerning CPU functional
defect TC113_CPU14
```

You can suppress this warning with the option **-w261**.

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:

```
w262: suspicious instruction concerning CPU functional
defect TC113_CPU15
```

You can suppress this warning with the option **-w262**.

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 between an LDA, LDDA, LD16A and the JI or CALLI instruction with the same address register as parameter. 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 an LDA, LDDA or LD16A instruction is directly followed by a JI or CALLI instruction with the same address register as parameter. The assembler also gives a warning when there is no or just one instruction (not a NOP instruction) between label and RET or RET16:

```
W263: suspicious instruction concerning CPU functional
defect TC113_CPU16
```

You can suppress this warning with the option **-w263**.

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 for SWAP, LDMST and ST.T instructions:

```
w264: suspicious instruction concerning CPU functional
defect TC113_DMU1
```

You can suppress this warning with the option **-w264**.

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:

```
W265: suspicious instruction concerning CPU functional
defect TC113_LFI2
```

You can suppress this warning with the option **-w265**.

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:

```
W266: suspicious instruction concerning CPU functional
defect TC113_LFI3
```

You can suppress this warning with the option **-w266**.

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.



CPU FUNCTIONAL PROBLEMS

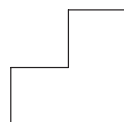
CHAPTER

6

MISRA C RULES



TASKING



9

CHAPTER

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 106460-1
7. Trigraphs shall not be used
8. Multibyte characters and wide string literals shall not be used
9. Comments shall not be nested
10. Sections of code should not be "commented out"

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 name spaces
13. Specific-length typedefs should be used instead of the basic types
14. Use 'unsigned char' or 'signed char' instead of plain 'char'
- * 15. Floating point implementations should comply with a standard
16. The bit representation of floating point numbers shall not be used

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/atol/atoi 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 5.8, *C Code Checking: MISRA C*, in Chapter *Using the Compiler* of the *User's Guide*.

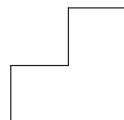


INDEX

INDEX



TASKING



INDEX

Symbols

#define, 4-12, 4-100
 #include, 4-25
 #undef, 4-49
 _close, 2-9
 _lseek, 2-9
 _open, 2-9
 _read, 2-9
 _tolower, 2-10
 _toupper, 2-10
 _unlink, 2-10
 _write, 2-10

A

abort, 2-11
