User's Guide

HighTec GNU Toolchain for TriCore

Version 1.21

HighTec EDV-Systeme GmbH
Feldmannstraße 98
D-66119 Saarbrücken

Phone: +49 681-9 26 13-16
Email: info@hightec-rt.com
http://www.hightec-rt.com

HighTec EDV-Systeme GmbH
Stammsitz & Entwicklung
Feldmannstraße 98
D-66119 Saarbrücken
Tel.: 0681/92613-16 Fax: -26
mailto:htc@hightec-rt.com

www.hightec-rt.com

Contents

If you can not convince them, confuse them.

– Harry S Truman

Part I

Introduction

1 The TriCore Development Platform

Purpose

This book is intended to help using the TriCore Development Platform. It describes the structure and configuration of the TriCore Development Platform. To understand this manual it is assumed that you are familiar with the C language.

This book describes the tools of the TriCore Development Platform, which are used to create a executable binary file from one or more C, C++ or assembly source files. For each of these programs the supported options for TriCore based platforms are described. These supported options may be a subset of the options supported by the toolchain ports for other target platforms.

1.1 Conventions

The following typographic conventions are used in this book:

`command`	A command, which is to be entered in the command line.
statement	C-, C++ or assembler statement.
|	This symbol separates a list of parameters or options. Exactly one of the elements of this list has to be chosen.
[]	Optional parameters or options are surrounded by square brackets. These square brackets may contain a list of parameters or options.
`<parameter>`	Parameter for a command or a statement. This item must be substituted by a reasonable value.

Table 1.1: Conventions

Parameters are values, which are passed to a program or statement as input or output or to process them. A parameter may be the input file that is passed to the compiler. Options, or switches, are setting a program in a certain mode. They may also be used to pass additional parameters to a program. A program can do its work without options but normally not without parameters.

Note:

Useful hints are printed as notes in blue marked boxes.

Example

This command may be printed in this book:

```
tricore-gcc <inputfile> [-o <outputfile>] [-c | -E]
```

It may be entered in the command-line like that:

```
tricore-gcc main.c -o main.o -c
```

Note:

C code within the text is printed in serif fonts. C keywords are typed boldface (e.g. **if**). Listings are printed in courier font.

As regular users of the World Wide Web will know, keeping track of URLs is tricky, error-prone process as sites continually disappear or change their structure. In the manual, therefore, we do not give formal URLs in the text, but rather give pointers to a URL catalog in the appendix. This catalog will be kept up to date.

Internal hyperlinks are printed in dark blue and file links are green.

1.2 Introduction to the TriCore Development Platform

The TriCore Development Platform is a complete program suite to develop programs for microcontrollers of the Infineon TriCore [↪TriCore] family. The TriCore Development Platform contains the programs to create an executable from high level language source code or assembler code. These programs are based on the development tools provided by the *GNU* project founded in the mid-1980s. The *GNU* development tools are set up in millions of installations worldwide and large projects like GNU/Linux are based on the reliability of the *GNU* tools.

These development tools are adapted by HighTec to create executables for TriCore microcontrollers. This ensures the stability and reliability of the *GNU* tools. Controlled by a common compiler driver program the TriCore Development Platform consists of the following programs:

The Preprocessor The Preprocessor prepares the C- or C++-source code for the compiler. In this first state of the compiling process the preprocessor directives like **#include** or **#define** are interpreted. The preprocessor merges files, replaces macros and removes comments from the source code. Additionally the built in parser checks the syntax of the source code to find errors.

The Preprocessor accepts files with C, C++ or assembler source code and outputs files with the preprocessed source code.

For a detailed description of the preprocessor see chapter 17 on page 139.

The Compiler The Compiler converts the target independent high level language source code to target specific assembler code. Since the assembler code is specific to the target hardware the compiler is specific for the target, too.

While compiling the source code to assembler code several features of the compiler may be activated. These features are used to optimize the generated assembler code. These optimizations contain among other things the reordering of instructions for a better performance or workarounds for CPU errata. Additionally optional debug information is created. This information is used to inspect the executable, which is to be created by the linker, in the debugger later on.

The input file for the compiler must contain source code, which was preprocessed by the preprocessor. The compiler outputs assembler code, which conforms with the TriCore architecture Manual.

For a detailed description of the compiler see chapter 18 on page 147.

The Assembler The assembler generates an object file from the assembler code, which is passed to it in the input file. This object file is created in the tricore-elf object format. The data in the object file consists of the instructions, that will be executed by the target machine, and data used by the program. After linking the content of the object file will be ready to be loaded into the target. No addresses are assigned to the dates in the object file, they are not located yet.

For a detailed description of the assembler see chapter 19 on page 187.

The Linker After all C- or C++ source files of a project are compiled in object files these object files are put together to a executable. This executable is generated by the linker by merging the object files of the project with objects from libraries and target specific startup code. This merge is controlled by a linker script file. By using the linker script file the linker assigns addresses to the dates in the passed files.

Note:

The linker must be invoked only once for every build process whereas preprocessor, compiler and assembler must be invoked for every source file.

For a detailed description of the linker see chapter 20 on page 221.

The BinUtils Strictly speaking the assembler and linker are parts of the binutils, too. The binutils are a collection of programs to inspect the files generated by the compiler tools. These programs are not invoked while executing the compiler driver program but must be started separately. Creating disassembly, copying output files from one file format to another or combining objects in archives are just a few of the tasks that can be done using a program of the binutils.

For a detailed description of the binutils see chapter 21 on page 287.

While compiling high level language source files various intermediate files may be generated. By means of the file suffix the content of the file can be determined.

While compiling an input file the name of the file stays the same, only the suffix changes: If the file `main.c` is compiled to an object file, the linker outputs the file `main.o`.

It is convention to use the following file extensions:

Note:

The files that are used as input files are case sensitive. `main.c`, `Main.c` and `MAIN.C` are not recognized to be the same file.

Extension	Description
.c	C source code file. Passed to the preprocessor by the control program.
.cpp, .c++, .C, .cc, .cp, .cxx	C++ source code file. Passed to the preprocessor by the control program.
.h	Header file. These files are not compiled or linked. Their content is merged with the C or C++ source code files by the preprocessor.
.i	File with preprocessed C source code. Passed to the compiler by the control program.
.ii	File with preprocessed C++ source code. Passed to the compiler by the control program.
.s, .S	File with assembler code. Passed to the assembler by the control program.
.o	Dynamically linked object file.
.d	`make` dependency file generated by the preprocessor.

Table 1.2: File extensions

Since the components of the TriCore Development Platform are interdependent it is necessary to invoke them correctly and in the right order. This is done by a common control program, sometimes called compiler driver. Besides setting some defaults for the invoked programs the compiler driver is responsible for passing the right in- and output files to the components of the toolchain. These defaults may be overwritten by user defined settings. This guarantees both the flexibility required by professional developers and the easiness of creating a program without having to think too much (see Figure 1.1 on page 7).

Basing on the file extension of the file the control program gets as input it starts the toolchain with the appropriate tool. If no other options are set the toolchain is run until the linker has finished and the executable is created. The generated program may be directly loaded into the target hardware afterwards. However the TriCore Development Platform may be stopped after a defined state. This makes it possible to generate assembler code files from high level language files or object files to add to a library or to link together.

Note:

It is not recommended to start the components of the toolchain individually. The compilation of a program should always be coordinated by the control program.

The TriCore Development Platform moreover includes programs to inspect the generated object code. These tools are, together with the assembler and the linker, the so called binutils.

Besides the programs directly involved in the creation of executables from source files the TriCore Development Platform comes with additional tools to manage the software build

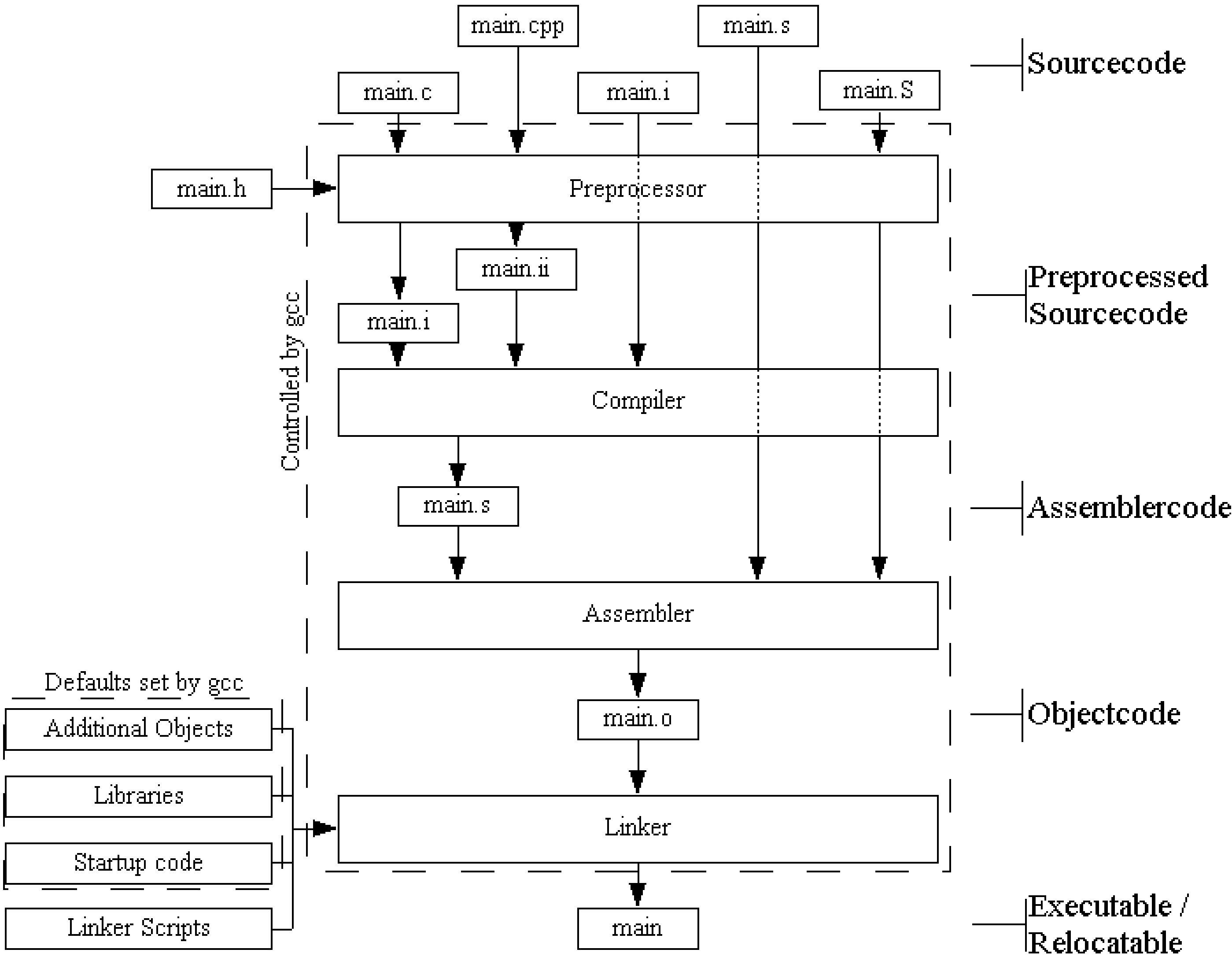

Figure 1.1: The TriCore Development Platform

process or to debug the created program. Among these are:

The GNU make `make` is a tool to support the management of large projects with more than just a few source files. This program coordinates the invocation of the compiler driver and passes the various input files to it. Its main purpose is to provide a consistent frontend to the user regardless of the size of the project.

The GNU Debugger The debugger is a program to help finding errors in programs. The executable can be downloaded to the target and run. The execution of the program on the target can be stopped at user defined break points and the content of memory, registers and variables may be examined.

The TriCore JTAG-Server The TriCore JTAG-Server provides a TCP/IP-Gateway to the JTAG-Interface of a target to download and debug a program via the *GNU* debugger.

The TriCore Simulator By using the TriCore simulator executables for the target machine may be tested without the physical hardware. The TriCore Simulator provides the same interface for the debugger as the TriCore JTAG-Server does. Thus it is transparent for the debugger whether it connects to a real hardware or a simulator.

1.3 Working with the TriCore Development Platform

The compiler driver `tricore-gcc` is a program which must be started from a command line interface (The DOS-Prompt on Microsoft Windows systems, a shell like the bash, the ksh or the csh on UNIX systems or Linux). By adding options to the invocation of the control program its behavior may be affected.

The control program is started in the following way:

```
tricore-gcc [options] <input file>
```

The TriCore Development Platform does not come with a built-in graphical user interface. All tools must be started via the command line. But many integrated development environments (IDEs), such as RedHat's Source Navigator, Borland's Codewright, KDevelop or Anjuta, may be used to add a GUI to the toolchain.

The control program accepts a huge amount of command line options. A few of them are used to control the compiler driver itself, most of them are passed through to the appropriate program of the toolchain. On the opposite, not all options available for the component programs are accessible from the compiler frontend. These are some of the most important options accepted by the compiler driver program:

`--version` Prints the version and the release number of the compiler driver program.

`-o <output file>`
The output file resulting from this compiler run is stored in <output file>.

`-c` The input file is compiled to a object file. The toolchain is stopped after the assembler. Normally all object files of a project are merged in one single executable file in a final link step.

`-g` Debug information is added to the generated executable. The debug information is generated in the default debug format. The only debug format currently supported by the TriCore compiler tools is DWARF version 2.

Note:

With `-g3` debug info is added to defines.

`-Wall` Nearly all warnings that occurred while compiling a program are displayed. These warnings include warnings originated by syntactically wrong code or by unused variables. These warnings are not treated as errors. The compilation of the source files will continue even if a warning was issued. Warnings are only for informational purposes.

For a detailed description of the compiler driver, the component programs and their available options see Part III on page 131.

2 Software Installation

This chapter describes how you can install the TriCore Development Platform on Windows hosts.

2.1 System Requirements

Use of TriCore Development Platform requires:

- the TriCore Development Platform CD
- an IBM-compatible host-PC (486DX or higher)
- minimum 128 MB RAM
- Windows NT/Service Pack 3, Windows 2000/XP or
- LINUX SuSE 7.x or Red Hat 7.x

2.2 Components of TriCore Development Platform

GNU provides a full line of tools to take source code through each step of the translation process. The result is a quick and efficient executable binary file. The *GNU* system is more than a cost effective alternative to other commercial programs; it rounds off any system with an exceptional mixture of functionality and comfort. The *GNU* system in itself is a 'work of art'.

The *GNU* Tools for the TriCore consist of

- compiler driver tricore-gcc
- preprocessor and compilers
- assembler tricore-as
- linker/locator tricore-ld
- source debugger tricore-gdb

With *GNU* `tricore-gcc` you can use:

- inline functions
- inline assembler (see chapter 8 on page 49)
- attributes (see chapter 6 on page 27)

A number of *GNU* utilities for analyzing and post-processing object files are available for the TriCore.

- Convert addresses into file names and line numbers with tricore-addr2line.

- Create, modify, and extract from archives with tricore-ar.
- List symbols from object files with tricore-nm.
- Copy and translate object files with tricore-objcopy.
- Display information from object files with tricore-objdump.
- Generate index to archive contents with tricore-ranlib.
- Display the contents of ELF format files with tricore-readelf.
- List file section sizes and total size with tricore-size.
- List printable strings from files with tricore-strings.
- Discard symbols with tricore-strip.

For detailed description see chapter 21 on page 287.

2.3 Installation

When you insert the TriCore Development Platform CD into the CD-ROM drive of your host-PC, the CD should automatically launch a setup program that installs the required software. Otherwise the setup program *setup.exe* can be executed manually from the root folder of the CD.

The installation can be done with normal user permissions except the system driver *givio*. So, if you install the CD without administrator permissions you have to install the system drivers manually with administrator permissions: Login as administrator and call the batch file *jtaginst.bat* in the destination location, directory `tricore\jtag`.

The default destination location is `C:\HighTec\tricore`. All path and file statements within this manual are based on the assumption that you accept the default <install-dir> as install paths and drives. If you decide to individually choose different paths and/or drives you must consider this for all further file and path statements.

3 A Sample Program

This chapter describes how you can create your first executable from a very simple sample program.

This is the source code of this program, saved in the file main.c:

```
/*  Description:

    A simple sample program for the introduction in the user's guide.
    this example should be target independent for using as a general example.

    (c) HighTec EDV-Systeme GmbH 2003 */

const char table[7] = {'T','H','E','C','H','I','G'};
const char indices[7] = {4, 0, 5, 6, 3, 1, 2};

int main (void) {

int i;
char result[7];

char place;

    for (i = 0; i < 7; i++) {
  place = indices[i];
  result[place] = table[i];
    }

}
```

To compile this program in a executable open a command line interface. This is the DOS-Box on a Microsoft Windows system or a shell like the bash or the ksh on UNIX or Linux. Then change to the directory the file is stored at and type:

```
tricore-gcc -o main main.c
```

Now the compiler is started and the input file `main.c` is compiled to the executable `main`. This file is ready to be loaded into the target.

Note:

If you want to use a makefile for linux and other operating system you can check the system with `ifeq ($(OSTYPE),linux)`

Here a small example:

```
ifeq ($(OSTYPE),linux)
    RDF = rm -rf
else
    RDF = rd
endif
```

If you test `$(OSTYPE)` under linux it returns `linux`, independent of your linux version. Under windows it depends on the windows version (NT,2000,XP).

4 Another Sample Program

In this chapter you will learn how to generate a more complex program with more than one input file. These input files require additional header files which are not in the build directory. This example is for the target board TriBoard.

HelloSerial is a simple application, which demonstrates how to use the UART-Interface. The main-routine reads a character from serial interface. Depending on the value of this character, the main-routine does one of following

- switches LED to off (if character is "0")
- switches LED to on (if character is "1")
- print "Hello world (if character is "2")
- finish program (if character is "E")

otherwise read a character again.

This example shows how to compile each source file of a project in a object file and how to link them afterwards together to a executable. The sources for this example are delivered with the compiler and are installed in the subdir `Examples\HelloSerial\src`.

The first step of compiling this sample program to a executable is to compile all C source files to object files. These object files contain the program code as the executable would but this code has no addresses associated yet. The addresses are not defined until the linking has been done. To compile these files open a command line interface, change to the example's directory and type the following:

```
tricore-gcc -c -I defs -o triuart.o triuart.c
```

This command will compile the source file `triuart.c` to the object file `triuart.o`. The files in the subdirectory `defs` are included in the compiler's search path for include files. The same command is used for the second source file:

```
tricore-gcc -c -I defs -o hello.o hello.c
```

Now the two objects of this project are created and they must be linked. This is done by the control program `tricore-gcc`, too. It can figure out which part of the toolchain must be started by inspecting the extension of the input files and if files with the extension `.o` are passed as input, the linker is started. To link the files, type the following:

```
tricore-gcc -o helloserial hello.o triuart.o
```

This will link the two previously generated object files `hello.o` and `triuart.o` together to the executable `helloserial`. Now this file can be downloaded to a target hardware, treated with the binutils or inspected with the debugger.

Open the "COM Direct"-Terminal. Now, you can type a "0" at your keyboard to switch a LED to off at your development board. A "1" to turn the LED on. If you type a "2" in the terminal the Text "Hello world!" appears. To finish the program type "E".

For a complete reference of `tricore-gcc` and what it can do for you see chapter 16 on page 131.

Part II

Implementation

5 Implementation Issues

5.1 Data Elements

The following ANSI-C data types are supported by TriCore Development Platform.

Type	Size	Value Range
_bit	$\frac{1}{8} to 1$	0 to 1
signed char	1	-128 to 127
unsigned char	1	0 to 255
signed short	2	-32768 to 32767
unsigned short	2	0 to 65535
signed int	4	-2147483648 bis 2147483647
unsigned int	4	0 to 4294967295
signed long	4	-2147483648 bis 2147483647
unsigned long	4	0 to 4294967295
signed long long	8	-9223372036854775808 to 9223372036854775807
unsigned long long	8	0 to 18446744073709551615
float	4	$\pm 1.1755 \cdot 10^{-38}$ to $\pm 3.402 \cdot 10^{38}$
double	8	$\pm 2.225 \cdot 10^{-308}$ to $\pm 1.798 \cdot 10^{308}$
enum	1 to 4	0 to 4294967295
pointer	4	0 to 4294967295

Table 5.1: Data types

The character type is treated a **signed char** by default.

An unsuffixed floating constant has the type **double**. If it is suffixed by the letter `f` or `F`, it has the type **float**.

Options

`-fsigned-char` variables of type **char** are evaluated as **signed char**

`-funsigned-char`
variables of type **char** are evaluated as **unsigned char**

5.2 Bit Data Type

The _bit is a known keyword of `tricore-gcc`.

5.2.1 Defining Bit Variables

For bit variables the value 0 or 1 is possible. This type or variable can be defined as global initialized, global uninitialized or local module initialized variable.

```
_bit bit1; //(global uninitialized variable)
_bit bit2 = 1; //(global initialized variable)
static _bit bit3; //(local module initialized variable)
```

The default section name for bit data type is .bdata and .bbss. In the first step the assembler reserves one byte per bit. Using the linker option `--relax-bdata` packs the bits in bytes and generates the byte address and bit offset, so it is possible to store module bits in bytes in a packed manner. For each module only one .bdata section is defined. All bit data types are allocated in this section. Only the linker can merge the .bdata section of various modules in absolute addressable sections. In the .bdata section all not explicitly initialized bits are cleared. For the bit offsets a section . boffs is generated by the linker, which has the size zero, because it is not a loadable section. The section . boffs contains symbol information of <name>.pos.

Note:
In the dissambly of an elf-file the `tricore-objdump` tries to find the corresponding symbol to an address. For instructions like `st.t` the symbol name for an address of the _bit variable may be incorrect.

In the Mapfile the address of bit variables is declared as <base address>.<bit offset>. For example for 9 defined global bits the following entries are generated in the Mapfile.

```
Allocating bit symbols (l/g = local or global scope)
Bit symbol bit address l/g file

test 0xa0000000.0 g ...
test1 0xa0000000.1 g ...
test2 0xa0000000.2 g ...
test3 0xa0000000.3 g ...
test4 0xa0000000.4 g ...
test5 0xa0000000.5 g ...
test6 0xa0000000.6 g ...
test7 0xa0000000.7 g ...
test8 0xa0000001.0 g ...
```

In this small example 0xa0000000 is a base address and .0 the bit offset of the variable test to the base address.

The datatype _bit is implemented as **unsigned int** with the size one. This datatype is an C/C++-language extension and should not be mixed up with the datatype _Bool.

A variable of the type _bit can only have the values 0 or 1, a assignment of a value larger than 1 is not allowed. In this case a warning is output. If a constant or variable integer value is assigned to a _bit-variable the bitposition zero (the LSB) of this integer value is assigned to the _bit-variable. If a variable of an integer type is casted to a _bit-variable, the least significant bit of the integer is used, too. The result of a cast from an integer type to a _bit is always 1 for odd integers and always 0 for even integers.

While casting a float to a _bit-variable, the float is transformed to an unsigned integer by using the function __f_ftoui. The resulting integer is casted to a _bit afterwards.

The compiler differs between assignments and conditionals. Generally is:

Note:

In conditional statements the parameters are always expanded to an **int**, in assignments they are decreased to the smallest possible datatype

5.2.2 Bit operations

Allowed bit operations are:

- assignment
- logical unary operators
- logical binary operators
- test operators

Assignement

```
bit1 = 0;
bit2 = 1;
bit3 = bit1;
```

logical unaray operators

```
bit1 = ~bit2;
```

logical binary operators

```
bit1 = bit2 | bit3;
bit1 = bit2 & bit3;
bit1 = bit2 ^ bit3;
```

test operators

```
if(bit1 ==1) ...
if(bit1 ==0) ...
if(bit1!=1) ...
if(bit1!=0) ...
if(bit1 == bit2) ...
if(bit1 != bit2) ...
```

5.2.2.1 Assignment

Examples for the combination of _bit-datatypes and other datatypes (e.g. **char**).

```
_bit b1, b2;
char c1, c2;

void char_assign1(void)
{
    b1 = c1;
    /*
        b1 = 1 if c1 odd,
        b1 = 0 if c1 even
    */
}
```

An assignment of a **char** to a _bit

```
    b1 = c1;
```

is interpreted by the compiler as:

```
    b1 = (_bit)c1;
```

b1 gets either the value 1 if c1 is odd (LSB set) or the value 0 if c1 is even (LSB not set).

This behavior is valid for all integer data types.

5.2.2.2 Conditionals

Examples for the combination of _bit-datatypes and other datatypes.

if-Statement

```
void char_equal(void)
{
    if (b2 == c2)
     {
        b1 = 1;
     }
}
```

This code is interpreted by the compiler as follows:

```
    if ( (int)b2 == (int)c2 )
```

The condition b2 == c2 is true, if c2 = b2 = 1 or c2 = b2 = 0. In all other cases the condition is false.

Another example:

```
int i, j;
_bit b;

int main (void)
{
    j = b ? 0 : 1;
    return 0;
}
```

The compiler generates this assembler code with the option -O2:

```
    movh.a %a15,HI:j
    call __main
    mov %d2, 0
    ld.b %d0, b
    extr.u %d15, %d0, bpos:b, 1
    xor %d15,%d15, 1
    st.w [%a15] LO:j, %d15
    ret
```

switch-Statement

The evaluation of a _bit in a **switch**-statement is similar to the evaluation of a **if**-Statement. The _bit is expanded to an integer, too.

```
int i;
_bit b;

int main (void)
{
    switch (b)
    {
        case 0:
            i = 4;
            break;
        case 1:
            i = 5;
            break;
        default:
            i = 1;
    }
    return 0;
}
```

The compiler generates this assembler code with the option `-O2`:

```
    call __main
    ld.b %d0, b
    extr.u %d0, %d0, bpos:b, 1
    mov %d15, 4
    movh.a %a15,HI:i
    jeq %d0, 0, .L7
    mov %d15, 5
    movh.a %a15,HI:i
    jeq %d0, 1, .L7
    mov %d15, 1
    movh.a %a15,HI:i
.L7:
    mov %d2, 0
    st.w [%a15] LO:i, %d15
    ret
```

5.2.2.3 Bits in functions

Parameters and return values of functions may be of the type _bit:

```
int i;

_bit foo (_bit b)
{
    i = b;
    return b;
}
```

5.2.3 Invalid operators

These operations are not allowed for the _bit-datatype:

- ++, −− (post/pre increment/decrement)
- unary minus (e.g. -b1)

- indirection (array/pointer/address)
- +, −, ∗, /, %, <<, >>
- +=, −=, ∗=, /=, %=, <<=, >>=

Note:

If the type _bit is defined within a **struct** or **union** the compiler will issue an error.

Examples

The addition of a not negated bit and any other datatype is not allowed and causes an error, since the addition of bits is not allowed.

```
int main (void)
{
    b1 = 1;
    b2 = 1;
    res = !b1 + b2;
    return 0;
}
```

This code does not cause an error message:

```
_bit b1, res;
unsigned int u1;

int main (void)
{
    res = !b1 + u1;
    return 0;
}
```

This assembler code is generated with the compiler option `-O2`:

```
    movh.a %a15,HI:u1
    call __main
    mov %d2, 0
    ld.b %d0, b1
    extr.u %d15, %d0, bpos:b1, 1
    xor %d15,%d15, 1
    lea %a15,[%a15] LO:u1
    ld.b %d0, [%a15] 0
    add %d15, %d15, %d0
    jz.t %d15,0,0f
    st.t res,bpos:res,1
    j 1f
0: st.t res,bpos:res,0
1:
    ret
```

The negation operator ! expands `b1` to an integer. Since the addition of integer values is allowed, no error message is output. So this code is valid, too:

```
_bit b1, b2, res;

int main (void)
{
    res = !b1 + !b2;
    res = !b1 + ~b2;
    return 0;
}
```

5.3 Inquiring on Alignment of Types or Variables

The keyword __alignof__ allows you to inquire about how an object is aligned, or the minimum alignment usually required by a type. Its syntax is just like **sizeof**. For example, if the target machine requires a double value to be aligned on an 8-byte boundary, then __alignof__ (**double**) is 8.

Note:

This is true on many RISC machines. On more traditional machine designs, __alignof__ (**double**) is 4 or even 2. Some machines never actually require alignment; they allow reference to any data type even at an odd address. For these machines, __alignof__ reports the recommended alignment of a type.

If the operand of __alignof__ is an lvalue rather than a type, its value is the required alignment for its type, taking into account any minimum alignment specified with GCCs __attribute__ extension (see chapter 6 on page 27).

6 Attributes

The __attribute__ keyword can be used to assign an attribute to a function or data declaration.

Note:

Attributes are a powerful feature of *GNU* compiler and are not known in ANSI C.

This keyword is followed by an attribute specification inside double parentheses. You may also specify attributes with "__" preceding and following each keyword. This allows you to use them in header files without being concerned about a possible macro of the same name. For example, you may use __aligned__ instead of aligned. Multiple attributes can be assigned in the same declaration by including them in a comma-separated list.

6.1 Specifying Attributes of Functions

For `tricore-gcc` you can specify attributes for functions. An attribute is assigned to a function in the declaration of the function prototype:

```
<Type> <name of function> (<parameter>) __attribute__ ((<attribute>));
```

The following example illustrates the use of an attribute:

```
void sayhello (void) __attribute__ ((const));
```

List of attributes

`alias` A function definition with this attribute causes the definition to become a weak alias of another function. It can be used in combination with weak attribute to define a weak alias. For instance,

```
void __f () { /* do something */; }
void f () __attribute__ ((weak, alias ("__f")));
```

declares "f" to be a weak alias for __f. In C++, the mangled name for the target must be used.

Note:

Not all target machines support this attribute.

`always_inline` A function with this attribute that is declared as being inline, will always be expanded as inline code, even when no optimization is specified. This attribute forces the compiler to inline the function ignoring the size of the function

`const`
: Many functions do not examine any values except their arguments, and have no effects except the return value. Basically this is just slightly more strict class than the pure attribute, since function is not allowed to read global memory. Note that a function that has pointer arguments and examines the data pointed to must not be declared **const**. Likewise, a function that calls a non-const function usually must not be **const**. It does not make sense for a const function to return void. The **const** attribute gives the optimizer more freedom than pure because there is no need to make certain that all global values are updated before the function is called. With this attribute Common Subexpression Elimination (CSE) and loop optimization is improved.

`constructor`
: The constructor attribute causes the function to be called automatically before execution enters main(). Functions with these attributes are useful for initializing data that will be used implicitly during the execution of the program. Also see the destructor attribute.

> **Note:**
>
> These attributes are not currently implemented for Objective-C.

`deprecated`
: A function with this attribute will cause the compiler to issue a warning message whenever it is called. This is useful when identifying functions that are expected to be removed in a future version of a program. The warning also includes the location of the declaration of the deprecated function, to enable users to easily find further information about why the function is deprecated, or what they should do instead.

```
int old_fn () __attribute__ ((deprecated));
int old_fn ();
int (*fn_ptr)() = old_fn;
```

> **Note:**
>
> Results in a warning on line 3 but not line 2.

The deprecated attribute can also be used for variables and types (see section 6.2 on page 31).

`destructor`
: A function with this attribute is called automatically after main() has returned or exit() has been called.

`format`
: A function with this attribute has one argument that is a format string and a variable number of arguments for the values to be formatted. This makes it possible for the compiler to check the format content against the list of arguments to verify that the types match the formatting. There are different types of formatting, so it is also necessary to specify whether validation is to be for the printf, scanf, strftime, or strfmon style. For example the following attribute specifies that the second argument passed to the function is the formatting string, the

formatting string is expected to be of the printf type, and the variable-length argument list begins with third arguments:

```
extern int
my_printf (void *my_object, const char *my_format, ...)
__attribute__ ((format (printf, 2, 3)));
```

Warning messages are issued when format string is found to be invalid only if the `-Wformat` option is specified. (You can also use __printf__, __scanf__, __strftime__ or __strfmon__.) For functions where the arguments are not available to be checked (such as vprintf), specify the third parameter as zero. In this case the compiler only checks the format string for consistency. For strftime formats, the third parameter is required to be zero. In the example above, the format string (my_format) is the second argument of the function my_print, and the arguments to check start with the third argument, so the correct parameters for the format attribute are 2 and 3. The format attribute allows you to identify your own functions which take format strings as arguments, so that GCC can check the calls to these functions for errors. The compiler always (unless `-ffreestanding` is used) checks formats for the standard library functions printf, fprintf, sprintf, scanf, fscanf, sscanf, strftime, vprintf, vfprintf and vsprintf whenever such warnings are requested (using `-Wformat`), so there is no need to modify the header file stdio.h. In C99 mode, the functions snprintf, vsnprintf, vscanf, vfscanf and vsscanf are also checked.

`format_arg` A function with this attribute accepts a formatting string as one of its arguments and makes a modification to the string so that the result can be passed on to a print(), scanf(), strftime, or strfmon(() type function. This attribute will suppress warning messages issued when the option `-Wformat-nonliteral` is set to detect nonconstant formatting strings. The following example demonstrates the setting of this attribute for a function that has such a format string as its second argument:

```
extern char *
my_dgettext (char *my_domain, const char *my_format)
__attribute__ ((format_arg (2)));
```

This causes the compiler to check the arguments in calls to a printf, scanf, strftime or strfmon type function, whose format string argument is a call to the my_dgettext function, for consistency with the format string argument my_format. The format_arg attribute allows you to identify your own functions which modify format strings, so that GCC can check the calls to printf, scanf, strftime or strfmon type function whose operands are a call to one of your own function.

`malloc` A function with this attribute informs the compiler that it can be treated as if it were the malloc() function. For the purposes of optimization, the compiler is to assume the returned pointer cannot alias anything.

`no_instrument_function`
A function with this attribute will not be instrumented and will not have profiling code inserted into it by the compiler, even if the option `-finstrument-function` is set.

`noinline`
A function with this attribute will never be expanded as inline code.

`noreturn`
A function with this attribute does not return to its caller (e.g. function exit). A call of a function with this attribute within another function implies that this function will also have no return. All functions that will not return the attribute noreturn is meaningfully. This attribute suppresses warnings and makes slightly better code because the compiler does not regard to what would happen if fatal ever did return. More importantly, it helps avoid spurious warnings of uninitialized variables.

With `-Wmissing-noreturn` a warning is issued, if a function that will not return the attribute noreturn is not used.

`pure`
Many functions have no effects except the return value and their return value depends only on the parameters and/or global variables. That is, there will be no changes to global values, locations addressed by arguments, or the contents of files. Unlike the **const** attribute, this function may read global values. Such a function makes it possible for the compiler to perform common subexpression optimization because all the values are guaranteed to be stable. These functions should be declared with the attribute pure. For example,

```
int square (int) __attribute__ ((pure));
```

says that the hypothetical function square is safe to call fewer times than the program says.

Some of common examples of pure functions are strlen or memcmp. Interesting non-pure functions are functions with infinite loops or those depending on volatile memory or other system resource, that may change between two consecutive calls (such as feof in a multithreading environment).

Note:
The attribute pure is not implemented in GCC versions earlier than 2.96.

`section`
A function with this attribute section(<section−name>) will have its assembly language code placed into the named section instead of the default .text section.

```
void sayhello (void) __attribute__ ((section(".foo")));
```

The use of the tricore-gcc with `-ffunction-sections` option advises the compiler to create a section with the function name in the output file. E.g. the function main is located in the section .main.

Note:

To get access to the section for debugging the sections must be declared in the linker description file (see section 20.2 on page 237). Some file formats do not support arbitrary sections so the section attribute is not available on all platforms. If you need to map the entire contents of a module to a particular section, consider using the facilities of the linker instead.

`unused` This attribute, attached to a function, means that the function is meant to be possibly unused. GCC will not produce a warning for this function.

Note:

GNU C++ does not currently support this attribute as definitions without parameters are valid in C++.

`used` A function with this attribute causes the compiler to generate code for the function body, even if the compiler determines the function is not being used. This can be useful for functions that are called only from inline assembly.

`weak` A function with this attribute has its name emitted as a weak symbol instead of a global name. This is primarily for the naming of library routines that can be overridden by user code. Weak symbols are supported for ELF targets, and also for a.out targets when using the *GNU* assembler and linker.

6.2 Specifying Attributes of Variables

The keyword __attribute__ allows you to specify special attributes of variables or structure fields. This keyword is followed by an attribute specification inside double parentheses. Ten attributes are currently defined for variables: aligned, deprecated, mode, nocommon, packed, section, transparent_union, unused, vector_size, and weak. Some other attributes are defined for variables on particular target systems. Other attributes are available for functions (see section 6.1 on page 27) and for types (see section 6.3 on page 34).

You may also specify attributes with __ preceding and following each keyword. This allows you to use them in header files without being concerned about a possible macro of the same name. For example,you may use __aligned__ instead of aligned.

`aligned` This attribute specifies a minimum alignment for the variable or structure field, measured in bytes. For example, the declaration:

```
int x __attribute__ ((aligned (16))) = 0;
```

causes the compiler to allocate the global variable x on a 16-byte boundary.

You can also specify the alignment of structure fields. For example, to create a double-word aligned **int** pair, you could write:

```
struct foo { int x[2] __attribute__ ((aligned (8))); };
```

This is an alternative to creating a union with a **double** member that forces the union to be double-word aligned.

As in the preceding examples, you can explicitly specify the alignment (in bytes) that you wish the compiler to use for a given variable or structure field. Alternatively, you can leave out the alignment factor and just ask the compiler to align a variable or field to the maximum useful alignment for the target machine you are compiling for. For example, you could write:

```
short array[3] __attribute__ ((aligned));
```

Whenever you leave out the alignment factor in an aligned attribute specification, the compiler automatically sets the alignment for the declared variable or field to the largest alignment which is ever used for any data type on the target machine you are compiling for. Doing this can often make copy operations more efficient, because the compiler can use whatever instructions copy the biggest chunks of memory when performing copies to or from the variables or fields that you have aligned this way.

The aligned attribute can only increase the alignment; but you can decrease it by specifying packed as well. See below.

Note that the effectiveness of aligned attributes may be limited by inherent limitations in your linker. On many systems, the linker is only able to arrange for variables to be aligned up to a certain maximum alignment. (For some linkers, the maximum supported alignment may be very very small.) If your linker is only able to align variables up to a maximum of 8 byte alignment, then specifying aligned(16) in an __attribute__ will still only provide you with 8 byte alignment. See your linker documentation for further information.

`deprecated` A variable with this attribute will cause the compiler to issue a warning every place it is referenced. This is useful when identifying types that are expected to be removed in a future version of a program. If possible, the warning also includes the location of the declaration of the deprecated type, to enable users to easily find further information about why the type is deprecated, or what they should do instead. Note that the warnings only occur for uses and then only if the type is being applied to an identifier that itself is not being declared as deprecated.

```
typedef int T1 __attribute__ ((deprecated));
T1 x;
typedef T1 T2;
T2 y;
```

```
typedef T1 T3 __attribute__ ((deprecated));
T3 z __attribute__ ((deprecated));
```

results in a warning on line 2 and 3 but not lines 4, 5, or 6. No warning is issued for line 4 because T2 is not explicitly deprecated. Line 5 has no warning because T3 is explicitly deprecated. Similarly for line 6.

The deprecated attribute can also be used for functions and variables (see section 6.1 on page 27 and section 6.2 on page 31.)

`mode`
: A variable with this attribute is sized to match the size of the specified mode. The mode can be set to byte, word, or pointer. The mode attribute determines the data type. For example, the following creates an **int** that is size of a single byte:

```
int x __attribute__ ((mode(byte)));
```

`nocommon`
: This attribute requests GCC not to place a variable into the common section pool but instead to allocate space for it directly in .bss section. If you specify the **-fno-common** flag, GCC will do this for all variables.

Specifying the nocommon attribute for a variable provides an initialization of zeros. A variable may only be initialized in one source file.

`packed`
: The packed attribute specifies that a variable or structure field should have the smallest possible alignment—one byte for a variable, and one bit for a field, unless you specify a larger value with the aligned attribute.

Here is a structure in which the field x is packed, so that it immediately follows a:

```
struct foo
{
  char a;
  int x[2] __attribute__ ((packed));
};
```

`section`
: A variable with this attribute will be placed into the named section instead of the default .data or .bss section. Sometimes, however, you need additional sections, or you need certain particular variables to appear in special sections, for example to map to special hardware. The section attribute specifies that a variable (or function) lives in a particular section. For example, this small program uses several specific section names:

```
struct duart a __attribute__ ((section ("DUART_A"))) = { 0 };
struct duart b __attribute__ ((section ("DUART_B"))) = { 0 };
char stack[10000] __attribute__ ((section ("STACK"))) = { 0 };
int init_data __attribute__ ((section ("INITDATA"))) = 0;
```

Because of the way the linker handles data, data declared in its own section must be initialized data. This attribute will be ignored on system that do not support sectioning.

Use the section attribute with an <initialized> definition of a <global> variable, as shown in the example. GCC issues a warning and otherwise ignores the section attribute in uninitialized variable declarations.

You can force a variable to be initialized with the `-fno-common` flag or the nocommon attribute.

`unused`
: This attribute, attached to a variable, means that the variable is meant to be possibly unused. GCC will not produce a warning for this variable.

`vector_size`
: This attribute specifies the vector size for the variable, measured in bytes. For example, the declaration:

```
int foo __attribute__ ((vector_size (16)));
```

causes the compiler to set the mode for foo, to be 16 bytes, divided into **int** sized units.

This attribute is only applicable to integral and float scalars, although arrays, pointers, and function return values are allowed in conjunction with this construct.

Aggregates with this attribute are invalid, even if they are of the same size as a corresponding scalar. For example, the declaration:

```
struct S { int a; };
struct S __attribute__ ((vector_size (16))) foo;
```

is invalid even if the size of the structure is the same as the size of the **int**.

`weak`
: A variable with this attribute has its name emitted as a weak symbol instead of as a global name. This is primarily for the naming of library variables that can be overridden by user code.

6.3 Specifying Attributes of Types

The keyword __attribute__ allows you to specify special attributes of **struct** and **union** types when you define such types. This keyword is followed by an attribute specification inside double parentheses. Five attributes are currently defined for types:

- aligned
- deprecated
- packed
- transparent_union
- unused

Other attributes are defined for functions (see section 6.1 on page 27) and for variables (see section 6.2 on page 31).

You may also specify any one of these attributes with __ preceding and following its keyword. This allows you to use these attributes in header files without being concerned about a possible macro of the same name. For example, you may use __aligned__ instead of aligned.

You may specify the aligned and transparent_union attributes either in a **typedef** declaration or just past the closing curly brace of a complete **enum**, **struct** or **union** type and the packed attribute only past the closing brace of a definition.

You may also specify attributes between the enum, struct or union tag and the name of the type rather than after the closing brace.

`aligned`

A type declared with this attribute is aligned on a memory address that is an even multiple of the number specified for the alignment. For example, the declarations:

```
struct S { short f[3]; } __attribute__ ((aligned (8)));
typedef int more_aligned_int __attribute__ ((aligned (8)));
```

force the compiler to insure (as far as it can) that each variable whose type is **struct** S or more_aligned_int will be allocated and aligned 'at least' on a 8-byte boundary. The aligned attribute can only be used to increase the alignment, not reduce it.

Note that the alignment of any given **struct** or **union** type is required by the ISO C standard to be at least a perfect multiple of the lowest common multiple of the alignments of all of the members of the **struct** or **union** in question. This means that you 'can' effectively adjust the alignment of a **struct** or **union** type by attaching an aligned attribute to any one of the members of such a type, but the notation illustrated in the example above is a more obvious, intuitive, and readable way to request the compiler to adjust the alignment of an entire **struct** or **union** type.

As in the preceding example, you can explicitly specify the alignment (in bytes) that you wish the compiler to use for a given **struct** or **union** type. Alternatively, you can leave out the alignment factor and just ask the compiler to align a type to the maximum useful alignment for the target machine you are compiling for. For example, you could write:

```
struct S { short f[3]; } __attribute__ ((aligned));
```

Whenever you leave out the alignment factor in an aligned attribute specification, the compiler automatically sets the alignment for the type to the largest alignment which is ever used for any data type on the target machine you are compiling for. Doing this can often make copy operations more efficient, because the compiler can use whatever instructions copy the biggest chunks of memory when performing copies to or from the variables which have types that you have aligned this way.

The aligned attribute can only increase the alignment; but you can decrease it by specifying packed as well.

Note:

Note that the effectiveness of aligned attributes may be limited by inherent limitations in your linker.

`deprecated`
A type declared with this attribute cause a warning message to be issued each time the type is used in a declaration. The message includes location information for the type declaration. The deprecated attribute can also be used for functions and variables (see section 6.1 on page 27 and section 6.2 on page 31).

`packed`
This attribute, attached to an **enum**, **struct**, or **union** type definition, specified that the minimum required memory be used to represent the type. Specifying this attribute for **struct** and **union** types is equivalent to specifying the packed attribute on each of the structure or union members. The command-line option `-fshort-enums` is the same as using packed attribute on all **enum** declarations. Likewise is the option `-fpack-struct` for a **struct**. You may only specify this attribute after a closing curly brace on an **enum** definition, not in a **typedef** declaration, unless that declaration also contains the definition of the **enum**.

`transparent_union`
This attribute, attached to a **union** type definition, indicates that any function parameter having that **union** type causes calls to that function to be treated in a special way.

First, the argument corresponding to a transparent **union** type can be of any type in the union; no cast is required. Also, if the union contains a pointer type, the corresponding argument can be a null pointer constant or a void pointer expression; and if the union contains a void pointer type, the corresponding argument can be any pointer expression. If the union member type is a pointer, qualifiers like **const** on the referenced type must be respected, just as with normal pointer conversions.

Second, the argument is passed to the function using the calling conventions of first member of the transparent union, not the calling conventions of the union itself. All members of the union must have the same machine representation; this is necessary for this argument passing to work properly.

Transparent unions are designed for library functions that have multiple interfaces for compatibility reasons. For example, suppose the wait function must accept either a value of type **int** ∗ to comply with Posix, or a value of type **union** wait ∗ to comply with the 4.1BSD interface. If wait's parameter were **void** ∗, wait would accept both kinds of arguments, but it would also accept any other pointer type and this would make argument type checking less useful. Instead, <sys/wait.h> might define the interface as follows:

```
typedef union
  {
    int *__ip;
    union wait *__up;
  } wait_status_ptr_t __attribute__ ((__transparent_union__));

pid_t wait (wait_status_ptr_t);
```

This allows either **int** $*$ or **union** wait $*$ arguments to be passed, using the **int** $*$ calling convention. The program can call wait with arguments of either type:

```
int w1 () { int w; return wait (&w); }
int w2 () { union wait w; return wait (&w); }
```

With this interface, wait's implementation might look like this:

```
pid_t wait (wait_status_ptr_t p)
{
  return waitpid (-1, p.__ip, 0);
}
```

`unused` When attached to a type (including a **union** or a **struct**), this attribute means that variables of that type are meant to appear possibly unused. GCC will not produce a warning for any variables of that type, even if the variable appears to do nothing. This is often the case with lock or thread classes, which are usually defined and then not referenced, but contain constructors and destructors that have nontrivial bookkeeping functions.

6.4 TriCore Specific Attributes

6.4.1 Specifying Attributes of Functions

`asection` In programming embedded systems more control for locating functions and variables is needed. Therefore the attribute asection was introduced. The syntax of this attribute is:

```
__attribute__ ((asection("<name>", "a=<align>", "f=<flags>")))
```

The parameter <name> specifies the name of the section in which the object (variable, type or function) should be located. The alignment of the section, which is set by the parameter a=<alignment> must be a power of 2 value (eg 2^1, 2^2, 2^4 …). Additional flags may be set by the parameter f=<flags>. These flags for attributed sections are available:

a allocatable. (This is always set.)

x executable

p PCP section

Setting the alignment and the flags for the attributed section is optional. The default alignment depends on the data stored in the section and the default flags are a and w.

If code is to be allocated in a section defined by the attribute asection the x-flag has to be set.

Note:

That sections containing executable code must be word aligned and must have the x-flag set.

Restriction

In 1-byte aligned sections integers may reside on odd addresses. This will lead to a malfunction of the program since the compiler uses st.w and ld.w instructions to access integers. However st.w and ld.w instructions are only valid to access 16-bit aligned memory locations.

Example

In this example attribute asection is used to locate the **int** foo (**void**) function in the user defined section .fastram.

```
int foo (void) __attribute__ ((asection(".fastram", "a=4", "f=x")));
```

`interrupt` A function declared with the attribute interrupt uses a jump and link instruction instead of a call instruction. Such functions return with the instruction ji a11 instead of ret (see section 14.7 on page 86 for details).

`interrupt_handler` A function declared with the attribute interrupt_handler uses a jump instruction instead of a call instruction. Such functions return with the instruction rfe instead of ret (see section 14.7 on page 86 for details).

`longcall` With the function attribute longcall arbitrary function can be called by calli instead of call instruction. The code and the execution time is reduced.

6.4.2 Specifying Attributes of Variables

`asection` Besides the attribute section the attribute asection may be used to locate variables. This attribute has a more sophisticated interface to adjust the settings of the section the variable is to be located in. The syntax of the attribute asection is:

```
__attribute__ ((asection("<name>", "a=<align>", "f=<flags>")))
```

The parameter <name> specifies the name of the section in which the variable should be located. The alignment of the section, which is set by the parameter a=<alignment> must be a power of 2 value (e.g. 2^0, 2^1, 2^2, 2^4 ...). Additional flags may be set by the parameter f=<flags>. These flags for attributed sections are available:

a allocatable. (This is always set.)

b section flag for bits (for details see section 14.1 on page 73)

w writable

p PCP section

s using small addressing

z using absolute addressing

If data is to be put into small or absolute addressable memory regions the name of the section must start with .sdata. or .sbss. for small addressable memory regions or .zdata. or .zbss. for absolute addressable memory regions. This is the only way to tell the linker to put the data in the appropriate memory region.

Example

This example shows how to define the section of a variable with pragma section. The integer variable a is located within a 2-byte aligned section named .zdata.absolute. This section is allocatable, writable and absolutely addressable:

```
int a __attribute__ ((asection(".zdata.absolute","a=2","f=awz")));
```

`ptr64` Types or values with this attribute will be handled circular buffer pointer.

6.4.3 Specifying Attributes of Types

`alignedaccess` This type attribute ensures for 4 byte access within PRAM. The option `-maligned-access` is required. A small example illustrates the use of this attribute.

```
typedef int pram_int __attribute__((alignedaccess("4")));

pram_int *pi = 0x10000;
```

Here for an **int** type a typedef called e.g. pram_int is defined. The typedef for PRAM access is specified by a type attribute alignedaccess. The argument "4" is used to indicate the 4 byte access. The shown method constricts the 4 byte access to the declared typedef pram_int. For other types than **int** use an equivalent **typedef**.

6.5 Pragma section

Another way for setting the section of an object is to use the pragma section. By using pragma section it is possible to easily locate more than one object into a user defined section by setting the pragma section for a whole group of objects. This group must be embraced by the pragma section directive to set the section and the attributes for the section properly:

```
#pragma section <name> [<flags>] [<alignment>]

<objects>

#pragma section
```

<name> specifies the name of the section the objects are to be put in. For the pragma section these flags may be specified:

The parameter <name> specifies the name of the section in which the object (variable, type or function) should be located. The alignment of the section, which is set by the parameter a=<alignment> must be a power of 2 value (eg 2^1, 2^2, 2^4 ...). Additional flags may be set by the parameter f=<flags>. These flags for pragma section are available:

a allocatable. This is always set

b section flag for bits (for details see section 14.1 on page 73)

w writable

x executable

p PCP section

s using small addressing

z using absolute addressing

The alignment of the section defined by the pragma section may be provided as a parameter. It must be a power of 2 value (e.g. 1, 2, 4, 8 ...). Setting flags or the alignment of a pragma section is optional. The default alignment depends on the defined data, the default flags are a and w.

To allocate code in a section defined by the pragma section directive the x-flag has to be set. Executable means that only code will be located in this section and all data within the pragma section will be located in the default data section. A pragma section without the x-flag is for allocating data.

If data is to be put into small or absolute addressable memory regions the name of the section must start with .sdata. or .sbss. for small addressable memory regions or .zdata. or .zbss. for absolute addressable memory regions. This is the only way to tell the linker to put the data in the appropriate memory region.

Note:
Code and data pragmas may be nested.

Examples

v3.4 In the both examples all code within **#pragma** section .text will be put in the section .text. The command **#pragma** section .data.mysection puts all global un-/initialized data in the section .data.mysection.

Example 1

```
#pragma section .text
#pragma section .data.mysection
int foo_data;
#pragma section

int
foo(void)
 {
  int i;
  ...
  return (foo_data + i);
 }
#pragma section
```

Example 2

```
#pragma section .data.mysection
#pragma section .text
int foo_data;

int
foo(void)
 {
  int i;
  ...
  return (foo_data + i);
 }
#pragma section
#pragma section
```

Restriction

In 1-byte aligned sections integers may reside on odd addresses. This will lead to a malfunction of the program since the compiler uses st.w and ld.w instructions to access integers. However st.w and ld.w instructions are only valid to access 16-bit aligned memory locations.

Example 3

If you use global variables that are also accessed within interrupts, the atomar load modify and store of these variables will be essential. To solve this problem the user should define these variables in an absolute addressable area with 4 byte alignment. In the following example a bitfield Bits is allocated via the #prama section .zdata. Corresponding to the TriCore EABI the alignment of a bitfield depends on its size. You get a 4 byte alignment for bitfield if the size is 4 byte.

```
volatile struct {
   unsigned int    bit0       :1;
   unsigned int    bit1      : 1;
   unsigned int    bit2      : 1;
   unsigned int    field    : 3;
   unsigned int     dummy    :16;
} Bits;

#pragma section .zdata
T_Bits  Bits;
#pragma section
```

Examples

This example shows how to define the section of a variable with pragma section. The integer variable a is located within a 2-byte aligned section named .zdata.absolute. This section is allocatable, writable and absolutely addressable:

Please note the leading dot in the name of the section.

```
#pragma section .zdata.absolute awz 2 //opening pragma section

int a;

#pragma section //closing pragma section
```

In this example the pragma section is used to locate the function **int** foo (**void**) in the user defined section .fastram.

```
#pragma section .fastram x

int foo (void);

#pragma section
```

In this example the functions **int** foo (**void**) and **int** bar(**void**) are located in the section .code. Since this pragma section has the x-flag set, it is ignored for all objects except functions. So the variable a is allocated as common symbol.

```
#pragma section .fastram x

int a;
int foo (void);
int bar (void);

#pragma section
```

In this example a function declaration is embraced by a pragma section without the x-flag set. This pragma section has only effect for variables. The variables a and b are located in the section .zdata.absolute. The pragma section is ignored for the function **int** foo(**void**), which is located in the default section .text.

```
#pragma section .zdata.absolute awz

int a;
int b;
int foo (void);

#pragma section
```

The compiler supports pragma for static variables.

```
#pragma section .zdata.foo awz
static unsigned short s1;
#pragma section
```

Input section `.zdata.foo` will be located by the default linker script in the output section `.zdata`. To get the symbol information about `s1` use one of the following commands. `tricore-objdump` displays the start address, the output section, the size and the symbol name.

```
tricore-objdump --syms test.o | grep s1
```

```
tricore-nm -f sysv test.o | grep s1
```

The entry `d` marks the variable `s1` as static variable. Global symbols are marked by `D`.

Note:

Keep in mind that static variables are not displayed in the Mapfile of the linker.

7 Inline Functions

At the reference point of a function normally a instruction call <name of function> is generated by the compiler. The programm counter is moved to the address of the referred function and the programm execution continues at this address. At the end of a function it will return to the reference point.

For tricore-gcc a function can be declared inline, and its code will be expanded much like a macro at the point it is called. This makes execution faster by eliminating the function-call overhead; in addition, if any of the actual argument values are constant, their known values may permit simplifications at compile time so that not all of the inline function's code needs to be included. The effect on code size is less predictable; object code may be larger or smaller with function inlining, depending on the particular case. Inlining code will be expanded much like a macro at the point it is called, therefore normally it is used only for small functions. Inlining of functions is an optimization and it really "works" only in optimizing compilation.

Note:

No functions are actually expanded inline unless you use `-O` option to specify some level optimization. A function can be forced to be expanded inline by assigning it the always_inline attribute. If this attribute is set for a function it will always be inlined, regardless of its size and the optimization level. The function prototype and the code of a inline function must be defined in the same source file before it is referenced.

For using an inline function in several source files you can define the inline function in a header file.

Inline functions are included in the ISO C99 standard, but there are currently substantial differences between what GCC implements and what the ISO C99 standard requires.

If you are writing a header file to be included in ISO C programs, use the keyword inline for inline functions.

You may use the keyword **extern** together with the declarations of inline-functions. In this case, the compiler will always inline this function ignoring any biases you have set using the option `-finline-limit` and will never create a callable instance of the function. This is necessary because the compiler can not rely on the implementation of the extern inline-function in another module. By inlining all of these functions linker errors are prevented.

You may also use the keyword **static** together with the declaration of inline-functions.

```
static inline void foo(void)
{
  /* some code */
}
```

In this case a callable instance of the function is created by the compiler if the function is called via a function pointer or the size of the function exceedes the maximum size of an inline function. If the function is too large to be inlined it will always called using the instruction call. If it is small enough to be inlined, the function will be inlined everywhere it is called. Everywhere a function pointer is used, the function will be called using the instruction call.

If you do not use **extern** or **static**, the compiler will always create a callable instance of the function. The decision whether the function is inlined or called using the instruction call is made based on the size of the function. Even if the function is inlined the compiler will output a callable instance of the function because this function may be called from other modules.

```
extern inline void foo(void)
{
  /* some code */
}
```

Options

`-O` This option enables optimization, so the a function declared as inline is expanded like a macro. The following options are only valid if the `-O` is specified.

`-fargument-noalias-global`
Specifies that arguments do not alias each other and do not alias global storage.

Each language will automatically use whatever option is required by the language standard. You should not need to use these options yourself.

`-fno-inline` The keyword inline is ignored by the compiler.

`-finline-limit=<number>`
Limits the size of inlined functions to <number>.

`-fkeep-inline-functions`
If an inline function is not declared as static, its body must be also generated by the compiler because it could be called from another module. Declaring a function as both inline and static will cause all occurrences of the function to be expanded inline, and the code for function is never generated. The option `-fkeep-inline-functions` of `tricore-gcc` will override this behavior and cause the function body to always be created.

`-finline-functions`
This option can be used to instruct the compiler to automatically select functions that are appropriate for being expanded inline, even if the function is not declared as inline. These functions are selected if their size does not exceed the limit set by `-finline-limit=<number>`.

`-Winline` This command-line option will issue a warning when a function declared as inline cannot be expanded inline.

To declare a function inline, use the `inline` keyword in its declaration, like this:

Example

```
int a,b,c;

inline int add1 (int p, int q) {

    return (p + q);

}

int main (void) {

    a = add1 (b, c);

    return 0;
}
```

Without using the keyword `inline` for the function `add` the compiler generates the following assembly code using `-O`:

```
add1:
    mov %d15, %d4
    mov %d0, %d5
    add %d1, %d15, %d0
    mov %d2, %d1
    ret

main:
    movh.a  %a15, HI:b
    movh.a  %a2, HI:c
    ld.w    %d4, [%a15] LO:b
    ld.w    %d5, [%a2] LO:c
    call    add1
    mov     %d15, %d2
    movh.a  %a15, HI:a
    st.w    [%a15] LO:a, %d15
```

With the option `-O` and the use of keyword `inline` the assembly language code results in:

```
main:
    movh.a  %a15, HI:b
    ld.w    %d0, [%a15] LO:b
    movh.a  %a15, HI:c
    ld.w    %d15, [%a15] LO:c
    add     %d0, %d0, %d15
    movh.a  %a15, HI:a
    st.w    [%a15] LO:a, %d0
```

8 Inline Assembly

By its very nature, there is nothing portable about assembly language. In most cases it does not make sense writing an entire program or module in assembly. The things that need to be done at the machine level can usually best be done by including a passage of assembly language inside the code of a higher level language. To this end, tricore-gcc provides the capability of inserting assembly language commands directly into a C function. In an assembler instruction using keyword asm, you can specify the operands of the instruction using C expressions. This means you need not guess which registers or memory locations will contain the data you want to use. You must specify an assembler instruction template much like what appears in an assembler language, plus a constraint string for each operand.

```
asm("mov %1, %0": "=d" (result): "m" (source) );
```

The following is the syntax of the asm construct:

```
asm(<assembly language template>
    :<output operands>
    :<input operands>
    :<clobber list>);
```

If you want to prevent the compiler from trying to optimize your assembly language code, you can use the **volatile** keyword. Also, if you need to be POSIX compliant, you can use the keywords __asm__ and __volatile__ instead of asm and **volatile**.

Note:

Please note that the assembler optimise the assembler code, e.g. the add d0,d0,1 will be optimised in a 16bit instruction add d0,1. So the dissasembly of your object file may be different from your inline assembly code. To disable such optimisations you can specify different assembler options or directives.

You may not use the **volatile** or __volatile__ if you are using the asm-statement outside a function, for example to define a user-defined function prologue.

`source`	Is the C expression for the input operand.
`result`	Is the C expression for the output operand.
`=`	Indicates, that the operand is an output.
`m and d/a`	Are constraints and indicate which types of addressing mode gcc has to use in the operand.

Each input and output operand is described by a constraint string followed by a C expression in parentheses. Commas separate the operands within each group.

Options

`-fno-asm`	The keyword `asm` is ignored by the compiler. GCC does not invoke the assembler anymore.

8.1 The Assembly Language Template

The first entry in the `asm` statement is called assembly language template. The assembly language template consists of one or more statements of assembly language and is the actual code to be inserted inline. All the assembly language is single-quoted string, and each line of the code requires a terminator. The terminator can be semicolon or a new line. Also, tabs can be inserted to improve readability. A small example shows the correct syntax for statements:

```
asm("statement \n\
     statement \n\
     statement"
...
```

The backslash n control sequence is used for generating a new line. The second backslash says that the statement continues. The start and end of statement are in quotation marks.

For the assembler code use quotation marks. Each operand is described by an operand-constraint string followed by the C expression in parentheses. A colon separates the assembler template from the first output operand and another separates the last output operand from the first input, if any. Commas separate the operands within each group.

If there are no output operands but there are input operands, you must place two consecutive colons surrounding the place where the output operands would go. Usually the most convenient way to use these `asm` instructions is to encapsulate them in macros that look like:

```
#define setasmlabel(lbl) asm volatile (#lbl":")
```

In this case do not use quotation marks. Within an assembly language template you can use mnemonic opcodes, labels or comments. The opcodes can address immediate (constant) values, the contents of registers, and memory locations. The following is a summary of the syntax rules for addressing values:

- Labels are written as symbols and are followed by a colon, e.g. `main`:
- A register name begins with two percent strings, e.g. `%%d0`. The register names normally begin with %, and the `asm` template also requires a %. But please note: If the assembly language template does not use either input nor output operands the second % must not be set.
- A memory location is one of the input or output operands. Each of these is specified by a number according to the order of its declaration following the colons. The first output operand starts with `%0`. If there is another output operand, it will be `%1` and

so on. The numbers continue with the input operands %2. The memory locations may have register modifiers.

- To get access to a special function register (SFRs) use $ as prefix, e.g. $psw. The names of SFRs can be found in the TriCore architecture manual.
- Comments are inserted using # and are only valid for each line.
- To access the high or low word of a **long long** variable use the characters 'H' or 'L' between the percent sign and the number of the memory location:

```
long long c;
long a, b;

c = 42;
asm ("mov %0, %L1": "=d"(a): "d"(c));
asm ("mov %0, %H1": "=d"(b): "d"(c));
```

8.2 Input and Output Operands

The input and the output operands consists of a list of variable names that you wish to be able to reference in the assembly code. You can use any valid C expression to address memory. This means you need not guess which registers or memory locations will contain the data you want to use. You must specify an assembler instruction template much like what appears in an assembler language, plus an operand constraint string for each operand.

Constraints are rules for specifying the input and output variables as follows:

- The C expression, which results in an address in your program, is enclosed in parentheses.
- A variable can be constrained to be addressed in memory instead of being loaded into a register by using m constraint.
- You may use any number of input and output operands by separating them with commas.
- The output and input operands are numbered sequentially beginning with $0 and continuing through $n−1, where n is the total number of both input and output operands.
- The constraint characters can have constraint modifiers

8.2.1 Constraint characters

`tricore-as` knows the following constraint characters:

m memory label

n variable or operand is const

i immediate integer operand. This includes symbolic constants whose values will be known only at assembly time.

The following additional constraints are available for TriCore inline assembly:

a address register

A address register a15

d data register

D data register d15

I signed character constant

J unsigned character constant

K signed short constant

L constant with $value < 15\&\&value >= 0$

M constant with $value < 8\&\&value >= -8$

N constant with $value < 16\&\&value >= 0$

O operand is zero constant

P signed 10 bit constant

Q no sdata symbol and no bss + 16 Bit offset

S small addressable memory (sdata symbol)

Z absolute addressable memory

B absolute addressable bit

T valid CONST_INT for absolut addressing

b a 32-bit constant with only one bit set

c a 32-bit constant with only one bit zero (complement of 'b')

U a 4-bit constant positiv positive signed 4-bit constant

k a 32-bit constant with low 16-bit all zero (a const16 << 16)

When the constraints for the read-write operand (or the operand in which only some of the bits are to be changed) allows a register, you may, as an alternative, logically split its function into two separate operands, one input operand and one write-only output operand. The connection between them is expressed by constraints which say they need to be in the same location when the instruction executes. You can use the same C expression for both operands, or different expressions. For example, here we write the (fictitious) <combine> instruction with bar as its read-only source operand and foo as its read-write destination:

```
asm ("combine %2,%0" : "=d" (foo) : "0" (foo), "d" (bar));
```

Note:

The constraint <"0"> for operand 1 says that it must occupy the same location as operand 0. A number in constraint is allowed only in an input operand and it must refer to an output operand.

Only a number in the constraint can guarantee that one operand will be in the same place as another. The mere fact that foo is the value of both operands is not enough to guarantee that they will be in the same place in the generated assembler code. The following would not work reliably:

```
asm ("combine %2,%0" : "=d" (foo) : "d" (foo), "d" (bar));
```

Various optimizations or reloading could cause operands 0 and 1 to be in different registers; GCC knows no reason not to do so. For example, the compiler might find a copy of the value of foo in one register and use it for operand 1, but generate the output operand 0 in a different register (copying it afterward to foo's own address). Of course, since the register for operand 1 is not even mentioned in the assembler code, the result will not work, but GCC can't tell that.

It is possible to specify input and output operands using symbolic names which can be referenced within the assembler code. These names are specified inside square brackets preceding the constraint string, and can be referenced inside the assembler code using %[name] instead of a percentage sign followed by the operand number. Using named operands the above example could look like:

```
asm ("add %[angle],%[output]"
     : [output] "=d" (result)
     : [angle] "d" (angle));
```

Note:

The symbolic operand names have no relation whatsoever to other C identifiers. You may use any name you like, even those of existing C symbols, but must ensure that no two operands within the same assembler construct use the same symbolic name.

8.2.2 Constraint modifiers

The constraint characters can have constraint modifiers. These modifiers instruct the compiler how the registers for the operands are to be allocated.

These constraint modifiers are available:

= The operand is written in the inline assembly statement and is an output operand

+ The operand is read and written in the inline assembly statement, it is input and output operand. This constraint modifier is to be assigned only to operands placed in the list of the output operand of the asm-statement.

& The operand is an early clobber. This modifier is used for output operands. The operand is used in the inline assembly statement before the last input operand is evaluated. So this modifier is necessary, because the compiler sometimes uses the same registers for input and output operands. If an input and an output operand are using the same register, the content of the register may have an undefined value within the asm-statement.

Example

```
signed short *ptr;
signed int n;
signed long long temp;
signed int i;
signed short read;

asm volatile("ld.h %L1,[%4+]2 \n\
              jlt %3, 2, 3f \n\
              add %0, %3,-2 \n\
              mov.a %2, %0 \n\
           2: ld.h %0,[%4+]2 \n\
              add %L1, %L1, %0 \n\
              loop %2, 2b \n\
              dvinit %A1, %L1,%3 \n\
              dvstep %A1, %A1,%3 \n\
              dvstep %A1, %A1,%3 \n\
              dvstep %A1, %A1,%3 \n\
              dvstep %A1, %A1,%3 \n\
              dvadj %A1, %A1,%3 \n\
           3: \n\"
           : "=&d"(read),"=&d" (temp), "=&a"(i) : "d"(n), "a"(ptr) );
```

In this example the output operands must have the constraint modifier & set because the output operands %0 to %2 are used the first time before the input operands %3 and %4 are used the last time.

% The operand with the modifier and the following one are commutative. This means that the compiler may interchange the two operands if that is the cheapest way to make all operands fit the constraints. tricore-gcc can only handle one commutative pair in an asm, if you use more, the compiler may fail.

8.2.3 Register modifiers

The memory locations in the Assembly Language Template may have register modifiers. These modifiers are used to handle extended registers in the assembler code. Sometimes the contents of an extended register %e2 must be referenced by the name of the extended register %e1, sometimes as %d2 and %d3. To distinguish between those two kinds of reference, register modifiers may be added to memory locations which are used to reference **long long**-variables.

These register modifiers are available:

A References to the alternative name of the register, e.g. %e2

L References to the lower 32-Bit-register of the extended register, e.g. %d2 for the extended register %e2

H References to the higher 32-Bit-register of the extended register, e.g. %d3 for the extended register %e2

For a detailed example please see the example for the constraint modifier & in subsection 8.2.2 on page 54.

8.3 Clobber List

The list of registers that are clobbered by your code is simply a list of the register names (a or d) separated by commas.

```
asm(<assembly language template>
    :<output operands>
    :<input operands>
    : "d0", "d1", "d2", "d3", "d4", "d5");
```

This information is passed on to the compiler so it will know not to expect any values to be retained in these registers.