 abs, 2-11, 3-6
 access, 2-11
 accum, 3-21
 acos, 2-11
 acs, 3-6
 align, 3-22
 architecture definition, 7-3, 7-19
 archiver options
 -?, 4-192
 -d, 4-193
 -p, 4-195
 -m, 4-194
 -r, 4-196
 -t, 4-198
 -V, 4-199
 -w, 4-201
 -x, 4-200
 add module, 4-196
 create library, 4-196
 delete module, 4-193
 extract module, 4-200
 move module, 4-194
 print list of objects, 4-198
 print list of symbols, 4-198

print module, 4-195
 replace module, 4-196
 arg, 3-7
 ascii, 3-23
 asciiz, 3-23
 asctime, 2-12
 asin, 2-12
 asn, 3-7
 aspcp, 3-7
 assembler controls
 case, 3-68
 debug, 3-69
 detailed description, 3-67
 fpu, 3-70
 bw_only, 3-71
 ident, 3-72
 list, 3-75
 list on/off, 3-73
 listing controls (overview), 3-66
 miscellaneous (overview), 3-66
 mmu, 3-77, 3-85
 noprnt, 3-82
 overview, 3-66
 page, 3-79
 prctl, 3-81
 prnt, 3-82
 stitle, 3-83
 tc, 3-78, 3-84
 title, 3-86
 warning off, 3-87
 assembler directives
 accum, 3-21
 align, 3-22
 ascii, 3-23
 asciiz, 3-23
 assembly control (overview), 3-18
 byte, 3-24
 calls, 3-26
 comment, 3-27
 conditional assembly (overview),
 3-19
 data definition (overview), 3-19

- debug information (overview)*, 3-20
- define*, 3-28
- detailed description*, 3-20
- double*, 3-39
- dup/endm*, 3-29
- dupa/endm*, 3-30
- dupc/endm*, 3-31
- dupf/endm*, 3-32
- end*, 3-34
- equ*, 3-35
- exitm*, 3-36
- extern*, 3-37
- fail*, 3-38
- float*, 3-39
- fract*, 3-40
- global*, 3-41
- half*, 3-64
- if*, 3-42
- include*, 3-44
- local*, 3-45
- macro/endm*, 3-46
- macros (overview)*, 3-19
- message*, 3-48
- name*, 3-49
- org*, 3-50
- overview*, 3-18
- pmacro*, 3-52
- sdecl*, 3-53
- sect*, 3-56
- set*, 3-57
- sfract*, 3-40
- size*, 3-58
- space*, 3-59
- storage allocation (overview)*, 3-19
- symb*, 3-60
- symbol definitions (overview)*, 3-19
- type*, 3-61
- undef*, 3-62
- warning*, 3-63
- word*, 3-64
- assembler list file, 4-78
- assembler options
 - ?, 4-58
 - case-sensitive, 4-61
 - cpu, 4-59
 - debug-info, 4-70
 - define, 4-62
 - diag, 4-64
 - error-file, 4-66
 - fpu-present, 4-69
 - help, 4-58
 - include-directory, 4-73
 - include-file, 4-72
 - is-tricore2, 4-76
 - keep-output-files, 4-77
 - list-file, 4-80
 - list-format, 4-78
 - mmu-present, 4-82
 - no-tasking-sfr, 4-83
 - no-warnings, 4-91
 - optimize, 4-84
 - option-file, 4-67
 - output, 4-85
 - preprocessor-type, 4-81
 - section-info, 4-88
 - silicon-bug, 4-86
 - symbol-scope, 4-75
 - version, 4-90
 - warnings-as-errors, 4-93
 - C, 4-59
 - c, 4-61
 - D, 4-62
 - f, 4-67
 - g, 4-70
 - H, 4-72
 - I, 4-73
 - i, 4-75
 - k, 4-77
 - L, 4-78
 - l, 4-80
 - m, 4-81
 - O, 4-84
 - o, 4-85
 - t, 4-88
 - V, 4-90
 - w, 4-91

assembly functions

- abs*, 3-6
- acs*, 3-6
- address calculation (overview)*, 3-6
- arg*, 3-7
- asn*, 3-7
- aspcp*, 3-7
- assembler mode (overview)*, 3-6
- astc*, 3-7
- at2*, 3-7