If you want to clobber an extended data register the clobber list must not contain the extended register name e.g. %%e0, but must contain the corresponding data registers %%d0, %%d1 instead. This also applies for extended address registers.

Example

The identified number of leading zeros of variable val shall be stored in clz.

```
  int val, zeros;
  ...
  asm ("clz %0, %1": "=d"(zeros): "d"(val));
```

If your assembler instruction modifies memory in an unpredictable fashion, add <memory> to the list of clobbered registers. This will cause GCC to not keep memory values cached in registers across the assembler instruction. You will also want to add the **volatile** keyword if the memory affected is not listed in the inputs or outputs of the asm, as the <memory> clobber does not count as a side-effect of the asm.

Note:

The construct __asm__ __volatile__("": : : "memory"); generates a sequence point. The compiler will not schedule any instruction over this point.

If a program is written like

```
  <some instructions>
  __asm__ __volatile__( "": : : "memory");
  <some more instructions>
```

no assembler instructions generated from the instructions in <some instructions> will be placed after the sequence point.

9 Intrinsic Functions

Inline assembly is very powerful method of inserting a passage of assembly language inside the C code. Often used assembly instruction are collected in header file. The intrinsic functions distinguish from the assembly instruction in a leading __. Some of these intrinsics functions request a parameter.

The intrinsic functions are declared in the header file machine/intrinsics.h

`_bisr(int)`	Begin ISR. Save the lower context of the currently executing task, set the current CPU priority number to a value, and enable interrupts.
`_mfcr(int)`	Move from core register. Move the contents of the core SFR register to a data register.
`_mtcr(int, int)`	Move to core register. Move the value from a data register to the core SFR. **Note:** This instruction can be executed in supervisor mode only.
`_syscall(<tin>)`	Generates a system call, this means generating a trap of class 6 with the trap identification number <tin>. The parameter <tin> must be a literal from 0 to 255, otherwise the assembler will issue a error message. For the declaration of a trap service routine see chapter 11 on page 61.
`_enable()`	Global interrupt enable.
`_disable()`	Disables global interrupt.
`_debug()`	If the debug mode is enabled, cause a debug event; otherwise, execute a NOP.
`_dsync()`	All data access is served before the next data access.
`_isync()`	All preceding instructions are executed by the CPU. Then the pipeline is flushed before the next instruction is executed.
`_rstv()`	Reset overflow status flags/bits in PSW.
`_rslcx()`	Restore lower context and load the contents of the memory location pointed to by the PCX field in PCXI into register A2-A7,D0-D7, and the PC. This operation in effect restores the register contents of a previously saved lower context.
`_svlcx()`	Save lower context. Store the contents of register A2-A7, D0-D7, and the current PC to the memory location pointed to by the PCX field in PCXI. This operation in effect saves the lower context of the currently executing task.

`_nop()`	no operation.

10 Interrupts

The TriCore architecture provides two handler vectors, one for interrupts and one for traps. Each peripheral or an external input can generate service requests. Also software interrupts are possible. With _enable() or _disable() global interrupt is enabled or disabled (see enable irq on page 57). All service requests are assigned priority numbers (IPN).

To install an interrupt service routine the function _install_int_handler must be used.

```
int _install_int_handler (int intno, void (*handler) (int), int arg)
```

The interrupt service routine is assigned to the ISR priority <intno>. The ISR is specified with <handler>. If the installation of a interrupt handler is successful the return value of the function will be different from zero. In case that the return value is zero the installation has failed. A reason can be a illegal priority. If an interrupt occurs the interrupt service routine with the argument arg is served.

Note:

The argument <handler> in _install_int_handler is a pointer to the ISR in the vector table and does not automatically enable the appropriate interrupt. _install_int_handler is defined in machine/cint.h.

Example

In our example **void** rxint (**void**) will be an interrupt service request of the asynchronous serial interface. The first argument of _install_int_handler is the priority number of the interrupt service request.

Note:

The argument allows to use the same code for two different interrupts.

Here the argument is 0, that means uart0 is used. With setting the argument to 1 you get access to uart1 interrupt. The intrinsic function _enable() is necessary for enabling global interrupt.

```
#include <machine/cint.h>
#include <machine/intrinsics.h>
#include <tc1796Regs.h>

void rxint (void) {
    //do something...
}

void init (void) {
    ...
    ASC0_RSRC = 0x00001002;
    _install_int_handler(2, (void (*) (int)) rxint, 0);
    _enable();
    ...
}
```

With **void** *_install_chained_int_handler(**int** intno, **void** (*handler) (**int**), **int** arg) another possibility is given to install a handler. With this function several interrupts of the same priority can be combined and use the same interrupt vector. Within the interrupt service routine the user code detects the appropriate service request.

11 Traps

A trap occurs as a result of an event such as a non-maskable interrupt, an instruction exception, a _syscall (), or illegal access. Traps are always active; they cannot be disabled.

The available traps are divided into 8 classes. The trap class is used to index into the trap vector table. Each trap is assigned a Trap Identification Number (TIN), that identifies the cause of the trap within its class. Each class is assigned a Trap Service Routine (TSR), which is served as soon as the trap of this class occurs. The TIN is identified before the first instruction of the trap handler is executed. A TSR requires a parameter of type **int** and a return value **void**.

Here a small example of trap service routine:

```
void class4_tsr (int tin)
{
  switch (tin)
    {
    case 1:
      // serve trap with TIN 1
      break;

    case 2:
      // serve trap with TIN 2
      break;

    case 3:
      // serve trap with TIN 3
      break;
    }
}
```

The function _install_trap_handler installs a trap handler, this means a pointer to TSR, for a trap class. The syntax is:

```
int _install_trap_handler (int trapno, void (*handler) (int))}
```

The trap service routine is assigned to the trap class <trapno>. The TSR is specified with <handler>. If the installation of a trap handler is successful the return value of the function will be unequal zero. In case that the return value is zero the installation has failed. A reason can be an illegal trap class number. When no TSR is specified the default trap service routine is assigned. The default trap routine only executes the _debug() instruction.

Note:

This function is located in the header file machine/cint.h.

Example

A small example illustrates the installation of trap service routine class4_tsr for a trap class. The class4_tsr returns the TIN. In install_trap_handler the first parameter specifies the trap class 4, and **void** (∗) (**int**) class4_tsr is the name of the TSR.

```
#include <machine/cint.h>

void class4_tsr (int tin) {

  switch (tin) {

    case 1:
      // serve trap with TIN 1
      break;

    case 2:
      // serve trap with TIN 2
      break;

    case 3:
      // serve trap with TIN 3
      break;
  }
}

void init (void) {
  ...
  install_trap_handler (4, (void (*) (int) class4_tsr);
  ...
}
```

12 Errata

The following table shows the mapping between the old and the new naming convention of TriCore erratas. The supported erratas are implemented in the compiler and/or assembler. For some erratas there is no software workaround available, so the user has to take care of the known erratas.

Erratum	**Old name**	**cc1**	**as**	Comment
CPU_TC.004	**CPU.4 (TC V1.2)**			No Workaround available
CPU_TC.008				Must be solved in software
CPU_TC.012				PACK is not generated by the compiler, the assembler is not able to create a WA
–	**CPU.9 (TC V1.3)**	**X**	**X**	Not relevant for TC1796/TC1766
–	**CPU.13 (TC V1.3)**	**X**		Not relevant for TC1796/TC1766
CPU_TC.014	**CPU.14 (TC V1.2)**			No Workaround available
CPU_TC.046	**CPU.19 (TC V1.2)**			Must be solved in software
CPU_TC.048	**CPU.16 (TC V1.3)**	**X**	**X**	
CPU_TC.053				Workaround in debugger
CPU_TC.059				Must be solved in software
CPU_TC.060		**X**	**X**	
CPU_TC.061				No Workaround available
CPU_TC.062				Must be solved in software
CPU_TC.063				No Workaround available
CPU_TC.064				No Workaround available
CPU_TC.067				No Workaround available, because no information on how context was saved
CPU_TC.068				Must be solved in software"
CPU_TC.069		**X**	**X**	
CPU_TC.070		**X**	**X**	
CPU_TC.072		**X**	**X**	
CPU_TC.073				No Workaround available
CPU_TC.074				opcode can be changed in as
CPU_TC.075				No Workaround available
CPU_TC.076		**X**		
CPU_TC.078				Must be solved in software. The compiler does not generate code the affected instructions.
CPU_TC.079				No Workaround available
CPU_TC.080				Must be solved in software
CPU_TC.081		**X**	**X**	
CPU_TC.082		**X**	**X**	
CPU_TC.083		**X**	**X**	

CPU_TC.084				Workaround in debugger
CPU_TC.086				Must be solved in software
CPU_TC.087				Must be solved in software
CPU_TC.088				No Workaround available
CPU_TC.089				Workaround in debugger
CPU_TC.094		**X**	**X**	
CPU_TC.095		**X**	**X**	
CPU_TC.096		**X**		
CPU_TC.097				Instruction is not generated by the compiler
CPU_TC.098				The compiler does not use PSW or its status
CPU_TC.099				The compiler does not use PSW or its status
CPU_TC.100				The erratum affects the MAC-Unit. The compiler does not use this unit.
CPU_TC.101		**X**		A workaround in the assembler is not possible, as an additional address register is necessary. The workaround is implemented in the corresponding compiler built-in.
CPU_TC.102				Instruction is not generated by the compiler
CPU_TC.104				The compiler ensures the correct alignment of variables. If a wrong alignment is used (e.g. inline assembly), the assembler will abort.
CPU_TC.105				PSW.IO is not used within the compiler. A workaround must be included in e.g. OS implementation that uses the MTCR instruction to switch between supervisor and user-modes.
CPU_TC.107				SYSCON.FCDSF is not used within the compiler. A workaround must be included in the application (e.g. trap handler).
CPU_TC.108, CPU_TC.109				Is not relevant for the compiler, since the code generation does not use it to access circular buffers. The user has to ensure the correct alignment of such buffers.
CPU_TC.112				The value PC is not relevant for the compiler

CPU_TC.114		**X**		CAE Trap may be generated by UPDFL instruction

Table 12.1: Erratas

Note:

The workarounds of the errata 100-102 are included in built-in the compiler.

13 Startup Process

HighTec has set some defaults for the startup behavior of your microcontroller. In this startup process the target is initialized and set to its defaults. The code of the startup files is executed before entering `main()`. Afterwards the constructors for `main()` are executed.

> **Note:**
>
> The startup code shipped with the compiler is only an example. To fit your needs you can modify the startup code.

The default startup process is done in two steps:

1. The startup code of `crt0.o` is executed
2. The constructors for `main()` are called

13.1 Startup code in crt0.o

The compiler suite contains an example of a startup code. This code is executed after a target reset and has to be placed at the address where the target expects its first instruction (the entry point). The default code initializes:

- The user and interrupt stack pointers
- The access to system global register
- Software breakpoint service for OCDS
- SDA base pointers (see section 14.2 on page 74)
- Initialize context save areas (CSAs)

Furthermore uninitialized data is set to zero and initialized data is copied from ROM to RAM. This is done by using the tables `__clear_table` and `__copy_table` defined in the default linker description file.

`__clear_table` is used to set the uninitialized data, that is the sections `.bss`, `.sbss` and `.zbss`, to zero before `main()` is entered. This is done because ANSI defines uninitialized data to be zero.

With `-fzero-initialized-in-bss` you can advise `tricore-gcc` to put variables that are initialized to zero in the `.bss` section instead of the `.data` section. Variables that use the modifier **static** are not affected by this option.

Initialized data is located in a RAM area on reset. This data has to be copied to its RAM address before the program gets access to it.

In the default linker script the `__copy_table` is initialized. By means of this table, the startup code can copy the contents of the `.data*` sections from their load memory address (LMA) in the ROM to their virtual memory address (VMA) in the RAM.

VMA Virtual memory address used internally by the module when it is run.

LMA Loadable memory address specifying where the section is to be loaded.

The source code of the default startup code is located in the file `crt0.S`. The corresponding object, `crt0.o`, is linked to the executable whenever an executable is generated by `tricore-gcc`. If you want to create your own startup file, you have to remove this default by using the compiler option `-nostartfiles`. Then you can include and link your own `crt0.o` to your project.

By default the linker adds the start files `crti.o`, `crtbegin.o`, `crt0.o` and the end files `crtend.o` and `crtn.o`. The option `-nocrt0` avoids that the default `crt0.o` are included, so the user can add his own startup code by using this option. With the option `-nostartfiles`, the startfiles and endfiles are skipped. The option `-nocrt0` is not documented in the help screen of the compiler.

The input sections `.init` and `.fini` are located in the output section `.init`. The function `__main` is replace by the function `_init`.

> **Note:**
>
> The order of linking object is important, so parenthesise the user objects in this manner `crti.o` <user object> `crtn.o`.

`.init`	Contains the calls of all constructors.
`.fini`	Contains the calls of all destructors.
`.jcr`	Contains java constructors. The section is located at the `.rodata` section.

In the file `crti.o`, the symbol `_init` is defined in the `.init` section, and the symbol `_fini` is defined in the `.fini` section. The file `crtn.o` contains the definition of the function epilog with the return instruction of the `.init` and `.fini` section. The symbol `_init` is called as a function before main from the startup file `crt0.o`. The section `.fini` will be generated to hold the code which has to be executed before `_exit`.

13.2 The constructors for main()

You may set a function as constructor via the attribute `constructor`. These constructors are called in the function `_init` in the section `.init`, which is called in `main()` before any other instruction is executed.

The libraries shipped with the TriCore Development Platform contain two constructors, which are both located in separate objects in the library `libos.a`:

`_init_vectab`	This function initializes the trap vector table for interrupts and traps. It is only added to the executable if a function to handle interrupts

or traps (e.g. _install_int_handler() or _install_trap_handler()) from the object cint.o of the library libos.o is referenced. Otherwise this constructor is not called from within _init.

`_init_hnd_chain` This function initializes the chained interrupt handlers. It is only added to the executable if a function to handle chained interrupt handlers (e.g. _install_chained_int_handler or _remove_chained_int_handler from the object `ccint.o` of the library `libos.o` is referenced. Otherwise this constructor is not called from within _init.

If you have referenced one of these functions, you cannot avoid the constructors to be called by __init. The only way to avoid this is to define your own _init as an empty function. In this case you have to take care of the initialization of the trap vector tables and the chained interrupt handlers if necessary. You may do this by explicitly calling the constructor-functions from your own _init or main().

13.3 Your program as ROM version

If you want to locate your program in the ROM of the TriCore, the following files and options are required:

- Selection of derivative with `-mcpu=<derivative>`
- Linker script file for memory layout: `memoryROM.x`
- Startup Code `crt0.S`
- Code to initialise your board in `initboard.S`
- The additional linker options `-nocrt0 memoryROM.x`

13.3.1 Memory layout

In the ROM version, the RAM addresses for code and data are adapted to the memory layout of your derivative. Below you find an example for a memory layout `memoryROM.x` for a TC1796. The derivative is specified with the compiler option `-mcpu=tc1796`.

```
/* __TC1796__ __TC13__ with Core TC1.3 */
__TRICORE_DERIVATE_MEMORY_MAP__ = 0x1796;
/* the external RAM description */
__EXT_CODE_RAM_BEGIN = 0x84000000;
__EXT_CODE_RAM_SIZE = 4M ;
__EXT_DATA_RAM_BEGIN = 0xa1000000;
__EXT_DATA_RAM_SIZE = 1M;
__RAM_END = __EXT_DATA_RAM_BEGIN + __EXT_DATA_RAM_SIZE;
/* the internal ram description */
__INT_CODE_RAM_BEGIN = 0xd4000000;
__INT_CODE_RAM_SIZE = 48K;
__INT_DATA_RAM_BEGIN = 0xd0000000;
__INT_DATA_RAM_SIZE = 56K;
/* the pcp memory description */
__PCP_CODE_RAM_BEGIN = 0xf0060000;
__PCP_CODE_RAM_SIZE = 32K;
```

```
__PCP_DATA_RAM_BEGIN = 0xf0050000;
__PCP_DATA_RAM_SIZE = 16K;

MEMORY
{
  ext_cram (rx): org = 0x84000000, len = 4M
  ext_dram (w!x): org = 0xa1000000, len = 1M
  int_cram (rx): org = 0xd4000000, len = 48K
  int_dram (w!x): org = 0xd0000000, len = 56K
  pcp_data (w!x): org = 0xf0050000, len = 16K
  pcp_text (rx): org = 0xf0060000, len = 32K
}

/* the symbol __TRICORE_DERIVATE_NAME__ will be defined in the crt0.S and is
 * tested here to confirm that this memory map and the startup file will
 * fit together
*/
_. = ASSERT ((__TRICORE_DERIVATE_MEMORY_MAP__ ==
__TRICORE_DERIVATE_NAME__), "Using wrong Memory Map. This Map is for
TC1796");
```

This memory layout is added to the default linker script to configure your memory. Please note that you must not use the option `-T` but only add your `memoryROM.x` to the inputfiles of the linker. If you have your own complete linker script, including memory layout and section assignment, then you have to provide this file using `-T`.

13.3.2 Board initialisation

The board is initialised with the files `crt0.S` and `initboard.S`. These files must be compiled and linked to your project.

> **Note:**
> If you are using your own startup code, then the function _init has to be called in `crt0.S` before calling main.

In the compiler the builtin define __TOOL_VERSION__ exists. So you can modify your own startup code via the following sequence.

```
#if defined(__TOOL_VERSION__) && (__TOOL_VERSION__ >= 20)
  /*
   * call the initializer, constructors etc.
  */
  call  _init
#endif /* __TOOL_VERSION__ && __TOOL_VERSION__ >= 20 */
```

In the startup code `crt0.S` the function _init_board is called from initboard .S. Below you find the code of `initboard.S` for TC1796.

```
#; initboard.S
.text
  .global  __board_init
  .type __board_init,@function
__board_init:
  .code32

#;
#;  initialize target environment (PLLCLC, BUSCONx, ADDSELx)
#;
#;  this is done by board specific setup table (address/value - pairs)
#;

  movh.a  %a15,hi:boardSetupTabSize
  ld.a  %a15,[%a15]lo:boardSetupTabSize  # %a15 = table size
  jz.a  %a15,no_setup
  add.a  %a15,-1       # correction for loop
  movh.a  %a14,hi:boardSetupTab
  lea  %a14,[%a14]lo:boardSetupTab  # %a14 points to setup table

setup_loop:
  ld.a  %a2,[%a14+]      # %a2 = boardSetupTab.addr
  ld.w  %d2,[%a14+]      # %d2 = boardSetupTab.val
  st.w  [%a2],%d2
  loop  %a15,setup_loop

   isync
   nop
   nop

#;  force PC to remapped ROM address
  movh.a  %a14,hi:__remapped
  lea  %a14,[%a14]lo:__remapped
  nop
  ji  %a14

__remapped:

#;  remap RA to new address range
  mov.d  %d15,%a11
  mov.d  %d14,%a14
  movh  %d2,hi:0xFFF00000
  and  %d14,%d14,%d2      # take highest bits from new address
  andn  %d15,%d15,%d2      # and lower bits from old address
  or  %d15,%d15,%d14
  mov.a  %a11,%d15

  isync

no_setup:

  ji  %a11
```

13.3.3 Loading to Flash

Before loading the executable to the flash, it must be converted into a binary format via tricore-objcopy

```
tricore-objcopy -O binary name.elf name.bin
```

Download the file with a flashloader.

14 Special Features

14.1 Multiple Bit-Data Sections

By default all data of the _bit type is allocated to the .bdata section. The TriCore Development Platform supports an arbitrary number of bit data sections, which can be defined by user.

Since both the assembler and the linker can process _bit data, the user-defined sections must have either the prefix .bdata. for initialised bits or .bbss. for uninitialised bits.

The default section name for bit data is .bdata and .bbss.

To ensure correct handling of bit data sections by the compiler, the section flag b and z for absolute addressable section is required. If a _bit is defined without having a section assigned, the data will be allocated to the default sections .bdata or .bbss.

```
_bit b1;
_bit b2 = 0;
_bit b3 = 1;
```

In the first example b1 is allocated to the .bbss section. The initialised bit data b2 and b3 are allocated to the .bdata section.

```
#pragma section .bdata.bits awbz
_bit b4;
_bit b5 = 1;
#pragma section
```

The bit data b4 and b5 are allocated to the user-defined .bdata.bits section.

```
#pragma section .bbss.bits awbz
_bit b6;
_bit b7;
_bit b8 = 1;
#pragma section
```

b6 and b7 are allocated to the section .bbss.bits which contains uninitialised bit variables. b8 issues a warning, because initialised data should not be allocated to the .bbss.∗ section.

Another possibility is the usage of attributes.

```
_bit b9 __attribute__ ((section(".bdata.bits")));
_bit b10 __attribute__ ((section(".bbss.bits,\"awbz\",@progbits")));
_bit b11 __attribute__ ((section(".bdata.bits,\"awbz\",@progbits"))) = 1;
_bit b12 __attribute__ ((asection(".bdata.bits","a=1","f=awbz")));
```

b9 is allocated to the .bdata.bits section. b10 is allocated to the section .bbss.bits which contains uninitialised bit variables, in the same manner as b7. The two notations are equivalent.

The linker allocates all sections which contain one of the prefixes .bdata. or .bdata to the object file. In the default linker script, the .bdata. section is listed in the __copy_table table. During runtime, the corresponding memory section is copied.

All sections carrying one of the prefixes .bbss. or .bbss are declared as NOLOAD by the linker and do not occupy any space in the object file. In the default linker script the section .bbss. is listed in __clear_table. During runtime the corresponding memory section is cleared.

14.2 Relative Addressing

In addition to the output sections .sdata/.sbss, which use the address A0 for addressing, the output sections

- .sdata2/.sbss2
- .sdata3/.sbss3
- .sdata4/.sbss4

can be defined in the linker script file for small-data addressing (SDA). These sections are addressed as A1, A8, A9. Address register A1 is assigned to .sdata2/.sbss2, A8 to .sdata3/.sbss3 and A9 to .sdata4/.sbss4.

The small data sections can be located at an arbitrary location in the linker script file. Every small data section is limited to a size of 64 kB.

For every small data section declared in the linker script file, a symbol, that points to the basis address of the corresponding section, is generated by the linker. The symbols are named:

_SMALL_DATA_ →.sdata/.sbss
_SMALL_DATA2_→.sdata2/.sbss2
_SMALL_DATA3_→.sdata3/.sbss3
_SMALL_DATA4_→.sdata4/.sbss4

By default all these symbols are initialized with the startign address of the assigned section plus 2^{15}. However it is also possible to assign any value in the linker script (see the example). The startup code uses these symbols to load the respective address register.

By default A0 is always used for small data address register; A1, A8 and A9 are used only if the small data sections .sdata2/sbss2, .sdata3/sbss3 or .sdata4/sbss4 are defined in the linker script file and the sections are not empty.

Note:

The address register A0, A1, A8 and A9 are not overwritten by the compiler.

The small data sections must be named either .sdata or .sdata.<name> and .sbss or .sbss.<name>. If the content of user-defined data sections are to have small adresses, the sections must have the section flag ”t” (for 10 bit relative addressing) or ”s” (for 16 bit relative addressing) set and the name of the sections must have the prefix `.sdata.` or `.sbss.` (e.g. `.sdata.byte`). Please note: There is a dot between the prefix and the section name! The allocation of all small data section (default and user defined) to the output sections `.sdata`, `.sdata2`, `.sdata3`, `.sdata4`, or `.sbss`, `.sbss2`, `.sbss3`, `.sbss4` is defined in the linker invocation file.

To achieve an efficient relative addressing, all 8 bit and 16 bit data must be combined within the small data sections at compilation time. This means that in every module at least a small data section with 8 bit data and one with 16 bit data has to be defined. The distinction between 8 bit and 16 bit data is necessary in order to avoid memory gaps due to the alignment.

Examples:

Small data addressing (10 bit) with 1 byte alignment:

```
#pragma section .sdata.byte 1 awt
    char c;
#pragma section
```

Small data (10 bit) addressing with using attribute:

```
char c __attribute__((asection (".sdata.word", "a = 1", "f = awt")));
```

Data small addressing (16 bit) with 4 byte alignment:

```
#pragma section .sdata.word 4 aws
    int c;
#pragma section
```

Small data (16 bit) addressing with using attribute:

```
int c __attribute__((asection (".sdata.word", "a = 4", "f = aws")));
```

Note:

The 10 bit and 16 bit relative addressing is specified by the flag `t`. The 16 bit relative addressing is enabled with the flag `s`. The flag `S` is reserved for string sections.

In these sections all data is addressed relatively to the section basis register. For 8 and 16 bit data in particular, a 10 bit relative addressing will be generated.

Example

The following example assumes that the linker option `-Wl,-T,<linker-file-name>,-Map,map` is used. The <linker-file-name> must be replaced by the user-defined linker description file.

```
#include <stdio.h>

/* These variables will be put in SDA2, provided the linker script
   defines the .sdata2/.sbss2 output sections, and assigns the
   input sections .{sdata,sbss}.foo to them.  */
```

```
#pragma section .sdata.foo aws
int sdata2_int1 = 42;
#pragma section
#pragma section .sbss.foo as
int sdata2_int2;
#pragma section

/* These variables will be put in SDA0.  */
#pragma section .sdata
int sdata_int1 = 0xdeadbeef;
#pragma section
#pragma section .sbss
int sdata_int2;
#pragma section

int
main ()
{
  printf ("sdata2_int1 = %d (should be 42)\n", sdata2_int1);
  printf ("sdata2_int2 = %d (should be 0)\n", sdata2_int2);
  printf ("sdata_int1 = 0x%08x (should be 0xdeadbeef)\n", sdata_int1);
  printf ("sdata_int2 = %d (should be 0)\n", sdata_int2);
}
```

In the linker description file you can see the user-defined sections in __clear_table and __copy_table.

```
    PROVIDE(__clear_table = .) ;
    LONG(0 + ADDR(.bss));     LONG(SIZEOF(.bss));
    LONG(0 + ADDR(.sbss));    LONG(SIZEOF(.sbss));
    LONG(0 + ADDR(.sbss2));   LONG(SIZEOF(.sbss2));
    LONG(0 + ADDR(.zbss));    LONG(SIZEOF(.zbss));
    LONG(-1);                 LONG(-1);
    PROVIDE(__copy_table = .) ;
    LONG(LOADADDR(.data));    LONG(ABSOLUTE(DATA_BASE)); LONG(SIZEOF(.data));
    LONG(LOADADDR(.sdata));   LONG(ABSOLUTE(SDATA_BASE));LONG(SIZEOF(.sdata));
    LONG(LOADADDR(.sdata2));  LONG(ABSOLUTE(SDA2_BASE)); LONG(SIZEOF(.sdata2));
    LONG(LOADADDR(.pcpdata)); LONG(ABSOLUTE(PRAM_BASE)); LONG(SIZEOF(.pcpdata));
    LONG(LOADADDR(.pcptext)); LONG(ABSOLUTE(PCODE_BASE));LONG(SIZEOF(.pcptext));
    LONG(-1);                 LONG(-1);                  LONG(-1);
```

In the following excerpt from the linker file, you can see the allocation of the input and output sections. The input section .sdata.foo is transferred to the output section .sdata2.

```
 /* Below are the definitions for SDA2.  */
  .sdata2 : AT(LOADADDR(.data) + SIZEOF(.data))
  {
    SDA2_BASE = ABSOLUTE(.) ;
    . = ALIGN(8) ;
    *(.sdata.foo)
    . = ALIGN(8) ;
  } > ext_dram
  . = ALIGN(8) ;
  .sbss2 (NOLOAD) :
  {
    . = ALIGN(8) ;
    *(.sbss.foo)
    . = ALIGN(8) ;
  } > ext_dram

  . = ALIGN(8) ;
  .sdata : AT(LOADADDR(.sdata2) + SIZEOF(.sdata2))
  {
    . = ALIGN(8) ;
    SDATA_BASE = ABSOLUTE(.) ;
    PROVIDE(__sdata_start = .);
    *(.sdata)
    *(.sdata.*)
    *(.gnu.linkonce.s.*)
    . = ALIGN(8) ;
  } > ext_dram
  _edata = . ;
  PROVIDE(edata = _edata) ;
  . = ALIGN(8) ;
```

Note:

The .sdata2 and .sbss2 sections must be declared before the output section .sdata in the linker description file, otherwise the wildcard *.(. sdata .*) will put all the input section .sdata.* in the output section .sdata.

Another example for the use of 10 bit and 16 bit relative addressing.

```
#pragma section .sdata.s10bit 2  awt
char ch10 = 1;
short s10 = 2;
int  i10 = 3;
#pragma section

#pragma section .sdata.s16bit 4  aws
char ch16 = 4;
short s16 = 5;
int  i16 = 6;
#pragma section

void foo(void)
{
    ch10 = 10;
    s10 = 10;
    i10 = 10;

    ch16 = 16;
    s16 = 16;
    i16 = 16;

}
```

The linker description file requires the definition of the section sdata2 in __copy_table.

```
LONG(LOADADDR(.sdata2));  LONG(ABSOLUTE(SDA2_BASE)); LONG(SIZEOF(.sdata2));
```

The SMALL_DATA2_ symbol must be an absolute address, therefore the symbol is *not* defined within an output section. The user is responsible for setting the address in such a way that the data is 10 or 16 bit addressable.

```
   .sdata2 : AT(LOADADDR(.data) + SIZEOF(.data))
    {
      SDA2_BASE = ABSOLUTE(.) ;
      . = ALIGN(8) ;
      *(.sdata.s10bit)
      *(.sdata.s16bit)
    } > ext_dram
    . = ALIGN(8) ;
   _SMALL_DATA2_ = SDA2_BASE ;
    .sdata : AT(LOADADDR(.sdata2) + SIZEOF(.sdata2))
  {
    SDATA_BASE = ABSOLUTE(.) ;
    PROVIDE(__sdata_start = .);
    *(.sdata)
    *(.sdata.*)
    *(.gnu.linkonce.s.*)
  } > ext_dram
  . = ALIGN(8) ;
  _edata = . ;
  PROVIDE(edata = _edata) ;
```

If you want to use 10 bit (± 512) relative addressing without defining an absolute symbol, you can customise the linker script like that:

```
    .sdata : AT(LOADADDR(.data) + SIZEOF(.data))
  {
    SDATA_BASE = ABSOLUTE(.) ;
    PROVIDE(__sdata_start = .);
    *(.sdata)
    . = 32768 - 512 ;
    *(.sdata.s10bit)
    ...
  } > ext_dram
  . = ALIGN(8) ;
  _edata = . ;
  PROVIDE(edata = _edata) ;
```

Note:

This method is not recommended, since the user himself must take care of the size of `.sdata`. This means the address of `.sdata.s10bit` is not fixed, and therefore this method is error-prone.

If you use a `.sbss` section, you need a similar entry in the linker description file.

14.3 Storage of constants and variables using different addressing modes

14.3.1 Small Addressing Mode

The compiler option `-msmall=<size>` enables the small (register relative) addressing mode for variables and constants, and allocates them to a `.sdata.*` section. The boundary value for the small addressing mode can be set by the optional <size> paramater. The boundary value will then be <size> x Bytes. If you set the boundary value <size> to zero, all variables and constants will be addressed using the small addressing mode.

The user can separately select the small addressing mode for variables (RAM) and constants (ROM) by the compiler options `-msmall-const` and `-msmall-data`.

`-msmall-const=<size>`
: This option selects the small addressing mode for constants. Constants are marked by the type modifier **const**.

`-msmall-data=<size>`
: This option selects the small addressing mode for variables. Variables do not have the type modifier **const**.

> **Note:**
> If the `-msmall` and `-msmall-data` options and/or the `-msmall-const` option are set simultaneously, the compiler will issue an error.

14.3.2 Examples

```
/* Variables */
unsigned char c1;
unsigned short s1;
unsigned int i1;

/* Constants */
const unsigned char c2;
const unsigned short s2;
const unsigned int i2;
...
```

`-msmall=4` All variables and constants with a size $\leq$ 4 Byte will be accessed using the small addressing mode. In this example c1, s1, i1, c2, s2, i2 are accessed like this.

`-msmall-data=2` All variables with a size $\leq$ 2 Byte will use the small addressing mode. In this example c1, s1 are accessed like this.

`-msmall-const=1`
: All constants with a size of 1 Byte will use the small addressing mode. In this example c2 are accessed like this.

`-msmall-const=0`
: All constants will use the small addressing mode. In this example c2, s2, i2 are accessed like this.

14.3.3 Definition and Declaration

```
/* Module 1 */
int a;
```

```
/* Module 2 */
extern const int a;
```

In the first module you can find the definition of the variable `a`. In the second module, `a` is declared as **extern** but with the modifier **const**. In this case the option `-msmall-const` will cause a linker error. In the first module the variable will be accessed using the normal addressing mode, because the modifier **const** is missing, so the variable will not be allocated to a `.sdata.*` section. In the second module, the constant `a` will be accessed using the small addressing mode. This contradiction will cause the linker error because the variable `a` is not in the small addressable section with the prefix `.sdata.*`. See section 14.2 on page 74 for details.

The following case will not result in a linker error. In the first module, the constant `a` is defined.

```
/* Module 1 */
int const a;
```

In the second module, the variable `a` is declared by **extern**.

```
/* Module 2 */
extern int a;
```

With `-msmall-const=4` the constant `a` in the first module will be accessed using the small addressing mode and allocated to a `.sdata.*` section. The second module will not be accessed using the small addressing mode because the modifier **const** is missing. A small addressing area `.sdata.*` can always be accessed using the normal addressing mode.

14.3.4 Absolute Addressing Mode

The compiler option `-mabs=<size>` enables the absolute (register relative) addressing mode for variables and constants and allocates them to a `.data.*` section. With the optional <size> parameter, the boundary value for absolute addressing mode is set. The boundary value will be <size> x Bytes. If you set the boundary value <size> to zero, all variables and constants will be addressed using the absolute addressing mode.

The user can separately select the absolute addressing mode for variables (RAM) and constants (ROM) by the compiler options `-mabs-const` and `-mabs-data`.

`-mabs-const=<size>`
: This option selects the absolute addressing mode for constants. Constants are marked by the type modifier **const**.

`-mabs-data=<size>`
: This option selects the absolute addressing mode for variables. Variables do not have the type modifier **const**.

Note:
If the `-mabs` and `-mabs-data` options and/or the `-mabs-const` option are set simultaneously, the compiler will issue an error.

14.4 Storage of constants in a read only section

In C, **const** should not be modified during runtime. To save RAM, constants should be stored in the Flash and not be copied to the RAM by the startup code. Constants, therefore, must not be stored in the sections for variables.

- By default, constants are stored in the `.rodata` section.
- Apart from this, they can also be stored in the `.sdata.rodata` section for relative addressing mode (`-msmall`) or in the `.zrodata` section for absolute addressable constants (`-mabs`).
- Constants will never be stored in sections that contain the attribute 'w'.
- Uninitialised constants are assumed to be zero.
- Write access to a constant will issue a compiler warning.
- Per default 32 bit addressing is used for constants.
- The settings can be changed by:
  - Compiler options `-muninit-const-in-rodata`, `-mno-uninit-const-in-rodata`, `-mvolatile-const-in-rodata` and `-mno-volatile-in-rodata`.
  - Using `__attribute__` for changing the addressing mode or the section.
  - Using **#pragma** for changing the addressing mode or the section.

Note:

The above properties are also valid for constants that use **volatile** as qualifier.

The options `-muninit-const-in-rodata` and `-mvolatile-const-in-rodata` are set by default.

The following table illustrates the effect of using these options.

	initialized const		uninitialized const	
	not volatile	volatile	not volatile	volatile
`-muninit-const-in-rodata` `-mvolatile-const-in-rodata`	.rodata	.rodata	.rodata	.rodata
`-mno-uninit-const-in-rodata` `-mvolatile-const-in-rodata`	.rodata	.rodata	.bss	.bss
`-muninit-const-in-rodata` `-mno-volatile-const-in-rodata`	.rodata	.data	.rodata	.data
`-mno-uninit-const-in-rodata` `-mno-volatile-const-in-rodata`	.rodata	.data	.bss	.bss

For constants which are to be adressed as absolute, the letter 'z' must be added (e.g. `.bss` → `.zbbs` etc.).

For constants which are to be adressed as small, the letter 's' must be added (e.g. `.bss` → `.sbss` etc.). The `.srodata` section is renamed to `.sdata.rodata`.

14.5 Modifying the Default Sections

14.5.1 Divided Default Sections

To avoid alignment gaps between modules, the default sections are divided depending on the alignment.

The Figure 14.1 on page 82 illustrates the behaviour of data alignment without and with the option `-maligned-data-sections`.

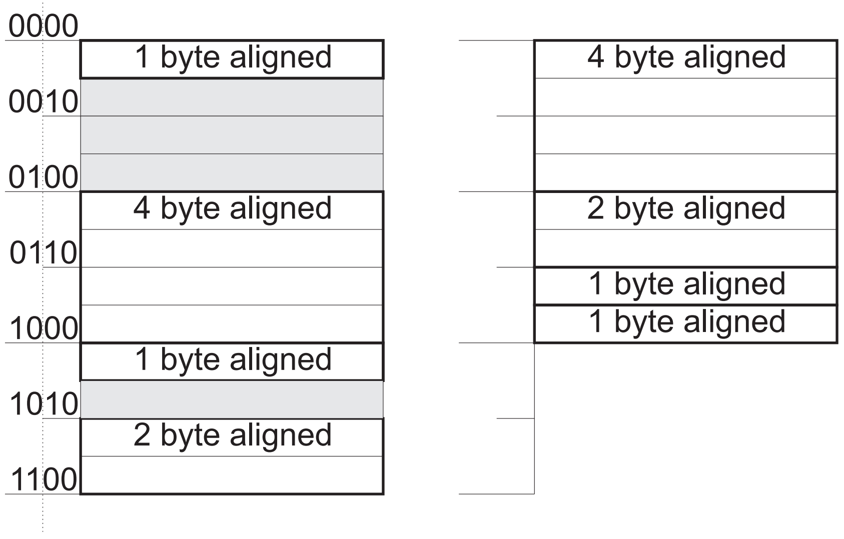

Figure 14.1: Comparison of aligned data

The alignment of data is considered for its addressing. Therefore an address of 4-byte of aligned data is a multiple of ”4” (0000b, 0100b, 1000b, 1100b ...). On the left hand side of the Figure 14.1 on page 82 1-byte of aligned data is followed by 4-byte of aligned data. The result is a gap of 3 bytes. For 2-byte of aligned data it is a gap of 1 byte.

By means of `-maligned-data-sections`, data can be sorted without having gaps. A multiple of 1-byte alignment is a 2-byte alignment. A multiple of 2-byte alignment is a 4-byte alignment. If data are sorted in descending order depending on their alignment, gaps will be avoided. This can be seen on the right hand side of Figure 14.1 on page 82.

The alignment of uninitialised variables in `.bss` section.

`.bss.a1`	uninitialised 1-byte variable
`.bss.a2`	uninitialised 2-byte variable
`.bss.a4`	uninitialised 4-byte variable
`.bss.a8`	uninitialised variable $\geq$ 8-byte

The alignment of initialised variables in `.data` section.

`.data.a1`	initialised 1-byte variable
`.data.a2`	initialised 2-byte variable

`.data.a4`	initialised 4-byte variable
`.data.a8`	initialised variable ≥ 8-byte

The alignment of constants in `.rodata` section.

`.rodata.a1`	1-byte constants
`.rodata.a2`	2-byte constants
`.rodata.a4`	4-byte constants
`.rodata.a8`	constants ≥ 8 byte

Note:

The described mechanism is also valid for the default sections `.zbss`, `.zdata`, `.zrodata`, `.sbss`, `.sdata` and `.sdata.rodata`. These sections are used in absolute (`-mabs`) and relative (`-msmall`) addressing mode.

This mechanism is not used for the following sections:

`.text`	code section
`.bbss`	section for uninitialised bit variables
`.bdata`	section for initialised bit variables

14.6 Circular Addressing

The TriCore Architecture supports the circular addressing mode. This mode implements a ring buffer structure in the hardware. The buffer has a predefined amount of items which can be accessed in a circular way: Each time a value is read from the buffer, the pointer to the item which is to be read on the next access, is incremented. If the last item is read, the pointer to this buffer is set to the first field. By using this technique it is possible to access a field of values in turn by using only one pointer. This is extremely useful in DSP applications such as digital filters.

Each ring buffer is defined by a control structure held by an instance of the type `circ_t`. This instance holds status information for each ring buffer. It defines the base address, the current index and the length of the ring buffer. By using this status information, the microcontroller calculates the effective address by adding the base address and the current index.

```
typedef struct {
    void *buf;          /* the base address     */
    short index;        /* the index            */
    short length;       /* the length in bytes  */
} circ_t __attribute__((__ptr64__));
```

Note:

The status information should not be written by the program.

This structure is defined in `machine\circ.h`. This headerfile also defines macros for initialising the ring buffer in order to put items to it or get items from it. The following functions are available:

`opPd(<type>,<gettype>,<puttype>,<size>,[A])`

Wherein <type> is the type of the items which are to be stored in the ring buffer, <gettype> and <puttype> are the TriCore types for the get and put instructions (e.g. w for **int**, h for **short** or b for **char**), size is the size of the ring buffer in bytes. If double names should be used inside the functions, the last parameter A may be added. This parameter is optional.

Depending on this macro the following inline functions are created for each type:

`circ_t init_circ_<type> (circ_t control, <type> *buffer, short length)`

Initialises the ring buffer at the location buffer with the control structure control and the length length for the storage of variables of the type type. This function returns the modified control structure.

`circ_t get_circ_<type> (circ_t control, <type> *ptrreturn);`

Reads the current value of the ring buffer defined by control and returns it in ptrreturn . This pointer must be of the same type as the items stored in the buffer. The function returns the modified control structure.

`circ_t put_circ_<type> (circ_t control, <type> value);`

Writes value to the current item of the ring buffer defined by control. value must be of the same type as the items stored in the buffer. The function returns the modified control structure.

Note:

The functions defined by the macro opPd() are inline functions. To expand them inline, either the option `-finline` or an optimisation level must be set. If this is not done, function calls are generated.

The functions for the most common used types are predefined in `machine\circ.h`. Since some of these types consist of two words, and a space is not allowed in a function call, these types are aliased by a **typedef** in `machine\circ.h`. Because of that it is normally not necessary to execute opPd() in user-defined programs. The functions for these types are available:

type	alias
long	long
int	int
short	short
char	char
unsigned long	ulong
unsigned int	uint
unsigned short	ushort
unsigned char	uchar
long long	llong
double	double
float	float

These predefined functions include init_circ_ulong, get_circ_short, put_circ_llong etc.

Example

This example shows how to create a buffer using circular addressing. The buffer buf contains 20 items which are controlled by ctrl. The buffer is initialised with the sequence 20, 19, 18, ... 2, 1. Afterwards the content of the buffer is read. Note that the variable i is never used to address the current item in the ring buffer; this is all done by the hardware.

```
#include "machine/circ.h"

#pragma section .zdata
long buf[20] __attribute__ ((aligned(8))) = { 1,2,3,4,5,6,7,8,9,0 };
#pragma section

main()
{

    circ_t ctrl;
    long ll;
    int i;

    ctrl = init_circ_long(ctrl, buf, 20*sizeof(long));

    //initialize the buffer
    for (i = 0; i < 20; i++)
        ctrl = put_circ_long(ctrl, 20-i);

    //read the buffer
    for (i = 0; i < 20; i++) {
        ctrl = get_circ_long(ctrl, &ll);
        //do something with ll
    }
    return 1;
}
```

14.7 CSA Overhead

For the TriCore architecture, a function call using the instruction call, an interrupt and a trap, uses a context save mechanism. This mechanism saves the upper context

- PSW (Processor Status Word)
- A10 to A15 (Address Register)
- D8 to D15 (Data Register)

in the Context Save Area (CSA). This is effected by the hardware within two instruction cycles. The upper context is restored by hardware after instructions like ret or rfe. In the instruction set of TriCore, alternative jump and link instruction like (jl, jla, jli) are available. These instructions need not save the context because the return address is stored in the address register a11.

If an interrupt occurs, the context will be saved. There is no need to save the context again while the corresponding interrupt routine is called in the interrupt table, if no register is modified before the call. If a jump or link instruction is used instead of call, an interrupt handler saves one context save and context restore process.

The function attributes interrupt and interrupt_handler are available (see subsection 6.4.1 on page 37 for function attributes).

The interrupt and interrupt_handler attributes are used for optimising the CSA usage. Normally, with every interrupt entry and every call, a CSA (context save area) of 64 bytes is allocated by the processor. The mentioned attributes are notes to the compiler to treat the marked functions in a different way from normal functions. This means in particular that a different function prologue and epilogue are to be used. With a function marked with the interrupt attribute, the compiler can assume that this function is called by a jump-and-link command, not by a call, i.e. no CSA was allocated when the function was called, and the function has to be quit, not by a ret but by an indirect jump via the a11 register.

The interrupt_handler attribute goes even further in its consequence. With functions of this type, the compiler assumes that they were called by a jump command, that the upper and lower context were saved beforehand, and that the functions must be quit by a defined interrupt return sequence which will restore the lower and upper context.

Thus, both attributes are used for optimising the interrupt behaviour, and make the usage of normal C functions as interrupt functions possible.

```
#include <machine/cint.h>
extern void lfoo(void);
extern void ifoo(int) __attribute__ ((interrupt));
extern void ihfoo(void) __attribute__ ((interrupt_handler));
extern void nfoo(void);

int lf, nf, iif, ihf;
unsigned int * IntSrc = (unsigned int *)0xf7e0fffc;
int main(void)
{
    _install_int_handler(3,ifoo,0);
    lfoo();
    nfoo();
    ifoo(1);
    *IntSrc = 0x1001;
    __asm__ volatile ("enable");
    *IntSrc |= 0x8000;
...
```

For the function ifoo() with attribute interrupt, a jump indirect instruction ji %a11 is used instead of a ret instruction. The generated assembler code looks like:

```
     call lfoo
    .loc 1 19 0
    call nfoo
    .loc 1 20 0
    mov %d4, 1
    jl ifoo
    ...
ifoo:
    ...
    movh.a %a15,HI:iif
    st.w [%a15] LO:iif, %d4
    ji %a11
```

A function declared with the attribute interrupt_handler uses a jump instruction instead of a call instruction. Such functions return with the instruction rfe instead of ret.

```
    .loc 1 54 0
    mov %d15, 1
    movh.a %a15,HI:ihf
    st.w [%a15] LO:ihf, %d15
    rslcx
    rfe
```

Without the attribute interrupt_handler the following code is generated.

```
    .loc 1 54 0
    mov %d15, 1
    movh.a %a15,HI:ihf
    st.w [%a15] LO:ihf, %d15
    ret
```

Example

The interrupt interface makes it possible to define standard C functions as interrupt functions. The third argument of the Function __install_int_handler is defined as the argument of the interrupt function to be called, and makes it possible to parameterise an interrupt function.

If you want to avoid the overhead resulting from this type of interrupt interface, you have the possibility to define the interrupt vector by yourself and to call the appropriate interrupt function directly. The abovementioned attributes interrupt and interrupt_handler are used for enabling an optimised usage of this method. You can, for example, define an interrupt function in the following way:

```
void interrupt1(void) __attribute__((interrupt_handler));

void interrupt1(void)
{
  ... some code..
}
```

Entry in the interrupt vector

```
__interrupt_1:
  bisr 1
  j  interrupt1
```

or

```
void interrupt1(void) __attribute__((interrupt));
void interrupt1(void)
{
  ... some code..
}
```

Entry in the interrupt vector

```
__interrupt_1:
  bisr 1
  jl  interrupt1
  ..addtional intructions
  rslcx


  rfe
```

Example

This additional example explains the __attribute__((interrupt)). This feature can be used to execute functions without a context save and therefore saving RAM ressources.

```
volatile unsigned int x_u32;

extern void abort(void);
extern void exit(int);

void __main(void);
void __main(void){ return; }


void csa_saving_func(void) __attribute__((interrupt));
void csa_saving_func(void)
{
    x_u32++;
}

void test (void);
void test (void)
{
    volatile unsigned int temp_a11;
    /* temp variable to save return address (register A11) on stack */
    /* volatile key word is required here */
    /* otherwise temp_a11 would be kept in a register which might be overridden. */

    /* save return address manually */
    __asm__ __volatile__("mov.d  %[temp_a11], %%a11"
      : [temp_a11]"=d"(temp_a11)
      :
      : "memory");

    csa_saving_func();
    /* with the JL command the content of A11 is modified and never restored */
    /* RET command at the end of this function would return to the wrong address */
    /* so return address has to be restored manually */

    /* restore return address manually */
    __asm__ __volatile__("mov.a  %%a11,  %[temp_a11]"
      :
      : [temp_a11]"d"(temp_a11)
      : "memory");

    /* now return address is restored - RET command will work again */

    return; /* RET command */
}


int main (void);
int main (void)
{
    test();
    if (x_u32 != 1) { abort(); }
    exit(0);
    return(0);
}
```

Note:

A function called with a JL instruction may override any registers of the calling function. So __attribute__((interrupt)) may only be used if one of both is fulfilled:

- the calling function does not use any values kept in registers
- the called function does not change any registers

14.8 Default Case

if-else constructions are used quite often in source files. In most cases, one branch is the default case while the other case is rarely used. An **else** case might for instance contain an error handling.

```
if (v_battery < LOW_BATTERY)
{
    FLAG |= FLAG_LOW_BATTERY;
}
```

```
else
{
    error_status (FLAG_LOW_BATTERY);
}
```

In the example the variable v_battery contains the value of A/D-converted battery voltage. For monitoring purposes, the battery voltage is compared against a lower limit LOW_BATTERY. If the voltage drops below LOW_BATTERY, an error handling function error_status (); is called. In the example, the error handling function reports the status FLAG_LOW_BATTERY. This error function is used only in the exception case, therefore it is advisable to optimise the code for the default case. To transfer this knowledge to the compiler, a **#pragma** statement with the following notation is introduced:

```
#pragma branch_if_default
if (v_battery < LOW_BATTERY)
{
    FLAG |= FLAG_LOW_BATTERY;
}
else
{
    error_status (FLAG_LOW_BATTERY);
}
```

Note:

The compiler uses this information on the code for optimising the cash and pipeline access for the default case. Thus the execution time is reduced.

`#pragma branch_if_default`
: Indicates the **if**-case as default case.

`#pragma branch_else_default`
: Indicates the **else**-case as default case.

`#pragma branch_no_default`
: Indicates that neither the **if**-case nor the **else** case is preferred.

`#pragma branch_case_default`
: Mark the default case within a **switch** statement.

14.8.1 Restriction

- The statements **#pragma** branch_if_default or **#pragma** branch_else_default must be followed immediately by an **if-else** or **if** statement (comments are the sole allowed insertion between these statements). Otherwise an error message will be issued.
- With **switch** blocks also, the **#pragma** branch_case_default statement must be followed immediately by a **case** statement, the only exception being comments. Otherwise an error message will be issued. The same applies for **#pragma** branch_no_default and **if-else-**, **if-** or **switch**-statements.
- Wrong use of pragma statements, e.g. **#pragma** branch_if_default before a **switch**-statement will issue an error.

- The scope of a pragma is only valid for the next **if**, **if-else** or **switch** statement. To select a conditional statement within a nested structure, you must select the statement by a separate **#pragma** instruction.
- If a conditional expression is used without specifying the default or non-default case, (**#pragma** branch_no_default), a warning will be issued. To enable the warning message, the option `-mwarn-pragma-branch` must be set. Per default the warnings are not enabled.
- Using contrary pragma statements within conditional statements will generate an error.

Note:

Comments and preprocessor directives between pragma and conditional statements are accepted.

14.8.2 Syntax Examples

```
#pragma branch_if_default
if (B_stend)
{
    ctemp =1;
}
else
{
    error_handler ();
}
```

```
#pragma branch_else_default
#ifdef OWNTEST
if (B_stend)
#endif
#ifndef OWNTEST
if (!B_stend)
#endif
{
    ctemp =1;
}
else
{
    error_handler ();
}
```

Using Comments

```
#pragma branch_if_default
// comment1
/* comment2 */
if (B_stend)
{
    temp =100;
}
else
{
    temp =0;
}
#pragma branch_else_default
// comment3
/* comment4 */
if (B_stend)
{
    temp =100;
}
...
```

Error Cases

```
#pragma branch_if_default
//Comment
/*Another comment */
temp_U32 =0x78546897;
if (B_stend)
{
    temp =100;
}
else
{
    temp =0;
}

#pragma branch_no_default
if (B_stend)
{
    temp =100;
}
#pragma branch_else_default
else
{
    temp =0;
}

#pragma branch_case_default
if (B_stend)
{
    temp =100;
}
else
{
    temp =0;
}

//Comment
/*Another comment */
if (B_stend)
{
    temp = 100;
}
else
{
    temp =0;
}
```

Nested Constructs

```
#pragma branch_else_default
if (B_stend)
{
    temp = 0;
}
else if (tmot > 15)
{
    temp = 1;
}
else
#pragma branch_if_default
if (tmot > 30)
{
   temp = 2;
}
else
#pragma branch_if_default
if (tmot > 100)
{
    temp = 3;
}
else
{
    temp = -1;
}
if (B_stend)
{
    temp = 0;
}
else
if (tmot > 15)
{
    temp = 1;
}
else
if (tmot > 30)
{
    temp = 2;
}
else
#pragma branch_if_default
if (tmot > 100)
{
    temp = 3;
}
else
{
    temp = -1;
}
```

Switch Statements

```
switch (state)
{
case INIT:
    temp = 0;
    state = STATE1;
    break;
case STATE1:
    temp = 1;
    state = STATE2;
    break;
#pragma branch_case_default
case STATE2:
    temp = 234;
    state = STATE3;
    break;
...
default:
    temp = 0;
    state = INIT;
    break;
}
```

Conditional Expression

```
#pragma branch_if_default
    temp = (B_stend = FALSE) ? temp1 : temp2;
```

14.9 Indirect Addressing by longcall

Because of the memory mapping of the TC1796 with internal RAM located at `0xD4000000`, all calls from external memory to internal RAM must be done via indirect addressing. The internal RAM is significantly faster in execution time, so it is advisable to map functions which are very often used to the internal RAM. Especially the runtime of operating system functions, which are called frequently, can be improved by factors.

The linker supports mapping of code to internal RAM with the option `--relax-24rel`. This is realised by having calls to a jump table and from there to the function with indirect addressing. This method is not very efficient in terms of runtime and memory consumption.

A better method is to select functions, which are called via indirect addressing, by an additional attribute. Because the operating system is supplied as a library, it is required that the addressing mode is selected at the compiler stage already. At the linker stage it is not possible to change the code to indirect addressing, because indirect addressing takes more space in the object code than absolute or relative addressing.

With the function attribute `longcall`, arbitrary functions can be called by `calli`. The code and the execution time is thus reduced.

This mechanism is available for all types of functions and should be applied to all functions that are not accessible by a normal `call` (e.g. internal RAM).

Note:

If a function is declared with the attribute `longcall` the function will be referenced by a `calli` instead of `call` instruction.

```
extern void func02 (void)__attribute__((longcall));

void func01 (void)__attribute__((longcall));

void func01 ()
{
/*do something */;
}
```

14.10 Position-Independent Code

Only relative jumps / jump tables are allowed for using position-independent code. With the option `-mcode-pic` the generation of position-independent code is activated. This option is not set per default.

If code consists of different modules, the user has to ensure, that the code is merged by linking at a defined section, without a reference to a different section.

If the option `-mcode-pic` is set, the compiler will generate PIC code and will be put functions in an own section called .pictext instead of the standard section .text. Please note that if you use any **#pragma** section directive for a function, the compiler will not generate PIC code for this function and will put it in the user defined section.

By using function pointers the absolute address of the function is stored in `a4`. Position-independent code is not compatible with a static initialisation of pointers. Therefore a warning or error is issued. The invocation of option `-merror-pic=on` will result in an error message. This is the default case but the option `-mcode-pic` is required.

By using the dynamic initialisation of a function pointer, the address of a function call can be calculated by the offset to the beginning of a section and the program counter. The dynamic initialisation routine must be started manually by the user.

If the option `-mno-dynamic-address-calc-with-code-pic` is set, the address of a function that is accessed by function pointer will not be calculated dynamically. It is not set per default in conjunction with `-mcode-pic`.

Note:

Position-independent code is only possible for code within the flash.

A Start address of section

B Start address of function

(B-A) Offset of function to beginning of section

C Start address of section after moving address

D Start address of function that is moved (D = New start address + Offset)

Note:

The optimisation level `-O3` activates the inlining of functions. This will cause erratic behaviour with position-independent code. To avoid inlining, simply add the `__attribute__ ((noinline))` for these functions.

Note:

The compiler writes the original address at the start of each function. This causes a disassembler to output strange and random mnemonics at the start of functions. Please note that this does not affect the correct working of the code!

Example

In the following example these flags are passed to the compiler driver

`-g3`	Generate debug info.
`-O2`	Use optimisation level 2.
`-mcpu=TC1775`	Select TC1775 derivative.
`-mcpu-specs=tricore.specs`	Use own configuration as specs file.
`-mcode-pic`	Generate position-independent code.
`-merror-pic=off`	Disable errors for pic.
`-Ttricore.ld`	Specify own linker description file.

Generate an object file by invoking the compiler with the corresponding flags.

```
tricore-gcc ... pic.c -o pic.o
```

With `tricore-objdump -s --section=.pictext` the context of the section `.pictext` is displayed in the command line.

In the user configuration file `tricore.specs` a `define` for derivative TC1775 is added.

```
*TC1775:
    -mtc12 %(tc1775_errata) -DTC1775
```

The linker description file `tricore.ld` inserts the input section `.MussInsRAM` into the output section `.data`.

```
  .data : AT(LOADADDR(.pcpdata) + SIZEOF(.pcpdata))
  {
    . = ALIGN(8) ;
    DATA_BASE = ABSOLUTE(.) ;
    *(.MussInsRAM)
    *(.data)
    *(.data.*)
    *(.gnu.linkonce.d*)
    SORT(CONSTRUCTORS)
    . = ALIGN(8) ;
    DATA_END = ABSOLUTE(.) ;
  } > ext_dram
```

In the first step, the target address of the position-independent code is defined. TC1775 is defined in the derivative-specific configuration file `tricore.specs`.

```
#ifdef TC1775
#define TARGET_ADDR_PIC (0xa00f0000)
#endif
```

The function Addieren returns the sum of two values.

```
uint32 Addieren(uint32 zahl1, uint32 zahl2)
{
    return (zahl1 + zahl2);
}
```

The function Multiplizieren returns the the result of a multiplication of two variables.

In the examples, different types of pointer arrays are defined.

Pointer Array in ROM

```
const uint32 FunktionsZeigerTabelle_ROM_uint32[2] = {
                                                     (uint32)&Addieren,
                                                     (uint32)&Multiplizieren
                                                    };
const void *FunktionsZeigerTabelle_ROM_void[2] = {
                                                  &Addieren,
                                                  &Multiplizieren
                                                 };
```

Pointer Array in RAM

```
/* Define CONST-RAM-Arrays with pointer to function */
#pragma section .MussInsRAM aw
/* .MussInsRAM is put in output section .data (modified linker description file) */
const uint32 FunktionsZeigerTabelle_CONST_RAM_uint32[2] = {
                                                           (uint32)&Addieren,
                                                           (uint32)&Multiplizieren
                                                          };
const void *FunktionsZeigerTabelle_CONST_RAM_void[2] = {
                                                        &Addieren,
                                                        &Multiplizieren
                                                       };
#pragma section
```

A dynamic initialisation of pointer arrays to the functions Addieren and Multiplizieren is required to ensure that these functions are referenced correctly after moving the position-independent code to a new target address.

```
/* dynamic initialisation of CONST-RAM-Arrays */
void InitFunktionsZeigerTabelle_CONST_RAM_uint32(void)
{
    FunktionsZeigerTabelle_CONST_RAM_uint32[0] = (uint32) &Addieren;
    FunktionsZeigerTabelle_CONST_RAM_uint32[1] = (uint32) &Multiplizieren;
}
```

In doit the functions Addieren and Multiplizieren are called by pointer arrays. In the example the pointer array FunktionsZeigerTabelle_ROM_void is not initialised by a function. Functions that are not initialised by a function will cause an error after moving the position-independent code to a different memory location.

```
void doit (void)
{
    /* Initialize values */
    val1 = 3;
    val2 = 4;
    val3 = Funktion(val1, val2,
                    (uint32 (*)(uint32, uint32))FunktionsZeigerTabelle_ROM_void[0]);
    val4 = Funktion(val1, val2,
                    (uint32 (*)(uint32, uint32))FunktionsZeigerTabelle_ROM_void[1]);

    InitFunktionsZeigerTabelle_RAM_uint32();
    InitFunktionsZeigerTabelle_RAM_void();
    val1 = 5;
    val2 = 6;
    val5 = Funktion(val1, val2,
                    (uint32 (*)(uint32, uint32))FunktionsZeigerTabelle_RAM_uint32[0]);
    val6 = Funktion(val1, val2,
                    (uint32 (*)(uint32, uint32))FunktionsZeigerTabelle_RAM_void[1]);
...
    return;
}
```

Funktion is an error function that returns -1 if the pointer array is not initialised dynamically by a function. Otherwise this function will return the result of the functions Addieren or Multiplizieren which are defined in the corresponding pointer arrays.

```
uint32 Funktion(uint32 zahl1, uint32 zahl2, uint32 (*func_ptr)(uint32, uint32))
{
    void *tmp;

    __asm__ volatile ("mov.aa %%a2, %%a11 \n"
                "\tjl 0f \n"
                "\t0: \n"
                "\tmov.aa %0, %%a11 \n"
                "\tmov.aa %%a11, %%a2" : "=a" (tmp):);
    if ((((uint32)func_ptr) & 0xffff0000) != (((uint32)tmp) & 0xffff0000))
        return(-1);
    return((*func_ptr)(zahl1, zahl2));
}
```

In main, the position-independent code is copied to the target memory location TARGET_ADDR_PIC by the memcpy instruction.

```
memcpy((void *)TARGET_ADDR_PIC,&PicSectionBegin,&PicSectionEnd-&PicSectionBegin);
```

The first parameter specifies the target address C, the second one the original start address A and the last parameter is the size of the section. Subtracting the start address A from the end address B provides the size $B - A$ of the section.

With the gnu debugger a snippet of the section can be easily displayed by:

```
x/10i PicSectionBegin
```

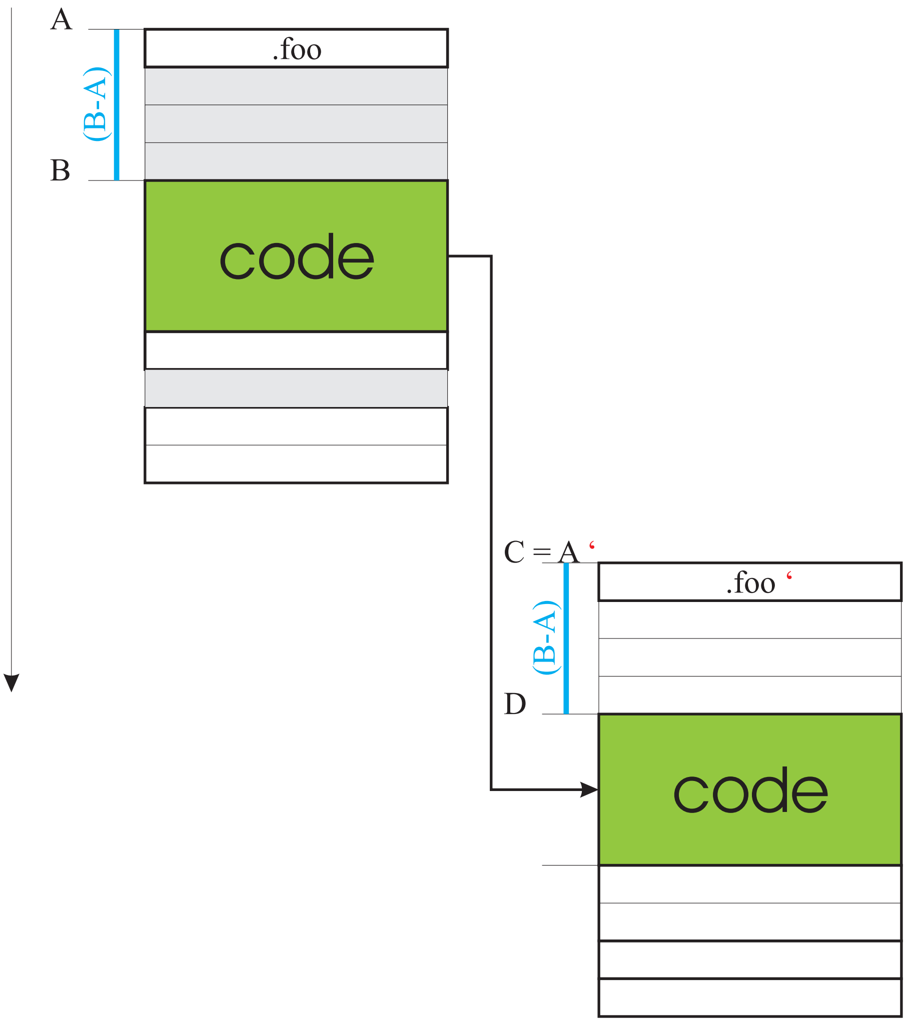

Figure 14.2: Position Independent Code

and

`x/10i TARGET_ADDR_PIC`

The instruction displays the contents of the next 10 instruction beginning at the specified address.

In main the function doit is referenced. The return values of the functions Addieren and Multiplizieren are stored in the global variables $val3 \ldots val8$.

```
    doit();
    if (val2 != 1) abort();
    if (val3 != 7) abort();
    if (val4 != 12) abort();
    if (val5 != 11) abort();
    if (val6 != 30) abort();
    if (val7 != 15) abort();
    if (val8 != 56) abort();
```

With fillmem the contents of the code at the start address A is filled with debug instruction. The debug instruction is 0xa000.

```
    fillmem(&PicSectionBegin,0xa000,&PicSectionEnd-&PicSectionBegin);
```

The new address of the doit function after the memory copying process is defined by the sum of target address and offset. A pointer fptr is initialised with the new address of doit.

```
fptr = (void (*)(void))(TARGET_ADDR_PIC
                        + ((unsigned int)doit - (unsigned int)&PicSectionBegin));
```

The function doit is called by:

```
    /* fptr points to new address of function doit -> call doit' */
    (*fptr)();
```

You can debug the function in the gnu debugger command line. First set a breakpoint at the new address of doit.

```
b *(TARGET_ADDR_PIC+(unsigned int)doit-(unsigned int)PicSectionBegin)
```

With the single step instruction you can debug the function doit at the new memory location.

```
si
```

The pointer array FunktionsZeigerTabelle_ROM_void is not initialised by a function. Therefore the error function Funktion will return (-1). The result is stored in variable $val3$ and $val4$.

```
    if (val2 != 1) abort();
    if (val3 != -1) abort();
    if (val4 != -1) abort();
    if (val5 != 11) abort();
    if (val6 != 30) abort();
```

See the attached source code `pic.c` for details.

```
#ifdef TC1775
#define TARGET_ADDR_PIC (0xa00f0000)
#endif
typedef unsigned long uint32;

volatile uint32 val1, val2, val3, val4, val5, val6, val7, val8;

/* Declare a function which is use function pointer as argument */
extern uint32 Funktion(uint32, uint32,
                       uint32 (*)(uint32, uint32));

/* Test functions */
extern uint32 Addieren(uint32, uint32);
uint32 Addieren(uint32 zahl1, uint32 zahl2)
{
    return (zahl1 + zahl2);
}

extern uint32 Multiplizieren(uint32, uint32);
uint32 Multiplizieren(uint32 zahl1, uint32 zahl2)
{
    return (zahl1 * zahl2);
```

```
}

/* Define ROM-Arrays with pointer to function */
const uint32 FunktionsZeigerTabelle_ROM_uint32[2] = {
                                                     (uint32)&Addieren,
                                                     (uint32)&Multiplizieren
                                                    };
const void *FunktionsZeigerTabelle_ROM_void[2] = {
                                                  &Addieren,
                                                  &Multiplizieren
                                                 };

/* Define RAM-Arrays with pointer to function */
uint32 FunktionsZeigerTabelle_RAM_uint32[2] = {
                                               (uint32)&Addieren,
                                               (uint32)&Multiplizieren
                                              };
void *FunktionsZeigerTabelle_RAM_void[2] = {
                                            &Addieren,
                                            &Multiplizieren
                                           };

/* Define CONST-RAM-Arrays with pointer to function */
#pragma section .MussInsRAM aw
/* .MussInsRAM is put in output section .data (modified linker description file) */
const uint32 FunktionsZeigerTabelle_CONST_RAM_uint32[2] = {
                                                           (uint32)&Addieren,
                                                           (uint32)&Multiplizieren
                                                          };
const void *FunktionsZeigerTabelle_CONST_RAM_void[2] = {
                                                        &Addieren,
                                                        &Multiplizieren
                                                       };
#pragma section

/* dynamic initialization of RAM-Arrays */
void InitFunktionsZeigerTabelle_RAM_uint32(void)
{
    FunktionsZeigerTabelle_RAM_uint32[0] = (uint32) &Addieren;
    FunktionsZeigerTabelle_RAM_uint32[1] = (uint32) &Multiplizieren;
}
void InitFunktionsZeigerTabelle_RAM_void(void)
{
    FunktionsZeigerTabelle_RAM_void[0] = &Addieren;
    FunktionsZeigerTabelle_RAM_void[1] = &Multiplizieren;
}

/* dynamic initialization of CONST-RAM-Arrays */
void InitFunktionsZeigerTabelle_CONST_RAM_uint32(void)
{
    FunktionsZeigerTabelle_CONST_RAM_uint32[0] = (uint32) &Addieren;
    FunktionsZeigerTabelle_CONST_RAM_uint32[1] = (uint32) &Multiplizieren;
}
void InitFunktionsZeigerTabelle_CONST_RAM_void(void)
{
    FunktionsZeigerTabelle_CONST_RAM_void[0] = &Addieren;
    FunktionsZeigerTabelle_CONST_RAM_void[1] = &Multiplizieren;
}


void doit (void)
{
  /* Initialize values */
  val1 = 3;
  val2 = 4;

  /* Call functions by pointer arrays */
  /* ================================================= */
  val3 = Funktion(val1, val2,
          (uint32 (*)(uint32, uint32)) FunktionsZeigerTabelle_ROM_void[0]);
  val4 = Funktion(val1, val2,
          (uint32 (*)(uint32, uint32)) FunktionsZeigerTabelle_ROM_void[1]);

  InitFunktionsZeigerTabelle_RAM_uint32();
  InitFunktionsZeigerTabelle_RAM_void();
  val1 = 5;
  val2 = 6;
  val5 = Funktion(val1, val2,
          (uint32 (*)(uint32,uint32))FunktionsZeigerTabelle_RAM_uint32[0]);
  val6 = Funktion(val1, val2,
          (uint32 (*)(uint32, uint32))FunktionsZeigerTabelle_RAM_void[1]);

  InitFunktionsZeigerTabelle_CONST_RAM_uint32();
  InitFunktionsZeigerTabelle_CONST_RAM_void();
  val1 = 7;
  val2 = 8;
  val7 = Funktion(val1, val2,
          (uint32 (*)(uint32,uint32))FunktionsZeigerTabelle_CONST_RAM_uint32[0]);
  val8 = Funktion(val1, val2,
          (uint32 (*)(uint32,uint32))FunktionsZeigerTabelle_CONST_RAM_void[1]);


  val1 = 5;
  switch (val1)
  {
      case 1: case 3: case 5: case 7: case 9:
          val2 = 1;
          break;
      case 2: case 4: case 6: case 8: case 10:
          val2 = 2;
          break;
      default:
          val2 = 0;
          break;
  }
  return;
}

extern void doit(void);

#pragma section .text
void fillmem(short *ps, short val, int count)
```

```
{
  int i;
  for (i = 0; i < count/sizeof(short); i++)
      *ps++ = val;
}
int main(void)
{
  extern void PicSectionBegin, PicSectionEnd;
  void (*fptr)(void);

  memcpy((void *)TARGET_ADDR_PIC,
         &PicSectionBegin,
         &PicSectionEnd-&PicSectionBegin);
  /* function doit is copied to new memory address */
  fptr = (void (*)(void))(TARGET_ADDR_PIC +
                        ((unsigned int)doit - (unsigned int)&PicSectionBegin));
  val1 = val2 = val3 = val4 = val5 = val6 = val7 = val8 = 0;
  doit();
  if (val2 != 1) abort();
  if (val3 != 7) abort();
  if (val4 != 12) abort();
  if (val5 != 11) abort();
  if (val6 != 30) abort();
  if (val7 != 15) abort();
  if (val8 != 56) abort();
  val1 = val2 = val3 = val4 = val5 = val6 = val7 = val8 = 0;
  /* fill "old code" with debug instruction (0xa000) */
  fillmem(&PicSectionBegin,
          0xa000,
          &PicSectionEnd-&PicSectionBegin);
  /* fptr points to new address of function doit -> call doit’ */
  (*fptr)();
  if (val2 != 1) abort();
  if (val3 != -1) abort();
  if (val4 != -1) abort();
  if (val5 != 11) abort();
  if (val6 != 30) abort();
  if (val7 != 15) abort();
  if (val8 != 56) abort();
  exit(0);
}
#pragma section
```

14.11 Position Independent Data

The option `-msmall-pid` generates position-independent data for small addressable data. v3.4 This option is set per default and can be disabled via `-mno-small-pid` option. The compiler accesses small addressable areas via one of the address registers `A0`, `A1`, `A8` or `A9`. If the compiler cannot generate position-independent data for a small addressable area, a warning will be issued. Because of the unit compile mechanism, the warning will be issued for the last line of the compiled module, if optimisation is enabled.