- atn*, 3-8
- fract*, 3-8, 3-10
- cel*, 3-8
- cnt*, 3-8
- cob*, 3-8
- conversions (overview)*, 3-5
- cos*, 3-9
- cpu*, 3-9
- cvf*, 3-9
- cvi*, 3-9
- def*, 3-9
- exp*, 3-10
- fld*, 3-10
- flr*, 3-10
- hi*, 3-11
- his*, 3-11
- int*, 3-11
- l10*, 3-11
- len*, 3-11
- lng*, 3-12
- lo*, 3-12
- log*, 3-12
- los*, 3-12
- lsb*, 3-12
- lst*, 3-13
- lun*, 3-13
- mac*, 3-13
- macros (overview)*, 3-5
- mathematical (overview)*, 3-4
- max*, 3-13
- min*, 3-13
- msb*, 3-14
- mxp*, 3-14
- pos*, 3-14
- pow*, 3-14
- rnd*, 3-14
- rvb*, 3-15
- scp*, 3-15
- sfracl*, 3-15
- sgn*, 3-15
- sin*, 3-15
- snb*, 3-16
- sqi*, 3-16
- strings (overview)*, 3-5
- sub*, 3-16
- syntax*, 3-3
- tan*, 3-16
- tnb*, 3-16
- unf*, 3-17
- xpn*, 3-17
- assert, 2-12
- assert.h, assert, 2-12
- astc, 3-7
- at2, 3-7
- atan, 2-12
- atan2, 2-13
- atexit, 2-13
- atn, 3-8
- atoac, 2-13
- atof, 2-13
- atofr, 2-14
- atoi, 2-14
- atol, 2-14
- ltolac, 2-15
- atolfr, 2-15

B

- bit handling, 1-21
- board specification, 7-5, 7-29
- bsearch, 2-15
- bus definition, 7-4

byte, 3-24

C

C library, reentrancy, 2-63

C++ muncher, 4-142

calloc, 2-16

calls, 3-26

case, 3-68

case sensitivity, 4-97

cat, 3-8

ceil, 2-16

cel, 3-8

char type, treat as unsigned, 4-50

chdir, 2-16

clearerr, 2-16

clock, 2-17

close, 2-17

cnt, 3-8

coh, 3-8

command file, 4-20, 4-67, 4-109,

4-145, 4-177

comment, 3-27

common subexpression elimination,

4-11

compiler options

-?, 4-4

--align, 4-7

--cpu, 4-8

--cse-all-addresses, 4-11

--debug-info, 4-23

--default-a0-size, 4-55

--default-near-size, 4-34

--define, 4-12

--diag, 4-14

--error-file, 4-18

--fpu-present, 4-22

--help, 4-4

--include-directory, 4-25

--include-file, 4-24

--indirect, 4-27

--inline-max-incr, 4-28

--inline-max-size, 4-28

--integer-enumeration, 4-30

--is-tricore2, 4-31

--iso, 4-10

--keep-output-files, 4-32

--language, 4-5

--misrac, 4-33

--no-double, 4-19

--no-tasking-sfr, 4-37

--no-warnings, 4-52

--option-file, 4-20

--output, 4-41

--preprocess, 4-16

--rename-sections, 4-42

--silicon-bug, 4-44, 4-46

--source, 4-43

--stdout, 4-36

--tradeoff, 4-48

--uchar, 4-50

--undefine, 4-49

--version, 4-51

--warnings-as-errors, 4-54

-A, 4-5

-C, 4-8

-c, 4-10

-D, 4-12

-E, 4-16

-F, 4-19

-f, 4-20

-g, 4-23

-H, 4-24

-I, 4-25

-k, 4-32

-N, 4-34

-n, 4-36

--optimize, 4-38

-O, 4-38

-o, 4-41

-R, 4-42

-s, 4-43

-t, 4-48

- U, 4-49
- u, 4-50
- V, 4-51
- w, 4-52
- Z, 4-55
- conditional make rules, 4-163
- control program options
 - , 4-138
 - silicon-bug, 4-154
 - C, 4-139
 - c, 4-141
 - c++, 4-140
 - cc, 4-141
 - cl, 4-141
 - cm, 4-142
 - cp, 4-143
 - cs, 4-141
 - elf, 4-144
 - f, 4-145
 - fptrap, 4-147
 - ieee, 4-148
 - ibex, 4-149
 - noc++, 4-150
 - nolib, 4-151
 - nomap, 4-152
 - o, 4-153
 - srec, 4-156
 - tmp, 4-157