Note:

Example: If a pointer is statically initialised with the address of a small addressable variable (.sdata object) and the option `-msmall-pid` is set, then the compiler will issue a warning. This warning can be disabled via the `-mno-warn-smalldata-initializers` option.

To support position-independent data by the compiler, a new relocation type is required. Since this relocation type is not part of the EABI-specification, the option `-meabi` will implicitly set the option `-mno-small-pid`.

Note:

The inlining of functions is activated in different optimisation levels. The functions that are used in context with position-independent data may not be inlined, therefore the __attribute__ ((noinline)) is required.

Example

In the following example, these flags are passed to the compiler driver.

`-g3`	Generate debug info.
`-O2`	Use optimisation level 2.
`-mcpu=TC1796`	Select TC1796 derivative.
`-msmall-pid`	Generate position-independent data.
`-Ttc1796.ld`	Specify user-defined linker description file.

Example

In the example the user-defined small data section .sdata.a9 is applied for the **struct** A9__org and **struct** A9__new. This section is inserted into the output section .sdata4 of the linker description file.

```
.sdata4 : AT(LOADADDR(.data) + SIZEOF(.data) + SIZEOF(.sdata2))
  {
    . = ALIGN(8) ;
    SDA4_BASE = ABSOLUTE(.) ;
    *(.sdata.sdata4*)
    _SMALL_DATA4_ = ABSOLUTE(.);
        small_data_1 = ABSOLUTE(.);
    *(.sdata.a9_1*)
        small_data_2 = ABSOLUTE(.);
    *(.sdata.a9_2*)
    *(.sdata.a9*)
    . = ALIGN(8) ;
  } > int_dram
```

The .sdata4 section is addressed via a9. In main the address register a9 is modified to

```
A9 = A9 + (uint32)&A9__new - (uint32)&A9__org;
```

so the data set 2 is accessed via a9. The access data will thus be position-independent and can be configured by switching the start address of the corresponding address register.

```
typedef struct {
  unsigned short x0;
  short x1;
  unsigned int x2;
  int x3;
} Data_t;

/* Address registers
A0: .sdata/. sbss, A1: .sdata2/.sbss2, A8: .sdata3/.sbss3, A9: .sdata4/.sbss4
*/
unsigned int A0, A1, A8, A9;

/*  Structs for testing the content
  .sdata.a9 is put in sdata4 output section in linker description file.
  sdata4 is addressed via register a9
*/

#pragma section .sdata.a9 aws
/* Configuration data set 1*/
const volatile Data_t A9__org = {
  .x0 = 0,
  .x1 = 1,
  .x2 = 2,
  .x3 = 3
};
/* Configuration data set 2 */
const volatile Data_t A9__new = {
  .x0 = 10,
  .x1 = 11,
  .x2 = 12,
  .x3 = 13
};
int test;    // small addressable variable
#pragma section

int *test_p = &test;  // will issue a warning

/* Position Independent Data Test Function */
void pid_test1(const volatile Data_t *mydata) __attribute__ ((noinline));
void pid_test2(const volatile Data_t *mydata) __attribute__ ((noinline));

void pid_test1(const volatile Data_t *mydata)
{
  if (mydata->x0 != 0) { abort(); }
  if (mydata->x1 != 1) { abort(); }
  if (mydata->x2 != 2) { abort(); }
  if (mydata->x3 != 3) { abort(); }
}

void pid_test2(const volatile Data_t *mydata)
{
  if (mydata->x0 != 10) { abort(); }
  if (mydata->x1 != 11) { abort(); }
  if (mydata->x2 != 12) { abort(); }
  if (mydata->x3 != 13) { abort(); }
}

int main (void)
{
/* Test data against configuration data set 1*/
  pid_test1(&A9__org);

/* Get the value of address register */
  __asm__ volatile ("mov.d %[d], %%a9" : [d] "=d" (A9) : );
/* Modify the value of address register; set data to configuration data set 2 */
  A9 = A9 + (unsigned int)&A9__new - (unsigned int)&A9__org;
  __asm__ volatile ("mov.a %%a9, %[d]" : :[d] "d" (A9) );

/* Check whether data is position independent */
/* Test data against configuration data set 2*/
  pid_test2(&A9__org);

/* All checks passed */
  exit (0);
  return(0);
}
```

14.12 Builtins

The TriCore Development Platform supports the intrinsics listed below. Note, that by using an intrinsic you make the compiler inject algebraic representations of the intrinsic's semantic into the internal algebraic representation of your C code.

Thus, using intrinsics gives the compiler a notion of what it is doing, a precondition for optimising. For example, n = __SUBS (1, n); may be compiled as rsub %d0, %d0, 1, provided n is an integer living in register %d0. In most cases they are machine instruction equivalents to intrinsics but exceptions exist. Moreover, there is no guarantee that the compiler will choose a specific machine instruction, even if one exists. It will choose machine instructions that have the desired effect on the machine, that's all.

To insert specific machine instructions into your code, please use Inline Assembler. The disadvantage of using Inline Aassembler is that you cannot benefit from an optimising compiler, because the optimising compiler perceives Inline Assembler templates as mere strings and thus has *no* notion of what is going on.

Note that for builtins, the __FOO is just a predefined macro that resolves to __builtin_foo, the very builtin's name. This allows you to check for the presence of a specific builtin:

```
/* get absolute value of a, not caring for overflow */
int _abs (int a)
{
#if defined (__ABS)
   return __ABS (a);
#else
   return (a >= 0) ? a : -a;
#endif /* have __ABS */
}
```

Before listing all the intrinsics and their prototypes, we take a detailed look at some of them. Suppose you want an 'unsigned multiply-add', i.e. something equivalent to the madd.u instruction. From the handbook, you learn that this instruction has the format:

```
   madd.u   E[c], E[d], D[a], D[b]
```

which implements $c = d + a \cdot b$. To use it for a multiply-add of unsigned integers we use the __MADDU builtin and write

```
/* d + a*b */
unsigned int maddu (unsigned int d, unsigned int a, unsigned int b)
{
   return __MADDU (d, a, b);
}
```

The generated code is then

```
maddu:
   madd    %d2, %d4, %d5, %d6
   ret
```

Note that there is no difference between an unsigned multiply-add and a signed multiply-add, since only 32-bit values are involved and the MCU status is no concern. Therefore, the compiler uses a madd instruction that has less overhead than a madd.u. To get a 64-bit result, write:

```
/* d + a*b */
unsigned long long maddul (unsigned int d, unsigned int a, unsigned int b)
{
   return __MADDU (d, a, b);
}

unsigned long long maddull (unsigned long long d, unsigned int a, unsigned int b)
{
   return __MADDU (d, a, b);
}
```

Here, the compiler has to take the madd.u instruction.

```
maddul:
   mov     %d2, %d4
   mov     %d3, 0
   madd.u  %e2, %e2, %d5, %d6
   ret

maddull:
   madd.u  %e2, %e4, %d6, %d7
   ret
```

14.12.1 Overview

This section gives a short overview of supported builtins.

For clarity reasons, notations in this sections lack the ultimate mathematical strictness.

Most operators come in different flavours. The signed and unsigned incarnation of Operator $\circ$ will be denoted as subscript s and u, resp., or will become clear from the context.

The small suffixes appended to each intrinsic denote the number of machine instructions that you may expect when applying the respective builtin. However, as stated above, there is no guarantee for that number of instructions, since the compiler will optimise the inserted algebraic transcriptions against the transcriptions of other builtins or the transcriptions of plain C/C++ code.

Not all functionalities are implemented as intrinsics. Some of these functionalities – esp. those tagged with an asterisk – can easily be accomplished by using another intrinsic that is then tagged* and stands in place of the unimplemented builtin. Suppose you want to count the number of trainling ones of an integer. In the corresponding table you will find the entry:

__CTZ$^*_6$

which leads you to the __CTZ intrinsic and thereby states that the best way to implement the desired functionality is by using that intrinsic. In this particular case the solution to the problem may look like this:

```
#define __CTO(X) \
    __CTZ (~(X))
```

or like that

```
static inline int
__cto (int x)
{
   return __CTZ (~x);
}
```

In both cases, the assembler output will contain instruction sequences that accomplish the desired computation.

14.12.1.1 Extracting Components of Packed Vectors

$i =$ const. with $0 \leqslant i < n$	$\text{extend}_\text{s} a_i$	$\text{extend}_\text{u} a_i$
2×16 Bit	__GETHWORD $_1$	__GETUHWORD $_1$
4×8 Bit	__GETBYTE $_1$	__GETUBYTE $_1$

$0 < w =$ const. and $w + d \leqslant 31$	$\text{extend}_\text{s}(\lfloor a/2^d \rfloor \mod 2^w)$	$\text{extend}_\text{u}(\lfloor a/2^d \rfloor \mod 2^w)$
get w bits $a_d \ldots a_{d+w-1}$	__EXTR $_1$	__EXTRU $_1$

14.12.1.2 Saturation

	$\text{sat}_8\, a$	$\text{sat}_{16}\, a$
signed	__SATB $_1$	__SATH $_1$
unsigned	__SATBU $_1$	__SATHU $_1$

14.12.1.3 Saturated Addition and Subtraction

For ordinary addition resp. subtraction of packed vectors you will write $a + b$ resp. $a - b$.

$\text{sat}\,(a_i \pm b_i)$	$\text{sat}_\text{s}(a_i + b_i)$	$\text{sat}_\text{u}(a_i + b_i)$	$\text{sat}_\text{s}(a_i - b_i)$	$\text{sat}_\text{u}(a_i - b_i)$
32 Bit	__ADDS $_{1}$	__ADDSU $_{1}$	__SUBS $_{1}$	__SUBSU $_{1}$
2×16 Bit	__ADDSH $_{1}$	__ADDSHU $_{1}$	__SUBSH $_{1}$	__SUBSHU $_{1}$
4×8 Bit	__ADDSB $_{15}$	__ADDSBU $_{3}$	__SUBSB $_{15}$	__SUBSBU $_{3}$

14.12.1.4 Multiplication

	$a \cdot b$	$\text{sat}_{32}(a \cdot b)$
signed 32 Bit	__MUL $_{1}$	__MULS $_{1}$
unsigned 32 Bit	__MULU $_{1}$	__MULSU $_{1}$

14.12.1.5 Multiply-Add and Multiply-Subtract

$64 = 64 \pm 32{\cdot}32$	$d + a{\cdot}b$	$d - a{\cdot}b$
signed	__MADD $_{1}$	__MSUB $_{1}$
unsigned	__MADDU $_{1}$	__MSUBU $_{1}$

$\text{sat}(64 \pm 32{\cdot}32)$	$\text{sat}_{32}(d{+}a{\cdot}b)$	$\text{sat}_{64}(d{+}a{\cdot}b)$	$\text{sat}_{32}(d{-}a{\cdot}b)$	$\text{sat}_{64}(d{-}a{\cdot}b)$
signed	__MADDS32 $_{1}$	__MADDS64 $_{1}$	__MSUBS32 $_{1}$	__MSUBS64 $_{1}$
unsigned	__MADDSU32 $_{1}$	__MADDSU64 $_{1}$	__MSUBSU32 $_{1}$	__MSUBSU64 $_{1,7}$

14.12.1.6 Minimun and Maximum

	$\max_\text{s}(a_i, b_i)$	$\max_\text{u}(a_i, b_i)$	$\min_\text{s}(a_i, b_i)$	$\min_\text{u}(a_i, b_i)$
32 Bit	__MAX $_{1}$	__MAXU $_{1}$	__MIN $_{1}$	__MINU $_{1}$
2×16 Bit	__MAXH $_{1}$	__MAXHU $_{1}$	__MINH $_{1}$	__MINHU $_{1}$
4×8 Bit	__MAXB $_{1}$	__MAXBU $_{1}$	__MINB $_{1}$	__MINBU $_{1}$

14.12.1.7 Absolute Value, Absolute Value of Difference

Note that the absolute value is defined as

$$\begin{aligned} \|x - y\| &:= \begin{cases} x - y, & \text{if } x \geqslant y \\ y - x, & \text{if } x < y \end{cases} \\ \|x\| &:= \|x - 0\| \end{aligned}$$

which may be negative provided you interpret the values as signed values. The result may be negative even if no underflow/overflow occurs in the computation of the difference. Just imagine the 32 bit integer example $\|-2^{31}\| = -2^{31}$.

	$\|a_i\|$	$\text{sat}_\text{s}\lvert a_i\rvert$	$\|a_i - b_i\|$	$\text{sat}_\text{s}\lvert a_i - b_i\rvert$
32 Bit	__ABS $_{1}$	__ABSS $_{1}$	__ABSDIF $_{1}$	__ABSDIFS $_{1}$
2×16 Bit	__ABSH $_{1}$	__ABSSH $_{1}$	__ABSDIFH $_{1}$	__ABSDIFSH $_{1}$
4×8 Bit	__ABSB $_{1}$	__ABSDIFSB $^{*}_{4}$	__ABSDIFB $_{1}$	__ABSDIFSB $_{4}$

14.12.1.8 Comparison

These comparison operators set all bits if the comparison holds and clear all bits, otherwise.

	$a_i == b_i$	$a_i <_\text{s} b_i$	$a_i <_\text{u} b_i$	$\max(1, \sum_{i=1}^{n} \lvert a_i == b_i\rvert)$
2×16 Bit	__EQH $_{1}$	__LTH $_{1}$	__LTHU $_{1}$	__EQANYH $_{1}$
4×8 Bit	__EQB $_{1}$	__LTB $_{1}$	__LTBU $_{1}$	__EQANYB $_{1}$

14.12.1.9 Shift, Rotate and Saturated Shift

	$a_i \gg b$	$a_i \ll b$	$a_i \ll_{\text{rot}} b$
signed 32 Bit	$a \gg b$ $_2$	$a \ll b$ $_1$	__ROTATEL $_1$
signed 2×16 Bit	__ASHIFTRH $_2$	__ASHIFTLH $_1$	-
unsigned 2×16 Bit	__LSHIFTRH $_2$	__LSHIFTLH $_1$	-
unsigned 4×8 Bit	$a \gg 1^*_{4 \cdot b}$	$a \ll 1^*_{4 \cdot b}$	-

	$\text{sat}_s(a_i \ll 1)$	$\text{sat}_u(a_i \ll 1)$	$\text{sat}_s(a_i \ll b)$
32 Bit	__ADDS$^*_1$	__ADDSU$^*_1$	__SHAS $_1$
2×16 Bit	__ADDSH$^*_1$	__ADDSHU$^*_1$	-
4×8 Bit	__ADDSB$^*_{11}$	__ADDSBU$^*_3$	-

To compute $(a \gg b) \,\&\, (2^w - 1)$ see also __EXTR and __EXTRU.

14.12.1.10 Bit Inventory

Count Leading...	...Zeros	...Ones	...Signs
32 Bit	__CLZ $_1$	__CLZ$^*_1$	__CLS $_1$
2×16 Bit	__CLZH $_1$	__CLZH$^*_1$	__CLSH $_1$
4×8 Bit	-	-	-

Count Trailing...	...Zeros	...Ones	Find First Set
32 Bit	__CTZ $_5$	__CTZ$^*_6$	__FFS $_4$
2×16 Bit	-	-	-
4×8 Bit	-	-	-

	Parity
32 Bit	__PARITY $_4$

14.12.1.11 Core

	Move to Core Register	Move from Core Register
unsigned int	__MTCR $_1$	__MFCR $_1$

14.12.2 Short Descriptions and Prototypes

The following sections list the builtins with their respective prototypes and associated machine instructions. In case an assembler mnemonic is given, the builtin's semantic is the same (from the C programmer's point of view) as the semantic of the machine instruction. We do not repeat the instruction set manual here.

The entries look like this:

__FOO (`mnemonic`)	short description <return type> __FOO (<argument list>)

14.12.2.1 Extracting from Packed Vectors and Integers

cn denotes an n bit compile time constant.

__GETBYTE	extract component signed, packed byte int __GETBYTE (__vector signed char a, int c2)
__GETUBYTE	extract component unsigned, packed byte unsigned int __GETUBYTE (__vector unsigned char a, int c2)
__GETHWORD	extract component signed, packed half-word int __GETHWORD (__vector signed short a, int c1)
__GETUHWORD	extract component unsigned, packed half-word unsigned int __GETUHWORD (__vector unsigned short a, int c1)
__EXTR (extr)	extract signed Extract and extend signed c5 bits from a, starting at bit pos int __EXTR (int a, unsigned int pos, unsigned int c5)
__EXTRU (extr.u)	extract unsigned Extract and extend unsigned c5 bits from a, starting at bit pos unsigned int __EXTRU (unsigned int a, unsigned int pos, unsigned int c5)

14.12.2.2 Saturation

__SATB (sat.b)	saturate byte char __SATB (int a)
__SATBU (sat.bu)	saturate byte unsigned unsigned char __SATBU (int a)

__SATH (`sat.h`)	saturate halfword short __SATH (int a)
__SATHU (`sat.hu`)	saturate halfword unsigned unsigned short __SATHU (int a)

14.12.2.3 Saturated Addition and Subtraction

__ADDS (`adds`)	add signed with saturation int __ADDS (int a, int b)
__ADDSU (`adds.u`)	add unsigned with saturation unsigned int __ADDSU (unsigned int a, unsigned int b)
__ADDSH (`adds.h`)	add signed with saturation, packed half-word __vector signed short __ADDSH (__vector signed short a, __vector signed short b)
__ADDSHU (`adds.hu`)	add unsigned with saturation, packed half-word __vector unsigned short __ADDSHU (__vector unsigned short a, __vector unsigned short b)
__ADDSB	add signed with saturation, packed byte __vector signed char __ADDSB (__vector signed char a, __vector signed char b)
__ADDSBU	add unsigned with saturation, packed byte __vector unsigned char __ADDSBU (__vector unsigned char a, __vector unsigned char b)
__SUBS (`subs`)	subtract signed with saturation int __SUBS (int a, int b)

__SUBSU (subs.u)	subtract unsigned with saturation unsigned int __SUBSU (unsigned int a, unsigned int b)
__SUBSH (subs.h)	subtract signed with saturation, packed half-word __vector signed short __SUBSH (__vector signed short a, __vector signed short b)
__SUBSHU (subs.hu)	subtract unsigned with saturation, packed half-word __vector unsigned short __SUBSHU (__vector unsigned short a, __vector unsigned short b)
__SUBSB	subtract signed with saturation, packed byte __vector signed char __SUBSB (__vector signed char a, __vector signed char b)
__SUBSBU	subtract unsigned with saturation, packed byte __vector unsigned char __SUBSBU (__vector unsigned char a, __vector unsigned char b)

14.12.2.4 Multiplication

__MUL (mul)	multiply signed long int __MUL (int a, int b)
__MULU (mul.u)	multiply unsigned unsigned long int __MULU (unsigned int a, unsigned int b)
__MULS (muls)	multiply signed with saturation int __MULS (int a, int b)
__MULSU (muls.u)	multiply unsigned with saturation unsigned int __MULSU (unsigned int a, unsigned int b)

14.12.2.5 Multiply-Add and Multiply-Subtract

__MADD (madd)	multiply-add signed long int __MADD (long int d, int a, int b)
__MADDU (madd.u)	multiply-add unsigned unsigned long int __MADDU (unsigned long int d, unsigned int a, unsigned int b)
__MADDS32 (madds)	multiply-add signed with saturation int __MADDS32 (long int d, int a, int b)
__MADDS64 (madds)	multiply-add signed with saturation long int __MADDS64 (long int d, int a, int b)
__MADDSU32 (madds.u)	multiply-add unsigned with saturation unsigned int __MADDSU32 (unsigned long int d, unsigned int a, unsigned int b)
__MADDSU64 (madds.u)	multiply-add unsigned with saturation unsigned long int __MADDSU64 (unsigned long int d, unsigned int a, unsigned int b)
__MSUB (msub)	multiply-subtract signed long int __MSUB (long int d, int a, int b)
__MSUBU (msub.u)	multiply-subtract unsigned unsigned long int __MSUBU (unsigned long int d, unsigned int a, unsigned int b)
__MSUBS32 (msubs)	multiply-subtract signed with saturation int __MSUBS32 (long int d, int a, int b)
__MSUBS64 (msubs)	multiply-subtract signed with saturation long int __MSUBS64 (long int d, int a, int b)

__MSUBSU32 (msubs.u)	multiply-subtract unsigned with saturation unsigned int __MSUBSU32 (unsigned long int d, unsigned int a, unsigned int b)
__MSUBSU64 (msubs.u)	multiply-subtract unsigned with saturation Due to erratum CPU_TC.101 the code for this builtin depends on command line option `-mcpu101`. unsigned long int __MSUBSU64 (unsigned long int d, unsigned int a, unsigned int b)

14.12.2.6 Minimun and Maximum

__MAX (max)	maximum value signed int __MAX (int a, int b)
__MAXU (max.u)	maximum value unsigned unsigned int __MAXU (unsigned int a, unsigned int b)
__MAXB (max.b)	maximum value signed, packed byte __vector signed char __MAXB (__vector signed char a, __vector signed char b)
__MAXBU (max.bu)	maximum value unsigned, packed byte __vector unsigned char __MAXBU (__vector unsigned char a, __vector unsigned char b)
__MAXH (max.h)	maximum value signed, packed half-word __vector signed short __MAXH (__vector signed short a, __vector signed short b)
__MAXHU (max.hu)	maximum value unsigned, packed half-word __vector unsigned short __MAXHU (__vector unsigned short a, __vector unsigned short b)
__MIN (min)	minimum value signed int __MIN (int a, int b)

__MINU (min.u)	minimum value unsigned unsigned int __MINU (unsigned int a, unsigned int b)
__MINB (min.b)	minimum value signed, packed byte __vector signed char __MINB (__vector signed char a, __vector signed char b)
__MINBU (min.bu)	minimum value unsigned, packed byte __vector unsigned char __MINBU (__vector unsigned char a, __vector unsigned char b)
__MINH (min.h)	minimum value signed, packed half-word __vector signed short __MINH (__vector signed short a, __vector signed short b)
__MINHU (min.hu)	minimum value unsigned, packed half-word __vector unsigned short __MINHU (__vector unsigned short a, __vector unsigned short b)

14.12.2.7 Absolute Value, Absolute Value of Difference

__ABS (abs)	absolute value int __ABS (int a)
__ABSS (abss)	absolute value with saturation int __ABSS (int a)
__ABSH (abs.h)	absolute value, packed half-word __vector signed short __ABSH (__vector signed short a)
__ABSSH (abss.h)	absolute value with saturation, packed half-word __vector signed short __ABSSH (__vector signed short a)

__ABSB (abs.b)	absolute value, packed byte __vector signed char __ABSB (__vector signed char a)
__ABSDIF (absdif)	absolute value of difference int __ABSDIF (int a, int b)
__ABSDIFH (absdif.h)	absolute value of difference, packed half-word __vector signed short __ABSDIFH (__vector signed short a, __vector signed short b)
__ABSDIFB (absdif.b)	absolute value of difference, packed byte __vector signed char __ABSDIFB (__vector signed char a, __vector signed char b)
__ABSDIFS (absdifs)	absolute value of difference with saturation int __ABSDIFS (int a, int b)
__ABSDIFSH (absdifs.h)	absolute value of difference with saturation, packed half-word __vector signed short __ABSDIFSH (__vector signed short a, __vector signed short b)
__ABSDIFSB	absolute value of difference with saturation, packed byte __vector signed char __ABSDIFSB (__vector signed char a, __vector signed char b)

14.12.2.8 Comparison

__LTB (lt.b)	less than, packed byte __vector signed char __LTB (__vector signed char a, __vector signed char b)
__LTBU (lt.bu)	less than unsigned, packed byte __vector unsigned char __LTBU (__vector unsigned char a, __vector unsigned char b)

__LTH (lt.h)	less than, packed halfword __vector signed short __LTH (__vector signed short a, __vector signed short b)
__LTHU (lt.hu)	less than unsigned, packed half-word __vector unsigned short __LTHU (__vector unsigned short a, __vector unsigned short b)
__EQB (eq.b)	equal, packed byte __vector signed char __EQB (__vector signed char a, __vector signed char b)
__EQH (eq.h)	equal, packed half-word __vector signed short __EQH (__vector signed short a, __vector signed short b)
__EQANYB (eqany.b)	equal any byte int __EQANYB (__vector signed char a, __vector signed char b)
__EQANYH (eqany.h)	equal any half-word int __EQANYH (__vector signed short a, __vector signed short b)

14.12.2.9 Shift, Rotate and Saturated Shift

__ROTATEL (dextr)	rotate left The rotate index b must satisfy $0 \leqslant b \leqslant 31$. Otherwise, the result of this operation will be unspecified int __ROTATEL (int a, int b)
__SHAS (shas)	arithmetic shift with saturation int __SHAS (int a, int b)
__ASHIFTRH (sha.h)	arithmetic shift right, packed half-word __vector signed short __ASHIFTRH (__vector signed short a, int b)

__ASHIFTLH (sha.h)	arithmetic shift left, packed half-word __vector signed short __ASHIFTLH (__vector signed short a, int b)
__LSHIFTRH (sh.h)	logic shift right, packed half-word __vector signed short __LSHIFTRH (__vector signed short a, int b)
__LSHIFTLH (sh.h)	logic shift left, packed half-word __vector signed short __LSHIFTLH (__vector signed short a, int b)

14.12.2.10 Bit Inventory

__PARITY	parity int __PARITY (int a)
__CLS (cls)	count leading signs int __CLS (int a)
__CLSH (cls.h)	count leading signs, packed half-word __vector signed short __CLSH (__vector signed short a)
__CLZ (clz)	count leading zeros. To count leading ones of x write __CLZ (~x) int __CLZ (int a)
__CLZH (clz.h)	count leading zeros, packed half-word. To count leading ones of x write __CLZH (~x) __vector signed short __CLZH (__vector signed short a)
__CTZ	count trailing zeros. To count trailing ones of x write __CTZ (~x) __vector signed short __CTZ (int a)

__FFS	find first set return 0 if a is 0. Otherwise, return one plus the index of the first least significant bit that is set. int __FFS (int a)

14.12.2.11 Core

__MFCR (mfcr)	move from core register unsigned int __MFCR (unsigned int c16)
__MTCR (mtcr)	move to core register void __MTCR (unsigned int c16, unsigned int a)

14.12.3 Examples for Saturation/Packed Instruction

v3.4 The packed arithmetic instructions partition a 32-bit word into several identical objects which can then be fetched, processed and stored in parallel.

The TriCore architecture supports two packed formats. The first format divides a 32-bit word into two 16-bit (halfword) values. Instructions which handle data in this way, are denoted in the instruction memonic by the .h and .hu data type modifiers.

The second packed format divides a 32-bit word into four 8-bit values. Instructions which handle the data in this way, are denoted by the .b and .bu data type modifiers. For example, an addition to individual packed bytes or halfwords with saturation is performed by using adds.[bu] oradd.[hu] instruction, respectively.

The sat instructions can be used to saturate the result of a 32-bit calculation before storing it in a byte or halfword in memory or a register. In the following Example1 the compiler generates a sat.bu instruction for a test of the **unsigned char** boundaries.

Example1

```
unsigned char
satbu (unsigned short z)
{
    return z = (z > 0xff) ? 0xff : z;
}
```

Example2

```
int
adds (int i, int j)
{
    return __ADDS (i, j);
}
```

or

```
unsigned char
satfoo (int i, short s)
{
    /* Add signed saturate and saturate unsigned to a byte */
    return __SATBU (__ADDS (i, s));
}
```

The intrinsic function __ADDS(**int** i, **int** j) will saturate individual **int** to the highest positive 32-bit signed integer on individual overflow, or to the highest negative 32-bit signed integer on individual underflow.

14.12.4 Examples SIMD

In computing, SIMD (Single Instruction, Multiple Data) is a set of operations for efficiently handling large quantities of data in parallel, as in a vector processor or array processor. v3.4

TriCore’s data path supports SIMD operations; the processor can split its data path to process two 16-bit halfwords or four 8-bit bytes. Using its SIMD capabilities, TriCore can execute two 16-bit multiplications with single-cycle throughput, i.e. with a throughput that is twice as high as the multiplication throughput of today’s midrange DSP processors.

With the __vector attribute for a variable, the compiler tries to use the SIMD for arithmetic operations.

A vector consists either of an array of two **short**, or of an array of four **char**. The Figure 14.3 on page 121 shows the summation of two vectors named vector1 and vector2. It is also possible to use SIMD for two-dimensional arrays if one dimension consists of either two **short** or four **char**.

For other types of arrays the __vector is not supported, therefore the compiler will issue an error.

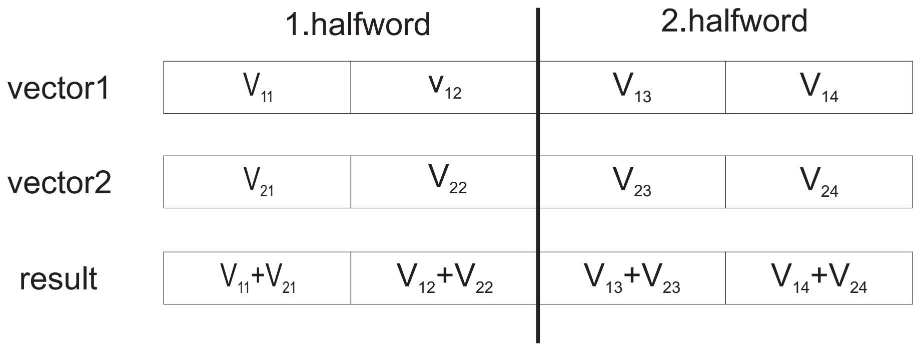

Figure 14.3: Single Instruction Multiple Data

Example

```
__vector unsigned char c1_vector = {10, 11, 12, 13};
__vector unsigned char c2_vector;

__vector unsigned char c1_array_vector[4] = {
    {10, 11, 12, 13},
    {20, 21, 22, 23},
    {30, 31, 32, 33},
    {40, 41, 42, 43}
};

__vector unsigned char c2_array_vector[4];

__vector unsigned short s1_vector = {10, 11};
__vector unsigned short s1_array_vector[4] = {
    {10, 11},
    {20, 21},
    {30, 31},
    {40, 41}
};

__vector unsigned short s2_array_vector[4];
__vector unsigned short s2_vector;

void foo(void)
{
    int i;
    /* Add multiple data */
    for (i = 0; i < 4; i++)
        c2_array_vector[i] += c1_array_vector[i];

    c2_vector = c1_vector;
    for (i = 0; i < sizeof(c2_array_vector) / sizeof(c2_array_vector[0]); i++)
        c2_array_vector[i] = c1_array_vector[i];
}

void sfoo(void)
{
    int i;

    for (i = 0; i < 4; i++)
        s1_vector += s1_array_vector[i];

    s2_vector = s1_vector;
    for (i = 0; i < sizeof(s2_vector) / sizeof(s2_vector[0]); i++)
        s2_array_vector[i] = s1_array_vector[i];
}
```

14.13 Options for testing purposes

These options are for testing purposes only. Do not use them in a productive environment! Some options may blast your temporary directory or the compiler may need very much more time to compile the code!

`-no-delete-temp-files`

`tricore-gcc` will delete the intermediate files that are generated during the compilation of the source code. If you use the option `-no-delete-temp-files` the temporary files will not be deleted. This can be useful if you have problems with intermediate files, that do not occcur together with

`-save-temps`. Do only use this option if you know exactly what you are doing, since temporary files need a lot of space in the temp-directory.