 - V, 4-159
 - v, 4-160
 - v0, 4-160
 - W, 4-158
 - Wa, 4-158
 - Wc, 4-158
 - wc++, 4-161
 - Wcp, 4-158
 - Wlc, 4-158
 - Wlk, 4-158
 - Wpl, 4-158
- controls
 - See also assembler directives*
 - detailed description, 3-67*
 - copy table, 4-123, 7-41

- copysign, 2-17
- core type, 4-59
- cos, 2-17, 3-9
- cosh, 2-18
- cpu, 3-9
- CPU type, 4-8, 4-59, 4-139
- CSE, 4-11
- ctime, 2-18
- ctype.h
 - _tolower, 2-10
 - _toupper, 2-10
 - isalnum, 2-28
 - isalpha, 2-28
 - isascii, 2-28
 - iscntrl, 2-28
 - isdigit, 2-28
 - isgraph, 2-29
 - islower, 2-29
 - isprint, 2-30
 - ispunct, 2-30
 - isspace, 2-30
 - isupper, 2-30
 - isxdigit, 2-31
 - toascii, 2-58
 - tolower, 2-59
 - toupper, 2-59
- cvf, 3-9
- cvi, 3-9
- cycle count, 4-88

D

- data types, 1-4
- debug, 3-69
- debug information, 4-23, 4-70, 4-131
- def, 3-9
- define, 3-28
- derivative definition, 7-4, 7-26
- difftime, 2-18
- directives
 - See also assembler directives*

detailed description, 3-20

div, 2-18

double, 3-39

dup, 3-29

dupa, 3-30

dupc, 3-31

dupf, 3-32

E

ELF/DWARF object format, 6-3

elif, 3-42

else, 3-42

end, 3-34

endif, 3-42

enum, 4-30

equ, 3-35

errno declaration, 2-71

errno.h, 2-71

exit, 2-19

exit macro, 3-36

exitm, 3-36

exp, 2-19, 3-10

extern, 3-37

F

fabs, 2-19

fail, 3-38

fclose, 2-19

fcntl.h, open, 2-37

feof, 2-19

ferror, 2-20

fflush, 2-20

fgetc, 2-20

fgetpos, 2-20

fgets, 2-21

fld, 3-10

float, 3-39

float.h

copysign, 2-17

isfinite, 2-29

isinf, 2-29

isnan, 2-30

scalb, 2-43

floating point, single precision, 4-22,
4-69

floor, 2-21

flr, 3-10

fmod, 2-21

fopen, 2-21

fprintf, 2-22

fputc, 2-22

fputs, 2-23

fract, 3-10, 3-40

fractional arithmetic support, 1-14

fread, 2-23

free, 2-23

freopen, 2-24

frexp, 2-24

fscanf, 2-24

fseek, 2-25

fsetpos, 2-25

ftell, 2-25

functional problems, 8-3

functions, assembly, 3-3

fwrite, 2-26

G

getc, 2-26

getchar, 2-26

getcwd, 2-27

getenv, 2-27

gets, 2-27

global, 3-41

gmtime, 2-27

H

half, 3-64

header files, 2-4

hi, 3-11
 his, 3-11
 hw_only, 3-71

I

ident, 3-72
 if, 3-42
 include, 3-44
 indirect function calling, 4-27
 inline functions, 1-28
 insert assembly instruction, 1-19
 int, 3-11
 Intel hex, record type, 6-8
 interrupt handling, 1-18
 intrinsic functions, 1-12
 bit handling, 1-21
 fractional data type, 1-14
 insert assembly instruction, 1-19
 interrupt handling, 1-18
 min/max of integers, 1-13
 miscellaneous, 1-23
 packed data type, 1-15
 register handling, 1-20
 iob structures, 2-70
 isalnum, 2-28
 isalpha, 2-28
 isascii, 2-28
 iscntrl, 2-28
 isdigit, 2-28
 isfinite, 2-29
 isgraph, 2-29
 isinf, 2-29
 islower, 2-29
 isnan, 2-30
 ISO C standard, 4-10
 isprint, 2-30
 ispunct, 2-30
 isspace, 2-30
 isupper, 2-30
 isxdigit, 2-31

L

l10, 3-11
 labs, 2-31
 language extensions, intrinsic
 functions, 1-12
 ldexp, 2-31
 ldiv, 2-31
 len, 3-11
 linker map file, 4-119
 linker options
 -?, 4-96
 --case-insensitive, 4-97
 --chip-format, 4-98
 --define, 4-100
 --diag, 4-103
 --error-file, 4-107
 --extern, 4-105
 --first-library-first, 4-111
 --format, 4-108
 --help, 4-96
 --ignore-default-library-path,
 4-113
 --incremental, 4-130
 --keep-output-files, 4-112
 --library, 4-115
 --library-directory, 4-113
 --link-only, 4-116
 --lsl-check, 4-117
 --lsl-dump, 4-118
 --map-file, 4-119
 --map-file-format, 4-120
 --misra-c-report, 4-122
 --no-rescan, 4-124
 --no-rom-copy, 4-123
 --no-warnings, 4-134
 --non-romable, 4-126
 --optimize, 4-127
 --option-file, 4-109
 --output-file, 4-129
 --strip-debug, 4-131
 --verbose, 4-133

- version*, 4-132
- warnings-as-errors*, 4-136
- c*, 4-98
- D*, 4-100
- d*, 4-101
- e*, 4-105
- F*, 4-108
- f*, 4-109
- H*, 4-96
- k*, 4-112
- L*, 4-113
- l*, 4-115
- M*, 4-119
- m*, 4-120
- N*, 4-123
- O*, 4-127
- o*, 4-129
- r*, 4-130
- S*, 4-131
- t*, 4-133
- V*, 4-132
- v*, 4-133
- w*, 4-134
- linker script file
 - architecture definition*, 7-3
 - board specification*, 7-5
 - bus definition*, 7-4
 - derivative definition*, 7-4
 - memory definition*, 7-4
 - processor definition*, 7-4
 - section layout definition*, 7-5
- list, 3-75
- list file, 4-80
 - assembler*, 4-78
 - linker*, 4-119
- list on/off, 3-73
- lng, 3-12
- lo, 3-12
- local, 3-45
- locale.h
 - localeconv*, 2-32
 - setlocale*, 2-46
- localeconv, 2-32
- localtime, 2-32
- log, 2-32, 3-12
- log10, 2-32
- longjmp, 2-33
- los, 3-12
- lsb, 3-12
- lseek, 2-33
- LSL expression evaluation, 7-18
- LSL functions
 - absolute()*, 7-9
 - addressof()*, 7-9
 - max()*, 7-9
 - min()*, 7-9
 - sizeof()*, 7-10
- LSL keywords
 - align*, 7-21, 7-35
 - allow_cross_references*, 7-36
 - architecture*, 7-20, 7-27
 - attributes*, 7-34, 7-35
 - bus*, 7-20, 7-23, 7-30
 - contiguous*, 7-36
 - copytable*, 7-22
 - create_section*, 7-36
 - derivative*, 7-26, 7-30
 - dest*, 7-23
 - dest_dbits*, 7-23
 - dest_offset*, 7-23
 - direction*, 7-32, 7-36
 - else*, 7-41
 - extends*, 7-20, 7-26
 - fill*, 7-27, 7-30
 - group*, 7-33, 7-35
 - grows*, 7-21
 - heap*, 7-22, 7-39
 - id*, 7-21
 - if*, 7-41
 - load_addr*, 7-37
 - map*, 7-23
 - mau*, 7-20, 7-21, 7-27, 7-30
 - mem*, 7-37
 - memory*, 7-27, 7-30
 - min_size*, 7-21
 - ordered*, 7-36

overlay, 7-36
page, 7-37
page_size, 7-21
processor, 7-29
reserved, 7-39
run_addr, 7-37
section_layout, 7-32
select, 7-34
size, 7-23, 7-27, 7-30, 7-39
space, 7-21, 7-23
speed, 7-27, 7-30
src_dbits, 7-23
src_offset, 7-23
stack, 7-21, 7-39
start_address, 7-22
table, 7-39
type, 7-27, 7-30
width, 7-20
 lst, 3-13
 lun, 3-13

M

mac, 3-13
 macro, 3-46
 macros, 1-25
 make utility, 4-163
 macros, predefined
 __DATE__, 4-49
 __FILE__, 4-49
 __LINE__, 4-49
 __TIME__, 4-49
 make utility options