15 PCP Programming

The TriCore uses the following e-flags to identify the TriCore/PCP derivatives:

Name	Value
EF_TRICORE_V1_2	0000 0002H
EF_TRICORE_V1_3	0000 0002H
EF_TRICORE_V1_3_1	0000 0100H

TriCore supports programming of the PCP co-processor as follows:

- The TriCore assembler understands both TriCore and PCP instructions; while TriCore is the default mode, you can easily switch to the PCP mode by using the pseudo-opcodes

  - `.pcptext` or `.pcptext.some_section_name`
  - `.pcpdata` or `.pcpdata.some_section_name`

  you can return to TriCore mode by entering any other section whose name doesn't start with `.pcptext` or `.pcpdata` (e.g., `.text`, `.data`, `.bss`, etc.).

  Note:

  There is no assembler command line option to enter PCP mode, so you have to explicitly use the `ext,data.pcpt` pseudo-opcodes.

- The TriCore linker provides the default output sections `.pcptext` and `.pcpdata` which will be filled with any PCP text and data sections found in the input objects of the current link run; in addition, the start and target addresses, as well as the sizes of these `.pcptext` and `.pcpdata` output sections, will be entered into the `__copy_table`; in addition, the linker can be directed to automatically translate data addresses between TriCore and PCP (as required by certain TriCore MCUs)

- The TriCore startup code (`crt0.o`) automatically copies the `.pcptext` and `.pcpdata` sections contained in a TriCore executable to the PCP's code and data RAM areas, so that all PCP code and data will be in place before the application's `main` function is being executed

- The TriCore debugger `tricore-gdb` and the `tricore-objdump` tool can disassemble PCP code sections; they automatically switch to PCP mode whenever they are requested to disassemble some code section whose name starts with `.pcptext` or `pcptext.`

A TriCore application that uses the PCP would typically consist of some C/C++ code (called by, or directly implemented in the program's `main()` function) that initializes all relevant hardware registers ('SFRs'), plus one or more assembler files (`*.s` or `*.S`) that implement the PCP-specific code and data. Further information can be obtained from the following documents:

- PCP Assembler Mnemonics Specification (Infineon)
- The TriCore Assembler Manual
- Smart Interrupt Service via PCP; Application Note AP32025 (Infineon)

15.1 PCP instruction syntax

15.1.1 Condition codes

cc_A	cc_UC	unconditional
	cc_Z	Zero/Equal
	cc_NZ	Not Zero/Not Equal
	cc_V	Overflow
	cc_C	Carry
	cc_ULT	unsigned less than
	cc_UGT	unsigned greater than
	cc_SLT	signed less than
	cc_SGT	signed greater than
cc_B (all of cc_A and)	cc_N	Negative
	cc_NN	Not Negative
	cc_NV	Not Overflow
	cc_NC	Not Carry
	cc_UGE	unsigned greater than or equal
	cc_SGE	signed greater than or equal
	cc_SLE	signed less than or equal
	cc_CNZ	CNT1 equal zero
	cc_CNN	CNT1 not equal zero
constants	immN	immediate value with width N bits
	offsetN	address offset with width N bits
	addressN	address constant with width N bits

constants can be decimal `[1−9][0−9]∗` or hexadecimal `0x[0−9]+`.

15.1.2 Intruction syntax

ADD	Ra, cc_A
ADD.I	Ra, imm6
ADD.F	Rb, [Ra], Size=[8,16,32]
ADD.PI	Ra, [offset6]
BCOPY	DST[,+.-], SRC[,+,-], CNC=[0,1,2], CNT=[2,4,8]
AND	Rb Ra, cc_A
AND.F	Rb, [Ra], Size=[8,16,32]
AND.PI	Ra, [offset6]
CHKB	Ra, imm5, [SET,CLR]
CLR	Ra, imm5

CLR.F	[Ra],imm5,Size=[8,16,32]
COMP	Rb Ra, cc_A
COMP.I	Ra, imm6
COMP.F	Rb, [Ra], Size=[8,16,32]
COMP.PI	Ra, [offset6]
COPY	DST[,+.-], SRC[,+,-], CNC=[0,1,2], CNT=[2,4,8], Size=[8,16,32]
DEBUG	EDA=[0,1], DAC=[0,1], RTA=[0,1], SDB=[0,1], cc_B
DINIT	Rb, Ra
DSTEP	Rb, Ra
INB	Rb, Ra, cc_A
INB.I	Ra, imm5
EXIT	EC=[0,1],ST=[0,1],INT=[0,1],EP=[0,1], cc_B
JC	offset6, cc_B
JC.A	address16, cc_B
JC.I	Ra, cc_B
JC.IA	Ra, cc_B
JL	offset10
LD.F	Rb, [Ra], Size=[8,16,32]
LD.I	Ra, imm6
LD.IF	[Ra], offset5, SIZE=[8,16,32]
LD.P	Rb, [Ra], cc_A
LD.PI	Ra, offset16
LDL.IL	Ra, imm16
LDL,IU	Ra, imm16
MINIT	Rb, Ra
MSTEP32	Rb, Ra
MSTEP64	Rb, Ra
MOV	Rb, Ra, cc_A
NEG	Rb, Ra, cc_A
NOP	
NOT	Rb, Ra, cc_A
OR	Rb, Ra, cc_A
OR.F	Rb, [Ra], SIZE=[8,16,32]
OR.PI	Ra, [offset6]
PRI	Rb, Ra, cc_A
MCLR.PI	Ra, [offset6]
MSET.PI	Ra, [offset6]
RL	Ra, [1,2,4,8]
RR	Ra, [1,2,4,8]
SET	Ra, imm5
SET.F	[Ra], imm5, SIZE=[8,16,32]
SHL	Ra, [1,2,4,8]
SHR	Ra, [1,2,4,8]
ST.F	Rb, [Ra], Size=[8,16,32]
ST.IF	[Ra], offset5, SIZE=[8,16,32]
ST.P	Rb, [Ra], cc_A
ST.PI	Ra, offset16

SUB	Rb Ra, cc_A
SUB.I	Ra, imm6
SUB.F	Rb, [Ra], Size=[8,16,32]
SUB.PI	Ra, [offset6]
XCH.F	Rb, [Ra], SIZE=[8,16,32]
XCH.PI	Ra, [offset6]
XOR	Rb, Ra, cc_A
XOR.F	Rb, [Ra], SIZE=[8,16,32]
XOR.PI	Ra, [offset6]

Part III

Components of the TriCore Development Platform

16 The TriCore Control Program

16.1 Description

The build process is coordinated by one control program. This control program invokes the necessary tools of the TriCore Development Platform in the correct sequence and passes options and parameters to them. It provides a consistent user interface irrespective of the user's current programming task. Sometimes the control program is called compiler driver.

The control program must be executed in a command line (DOS command line on DOS/Windows based systems, a shell like the bash on UNIX based systems). It accepts lots of options. Most of these options are passed through to appropriate program but some options are used to control the compiler driver itself.

The control program is called `gcc`, which stands for *GNU* Compiler Collection. Since our port of `gcc` is target specific, all executables are preceded by `tricore-gcc`. The resulting command to start the control program for compiling source code for TriCore based microcontrollers is:

```
tricore-gcc <options>
```

To compile programs without concerning the various settings of the tools in the toolchain it is necessary to have defaults. These defaults are set by the control program `tricore-gcc`. It invokes the programs of the toolchain with predefined settings. Among these settings are default include pathes for libraries and header files, a default type of target hardware, if there is more than one possible type of target hardware, or directives to include the startup code for the microcontroller.

Note:

You can find a complete list of the default options for each program of the toolchain in the section "Default options" in the chapters describing the program. These default options are overwritten by user defined options passed to the control program.

Besides managing the options for the tools of the toolchain, `tricore-gcc` is responsible to invoke the right programs in the right order and to start or stop the build process with a given program.

Note:

It is not recommended to start the components of the toolchain individually. The compilation of a program should always be coordinated by the control program.

16.2 Invocation

The control program `tricore-gcc` is controlled by command line options. This chapter describes these options available for the control program of the TriCore Development Platform.

A subset of the options available for a component of the toolchain are accessible from `tricore-gcc`. These options are accepted by the control program and passed to the appropriate component. This chapter describes the options accepted by `tricore-gcc`. The descriptions of these options include, if available, the component the option is passed to.

16.2.1 Overview

For a description of the available options of `tricore-gcc` see the according section.

16.2.2 Options controlling the control program

`--help` Prints a description of the command line options that are accepted by `tricore-gcc`. If the option `-v` is specified in combination with `--help`, then this option is passed to the programs invoked by `tricore-gcc` and a complete command reference of the programs included in the toolchain is generated.

`--target-help` Prints the description of TriCore specific options for each tool of the toolchain.

`-dumpspecs` The behavior of `tricore-gcc` is controlled by so called specs. Per default `tricore-gcc` uses the built in specs. These built in specs can be displayed by this option. For changing the built in specs see option -specs.

`-dumpversion` Displays the version of the compiler

`--version` Displays the version of the compiler and additional copyright information

`-print-search-dirs`
This option prints out the default directories the control program searches for executables and libraries.

`-print-libgcc-file-name`
Displays the name and the path to the library libgcc

`-print-file-name=<library>`
Displays the full path to the library <library>. In a default installation this would be:

```
C:\>tricore-gcc -print-file-name=libgcc.a
C:\<install-dir>\bin\..
\lib\gcc-lib\tricore\<gcc-version>\libgcc.a
```

`-print-prog-name=<program>`
Displays the full path to the toolchain component <program>. In a default installation this would be:

```
C:\>tricore-gcc -print-prog-name=cc1
C:\<install-dir>\tricore\bin\..
\lib\gcc-lib\tricore\<gcc-vercion>\cc1.exe
```

`-save-temps` The files produced while compiling a source file are normally generated in a temporary directory and deleted after the compilation. Using the option `-save-temps` prevents generating the intermediate files in the temporary directory but saves them in the build directory.

`-save-temps-dir=<directory>`
The files produced while compiling a source file are normally generated in a temporary directory and deleted after the compilation. Using the option `-save-temps-dir` prevents generating the intermediate files in the temporary directory but saves them in the directory specified by an user. If you also use -no-integrated-cpp then the `.i` files are generated.

`-Wp,<options>` Pass options to the preprocessor. The options must be specified as comma separated list. This option must be used for all preprocessor options not directly accessible via `tricore-gcc`

`-Wa,<options>` Pass options to the assembler. The options must be specified as comma separated list. This option must be used for all assembler options not directly accessible via `tricore-gcc`

`-Wl,<options>` Pass options to the linker. The options must be specified as comma separated list. This option must be used for all linker options not directly accessible via `tricore-gcc`

`-Xlinker <arg>` Pass one option to the linker. This option <arg> may be either a command line option understood by the linker or an additional object file that should be linked into the executeable.

`-v` Generate verbose output. The programs invoked by the control program and the options passed to them are written to the standard error output. This option also prints the version number of the compiler driver program and the invoked tools.

`-###` Equivalent to -v but the components are not executed. The way they would be started by `tricore-gcc` is printed to the standard error output. This option can be used to check wether the right options are passed to the components without starting them.

`-specs=<file>` This option passes the additional specs file <file> to the control program. The specs in this file define which options are passed to the components of the toolchain. The settings in this specs file override the default settings made by the built in specs. If more than one option `-specs=<file>` is specified the specs files are read from left to right. Also see: -dumpspecs.

`-B <directory>` Adds <directory> to the library search path of the compiler.

`-std=<standard>`

Specify the standard the code should be checked against. Allowed standards are:

iso9899:1990, c89 The ISO C standard from 1990. c89 is the customary shorthand for this version of the standard.

gnu89 the ISO C standard from 1990 plus *GNU* extensions. This is the default.

iso9899:199409 The 1990 C standard, as amended in 1994.

iso9899:1999, iso9899:199x, c99, c9x The revised ISO C standard, published in December 1999. Before publication, this was known as C9X.

gnu99, gnu9x The 1999 ISO C standard plus *GNU* extensions.

`-x <language>` Specify the language of the following input files. Permissable languages are:

`c c++ assembler none`

`none` means revert to the default behavior of guessing the language based on the file's extension. For valid file extensions see Table 1.2 on page 6. Depending on the language of the input files the corresponding compiler is invoked. For C and C++ input files `tricore-gcc` starts the C or C++ compiler respectively. If the input source files contain assembler code the starting of the C or C++ compiler is omitted and only the assembler is started.

`-E` Stop after preprocessing, do not compile anything. The output is preprocessed source code, which is sent to the standard output if the option `-o <file>` is not given. Input files which are not preprocessed (e.g. `.h` or `.i` files) are ignored.

`-S` Stop the toolchain after the compiler. The output is one assembler code file for each input source file. The default file extension for files containing assembler code is .s, so the input file main.c becomes main.c. Input files which are not compiled (e.g. `.s` or `.S` files) are ignored.

`-c` Compile or assemble the source files, but do not link. The toolchain is stopped after the assembler. The output generated at this stage of comiling is a object file for each input file. By default, the object file name for a source file is made by replacing the extension `.c`, `.i`, `.s`, etc., with `.o`. Input files which are not assembled (e.g. `.o`) are ignored.

`-o <file>` Place output in <file>. This applies regardless to whatever sort of output is being produced, whether it be an executable file, an object file, an assembler file or preprocessed C code. Since only one output file can be specified, it does not make sense to use `-o` when compiling more than one input file, unless you are producing an executable file as output. If `-o` is not specified, the default is to put an executable file in `a.out`, the object file for `source.suffix` in `source.o`, its assembler file in `source.s`, and all preprocessed C source on standard output.

`-pass-exit-codes`
: Normally the `tricore-gcc` program will exit with the code of 1 if any phase of the compiler returns a non-success return code. If you specify '-pass-exit-codes', the gcc program will instead return with numerically highest error produced by any phase that returned an error indication.

`-nostartfiles`
: The standard system libraries are used normally, unless `-nostdlib` or `-nodefaultlibs` is set. With `-nostartfiles` the standard system startup files are not used when linking.

`-nodefaultlibs`
: Do not use the standard system libraries when linking. Only the libraries you specify will be passed to the linker. The standard startup files are used normally, unless `-nostartfiles` is used. The compiler may generate calls to memcmp, memset, and memcpy for System V (and ISO C) environments or to bcopy and bzero for BSD environments. These entries are usually resolved by entries in libc. These entry points should be supplied through some other mechanism when this option is specified.

`-nostdlib`
: Only search library directories explicitly specified on the command line. Library directories specified in linker scripts (including linker scripts specified on the command line) are ignored.

16.3 TriCore Derivative Specs File

For the TriCore a derivative specific configuration file called `tricore.specs` is available in `lib/gcc-lib/tricore/<gcc-version>`. In this file you can configure every derivative with specific compiler options (e.g. setting errata for derivative). The derivative is selected by using the compiler option `-mcpu=<derivative>`. Here a code snippet of TriCore specs file.

```
...
*tc1775_errata:
-mcpu018 -mcpu024 -mcpu031 -mcpu034 -mcpu048 -mcpu050
...
*TC1775:
    -mtc12 %(tc1775_errata)
...
```

Note:

All tags must start with a leading ∗. It is also possible to define own tags.

With `%(tc1775_errata)` the contents of `∗tc1775_errata` is substituted. You can access the tag `∗TC1775` by invocation of `tricore-gcc -mcpu=TC1775 ...`

this corresponds to

`tricore-gcc -mtc12 -mcpu018 -mcpu024 -mcpu031 -mcpu034 -mcpu048 -mcpu050 ...`

Note:

With `-mcpu-specs=<file>` it is also possible to specify a <file> with your own configuration settings.

Example

The TriCore derivatives distinguish in memory location, port configuration etc. For this reason often preprocessor statements like

```
#ifdef TC1775
#define MEMCPY_START (0x...)
#endif
```

set the derivative specific settings. To enable the settings the option `-DTC1775` must be passed to the compiler. A more comfortable way is given with the possibility to declare the target specific 'defines' or parameters within a configuration file. For example modify the user specs file `tricore.specs` by adding a `define`.

```
*TC1775:
    -mtc12 %(tc1765_errata) -DTC1775
```

Example for TC1796

```
*tc1796_errata:
-mcpu048=1 -mcpu060 -mcpu069 -mcpu070 -mcpu072 -mcpu076 -mcpu081 -mcpu082 -mcpu083 -mcpu

*tc13_OP1_errata:
-mcpu048=1 -mcpu060 -mcpu069 -mcpu070 -mcpu072 -mcpu081 -mcpu082 -mcpu083 -mcpu095 -mcpu

*tc13_err13x_OP1_errata:
-mcpu048=1 -mcpu060 -mcpu069 -mcpu070 -mcpu072 -mcpu081 -mcpu082 -mcpu083 -mcpu095 -mcpu

*tc1796_OP1_errata:
-mcpu048=1 -mcpu060 -mcpu069 -mcpu070 -mcpu072 -mcpu081 -mcpu082 -mcpu083 -mcpu095 -mcpu

*tc1766_OP1_errata:
-mcpu048=1 -mcpu060 -mcpu069 -mcpu070 -mcpu072 -mcpu081 -mcpu082 -mcpu083 -mcpu095 -mcpu

*TC13_OP1:
  -D__TC13__  -D__TRICORE_NAME__=0x13 -mtc13 %(tc13_OP1_errata)

*TC13_ERR13X_OP1:
  -D__TC13__  -D__TRICORE_NAME__=0x13 -mtc13 %(tc13_err13x_OP1_errata)

*TC13-FPU_OP1:
  -D__TC13_FPU__ -D__TC13__  -D__TRICORE_NAME__=0x13 -mtc13 %(tc13_OP1_errata) -mhard-fl

*TC13-FPU_ERR13X_OP1:
  -D__TC13_FPU__ -D__TC13__  -D__TRICORE_NAME__=0x13 -mtc13 %(tc13_err13x_OP1_errata) -m
```

```
*TC1766_OP1:
  -D__TC1766__  -D__TRICORE_NAME__=0x1766 -mtc13 %(tc1766_OP1_errata) -mhard-float -mcpu

*TC1796_OP1:
  -D__TC1796__  -D__TRICORE_NAME__=0x1796 -mtc13 %(tc1796_OP1_errata) -mhard-float -mcpu
```

17 The TriCore Preprocessor

17.1 Description

The preprocessor is the first program started by `tricore-gcc` in a default compilation process. It analyzes the preprocessor directives like **#include** and removes the comments from the source code. The output generated by this program contains preprocessed source code. If the option `-o <file>` is not set the preprocessed source code is output to the console. If no input file is given to the preprocessor, it reads its input from stdin.

The preprocessor is controlled by the preprocessor directives. By analyzing these directives the preprocessor adds or removes parts of the input source file. This first state of compiling is the same for all targets since the preprocessor only changes the text of the source files. It even is not specific for programming because the functionality of the preprocessor can be used to add, remove and replace text in any text file.

The compiler has a built-in preprocessor. Because of that the preprocessor is invoked as independent process only if the compiler driver program `tricore-gcc` is invoked with the options `-E` or `-save-temps`.

17.2 Invocation

For a description of the available options of preprocessor see the according section.

`-$`
: Do not allow '$' in identifiers.

`-+`
: Allow parsing of C++ style features.

`-A <question>=<answer>`
: Assert the <answer> to <question>. One assertion may have more than one answers. This assertion can be evaluated by the following C construct:

```
#<question> (<answer>)
```

Conditional code can be created using an assertion:

```
#if #<question> (<answer>)
```

`-A -<question>=<answer>`
: Undefines the answer <answer> of <question>.

`-C`
: The comments of the input file are preserved in the output file. This option is accepted by `tricore-gcc` only in combination with the option `-E`.

`-D <macro>`
: Defines the <macro> with the string `1` as its value. This macro can be used together with the **#ifdef** directive.

`-D <macro>=<value>`
Defines the <macro> with <value> as its value.

`-dD` Generates a list of all defined macros and their values before preprocessing the input file. This list does not contain the predefined macros. If the preprocessor is invoked by `tricore-gcc` this option only produces output to standard output if used in conjunction with the option `-E`.

`-dI` Include **#include** directives in the output.

`-dM` Generate a list of all defined macros and their values at the end of the preprocessing instead of the normal output. If the preprocessor is invoked by `tricore-gcc` this option only produces output to standard output if used in conjunction with the option `-E`. Otherwise the output is written to the generated `.i`-file.

`-dN` Generates a list of all defined macros without their values before preprocessing the input file. This list does not contain the predefined macros. If the preprocessor is invoked by `tricore-gcc` this option only produces output to standard output if used in conjunction with the option `-E`.

`-ftabstop=<number>`
Set the number of whitespaces replacing a tabstop in the output.

`-h or --help` Display the online help.

`-H` Print the name of header files in the order they are used. For system header files the absolute path is included.

`-I-` See option `-I` for a description.

`-I <dir>` Add dir to the end of the main include path. If this option is set before the option `-I-` the specified directory will be only searched for header files included by **#include** "<file>". If this option is set after the option `-I-` the directory <dir> will be searched for header files included with both **#include** "<file>" and **#include** <<file>>. In addition, `-I-` inhibits the use of the directory of the current file as the first search directory for **#include** "<file>". The preprocessor automatically checks the content of the system environment variable CPATH. The content of this variable is processed after any of the `-I`-options processed, regardless of the location of the option `-I-`.

`-idirafter <dir>`
Add <dir> to the end of the system include path. The files in <dir> are searched for header files included with **#include** <<file>>.

`-imacros <file>`
The macros defined in <file> are defined before any input file is processed. This option does the same as `-include <file>` except that only the macro definitions are read. No function declarations from

<file> are read. The files specified with `imacros` are processed before the files specified with `include`.

`-include <file>`
: Include the contents of <file> to the preprocessed file before processing the **#include** directives of the input file. Specifying this option has the same effect as inserting **#include** <<file>> at the very beginning of all input files.

`-iprefix <path>`
: For the options `-iwithprefix` and `-iwithprefixbefore` <path> is set as the prefix.

`-isystem <dir>`
: Add dir to the start of the system include path. The directory <dir> is searched for header files before the predefined system include path. By using this option predefined header files can be replaced without changing the source code.

`-iwithprefix <dir>`
: Add <prefix><dir> to the end of the system include path. The prefix must be set by the option `-iprefix` before. The files in <dir> are searched for header files included with **#include** <<file>>.

`-iwithprefixbefore <dir>`
: Add <prefix><dir> to the end of the main include path. The prefix must be set by the option `-iprefix` before. The files in <dir> are searched for header files included with **#include** "<file>".

`-lang-asm`
: By default the language is recognized by its extension. With this option you can assume that the input sources are in assembler, even if a different extension is used.

`-lang-c`
: Assume that the input sources are in C.

`-lang-c++`
: Assume that the input sources are in C++.

`-M`
: This option generates the dependencies of the preprocessed file to include in a makefile.

: After generating this output the toolchain is stopped. No compilation is done.

`-MD`
: Generates `make` dependencies and continues the toolchain after the preprocessor. The make dependencies are output in a file with the extension `.d`.

`-MF <file>`
: Write the dependency output generated by the preprocessor to <file> instead of file <`inputfile`>`.d`

`-MG`
: Treat missing header file as generated files. This allows the preprocessor to generate a make dependency for header files not existing in the source file's directory yet. If the preprocessor is invoked with `-M` and a required header file does not exist, it will exit.

`-MMD`	Generates make dependencies and continues the toolchain after the preprocessor, but ignores system header files.
`-MP`	Generate phony targets for all headers included in the input file.
`-MQ <target>`	The default name of the target created by the option `-M` is the name of the output object file. By using the option `MQ` the target can be renamed to <target>. If <target> contains characters that are recognized by `make` as special characters these characters are quoted.
`-MT <target>`	The same as `-MQ` except that the special characters in the target name are not quoted.
`-nostdinc`	Do not search the predefined system include directories. The user defined directories specified by `-isystem` will be searched for header files.
`-nostdinc++`	Do not search the predefined C++ standard include directories. All other predefined directories are searched for header files.
`-o <file>`	The output generated by the preprocessor is put into <file> instead of stdout. Convention is to write the preprocessor's output into a file with the extension `.i`.
`-P`	Do not generate #line directives and line information in the output.
`-pedantic`	Issue a warning if the source code is not confirm with the standard set by the option `-std`.
`-pedantic-errors`	Issue an error if the source code is not confirm with the standard set by the option `-std`.
`-remap`	Enable special code to work around file systems which only permit very short file names, such as MS-DOS.
`-std=<standard>`	Specify the standard the code should be checked against. Allowed standards are:

iso9899:1990, c89 The ISO C standard from 1990. c89 is the customary shorthand for this version of the standard.

gnu89 the ISO C standard from 1990 plus *GNU* extensions. This is the default.

iso9899:199409 The 1990 C standard, as amended in 1994.

iso9899:1999, iso9899:199x, c99, c9x The revised ISO C standard, published in December 1999. Before publication, this was known as C9X.

gnu99, gnu9x The 1999 ISO C standard plus *GNU* extensions.

`-trigraphs`	Support ISO C trigraphs.

`-U <macro>`	Undefines <macro>.
`--version`	Display version information of the preprocessor.
`-v`	Displays the version number.
`-w`	Suppress all warnings, including those which the preprocessor issues by default.
`-Wall`	Turns on all optional warnings which are desirable for normal code. At present this is `-Wcomment` and `-Wtrigraphs`. Note that many of the preprocessor's warnings are on by default and have no options to control them.
`-Wcomment`	Same as `-Wcomments`.
`-Wcomments`	Warn whenever a comment-start sequence /* appears in a /* comment, or whenever a backslash-newline appears in a // comment.
`-Werror`	Make all warnings into hard errors. Source code which triggers warnings will not be preprocessed and the compilation will be stopped.
`-Wimport`	Warn the first time #import is used.
`-Wno-trigraphs`	Do not warn about trigraphs.
`-Wno-comments`	Do not warn about comments.
`-Wno-traditional`	Do not warn about traditional C.
`-Wno-undef`	Do not warn about testing undefined macros.
`-Wno-import`	Do not warn about the use of #import
`-Wno-error`	Do not treat warnings as errors.
`-Wno-system-headers`	Suppress warnings from system headers.
`-Wsystem-headers`	Issue warnings for code in system headers. These are normally unhelpful in finding bugs in your own code, therefore suppressed. If you are responsible for the system library, you may want to see them.
`-Wtrigraphs`	Warn if trigraphs are encountered. Trigraphs are special control sequences defined in ANSI C for generating special characters. These trigraphs are rarely used and replaced by other characters while preprocessing. They should be avoided if possible.
`-Wtraditional`	Warn about certain constructs that behave differently in traditional and ISO C. Also warn about ISO C constructs that have no traditional C equivalent, and problematic constructs which should be avoided.
`-Wundef`	Warn whenever an identifier which is not a macro is encountered in an #if directive, outside of defined. Such identifiers are replaced by zero.

`-mno-warn-skipped-cpp-directives`
: Don't warn about invalid preprocessor directives in skipped blocks.

`-mwarn-skipped-cpp-directives`
: Warn about invalid preprocessor directives in skipped blocks. This option is not set by default.

17.2.1 Default options and defines

The following options are set by default for the preprocessor by the compiler driver program `tricore-gcc`:

- `-fomit-frame-pointer`
- `-funsigned-bitfields`
- `-quiet`
- `-mversion-info-string=`
- `-mtc12`
- `-mcpu018`
- `-mcpu024`
- `-mcpu031`
- `-mcpu034`
- `-mcpu048=all`
- `-mcpu050`
- `-mcpu060`
- `-mcpu069`
- `-mcpu070`
- `-mcpu072`
- `-auxbase`
- `-iprefix <install-dir>/bin/../lib/gcc-lib/tricore/<gcc-version>/`
- `-isystem`

Where <install-dir> represents the directory where the toolchain is installed to and <gcc-version> stands for the version of the *GNU* compiler collection that provided the base for the TriCore Development Platform.

These defines are set by default either by command line options or by internal settings of the preprocessor:

- –DRIDER_B

- –D__TC12__
- –D__RIDER_B__
- –D__SOFT_FLOAT__
- –DERRATA_CPU8
- –DERRATA_CPU31
- –DERRATA_CPU18
- –DERRATA_CPU34
- –DERRATA_CPU24
- –DERRATA_CPU60
- –DERRATA_CPU69
- –DERRATA_CPU70
- –DERRATA_CPU72
- –DERRATA_CPU16
- –DERRATA_CPU48
- –DERRATA_CPU48_1
- –DERRATA_CPU48_2
- –DERRATA_CPU50
- –D__TC12__
- –D__TRICORE_NAME__=0x12

18 The TriCore Compiler

18.1 Description

The Compiler is the central part of the TriCore Development Platform. It converts the input files written in a high level language such as C or C++ to machine specific assembler instructions. Since these assembler instructions are not interchangeable between different microprocessors a special compiler has to be created for every target platform.

The compiler is divided into two parts: the compiler frontend and the compiler backend. The compiler frontend reads the input source code, analyzes the objects in it and writes their relationship into a tree structure. This tree structure is transformed in a pseudo assembly language called Register Transfer Language (RTL) . This part of the compiler is not machine specific but the same for all target platforms.

After the creation of the RTL code the compiler backend generates code for the target platform. In this state the tree structure generated by the compiler frontend is optimized. Redundant or unused parts of the tree are removed. Parts of the tree may be moved to prevent statements to be executed more than necessary. Additionally some user defined optimizations may be done in this state of the compilation. After all of these tasks have been done the compiler outputs the target specific assembler code.

The TriCore compiler has an integrated preprocessor. That means that the compiler accepts raw source code as input which needs not to be preprocessed. By including the preprocessor in the compiler the processes invoked by `tricore-gcc` are reduced. However the preprocessor is included in the toolchain as stand-alone binary file. This binary is invoked by `tricore-gcc` every time the preprocessed source code is to be output.

> **Note:**
>
> The preprocessor is invoked as process if the options `-E` or `-save-temps` are specified for `tricore-gcc`.

The disadvantage of this feature is the loss of the clear mapping of the preprocessor options specified for `tricore-gcc`. These options are accepted by both the preprocessor as stand-alone binary and the compiler. If the preprocessor is invoked by `tricore-gcc`, these options are passed to it. The built-in preprocessor of the compiler is disabled by the option `-fpreprocessed`. Otherwise the preprocessor options are passed directly to the compiler.

The compiler is controlled by command line options. Most of these options add or remove features to or from the optimization algorithms applied to the RTL code. These options normally start with the character 'f'. Most of the options for adding a feature have a negated form, too for removing the feature. The options for removing the feature are starting with 'fno-' (e.g. `-fasm` for enabling the `asm` statement and `-fno-asm` for disabling the `asm` statement). These options to turn off a feature are often used to remove one feature form a predefined set of features enabled by one option. For example if all features enabled by optimization level 2 should be activated except the inlineing of functions. The resulting options for the compiler would be `-O2` and `fno-inline`.

Equivalent options are available for enabling and disabling warnings. The option enabling warnings start with 'W', the option to disable the respective warning starts with 'Wno-'.

This manual describes only one of the options, either the positive or the negative form.

18.2 Invocation

`-$`
: Do not allow '$' in identifiers.

`-A <question>=<answer>`
: Assert the <answer> to <question>. One assertion may have more than one answers. This assertion can be evaluated by the following C construct:

```
#<question> (<answer>)
```

Conditional code can be created using an assertion:

```
#if #<question> (<answer>)
```

`-A -<question>=<answer>`
: Undefines the answer <answer> of <question>.

`-ansi`
: Features that are not compatible with the ANSI C or ANSI C++ standard are disabled. Among these features are the keywords asm and typeof and some predefined macros like unix and vax. The alternate keywords __asm__, __typeof__, __extension__ and __inline__ continue to work. Comments in C++ style are not recognized any more in C files and the inline keyword is disabled. The option `-ansi` option does not cause non-ISO programs to be rejected gratuitously. For that, option `-pedantic` is required in addition to `-ansi`.

`-aux-info <file>`
: Emit protosize declaration info into <file>.

`-C`
: The comments of the input file are preserved in the output file. This option is accepted by `tricore-gcc` only in combination with the option `-E`

`-D <macro>`
: Defines the <macro> with the string `1` as its value. This macro can be used together with the **#ifdef** directive.

`-D <macro>=<value>`
: Defines the <macro> with <value> as its value.

`-dumpbase <file>`
: Set the base filename for the debug dumps made by `-d` to <file>.

`-falign-functions=<number>`
: Sets the alignment of the starting address of functions to a power of 2 value equal or greater than <number>. This alignment is done only if

it is not necessary to skip more than <number> bytes.

If <number> is set to 16, all functions are aligned on a 16-byte boundary. If <number> is set to 20 the alignment of functions is set to 32 (which is the next power of 2 value) but only if the gap between the original address of the function and the next boundary is smaller than 20 bytes.

The default value for <number> is 2. Because this is the lowest alignment possible for TriCore targets, this option is not affected by using optimization.

`-falign-jumps=<number>`

Sets the alignment of jump targets to a power of 2 value equal or greater than <number>. This alignment is done only if it is not necessary to skip more than <number> bytes.

If <number> is set to 16, all jump targets are aligned on a 16-byte boundary. If <number> is set to 20 the alignment of jump targets is set to 32 (which is the next power of 2 value) but only if the gap between the original address of the target and the next boundary is smaller than 20 bytes.

The default value for <number> is 2. Because this is the lowest alignment possible for TriCore targets, this option is not affected by using optimization.

`-falign-labels=<number>`

Sets the alignment of jump targets to a power of 2 value equal or greater than <number>. This alignment is done only if it is not necessary to skip more than <number> bytes. By using this option the resulting code may be slower and larger because of the insertion of dummy instructions before the jump.

If <number> is set to 16, all jump targets are aligned on a 16-byte boundary. If <number> is set to 20 the alignment of jump targets is set to 32 (which is the next power of 2 value) but only if the gap between the original address of the target and the next boundary is smaller than 20 bytes.

The default value for <number> is 2. Because this is the lowest alignment possible for TriCore targets, this option is not affected by using optimization.

`-falign-loops=<number>`

Sets the alignment of the start of loops to a power of 2 value equal or greater than <number>. This alignment is done only if it is not necessary to skip more than <number> bytes. By using this option the resulting code may be slower and larger because of the insertion of dummy instructions before the jump. But the execution of the loop may be faster because of faster jumps to aligned targets.

If <number> is set to 16, all jump targets are aligned on a 16-byte boundary. If <number> is set to 20 the alignment of jump targets is set to 32 (which is the next power of 2 value) but only if the gap between the original address of the target and the next boundary is smaller than 20 bytes.

The default value for <number> is 2. Because this is the lowest alignment possible for TriCore targets, this option is not affected by using optimisation.

`-fallow-single-precision`
: If the option `-traditional` is used all floating point operations are performed as double precision. To use single precision floating point operations in conjunction with the option `-traditional` the option `-fallow-single-precision` must be used.

`-fargument-alias`
: Parameters to a function may alias each other, that is they are connected via __attribute__ ((alias)) . A parameter may also be an alias of a global variable. This is the default.

`-fargument-noalias`
: Parameters to a function may not alias each other, that is they are connected via __attribute__ ((alias)) . But a parameter may be an alias of a global variable.

`-fargument-noalias-global`
: Parameters to a function may not alias each other, that is they are connected via __attribute__ ((alias)) . A parameter also must not be an alias of a global variable.

`-fbounded-pointers`
: Compile pointers as triples: value, base & end

`-fcaller-saves`
: Enable values to be allocated in registers that will be clobbered by functions. Around the function call extra instructions are generated to save and restore the content of these registers. This extra code is only generated for registers containing useful values.

This feature is enabled by `-O2` , `-O3` and `-Os` and may be reset by `-fno-caller-saves`

`-fcall-saved-<register>`
: Set the register <register> as an allocatable register that keeps its value even after an function call. Functions compiled with this option must save and restore the content of this register. This option must not be specified for registers such as the frame pointer or the stack pointer. Using this option to a register containing the return value of a function will fail, too.

`-fcall-used-<register>`
: Set the register <register> as an allocatable register that gets its value corrupted by function calls. This option must not be specified for reg-

isters such as the frame pointer or the stack pointer. Using this option to a register containing the return value of a function will fail, too.

`-fcond-mismatch`
: Allow conditional expressions created with constructs like `(a ? b : c)` with mismatched types in the second and third arguments. The value of such an expression is void. This option is not supported for C++.

`-fcprop-registers`
: After the register allocation the pattern of data copied into the registers is analyzed to try to find places where it is not necessary to copy a value into a register but to propagate a previous copy forward so it can be used again.

  This feature is enabled by `-O` , `-O1` , `-O2` , `-O3` , `-Os` and may be reset by `-fno-cprop-registers`

`-fcrossjumping`
: Perform cross-jumping transformation. This transformation unifies equivalent code and save code size. The resulting code may or may not perform better than without cross-jumping.

  This feature is enabled by `-O` , `-O1` , `-O2` , `-O3` , `-Os` and may be reset by `-fno-crossjumping`

`-fcse-follow-jumps`
: When running CSE, follow jumps to their targets

  This feature is enabled by `-O2` , `-O3` , `-Os` and may be reset by `-fno-cse-follow-jumps`

`-fcse-skip-blocks`
: When running CSE, follow conditional jumps

  This feature is enabled by `-O2` , `-O3` , `-Os` and may be reset by `-fno-cse-skip-blocks`

`-fdata-sections`
: Items normally placed in the `.data` are placed in their own sections. The section names are created by putting a dot before the name of the data object.

  Also see: `-ffunction-sections`

  Note:

  To avoid the loss of data you must ensure that all the generated sections are specified in the linker script file

`-fdefer-pop`
: The arguments for a function pushed to the stack are not popped off after the return of the function but are defered. The stack is later cleared of all the arguments.

  This feature is enabled by `-O` , `-O1` , `-O2` , `-O3` , `-Os` and may be reset by `-fno-defer-pop`

`-fdelete-null-pointer-checks`
: Use global dataflow analysis to identify and eliminate useless checks for null pointers. The compiler assumes that dereferencing a null pointer would have halted the program. If a pointer is checked after it has already been dereferenced, it cannot be null.

  This feature is enabled by `-O2` , `-O3` and `-Os` and may be reset by `-fno-delete-null-pointer-checks`

`-fdiagnostics-show-location=[once | every-line]`
: Defines how often source information should be printed while outputting diagnostic information (error or warning) on the standard error output. `once` means to output the source location is printed once, `every-line` means to print this information on every diagnostics line. Only implemented for C++ at the moment.

`-fdollars-in-identifier`
: Allow the use of `$` inside identifiers. The use of `$` was allowed in the traditional C standards but the modern standards prohibit them. If `$` must be used in identifiers it should be explicitly allowed.

  Also see: Option `-$` for the compiler and the preprocessor

`address`	Print the address of each node. Usually this is not meaningful as it changes according to the environment and source file. Its primary use is for tying up a dump file with a debug environment.
`slim`	Inhibit dumping of members of a scope or body of a function merely because that scope has been reached. Only dump such items when they are directly reachable by some other path.
`all`	Turn on all options.

`-fdump-unnumbered`
: When doing debugging dumps (option `-d`), suppress instruction numbers and line number note output. This makes it more feasible to use diff on debugging dumps for compiler invocations with different options, in particular with and without option `-g`.

`-feliminate-dwarf2-dups`
: Perform DWARF2 duplicate elimination.

`-fexpensive-optimizations`
: Perform a number of minor optimizations which are relatively expensive, that is they need more execution time. Some optimizations are repeated or are using additional memory.

  This is done in detail:

  - The CSE (Common Subexpression Elimination) is done additionaly after the der GCSE (Global Common Subexpression Eliminitation) and the jump-optimization.

- Some additional tests for the substitution of comples operand expressions in combine and cse are done.
- The search for the best register class while assigning registers to operands is done twice.
- The optimization for choosing registers for operands is done twice
- The compiler stores additional subexpressions. The subexpressions are regarded as additional pseudoregisters and are considered while optimizing. This increases the internal database of the compiler and influences the memory consumption and the execution time, because the compiler has to search a larger database.
- After the reload phase (allocation of registers for memory access) an additional CSE is done for the reload phase.

This feature is enabled by `-O2` , `-O3` , `-Os` and may be reset by `-fno-expensive-optimizations`

`-ffixed-<register>`
Treat the register <register> as fixed register. The code generated by the compiler should never use this register.

`-fforce-addr` Memory addresses must be copied into registers before doing arithmetic on them. This improves the code because addresses needed will have been previously loaded into a register and do not need to be loaded again.

`-fforce-mem` Values must be copied into registers before doing arithmetic on them. This improves the code because values needed will have been previously loaded into a register and do not need to be loaded again.

This feature is enabled by `-O2` , `-O3` and `-Os` and may be reset by `-fno-force-mem`.

`-ffreestanding` The compiled program is to be run in a freestanding environment where the standard libraries may not be available and the execution may not begin with `main ()`. This options sets `-fno-builtin` and is equivalent to `-fno-hosted`.

`-ffunction-cse` Function calls are made with the function address stored in a register. This is the default. Disabling this feature with `-fno-function-cse` will cause each instruction making a call to implicitly include the addess of the function and produce less efficient code.

`-ffunction-sections`
Functions normally placed in the `.text` are placed in their own sections. The section names are created by putting a dot before the name of the function.

Also see: `-fdata-sections`.

> **Note:**
>
> To avoid the loss of data you must ensure that all the generated sections are specified in the linker script file.

`-fgcse`
: Perform the global common subexpression elimination.

  This feature is enabled by `-O2` , `-O3` and `-Os` and may be reset by `-fno-gcse`.

> **Note:**
>
> When compiling a program using computed gotos you may get better runtime performance if you disable the global common subexpression elimination pass by adding `-fno-gcse` to the command line.

`-fgcse-lm`
: Perform global common subexpression elimination optimization by detecting load and store operations inside a loop, in which the load operation is in a form that can be moved in front of the loop and thus only occur once.

`-fgcse-sm`
: When `-fgcse-sm` is enabled, A store motion pass is run after global common subexpression elimination. This pass will attempt to move stores out of loops. When used in conjunction with `-fgcse-lm`, loops containing a load/store sequence can be changed to a load before the loop and a store after the loop. Enabled by default when gcse is enabled.

`-fgnu-linker`
: Output *GNU* ld formatted global initializers

`-fguess-branch-probability`
: Enables guessing of branch probabilities

  This feature is enabled by `-O` , `-O1` , `-O2` , `-O3` , `-Os` and may be reset by `-fno-guess-branch-probability`

`-fhosted`
: Assert that compilation takes place in a hosted environment. This implies `-fbuiltin`. A hosted environment is one in which the entire standard library is available, and in which `main` has a return type of **`int`**.

`-fident`
: The preprocessor will process the #`ident` directives.

`-fif-conversion`
: Attempt to transform conditional jumps into branch-less equivalents. This include use of conditional moves, min, max, set ags and abs instructions, and some tricks doable by standard arithmetics. The use of conditional execution on chips where it is available is controlled by `if-conversion2`.

  This feature is enabled by `-O` , `-O1` , `-O2` , `-O3` , `-Os` and may be reset by `-fno-if-conversion`

`-fif-conversion2`
: Use conditional execution (where available) to transform conditional jumps into branch-less equivalents. This feature is enabled by `-O` , `-O1` , `-O2` , `-O3` , `-Os` and may be reset by `-fno-if-conversion2`

`-finhibit-size-directive`
: Do not output a `.size` assembler directive, or anything else that would cause trouble if the function is split in the middle, and the two halves are placed at locations far apart in memory. This option is used when compiling `crtstuff.c`, you should not need to use it for anything else.

`-finline`
: By using this option the compiler is advised to expand a function inline at the place this function is called. This is valid only for functions declared with the `inline` keyword.

  This feature is enabled by `-O` , `-O1` , `-O2` , `-O3` , `-Os` and may be reset by `-fno-inline`

`-finline-functions`
: The compiler may inline simple functions into their callers. If all calls to a given function are integrated, and the function is declared static, then the function is normally not output as assembler code in its own right.

  Inlining of simple functions is enabled by using the option `-O3` .

  Also see: `-fkeep-inline-functions`.

`-finline-limit=<number>`
: The compiler will not inline functions that require more than <number> pseudo instructions. The default value for <number> is 600.

  You may use the keyword **extern** together with the declarations of inline-functions. In this case, the compiler will always inline this function ignoring any biases you have set using the option `-finline-limit`. This is necessary because the compiler can not rely on the implementation of the extern inline-function in another module. By inlining all of these functions linker errors are prevented.

```
extern inline void foo(void)
{
  /* some code */
}
```

`-finstrument-functions`
: Generate instrumentation calls for entry and exit to functions. Just after function entry and just before function exit, the following profiling functions will be called with the address of the current function and its call site. (On some platforms, __builtin_return_address does not work beyond the current function, so the call site information may not be available to the profiling functions otherwise.)

```
void __cyg_profile_func_enter (void *this_fn, void *call_site);
void __cyg_profile_func_exit (void *this_fn, void *call_site);
```

The first argument is the address of the start of the current function, which may be looked up exactly in the symbol table. This instrumentation is also done for functions expanded inline in other functions. The profiling calls will indicate where, conceptually, the inline function is entered and exited. This means that addressable versions of such functions must be available. If all your uses of a function are expanded inline, this may mean an additional expansion of code size. If you use **extern** inline in your C code, an addressable version of such functions must be provided. (This is normally the case anyways, but if you get lucky and the optimizer always expands the functions inline, you might have gotten away without providing static copies.)

A function may be given the attribute no_instrument_function, in which case this instrumentation will not be done. This can be used, for example, for the profiling functions listed above, high-priority interrupt routines, and any functions from which the profiling functions cannot safely be called (perhaps signal handlers, if the profiling routines generate output or allocate memory).

`-fkeep-inline-functions`
The compiler generates the body of a function even if it is fully inlined.

`-fkeep-static-consts`
Emit variables declared **static const** when optimization is not turned on, even if the variables are not referenced. This option is enabled by default. If the compiler shall check if a variable was referenced, regardless of whether or not optimization is turned on, the option `-fno-keep-static-consts` must be used.

`-fleading-underscore`
This option rewrites every symbol written to the object to have a leading underscore.

Note:
You should be knowing what you do when using this option since problems with linking may occur

`-floop-optimize`
Perform loop optimizations: move constant expressions out of loops, simplify exit test conditions and optionally do strength-reduction and loop unrolling as well.

This feature is enabled by `-O` , `-O1` , `-O2` , `-O3` , `-Os` and may be reset by `-fno-loop-optimize`

`-fmath-errno` Set errno after built-in math functions

`-fmem-report` Makes the compiler print some statistics about permanent memory allocation when it finishes.

`-fmerge-all-constants`
: Attempt to merge identical constants and identical variables. This option implies `-fmerge-constants`. In addition to `-fmerge-constants` this considers e.g. even constant initialized arrays or initialized constant variables with integral or floating point types.

`-fmerge-constants`
: Attempt to merge identical constants (string constants and floating point constants) accross compilation units.

  This feature is enabled by `-O` , `-O1` , `-O2` , `-O3` , `-Os` and may be reset by `-fno-merge-constants`

`-fmessage-length=<number>`
: The diagnostic output of the compiler (errors or warnings) are formatted to be not longer than <number>. If <number> is 0 no line wrapping is done and the output appears on a single line.

`-fmove-all-movables`
: All invariant expressions are moved out of loops. This may result in better code.

`-fno-asm`
: The asm keyword is disabled.

`-fno-builtin`
: Do not recognize any built in functions in C that do not begin with __builtin_ as prefix. GCC normally generates special code to handle certain built-in functions more efficiently; for instance, calls to alloca may become single instructions that adjust the stack directly, and calls to memcpy may become inline copy loops. The resulting code is often both smaller and faster, but since the function calls no longer appear as such, you cannot set a breakpoint on those calls, nor can you change the behavior of the functions by linking with a different library. In C++, `-fno-builtin` is always in effect. The `-fbuiltin` option has no effect. Therefore, in C++, the only way to get the optimization benefits of built-in functions is to call the function using the __builtin_ prefix. The GNU C++ Standard Library uses built-in functions to implement many functions (like std::strchr), so that you automatically get efficient code. With the `-fno-builtin-function` option, not available when compiling C++, only the built-in function function is disabled. function must not begin with __builtin_. If a function is named this is not built-in in this version of GCC, this option is ignored. There is no corresponding `-fbuiltin-function` option; if you wish to enable built-in functions selectively when using `-fno-builtin` or `-ffreestanding`, you may define macros such as:

```
#define abs(n) __builtin_abs ((n))
#define strcpy(d, s) __builtin_strcpy ((d), (s))
```

`-fno-common`
: In C, allocate even uninitialized global variables in the data section of the object file, rather than generating them as common blocks. This has the effect that if the same variable is declared (without extern) in two different compilations, you will get an error when you link them.

The only reason this might be useful is if you wish to verify that the program will work on other systems which always work this way.

`-fomit-frame-pointer`
: Do not store the frame pointer in a register for functions that do not need one. By doing this the code to store and retrieve the address is omitted and another register is available for general use.

`-foptimize-register-move`
: Register allocation is optimized by changing the assignment of registers used in operations that move data from one memory location to another. This is especially effective on machines that have instructions that can move data directly from one memory location to another.

This feature is enabled by `-O2` , `-O3` and `-Os` and may be reset by `-fno-optimize-register-move`

`-foptimize-sibling-calls`
: Optimize sibling and tail recursive calls. This code is an example of an recursive tail call:

```
int foo (int bar1, int bar2) {

  ...
  return foo (bar1 + 2, bar2 - 3);

}
```

By replacing the call for **return** foo (bar1 + 2, bar2 − 3); with a jump command to the top of the function this code can be optimized. A similar situation is shown in this code describing a sibling call:

```
int foo1 (int bar1, int bar2) {

  ...
  return foo2 (bar1 - 2, bar2 + 3);

}
```

In a sibling call, the function foo2 must be called but the stack frame of foo1 can be deleted. foo2 will return its return value directly to the caller of foo1.

This feature is enabled by `-O2` , `-O3` and `-Os` and may be reset by `-fno-optimize-sibling-calls`

`-fpack-struct` Members of structures are packed together in such a way that no space is wasted between them due to alignment rules. Specifying this option is the same as adding the attribute __attribute__ ((packed)) to all structures.

`-fpcc-struct-return`
: Generate code to return all structures in memory rather than in registers, even if they would fit in a register.

Also see: `freg-struct-return`

`-fpeephole` Enables peephole optimization at the time the assembly language is output by the compiler. The compiler searches for patterns of instructions and replaces them with optimized ones. This option is set as default.

`-fpeephole2` Enables RTL peephole optimization after registers have been allocated but before scheduling. The compiler searches for patterns of instructions and replaces them with optimized ones. This option only has effect if optimization is turned on.

This feature is enabled by `-O2` , `-O3` , `-Os` and may be reset by `-fno-peephole2`

`-fprefetch-loop-arrays`
If supported by the target machine, generate instructions to prefetch memory to improve the performance of loops that access large arrays.

`-fpreprocessed` Treat the input file as already preprocessed and disable the built-in preprocessor of the compiler. This option is passed to the compiler if the control program `tricore-gcc` is invoked with `-E` or `-save-temps`.

`-fpretend-float`
Pretend that host and target use the same FP format host and target.

`-fprofile-arcs` When the program compiled with `-fprofile-arcs` exits it saves arc execution counts to a file called `<sourcename>.da` for each source file. The information in this data file is very dependent on the structure of the generated code, so you must use the same source code and the same optimization options for both compilations. The other use of `-fprofile-arcs` is for use with `tricore-gcov`, when it is used with the `-ftest-coverage` option. These options used together create a flow graph for each function in the program, then finds a spanning tree for the graph. Only arcs that are not on the spanning tree have to be instrumented: the compiler adds code to count the number of times that these arcs are executed. When an arc is the only exit or only entrance to a block, the instrumentation code can be added to the block; otherwise, a new basic block must be created to hold the instrumentation code.

`-freduce-all-givs`
Forces all general induction variables (loop counters) to be strength reduced. If setting this option produces better or worse code is highly dependent on the structure of the loops in the source code.

`-fregmove` Equivalent to `-foptimize-register-move`

This feature is enabled by `-O2` , `-O3` , `-Os` and may be reset by `-fno-regmove`

`-freg-struct-return`
Generate code to return small structures in registers, if they are not too large.

Also see: `-fpcc-struct-return`

`-frename-registers`
: This optimization algorithm attempts to eliminate false dependencies in scheduled code by making use of registers left over after register allocation. This optimization will most benefit processors with lots of registers. It can, however, make debugging impossible, since variables will no longer stay in a 'home register'.

  This feature is enabled with `-O3` . With `-fno-rename-registers` it can be reset.

`-freorder-blocks`
: Reorder basic blocks to improve code placement.

  This feature is enabled by `-O2` and `-O3` and may be reset by `-fno-reorder-blocks`

`-freorder-functions`
: Reorder basic blocks in the compiled function in order to reduce number of taken branches and improve code locality. This is implemented by using special subsections text.hot for most frequently executed functions and text.unlikely for unlikely executed functions. Reordering is done by the linker so object file format must support named sections and linker must place them in a reasonable way. Also profile feedback must be available in to make this option effective. See `-fprofile-arcs` for details.

  This feature is enabled by `-O2` , `-O3` , `-Os` and may be reset by `-fno-reorder-functions`

`-frerun-cse-after-loop`
: Since loop optimizations may create new subexpressions it may desirable to rerun the common subexpression elimination a second time after the loop optimization. This is done by setting this option.

  This feature is enabled by `-O2` , `-O3` and `-Os` and may be reset by `-fno-rerun-cse-after-loop`.

`-frerun-loop-opt`
: Run the loop optimizer twice.

  This feature is enabled by `-O2` , `-O3` and `-Os` and may be reset by `-fno-rerun-loop-opt`

`-fsched-interblock`
: Enable scheduling across basic blocks.

`-fsched-spec`
: Allow speculative motion of non-load instructions.

`-fsched-spec-load`
: Allow speculative motion of some loads.

`-fsched-spec-load-dangerous`
: Allow speculative motion of more loads.

`-fschedule-insns`

Some instructions depend on data. If this data is not available at the time the instruction is executed, the processor must wait until this data can be read. To avoid these latency times, the instructions can be rescheduled. By doing rescheduling other instructions can be executed while the necessary input data for a pending instruction is collected. By using the option `-fschedule-insns`, the rescheduling is done before register allocation.

This feature is enabled by `-O2` , `-O3` , `-Os` and may be reset by `-fno-schedule-insns`

`-fschedule-insns2`

Some instructions depend on data. If this data is not available at the time the instruction is executed, the processor must wait until this data can be read. To avoid these latency times, the instructions can be rescheduled. By doing rescheduling other instructions can be executed while the necessary input data for a pending instruction is collected. By using the option `-fschedule-insns2`, the rescheduling is done after register allocation.

This feature is enabled by `-O2` , `-O3` , `-Os` and may be reset by `-fno-schedule-insns2`

`-fsched-verbose=<number>`

On targets that use instruction scheduling, this option controls the amount of debugging output the scheduler prints. This information is written to standard error, unless `-dS` `-dS` or `-dR` is specified, in which case it is output to the usual dump listing file, `.sched` or `.sched2` respectively.

However for <number> greater than nine, the output is always printed to standard error. For <number> greater than zero, `-fsched-verbose` outputs the same information as `-dRS`. For <number> greater than one, it also output basic block probabilities, detailed ready list information and unit/insn info. For <number> greater than two, it includes RTL at abort point, control-flow and regions info. And for <number> over four, `-fsched-verbose` also includes dependence info.

`-fshared-data` Mark data as shared rather than private. On operating systems where shared data is accessible among separate processes running the same program and each process has its own copy of private data

`-fshort-double` Use the same size for **double** as for **float**.

`-fshort-enums` Allocate to an enum type only as many bytes as it needs for the declared range of possible values. Specifically, the enum type will be equivalent to the smallest integer type which has enough room.

> **Note:**
>
> The `-fshort-enums` switch causes `tricore-gcc` to generate code that is not binary compatible with code generated without that switch. Use it to conform to a non-default application binary interface.

`-fshort-wchar` Changes the type of `wchar_t` from the target specific default type to **unsigned short**

`-fsigned-char` Variables of the type **char** are signed by default.

Also see: `-funsigned-char`

`-fsingle-precision-constant`
Constant declarations of floating point numbers are stored as single precision floating point numbers instead of doubles.

`-fstack-check` Generate code to verify that you do not go beyond the boundary of the stack. You should specify this flag if you are running in an environment with multiple threads, but only rarely need to specify it in a single-threaded environment since stack overflow is automatically detected on nearly all systems if there is only one stack.

> **Note:**
>
> Note that this switch does not actually cause checking to be done; the operating system must do that. The switch causes generation of code to ensure that the operating system sees the stack being extended. Insert stack checking code into the program

`-fstrength-reduce`
Perform the optimizations of loop strength reduction and elimination of iteration variables. This optimization feature replaces time-consuming instructions such as multiply and divide by faster instructions such as add and subtract:

```
for {int i = 0; i < 10; i++} {
  index = 2 * i;
  foo = table[index];
}
```

This optimization algorithm may replace this code snippet by this one:

```
for {int i = 0; i < 10; i++} {
  foo = table[i << 1];
}
```

This feature is enabled by `-O2`, `-O3` and `-Os` and may be reset by `-fno-strength-reduce`

`-fstrict-aliasing`
Allows the compiler to assume the strictest aliasing rules applicable. This activates optimizations based on the type of expressions. In particular, an object of one type is assumed never to reside at the same

address as an object of a different type, unless the types are almost the same. For example, an **unsigned int** can alias an **int**, but not a **void**∗ or a **double**. A character type may alias any other type.

This feature is enabled by `-O2` , `-O3` , `-Os` and may be reset by `-fno-strict-aliasing`

`-fsyntax-only` Check for syntax errors, then stop the toolchain.

`-ftabstop=<number>`
Set the number of whitespaces replacing a tabstop in the output.

`-ftest-coverage`
Create data files for the `tricore-gcov` code-coverage utility. The data file names begin with the name of your source file:

`<sourcename>.bb` A mapping from basic blocks to line numbers, which `tricore-gcov` uses to associate basic block execution counts with line numbers.

`<sourcename>.bbg` A list of all arcs in the program flow graph. This allows `tricore-gcov` to reconstruct the program flow graph, so that it can compute all basic block and arc execution counts from the information in the `sourcename.da` file.

`<sourcename>.da` Runtime arc execution counts, used in conjunction with the arc information in the file sourcename.bbg.

Use `-ftest-coverage` with `-fprofile-arcs` . `-fprofile-arcs` adds instrumentation to the program, which then writes execution counts to the `.da`-file.

Coverage data will map better to the source files if `-ftest-coverage` is used without optimization.

`-fthread-jumps` If the value and the target of a conditional jump are such that the target of the jump is another jump and this jump will also be taken, the first jump is redirected to the target of the second jump.

This feature is enabled by any level of optimization set by This feature is enabled by `-O` , `-O1` , `-O2` , `-O3` , `-Os` and may be reset by `-fno-thread-jumps`.

`-ftime-report` Report time taken by each compiler pass at end of run

`-funroll-all-loops`
Unroll all loops even if the number of cycles is not known.

`-funroll-loops` If the number of cycles of a loop is known at compile time and the number of instructions in the loop is not too large the loop is unrolled. The code of the loop is copied as often as the loop is run so that the instructions are executed the correct number of times. A loop is considered small enough to be unrolled if the number of instructions multiplied by the number of cycles is less than 100. Using this option may result in faster code.

`-funsafe-math-optimizations`
Allow math optimizations that may violate IEEE or ANSI standards.

`-funsigned-bitfields`
By using this option bit fields are treated as **unsigned int** data types. This option is set if `-traditional` is set.

`-funsigned-char`
Variables of the type **char** are unsigned by default.

Also see: `-fsigned-char`

`-fverbose-asm` More verbose information is added to the output of the compiler. This information makes the output more readable.

`-fwritable-strings`
The compiler allows data to be written into string constants, the strings are stored in writable data sections.

`-fzero-initialized-in-bss`
By default variables that are not initialized to zero are placed in the .bss section. Variables initialized to zero will be placed in the data section if the option '-fzero-initialized-in-bss' is *not* enabled. If the target supports a .bss section, `-fzero-initialized-in-bss` advises `tricore-gcc` to put variables that are initialized to zero into .bss. This option is set by default and can save space in the resulting code. The option `-fno-zero-initialized-in-bss` turns off this behavior because some programs explicitly rely on variables going to the data section. E.g., so that the resulting executable can find the beginning of that section and/or make assumptions based on that.

`-g[<level>]` Generate debugging information in the default debug format.

For TriCore the only debug format supported is DWARF v2.

The optional parameter <level> sets the debug level and defines which information should be included in the output file. Debug level 1 includes global information for making backtraces. This includes descriptions of functions and external variables, but no information about local variables and no line numbers. Level 2, which is the default, includes the information of debug level 1 plus information about local variables and line numbers. Level 3 adds to the information of level 2 extra information, such as all the macro definitions present in the program.

`-gdwarf-2` Generate DWARF-2 debug info. This option does not support the setting of the debug level by adding <level> as parameter to this option.

> **Note:**
>
> If DWARF v2 debug information in debug level 3 shall be included in the assembler code the debug level must be set by using the option `-g<level>`

`-ggdb` Generate debugging info in default extended format.

`-h or --help` Display the online help.

`-H` Print the name of header files in the order they are used. For system header files the absolute path to the files is includes.

`-I <dir>` Add dir to the end of the main include path. If this option is set before the option `-I-` the specified directory will be only searched for header files included by **#include** "<file>". If this option is set after the option `-I-` the directory <dir> will be searched for header files included with both **#include** "<file>" and **#include** <<file>>. In addition, `-I-` inhibits the use of the directory of the current file as the first search directory for **#include** "<file>"

`-I-` See option `-I` for a description.

`-idirafter <dir>`
Add <dir> to the end of the system include path. The files in <dir> are searched for header files included with **#include** <<file>>.

`-imacros <file>`
The macros defined in <file> are defined before any input file is processed. This option does the same as `-include <file>` except that only the macro definitions are read. No function declarations from <file> are read. The files specified with `imacros` are processed before the files specified with `include`

`-include <file>`
Include the contents of <file> to the preprocessed file before processing the **#include** directives of the input file. Specifying this option has the same effect as inserting **#include** <<file>> at the very beginning of all input files.

`-iprefix <path>`
For the options `-iwithprefix` and `-iwithprefixbefore` <path> is set as the prefix.

`-isystem <dir>` Add <dir> to the start of the system include path. The directory <dir> is searched for header files before the predefined system include path. By using this option predefined header files can be replaced without changing the source code.

`-iwithprefix <dir>`
Add <prefix><dir> to the end of the system include path. The prefix must be set by the option `-iprefix` before. The files in <dir> are searched for header files included with **#include** <<file>>.

`-iwithprefixbefore <dir>`
Add <prefix><dir> to the end of the main include path. The prefix must be set by the option `-iprefix` before. The files in <dir> are searched for header files included with **#include** "<file>".

`-lang-asm`
By default assembler files are recognized by the extensions. With this option the compiler assume that the specified input sources are in assembler, even if the use a different extension.

`-lang-c`
Assume that the input sources are in C.

`-lang-c++`
Assume that the input sources are in C++.

`-lang-c89`
Assume that the input sources are in C89.

`-lang-objc`
Assume that the input sources are in ObjectiveC.

`-lang-objc++`
Assume that the input sources are in ObjectiveC++.

`-M`
This option generates the dependencies of the preprocessed file to include in a makefile. After generating this output the toolchain is stopped. No compilation is done.

`-MD`
Generates `make` dependencies and continues the toolchain after the compiler. The make dependencies are output in a file with the extension `.d`.

`-MF <file>`
Write the dependency output generated by the preprocessor to <file> instead of file <`inputfile`>`.d`

`-MG`
Treat missing header file as generated files. This allows the compiler to generate a make dependency for header files not existing in the source file's directory yet. If the compiler is invoked with `-M` and a required header file does not exist, it will exit.

`-MM`
Generates make dependencies and stops the toolchain after doing this, but ignores system header files

`-MMD`
Generates make dependencies and continues the toolchain after doing this, but ignores system header files

`-MP`
Generate phony targets for all headers included in the input file.

`-MQ <target>`
The default name of the target created by the option `-M` is the name of the output object file. By using the option `MQ` the target can be renamed to <target>. If <target> contains characters that are recognized by `make` as special characters these characters are quoted.

`-MT <target>`
The same as `-MQ` except that the special characters in the target name are not quoted.

`-no-integrated-cpp`
Performs a compilation in two passes: preprocessing and compiling. This option allows a user supplied "cc1", "cc1plus", or "cc1obj" via

the `-B` option. The user supplied compilation step can then add in an additional preprocessing step after normal preprocessing but before compiling. The default is to use the integrated cpp (internal cpp) The semantics of this option will change if "cc1", "cc1plus", and "cc1obj" are merged.

`-nostdinc` Do not search the predefined system include directories. The user defined directories specified by `-isystem` will be searched for header files.

`-nostdinc++` Do not search the predefined C++ standard include directories. All other predefined directories are searched for header files.

`-o <file>` The output generated by the compiler is put into <file>. Convention is to write the compiler's output into a file with the extension `.s`

`-O<level>` Set the level of optimization. For each level more optimization algorithms are activated.

`-O0` Disables all optimization. Turns of all size optimization.

`-O` The same as `-O1`.

`-O1` The compiler tries to reduce code size and execution time, without performing any optimizations that take a great deal of compilation time. This implies these options:

```
-fcprop-registers   -fcrossjumping
-fdefer-pop         -fguess-branch-probability
-fif-conversion     -fif-conversion2
-floop-optimize     -fmerge-constants
-fthread-jumps
```

Additionally all functions declared with the keyword `inline` are expanded inline. For details see chapter 7 on page 45.

`-O2` The compiler turns on nearly all optimization algorithms that do not involve size and speed trade-offs. By using this optimization level all algorithms activated for level 1 are activated plus these:

```
-fcaller-saves               -fcse-follow-jumps
-fcse-skip-blocks            -fdelete-null-pointer-checks
-fexpensive-optimizations    -fforce-mem
-fgcse                       -foptimize-register-move
-foptimize-sibling-calls     -fpeephole2
-fregmove                    -freorder-blocks
-freorder-functions          -frerun-cse-after-loop
-frerun-loop-opt             -fschedule-insns
-fschedule-insns2            -fstrength-reduce
-fstrict-aliasing
```

`-O3` Optimize even more. The algorithms for optimization level 3 are the same as for level 2 plus

`-frename-registers`

Additionally small functions are expanded automatically as inline function if their size does not exceed the value set by the option `-finline-limit=<number>`. For details see chapter 7 on page 45.

`-Os` Optimize for space rather than speed. The same optimization options as for level 2 are set except the option `-freorder-blocks` and some internal optimization flags. The values for align_loops, align_jumps, align_labels, align_functions are set to 1, which results in a minimal alignment of $2^1 = 2$. Additionally the value of MOVE_RATIO is changed from 15 to 3. If MOVE_RATIO is exceeded for copying non-scalar types the function memcpy is used instead of a move. The second value, that is changed, is the threshold, which is used for inlining functions. The threshold is set to 1 instruction plus 1.5 instructions per argument for the function. The default is 8 per argument. This threshold is used by the compiler to check if it is possible to implicitly inline a function which has not the keyword inline set. The higher the threshold gets, the more probable the inlining of the function becomes.

This threshold is used to check in general, if a function should be expanded inline. If it is actually inlined is determined by the inline limit, which may be set using the option `-finline-limit`. If a function has a lot of arguments but has only very few code it is better for the compiler to expand this function inline because the preparation of the arguments (that is writing them to registers or on the stack) is much more expensive for the compiler than the inlined code. If `-Os` is set only the smallest functions with few arguments are expanded inline because the inling will always need space on the target.

Note:

Choosing a higher level of optimization does not imply to generate 'better' code since the algorithms activated in higher levels may compete with others. The option `-O2` use speed optimization strategies to optimize the size of your code use the option `-Os`.

`-P` Do not generate #line directives and line information in the output.

`-p` Enable function profiling

`-pedantic` Issue all all the warnings demanded by strict ISO C and ISO C++; reject all programs that use forbidden extensions, and some other programs that do not follow ISO C and ISO C++. For ISO C, follows the version of the ISO C standard specified by any `-std` option used. Valid ISO C and ISO C++ programs should compile properly with or without this option (though a rare few will require `-ansi` or a `-std` option specifying the required version of ISO C). However, without this option, certain GNU extensions and traditional C and C++ features are supported as well. With this option, they are rejected. `-pedantic` does not cause warning messages for use of the alternate keywords whose names begin and end with __. Pedantic warnings are also disabled in the expression that follows __extension__. However, only system header files should use these escape routes; application programs should avoid them. Some users try to use `-pedantic` to check programs for strict ISO C conformance. They soon find that it does not do quite what they want: it finds some non-ISO practices, but not all-only those for which ISO C requires a diagnostic, and some others for which diagnostics have been added. A feature to report any failure to conform to ISO C might be useful in some instances, but would require considerable additional work and would be quite different from `-pedantic`. Where the standard specified with `-std` represents a GNU extended dialect of C, such as 'gnu89' or 'gnu99', there is a corresponding base standard, the version of ISO C on which the GNU extended dialect is based. Warnings from `-pedantic` are given where they are required by the base standard. (It would not make sense for such warnings to be given only for features not in the specified GNU C dialect, since by definition the GNU dialects of C include all features the compiler supports with the given option, and there would be nothing to warn about.)

`-pedantic-errors`
Issue an error if the source code is not confirm with the standard set by the option `-std`.

`-quiet` Do not display functions compiled or elapsed time

`-remap` Enable special code to work around file systems which only permit very short file names, such as MS-DOS.

`-std=<stdandard>`
Specify the standard the code should be checked against. Allowed standards are:

iso9899:1990, c89 The ISO C standard from 1990. c89 is the customary shorthand for this version of the standard.

gnu89 the ISO C standard from 1990 plus *GNU* extensions. This is the default.

iso9899:199409 The 1990 C standard, as amended in 1994.

iso9899:1999, iso9899:199x, c99, c9x The revised ISO C standard, published in December 1999. Before publication, this was known as C9X.

gnu99, gnu9x The 1999 ISO C standard plus *GNU* extensions.