 -?, 4-165
 -a, 4-166
 -c, 4-167
 -D, 4-168
 -d, 4-169
 -DD, 4-168
 -dd, 4-169
 -e, 4-170

 -err, 4-171
 -f, 4-172
 -G, 4-173
 -i, 4-174
 -K, 4-175
 -k, 4-176
 -m, 4-177, 4-183
 -n, 4-179
 -p, 4-180
 -q, 4-181
 -r, 4-182
 -s, 4-184
 -t, 4-185
 -time, 4-186
 -V, 4-187
 -W, 4-188
 -w, 4-189
 -x, 4-190
 defining a macro, 4-163
 malloc, 2-33
 map file
 control program option, 4-152
 format, 4-120
 linker, 4-119
 math.h
 acos, 2-11
 asin, 2-12
 atan, 2-12
 atan2, 2-13
 ceil, 2-16
 cos, 2-17
 cosh, 2-18
 exp, 2-19
 fabs, 2-19
 floor, 2-21
 fmod, 2-21
 frexp, 2-24
 ldexp, 2-31
 log, 2-32
 log10, 2-32
 modf, 2-36
 pow, 2-37

sin, 2-47
sinb, 2-47
sqrt, 2-48
tan, 2-57
tanb, 2-57
 max, 3-13
 mblen, 2-34
 mbstowcs, 2-34
 mbtowc, 2-34
 memchr, 2-35
 memcmp, 2-35
 memcpy, 2-35
 memmove, 2-35
 memory definition, 7-4
 memory management instructions,
 4-82
 memset, 2-36
 message, 3-48
 min, 3-13
 min/max of integers, 1-13
 MISRA C, 4-33
 supported rules, 9-3
 mktime, 2-36
 fpu, 3-70, 3-77, 3-85
 modf, 2-36
 msb, 3-14
 muncher, 4-142
 mxp, 3-14

N

name, 3-49
 noprint, 3-82

O

offsetof, 2-36
 open, 2-37

optimization, 4-38, 4-84, 4-127
 option file, 4-20, 4-67, 4-109, 4-145,
 4-177
 org, 3-50
 output file, 4-41, 4-85, 4-129, 4-153
 output format, 4-98, 4-108, 4-144,
 4-148, 4-149, 4-156

P

packed data type support, 1-15
 page, 3-79
 pass option to tool, 4-158
 perror, 2-37
 pmacro, 3-52
 pos, 3-14
 pow, 2-37, 3-14
 pragmas, 1-24
 prctl, 3-81
 predefined macros, 1-25
 predefined macros in C
 __CTC__, 1-25
 __DOUBLE_FP__, 1-25
 __DSPC__, 1-25
 __DSPC_VERSION__, 1-25
 __FPU__, 1-25
 __SINGLE_FP__, 1-25
 __TASKING__, 1-25
 preprocessor, 4-81
 print, 3-82
 printf, 2-38
 processor definition, 7-4, 7-29
 putc, 2-40
 putchar, 2-40
 puts, 2-40

Q

qsort, 2-41

R

raise, 2-41
 rand, 2-41
 read, 2-41
 realloc, 2-42
 reentrancy, 2-63
 register handling, 1-20
 remove, 2-42
 rename, 2-42
 rename sections, 4-42
 rewind, 2-43
 rnd, 3-14
 rvb, 3-15

S

scalb, 2-43
 scanf, 2-43
 scp, 3-15
 sdecl, 3-53
 sect, 3-56
 section, summary, 4-88
 section activation, 3-56
 section attributes, 3-53
 section declaration, 3-53
 section layout definition, 7-5, 7-32
 section names, 3-54
 sections, rename, 4-42
 set, 3-57
 setbuf, 2-45
 setjmp, 2-45
 setjmp.h
 longjmp, 2-33
 setjmp, 2-45
 setlocale, 2-46

setvbuf, 2-46
 sfract, 3-15, 3-40
 sgn, 3-15
 SIGABRT, 2-47
 SIGFPE, 2-47
 SIGILL, 2-47
 SIGINT, 2-47
 signal, 2-47
 signal.h
 raise, 2-41
 signal, 2-47
 signals, 2-47
 SIGSEGV, 2-47
 SIGTERM, 2-47
 silicon bug workaround, 4-44, 4-86,
 4-154
 sin, 2-47, 3-15
 sinh, 2-47
 size, 3-58
 snh, 3-16
 space, 3-59
 sprintf, 2-48
 sqrt, 2-48
 sqt, 3-16
 srand, 2-48
 sscanf, 2-48
 stat, 2-49
 stdarg.h
 va_arg, 2-60
 va_end, 2-60
 va_start, 2-60
 stddef.h, offsetof, 2-36
 stdio.h
 _close, 2-9
 _lseek, 2-9
 _open, 2-9
 _read, 2-9
 _unlink, 2-10
 _write, 2-10
 clearerr, 2-16
 fclose, 2-19
 fEOF, 2-19
 ferror, 2-20

fflush, 2-20
fgetc, 2-20
fgetpos, 2-20
fgets, 2-21
fopen, 2-21
fprintf, 2-22
fputc, 2-22
fputs, 2-23
fread, 2-23
freopen, 2-24
fscanf, 2-24
fseek, 2-25
fsetpos, 2-25
ftell, 2-25
fwrite, 2-26
getc, 2-26
getchar, 2-26
gets, 2-27
perror, 2-37
printf, 2-38
putc, 2-40
putchar, 2-40
puts, 2-40
remove, 2-42
rename, 2-42
rewind, 2-43
scanf, 2-43
setbuf, 2-45
setvbuf, 2-46
sprintf, 2-48
sscanf, 2-48
tmpfile, 2-58
tmpnam, 2-58
ungetc, 2-59
vfprintf, 2-60
vprintf, 2-61
vsprintf, 2-61
 stdlib.h
abort, 2-11
abs, 2-11
atexit, 2-13
atoac, 2-13
atof, 2-13
atofr, 2-14
atoi, 2-14
atol, 2-14
atolac, 2-15
atofr, 2-15
bsearch, 2-15
calloc, 2-16
div, 2-18
exit, 2-19
free, 2-23
getenv, 2-27
labs, 2-31
ldiv, 2-31
malloc, 2-33
mblen, 2-34
mbstowcs, 2-34
mbtowc, 2-34
qsort, 2-41
rand, 2-41
realloc, 2-42
srand, 2-48
strtoac, 2-54
strtod, 2-54
strtofr, 2-54
strtol, 2-55
strtolac, 2-55
strtolfr, 2-56
strtoul, 2-56
system, 2-57
wcstombs, 2-61
wctomb, 2-62
 stitle, 3-83
 strcat, 2-49
 strchr, 2-49
 strcmp, 2-49
 strcoll, 2-50
 strcpy, 2-50
 strcspn, 2-50
 strerror, 2-50
 strftime, 2-51
 string.h
memchr, 2-35
memcmp, 2-35

memcpy, 2-35
memmove, 2-35
memset, 2-36
strcat, 2-49
strchr, 2-49
strcmp, 2-49
strcoll, 2-50
strcpy, 2-50
strcspn, 2-50
strerror, 2-50
strlen, 2-52
strncat, 2-52
strncmp, 2-52
strncpy, 2-52
strpbrk, 2-53
strrchr, 2-53
strspn, 2-53
strstr, 2-53
strtok, 2-55
strxfrm, 2-56
strlen, 2-52
strncat, 2-52
strncmp, 2-52
strncpy, 2-52
strpbrk, 2-53
strchr, 2-53
strspn, 2-53
strstr, 2-53
strtoac, 2-54
strtod, 2-54
strtofr, 2-54
strtok, 2-55
strtol, 2-55
strtolac, 2-55
strtolfr, 2-56
strtoul, 2-56
strxfrm, 2-56
sub, 3-16
switch statement, 4-46
symb, 3-60
system, 2-57
system libraries, 4-113, 4-115

T

tan, 2-57, 3-16
tanh, 2-57
tc, 3-78, 3-84
temporary files, 4-157
time, 2-57
time.h
 asctime, 2-12
 clock, 2-17
 ctime, 2-18
 difftime, 2-18
 gmtime, 2-27
 localtime, 2-32
 mktime, 2-36
 strftime, 2-51
 time, 2-57
tmpfile, 2-58
tmpnam, 2-58
tnh, 3-16
toascii, 2-58
tolower, 2-59
toupper, 2-59
trap handling, 4-147
TriCore 2 instructions, 4-31, 4-76
type, 3-61

U

undef, 3-62
unf, 3-17
ungetc, 2-59
unistd.h
 access, 2-11
 chdir, 2-16
 close, 2-17
 getcwd, 2-27
 lseek, 2-33
 read, 2-41
 stat, 2-49

unlink, 2-59

write, 2-62

unlink, 2-59

V

va_arg, 2-60

va_end, 2-60

va_start, 2-60

verbose, 4-133, 4-160

version information, 4-51, 4-90, 4-132,
4-159, 4-187, 4-188, 4-199

vfprintf, 2-60

vprintf, 2-61

vsprintf, 2-61

W

warning, 3-63

title, 3-86, 3-87

warnings, suppress, 4-91

warnings as errors, 4-54, 4-93, 4-136

warnings, suppress, 4-52, 4-134

wcstombs, 2-61

wctomb, 2-62

word, 3-64

write, 2-62

X

xpn, 3-17