`-traditional` Attempt to support traditional K&R style C

`-trigraphs` Support ISO C trigraphs

`-U <macro>` Undefines <macro>

`-v` Outputs verbose information about the program compiled. Among these information are the version and the release number.

`--version` Display version and release number information of the compiler.

`-w` Suppress all warnings, including those which the compiler issues by default. The same as `--no-warnings`

`-W` Enable additional warnings. The warnings enables by `-W` are issued because of code that could possibly cause a problem. This code may although be knowingly written. The following warnings are enabled:

Aggregate initializers

Warn if initial values of an aggregate are specified and values are not set for all aggregate members. The following example will cause a warning to be issued:

```
int table[10] = {42, 42, 42, 42};
```

Comparison

- A warning is issued if a unsigned variable is tested to be less than zero. Unsigned variables can never be less than zero so the conditional will always be false.
- The compiler warns if a signed value is compared with an unsigned one. An incorrect result can be produced when the unsigned value is converted to a signed value for the comparison. These warnings are also issued if `-Wsign-compare` is set and may be reset by `-Wno-sign-compare`
- Algebraic notation and C syntax are not the same. If the following comparison is detected, a warning will be issued:

  ```
  if (a < b < c)
  ```

  In algebraic notation this statement is only true if the value of `b` lies between the values of `a` and `c`. In C syntax this statement corresponds with the following C code:

  ```
  int temp;

  temp = a < b;

  if (temp < c) {
    ...
  }
  ```

Constant return

Issue a warning if the return value of a function is declared **const**.

No side effects

A warning is issued for if a statement has no effect:

```
int foo;

foo + 42;
```

Return value

Output a warning if a function may or may not return a value.

Static syntax

Issue a warning if the keyword **static** is not the first word in a declaration line.

Unused arguments

Warn if a function does not use a parameter. These warnings are activated if the option `-W` is used together with `-Wall` or `-Wunused` set and may be reset by `-Wno-unused`.

`-Waggregate-return`
Warn if a function returns a structure, union or array.

`-Wall` Enable almost all warnings.

`-Wbad-function-cast`
Warn whenever a function is cast to an non-matching type. The following example will cause a warning:

```
int foo () {
  return (42);
}

char *bar;

bar = (char *) foo();
```

`-Wcast-align` On some machines some variables have predefined alignments. For example variables of the type **int** may only be aligned to 2 or 4-byte boundaries. If a pointer is casted to a type with a potentially higher alignment.

`-Wcast-qual` Warn about casts which discard qualifiers

`-Wchar-subscripts`
Warn about subscripts whose type is **char**. Since chars often defaults to being **signed**, this can be the cause of an error

`-Wcomments and -Wcomment`
Warn whenever a comment-start sequence /* appears in a /* comment, or whenever a backslash-newline appears in a // comment.

`-Wconversion` If the prototype of a variable conflicts with with a type conversion a warning is generated. This includes conflicts while converting real and integer types, signed and unsigned values and changing the width of values. Warnings are issued only for implicit conversions but not for specific casts. In the following example the first of the assignment statements will issue a warning, the second wont:

```
unsigned char foo;

foo = -42;
foo = (unsigned char) -42;
```

`-Wdeprecated-declarations`
Warn whenever a variable or function with __attribute__ ((deprecated)) is encountered.

`-Wdisabled-optimization`
If a requested optimization could not be done a warning is issued. `tricore-gcc` will refuse to perform optimizations that are too complex or will take too long.

`-Werror` Make all warnings into hard errors. Source code which triggers warnings will not be preprocessed and the compilation will be stopped.

`-Werror-implicit-function-declaration`
Give an `error` whenever a function is used before being declared. With the option `-Wimplicit-function-declaration` a warning instead of an error is issued.

`-Wfloat-equal` If two floating point numbers are compared for equality a warning is generated. Comparing two floating point numbers for equality may result in an error in the program since two calculated floats rarely are exactly equal.

`-Wformat` Check the calls of printf, scanf, strftime and strfmon and issue a warning if the argument types do not match the ones set in the formatting string.

`-Wformat-nonliteral`
Warn if `-Wformat` is set and the formatting string on a function like printf or scanf is not a literal string.

`-Wformat-security`
Warn if `-Wformat` is set and the formatting string on a function like printf or scanf is not a constant but a variable. This is regarded as a security problem because these variable strings may contain a %n.

`-Wimplicit-function-declaration`
Warn whenever a function is used before it is declared. The option `-Werror-implicit-function-declaration` issues an error instead of a warning.

`-Wimplicit-int` Warn when a declaration does not specify a type.

`-Wimport` Warn the first time #import is used.

-Winline Warn when a function declared as inline cannot be inlined.

-Wlarger-than-<number>
Warn if an object is larger than <number> bytes.

-Wmain Issue a warning if the definition of main() looks suspicious. main() should have no, two or three arguments of the appropriate type and should return an **int**.

-Wmissing-braces
Print out a warning if the inital values of an array are not completely bracketed. The warning is reported only once even if more than one braces is missing.

```
typedef struct {
        unsigned int params[2][3];
        unsigned char buffer[2][2];
  unsigned char vector[2];
} Info_t;

Info_t Info = {
        .params = {1, 2, 3, 4, 5, 6},  /* missing braces */
        .buffer = {1, 2, 3, 4},    /* missing braces */
  .vector = {1, 2}
};
```

The first wrong initialization is in the array Info.params. The correct code looks like:

```
  ...
  .params = {{1, 2, 3}, {4, 5, 6}},
  ...
```

You have to correct the first error to get a warning for the second wrong initialization of array Info.buffer. The correct code looks like.

```
  ...
  .buffer = {{1, 2}, {3, 4}},
  ...
```

-Wmissing-declarations
Warn about global functions without previous declarations.

-Wmissing-format-attribute
Issue a warning if a function is encountered which may be candidate for the Attribute format. This option is ignored if not used together with the option -Wformat.

-Wmissing-noreturn
Issue a warning if a function is encountered which may be candidate for the Attribute noreturn.

-Wmissing-prototypes
Warn if a global function defined without a previous prototype declaration.

-Wmultichar If a variable of the type **char** is assigned a multicharacter literal such as 'ab' or 'ht' a warning is generated.e

`-Wnested-externs`
: Warn about **extern** declarations inside a function.

`-Wno-comments`
: Do not warn about comments.

`-Wno-error`
: Do not treat warnings as errors

`-Wno-format-extra-args`
: Do not output a warning if too many arguments are passed to function calls such as printf or scanf. This option is ignored if not used together with the option `-Wformat`.

`-Wno-format-y2k`
: Don't warn about strftime formats that produce 2 digit years.

`-Wno-import`
: Do not warn about the use of #import.

`-Wno-long-long`
: Do not warn about using the data type **long long** when the option `-pedantic` is set, too

`-Wno-system-headers`
: Suppress warnings from system headers.

`-Wno-traditional`
: Do not warn about traditional C

`-Wno-trigraphs`
: Do not warn about trigraphs

`-Wno-undef`
: Do not warn about testing undefined macros.

`-Wpacked`
: If a **struct** has the attribute packed set but this attribute has no effect, a warning is generated. The **struct** in the following example has the same size with or without the attribute packed set:

```
struct foo {
  int foo_int;
  short foo_short;
  char foo_char1;
  char foo_char2;
} __attribute__ ((packed));
```

`-Wpadded`
: If the compiler adds padding between members of a structure to either align the members of the structure or the structure itself.

`-Wparentheses`
: If a syntactically correct code is encountered, which may be confusing to the programmer because of operator precedence or the structure of the code, a warning is generated.

`-Wpointer-arith`
: Issue a warning for anything that depends on the size of a function type or the size of a void.

`-Wredundant-decls`
: Warn about multiple declarations of the same object even if these declarations are identical.

`-Wsequence-point`
: Warn about code that may have undefined semantics because of violations of sequence point rules in the C standard.

  The C standard defines the order in which expressions in a C program are evaluated in terms of sequence points, which represent a partial ordering between the execution of parts of the program: those executed before the sequence point, and those executed after it. These occur after the evaluation of a full expression (one which is not part of a larger expression), after the evaluation of the first operand of a &&, ||, ?, : or , operator, before a function is called (but after the evaluation of its arguments and the expression denoting the called function), and in certain other places. Other than as expressed by the sequence point rules, the order of evaluation of subexpressions of an expression is not specified. All these rules describe only a partial order rather than a total order, since, for example, if two functions are called within one expression with no sequence point between them, the order in which the functions are called is not specified. However, the standards committee have ruled that function calls do not overlap.

  It is not specified when between sequence points modifications to the values of objects take effect. Programs whose behavior depends on this have undefined behavior; the C standard specifies that 'Between the previous and next sequence point an object shall have its stored value modified at most once by the evaluation of an expression. Furthermore, the prior value shall be read only to determine the value to be stored.'. If a program breaks these rules, the results on any particular implementation are entirely unpredictable.

  Examples of code with undefined behavior are a = a++;, a[n] = b[n++] and a[i++] = i;. Some more complicated cases are not diagnosed by this option, and it may give an occasional false positive result, but in general it has been found fairly effective at detecting this sort of problem in programs. The present implementation of this option only works for C programs. A future implementation may also work for C++ programs.

`-Wshadow`
: Warn whenever a local variable shadows another local variable, parameter or global variable or whenever a built-in function is shadowed.

`-Wsign-compare`
: Enable if a comparison of a **signed** and a **unsigned** value is detected a warning is generated. These comparisons could generate incorrect results because of converting the signed value to an unsigned one.

`-Wstrict-prototypes`
: If a function is declared of defined without specifying the type and number of arguments a warning is generated. If this option is set the following line of code will cause a warning:

```
int foo ();
```

`-Wswitch` A warning is generated if a enumerated type is used as index for a **switch** statement and if neither do **default** statement is given or if **case** statements are not set for all members of the enumerated type.

`-Wsystem-headers`
Issue warnings for code in system headers. These are normally unhelpful in finding bugs in your own code, therefore suppressed. If you are responsible for the system library, you may want to see them.

`-Wtraditional` Warn about certain constructs that behave differently in traditional and ISO C. Also warn about ISO C constructs that have no traditional C equivalent, and problematic constructs which should be avoided.

`-Wtrigraphs` Warn if trigraphs are encountered. Trigraphs are special control sequences defined in ANSI C for generating special characters. These trigraphs are rarely used and replaced by other characters while preprocessing. They should be avoided if possible.

`-Wundef` Warn whenever an identifier which is not a macro is encountered in an **#if** directive, outside of defined. Such identifiers are replaced by zero.

`-Wuninitialized`
Warn if an automatic variable is used without first being initialized or if a variable may be clobbered by a setjmp call.

These warnings are possible only in optimizing compilation, because they require data flow information that is computed only when optimizing. These warnings occur only for variables that are candidates for register allocation. Therefore, they do not occur for a variable that is declared **volatile**, or whose address is taken, or whose size is other than 1, 2, 4 or 8 bytes. Also, they do not occur for structures, unions or arrays, even when they are in registers.

`-Wunknown-pragmas`
If unknown **#pragma** statement is encountered a warning is issued.

`-Wunreachable-code`
With the option `-Wunreachable-code` the compiler issues a warning if the compiler detects code that will never be executed. This option is intended to warn when the compiler detects that at least a whole line of source code will never be executed, because some condition is never satisfied or because it is after a procedure that never returns. It is possible for this option to produce a warning even though there are circumstances under which part of the affected line can be executed, so care should be taken when removing apparently-unreachable code. For instance, when a function is inlined, a warning may mean that the line is unreachable in only one inlined copy of the function. In the optimization path of the compiler e.g. tries to find redundant source code. Therefore the compiler attemps to generate code only once for equivalent source lines. In such cases the compiler will issue the warning that some code will never be executed because not all

equivalent source lines will generate code. So please be careful when removing apparently-unreachable code. To find dead code you should rather disable optimization.

`-Wunused-function`
: Warn when a static function is defined and not used or when a static function is declared and not defined.

`-Wunused-label`
: Warn when a label without the `unused` attribute is not used.

`-Wunused-parameter`
: Warn when a function parameter is unused.

`-Wunused-value`
: Warn when a local variable or a non-constant static variable without the attribute `unused` is not used.

`-Wunused-variable`
: Warn when a variable is unused.

`-Wwrite-strings`
: Issue a warning if string literals are not declared **const**.

18.2.1 TriCore specific options

`-maligned-data-sections`
: To avoid alignment gaps between modules, depending on their alignment, the default sections can be divided by alignment using this option.

`-maligned-access`
: In PRAM only 4 byte access is permitted. This option forces the compiler to avoid 2 byte access to PRAM even if an optimization is activated. See type attribute alignedaccess.

`-maligned-code=<64,256>`
: Insert align directives in the assembler to force the assembler to avoid allocation of 32-Bit instruction across 64 (256)-Bit boundaries. For `-maligned-code=64` the assembler instruction:

```
.p2alignw 3,...
```

is inserted. The `.p2alignw` directive treats the fill pattern as a two byte word value. `.p2alignw 3` advances the location counter until it is a multiple of $2^3 \rightarrow 8$. If the location counter is already a multiple of 8, no change is needed.

For `-maligned-code=256` the assembler instruction:

```
.p2alignw 5,...
```

appears. `.p2alignw 5` advances the location counter until it is a multiple of $2^5 \rightarrow 32$. If the location counter is already a multiple of 32, no change is needed.

`-mcode-pic` Enable the position independent code feature. If the option `-mcode-pic` is set the code will be located in an own section called `.pictext` instead of the standard section `.text`.

`-msmall-pid` Use base relative address computation instead of absolute address computation for addresses of small data objects.

`-mno-small-pid` To support position independent data by the compiler a new relocation type is required. Since this relocation type is not part of the EABI-specification, the option `-meabi` will implicitly set the option `-mno-small-pid`. If the option `-mno-small-pid` is set the compiler will generate code without using this additional relocation type.

`-mno-warn-smalldata-initializers`
If a pointer is statically initalized with the address of a small addressable variable (`.sdata` object) and the option `-msmall-pid` is set, then the compiler will issue a warning. This warning can be disabled via the `-mno-warn-smalldata-initializers` option.

`-mcpu=<derivative>`
Select derivatives specific options (e.g. erratas for derivative).

`-mcpu-specs=<file>`
Specify a derivative specs <file>, which contains the user defined configuration settings. The option `-mcpu=` is required.

`-mcpu008` Generates code for CPU-Bug CPU_TC.031 (former CPU8) for TriCore 1.2: Disables all interrupts while divide instruction is executed. This option is depreciated, use option `-mcpu0031` instead.

`-mcpu009` Generates code for CPU-Bug CPU9 for TriCore 1.3: Insert two `nop`-instructions after each `dsync`. This workaround is not relevant for TC1796 and TC1766.

`-mcpu010` Generates code for CPU-Bug CPU_TC.018 (former CPU10) for TriCore 1.2: Insert an `isync` before the `loop` instruction. This option is depreciated, use option `-mcpu018` instead.

`-mcpu013` Generates code for CPU-Bug CPU13 for TriCore 1.3: Insert a `dsync` at the start of a function. This workaround is not relevant for TC1796 and TC1766.

`-mcpu016` Generates code for CPU-Bug CPU_TC.048 (former CPU16): Insert a `nop` between `ld.a` and `calli` / `ji` on the same register. This option is depreciated, use option `-mcpu048` instead.

`-mcpu018` Generates code for CPU-Bug CPU_TC.018 (former CPU10) for TriCore 1.2: Insert an `isync` before the `loop` instruction. This option replaces `-mcpu010`.

`-mcpu024` Generates code for CPU-Bug CPU TC.024: Incorrect return address in `a11` when performing nested calls. An access from FPI bus to DMU memory which runs in parallel with a CPU access to DMU memory stalls the CPU for one cycle. If the 'DMU not ready' indication occurs during a two cycle call variant (e.g. caused by nested calls) the register `a11` - the return PC - is incorrectly updated. This is due to an error in the fetch unit block.

As workaround in any FPI to DMU access situation (always the case when a JTAG debugger is attached or if PCP/DMA/XMU is using DMU memory or if CPU is executing code from DMU memory) a `nop` instruction is inserted at the top of any subroutine if it was previously beginning with a `call` instruction.

```
procA:
NOP ; bug fix for nested call sequence
<one optional IP instruction>
CALL procB
...
RET
```

`-mcpu031` Generates code for CPU-Bug CPU_TC.031 (former CPU8) for TriCore 1.2: Disables all interrupts while divide instruction is executed. This option replaces `-mcpu08`.

`-mcpu034` Generates code for CPU-Bug CPU_TC.034 for TriCore 1.2: insert an `isync` after each `dsync`

`-mcpu048=<case>`

Generates code for CPU-Bug CPU_TC.048 (former CPU.16): The errata CPU_TC.48 consists of two parts. You may choose between those two cases by setting the case in the command line opion. Allowed cases are:

`1`	Activates only case 1
`2`	Activates only case 2
`all`	Activates cases 1 and 2

For the TC1796 processor, the case 2 of this errata is not needed any more. This option replaces `-mcpu016`.

`-mcpu060` Generates code for CPU-Bug CPU.60. A `nop` is included between `ld.a` and `ld.d`.

`-mcpu069` Generates code for CPU-Bug CPU.69. This option inserts a `NOP` after a `RSLCX` instruction.

`-mcpu070` Generates code for CPU-Bug CPU.70. Two nops are included between `jne` and `loop`.

`-mcpu072` This option enables a workaround for the CPU bug CPU_TC.072, which requires inserting a `nop` after a `ld.[a,da]` instruction and an immediately following `loop` instruction that uses the just loaded address register. If your particular TriCore chip is affected by this bug, you should use this option for both manually written and compiler-generated assembler files.

`-mcpu076`	Workaround for CPU bug CPU_TC.076 of TC1796; incorrect result for MADD.F and MSUB.F instructions. This option replaces the instruction by paired multiply MUL.F and Add/Subtract ADD.F/SUB.F instructions. The defines for the cpu erratas are set in the specs file.
`-mcpu081`	Generates code for CPU-Bug CPU.81. Forbid loading of `%a10` from memory.
`-mcpu082`	Generates code for CPU-Bug CPU.82. Data corruption may occur when a context store operation, STUCX or STLCX, is immediately followed by a memory load operation which reads from the last double-word address written by the context store. This option inserts a NOP between these instructions.
`-mcpu083`	Generates code for CPU-Bug CPU.83. This option inserts a NOP after a DISABLE instruction.
`-mcpu094`	The compiler inserts a NOP instruction between the IP jump and the CSA list instruction. For details see Infineon's Errata Sheet for this bug.
`-mcpu095`	The compiler inserts a single NOP instruction between any SAT.B/SAT.H instruction and a following Load-Store instruction with a DGPR source operand addsc.a, addsc.at, mov.a, mtcr. For details see Infineon's Errata Sheet for this bug.
`-mcpu096`	The compiler avoids generating that kind of loops, which are affected by CPU_TC.096, but this errata is not treated by the assembler. If you are using inline assembler or assembler files please inspect the relevant blocks of your source code. For details see Infineon's Errata Sheet for this bug. This option sets only a define for the cpu errata.
`-mcpu101`	Workaround for ERRATUM CPU.101 (TC1796/66 TC1.3). Under certain circumstances two variants of the MSUB.U instruction can fail to assert PSW.V when expected to do so. If this occurs for MSUBS.U, the result fails to saturate. The compiler does not generate sequences which are affected by CPU erratum 101. The sequence is only used in intrinsic functions of the compiler. If the customers use inline assembler with the instructions affected by 101, no workaround code will be generated, this means the user has to ensure the correct behaviour.
`-mcpu114`	Workaround for ERRATUM CPU.114 (TC1797/67 TC1.3.1).
`-mcpu116`	Workaround for ERRATUM CPU.116 (TC1.3).
`-meabi`	Generate EABI conform code.
`-meabi-bitfields`	Set EABI conformity for bitfields.

`-mtest-eabi-bitfields=<opt>`

If you compile your code without the option `-meabi` or `-meabi-bitfields` then the compiler may generate code for bitfields that is not EABI conform. In this case the compiler will issue a warning for those bitfields, if the option `-mtest-eabi-bitfields` is set. The optional parameter <`opt`> activates additional warnings and may be one or more of:

`all`	Activates all warnings.
`type`	Equivalent of `-mtest-eabi-bitfields`. Warn on non-EABI-conform types or on types, that contain non-EABI-conform subtypes.
`decl`	Warn on non-EABI-conform declarations.
`def`	Warn on non-EABI-conform definitions.

Note:

The parameters may be combined by commata, no spaces are allowed in the list of the parameters. `-mtest-eabi-bitfields=` disables all warnings.

`-merror-pic=on/off`

A **static** initialization of a pointer prevents position independent code. With `-merror-pic=on` an error instead of a warning will be issued, if a pointer is initialized statically. The default state is on.

`-mhard-float` Use floating instructions.

`-moptfp` Use optimized single float emulation.

`-msoft-fdiv` Use emulation code for floating point divide.

`-mnosoft-float-to-int-conversion`

Per default the option `-msoft-float-to-int-version` is set to ensure that a float to int conversion is interrupt save. If the option is set then soft float functions will be used instead of TriCore instructions ftoi /ftou. The corresponding functions are included in the library `libhtc.a` and `ligcc.a` of the linker search directory. To disable this behaviour use `-mno-soft-float-to-int-conversion` and this will enable the generation of ftoi /ftou instructions.

`-mnocallerrors` If callee, the function or subroutine being called by the caller, and the caller are in different sections no error will be issued. If the option `-ffunction-sections` is set, all functions are put in own section. In this case the option `-mnocallerrors` should be invoked. Per default the option is not set.

`-mno-dynamic-address-calc-with-code-pic`

The address calculation of a referenced function requires a dynamic initialization routine. Per default the address calculation is activated.

With `-mno-dynamic-address-calc-with-code-pic` you can prevent the calculation.

`-mno-errornumbers`
: For every error message an error number is issued. With this option the error numbers can be suppressed.

`-mstackparm`
: All function parameters are passed on the stack instead of registers. Setting this option is equivalent to adding `__attribute__ ((stackparm))` to all functions.

`-mtc12`
: Generate code for TriCore v1.2.

`-mtc13`
: Generate code for TriCore v1.3.

`-mtc131`
: Generate code for TriCore v1.3.1 core.

`-mwarnprqa=on|off`
: The pragmas PRQA_MESSAGES_ON and PRQA_MESSAGES_OFF can be used, but they have no effect, unless a warning is issued if these pragmas are used. With this option you can enable or disable warnings for QAC pragmas.

`-muninit-const-in-rodata`
: This option is used to store uninitialized constants in the section .rodata instead of .bss. So uninitialized constants behave like initialized data. This option is set by default.

`-mno-uninit-const-in-rodata`
: It is used to store uninitialized constants in the section .bss.

`-mvolatile-const-in-rodata`
: This option is used to store constants that use qualifier **volatile** in a .rodata section. It is a default option of the compiler. If you use this option in conjunction with `-mno-uninit-const-in-rodata`, only initialized constants are placed in the section .rodata.

`-mno-volatile-const-in-rodata`
: Constant that use qualifier **volatile** are not stored in .rodata.

`-mversion-info`
: Generate a section .version_info that contains version information. This option is set per default. The section `options passed` in .version_info includes all options which are passed to `tricore-cc1`. The section `options enabled` in .version_info includes all options which will affect the code generation of the compiler and the target/machine specific options. Target specific options start with the prefix `-m`. Options that will affect the code generation are `-f*` option. If an optimization option -O1, -O2, -O3, -Os is specified this option is not listed in the `option enabled`, instead all options that enabled by optimization are listed. See chapter 23 on page 323 for details.

`-mno-version-info`
: To suppress the generation of a section .version_info that contains version information.

`-mwarn-pragma-branch`
: If an **if-else** statement, **switch** or other conditional statements are written without **#pragma** for branches, a warning will be issued (see section 14.8 on page 89 for details). Per default the compiler option is not set.

`-masm-source-lines`
: The compiler outputs the generated assembler source in a file with the suffix `.s`. This file does not contain any information about the C-Sourcecode it was generated from. By setting this option, the compiler outputs the original C-Sourcecode in comments in the assembler code. This option requires one of these options `-g`, `-g2`, `-g3`, `-gdwarf`, `-gdwarf-2` for generating debug info.

> **Note:**
>
> If you compile your program using either debug info or no debug info, the resulting code/object may differ.

`-mversion-info-string=<text>`
: The section .version_info section contains information of the options used while compiling. These options are not identical to the options passed to `tricore-gcc`, because it passes only the options to the compiler, which are relevant for the compiling (e.g. the option `-E` is not passed to the compiler since it is a preprocessor option).

By using the option `-mversion-info-string=<text>` an additional user defined string <text> can be added to the section .version_info. If <text> contains spaces it must be embraced by double quotes. If the option `-mversion-info` is set (which is the default), `tricore-gcc` automatically passes its list of options to the compiler.

`-mabs=<number>` Allocate data smaller or equal than <number> in the absolute data section (.zbss). If <number> is zero, all data will be allocated in the absolute addressable data section .zbss.

`-mabs-const=<number>`
: This option selects the absolute addressing mode for constants. Constants are marked by the type modifier **const**. v3.4

`-mabs-data=<number>`
: This option selects the absolute addressing mode for variables. Variables do not have the type modifier **const**.

`-msmall=<number>`
: Allocate data (variables and constants) smaller than or equal to <number> in the small addressable data sections (.sdata, .sdata.∗ and .sbss, .sbss.∗). If <number> is zero, all data is allocated in the small addressable data sections.

`-msmall-const=<number>`
: This option selects the small addressing mode for constants. Constants are marked by the type modifier **const**. v3.4

`-msmall-data=<number>`
: This option selects the small addressing mode for variables. Variables do not have the type modifier **const**.

`-mno-bits-struct-unions`
: If the type _bit is defined within a **struct** or **union** the compiler will issue an error. This option is set per default.

`-mbits-struct-unions`
: The definition of _bit within a **struct** or **union** is supported.

`-mcase-threshold=<number>`
: Generate only jump tables for switches which number of case statements is $\leq$ <number>.

`-mcase-range-threshold=<range>`
: Generate an if-then-else construct instead of a jump table if the product of <range> and number of case label is smaller than the case label range. In the following example with 3 case labels the range of the case label is 20, so if you specify e.g. `-mcase-range-threshold=5` the compiler will generate an if-then-else construct, because the prod $5 \cdot 3 < 20$.

```
switch (value) {
case 1:
  ...
case 10:
  ...
case 20:
  ...
};
```

`-mjumptable-in-textsection`
: Allocate the jumptable in the code section .text instead of the section .rodata.

`-mdfa`
: Use deterministic finite automaton for instruction scheduling. This option is set per default.

`-mno-all-errata`
: This option has no effect.

`-mno-divloop`
: The 32bit-division is optimised for `-Os`. The four dvstep instructions are executed in a loop instead of just executing dvstep four times. This is the default for –Os, otherwise the non optimised division using the four consecutive dvstep instruction is used by the compiler. To use the non optimised divison for `-Os`, too, use the option `-mno-divloop`.

`-mlibgcc1`
: An optimised division can be performed with option `-Os` via a function call to a library. This will reduce the codesize to a minimum. To use the library call set the option `-mlibgcc1`. You do not have to change any linker settings, because the object, the `libgcc1`, is part of the standard compiler library `libgcc.a`.

`-muse-round2zero`
: Setting ISO-conform rounding type for float to int. This option is set per default. To disable the option use the compiler switch `-mno-use-round2zero`.

`-mlicense-dir=<directory>`
: The compiler includes a license manager to support different license models. The licenses are located per default in the subdirectory `licenses` of the TriCore installation directory e.g. `C:\HighTec\TriCore\licenses`. This directory will be searched by the compiler for valid licenses. If you want to include a user-defined search directory you have to pass the compiler option `mlicense-dir=<directory>`. Alternatively you can set the environment variable `RLM_LICENSE` and assign as value the path and file name of the license.

18.2.2 Default options

- `-feliminate-unused-debug-types`
- `-fomit-frame-pointer`
- `-fpeephole`
- `-ffunction-cse`
- `-fkeep-static-consts`
- `-freg-struct-return`
- `-fgcse-lm`
- `-fgcse-sm`
- `-fgcse-las`
- `-fsched-interblock`
- `-fsched-spec`
- `-fsched-stalled-insns`
- `-fsched-stalled-insns-dep`
- `-fbranch-count-reg`
- `-fcommon`
- `-fargument-alias`
- `-fzero-initialized-in-bss`
- `-fident`
- `-fmath-errno`
- `-ftrapping-math`

- `-mtc12`
- `-mno-bits-struct-unions`
- `-muninit-const-in-rodata`
- `-mvolatile-const-in-rodata`
- `-mdfa`
- `-mversion-info`
- `-mversion-info-string=`
- `-mcpu018`
- `-mcpu024`
- `-mcpu031`
- `-mcpu034`
- `-mcpu048=all`
- `-mcpu050`
- `-mcpu060`
- `-mcpu069`
- `-mcpu070`
- `-mcpu072`
- `-mround2zero`
- `-msoft-float-to-int-conversion`

19 The TriCore Assembler

If you use the GNU assembler on one architecture, you should find a fairly similar environment when you use it on another architecture. Each version has much in common with the others, including object file formats, most assembler directives (often called <pseudo-ops>) and assembler syntax.

The GNU assembler is primarily designed to assemble the output of the GNU C compiler into an object code format for use by the linker `ld`. Nevertheless, we've tried to make `tricore-as` assemble correctly everything that other assemblers for the same machine would assemble. If you need to write an assembly module, the best way to start is to write a simple program in C that contains all the structural elements you need and then use `tricore-gcc` with the `-S` option to generate assembly language source. If you do not need much assembly language, it may be easier to insert it as inline assembly (see chapter 8).

When you are writing in a higher level language, `tricore-gcc` normally invokes the assembler for you, so you seldom need to deal with the command-line options. But you can also use the `-Wa` option with `tricore-gcc` to pass arguments through to the assembler. The assembler arguments must be separated from each other (and the `-Wa`) by commas.

19.1 The GNU Assembler

This manual is a short introduction to the *GNU* assembler. Following is a brief summary of generally available command-line options; for machine specific options see subsection 19.1.2 on page 191.

19.1.1 Common Command-Line Options

`-a[opts][=file]`

Turn on the output listing. A combination of one or more `opts` can be used to specify the format and the content of the output listing. The default option is `-ahls`. The output listing can be directed to a file by assigning the file name as part of the sub-option. By itself, `-a` requests high-level, assembly, and symbols listing. You can use other letters to select specific options for the list: `-ah` requests a high-level language listing, Note if the assembler source is coming from the standard input (eg because it is being created by `tricore-gcc` and the `-pipe` command line switch is being used) then the listing will not contain any comments or preprocessor directives. This is because the listing code buffers input source lines from stdin only after they have been preprocessed by the assembler. This reduces memory usage and makes the code more efficient. Once you have specified one of these options, you can further control listing output and its appearance using the directives `.list`, `.nolist`, `.psize`, `.eject`, `.title`, and `.sbttl`. High-level listings require that a compiler debugging option like `-g` be used, and that assembly listings (`-al`) be requested also.

`-ac`	Use the `-ac` option to omit false conditionals from a listing. Any lines which are not assembled because of a false `.if` (or `.ifdef`, or any other conditional), or a true `.if` followed by an `.else`, will be omitted from the listing.
`-ad`	Omits debugging directives found in source files.
`-ah`	Include high-level language source.
`-al`	Includes assembled code. `-al` requests an output-program assembly listing.
`-am`	Includes macro expansions.
`-an`	Omit forms processing. The `-an` option turns off all forms processing. If you do not request listing output with one of the `-a` options, the listing-control directives have no effect.
`-as`	Includes a symbols cross-reference table.
`=file`	List to file (e.g. `-ahls=assembly.list`).

`-D` Ignored. This option is accepted for script compatibility with calls to other assemblers.

`--defsym <symbol>=<value>`
Defines the symbol <sym> to be <value> before assembling the input file. <value> must be an integer constant.

`-f` `-f` should only be used when assembling programs written by a (trusted) compiler. `-f` stops the assembler from doing whitespace and comment preprocessing on the input file(s) before assembling them.

Note:

If you use `-f` when the files actually need to be preprocessed (if they contain comments, for example), `tricore-as` does not work correctly.

`--fatal-warnings`
Treat warnings as errors.

`--gdwarf2` Generate DWARF2 debugging information for each assembler line and include it in the object file.

Note:

If you compile your program using either debug info or no debug info, the resulting code/object may differ.

`--help` Print a summary of the command line options and exit.

`-I <dir>` Use this option to add a <path> to the list of directories `tricore-as` searches for files specified in `.include` directives (subsection 19.2.1 on page 194). You may use `-I` as many times as necessary to include a variety of paths. The current working directory is always searched first; after that, `tricore-as` searches any `-I` directories in the same order as they were specified (left to right) on the command line.

`-J` Don't issue warnings about signed overflow.

`-K` It is permitted for compatibility with the *GNU* assembler on other platforms, where it can be used to warn when the assembler alters the machine code generated for `.word` directives in difference tables. The TriCore family does not have the addressing limitations that sometimes lead to this alteration on other platforms.

> **Note:**
>
> This option is accepted but has no effect for TriCore family.

`--keep-locals` Labels beginning with `.L` (upper case only) are called local label. Normally you do not see such labels when debugging, because they are intended for the use of programs (like compilers) that compose assembler programs, not for your notice. Normally both `tricore-as` and `tricore-ld` discard such labels, so you do not normally debug with them.

`-L` Same as `--keep-locals`

`--listing-lhs-width=<number>`
Set the maximum width, in words, of the output data column for an assembler listing to <number>.

`--listing-lhs-width2=<number>`
Set the maximum width, in words, of the output data column for continuation lines in an assembler listing to <number>.

`--listing-rhs-width=<number>`
Set the maximum width of an input source line, as displayed in a listing, to <number> bytes.

`--listing-cont-lines=<number>`
Set the maximum number of lines printed in a listing for a single line of input to <number> + 1.

`-MD <filename>` `tricore-as` can generate a dependency file for the file it creates. This file consists of a single rule suitable for `make` describing the dependencies of the main source file. The rule is written to the file named in its argument. This feature is used in the automatic updating of makefiles.

`--no-warn` Same as `-W`. Suppress warning messages.

`-o <objfile>`	You use this option (which takes exactly one filename) to give the object file a different name. Whatever the object file is called, `tricore-as` overwrites any existing file of the same name.
`-R`	`-R` tells `tricore-as` to write the object file as if all data-section data lives in the text section. This is only done at the very last moment: your binary data are the same, but data section parts are relocated differently. The data section part of your object file is zero bytes long because all its bytes are appended to the text section.
`--statistics`	Use `--statistics` to display two statistics about the resources used by `tricore-as`: the maximum amount of space allocated during the assembly (in bytes), and the total execution time taken for the assembly in seconds.
`--strip-local-absolute`	Any symbol that is local to this assembly and has a constant value is removed from the outgoing symbol table.
`--target-help`	Print a summary of all target specific options and exit.
`-v`	Same as `-version`.
`-version`	Same as `--version`.
`--version`	Print the `tricore-as` version and exit.
`-W`	Suppress warning messages.
`--warn`	Don't suppress warning messages.
`-Z`	Generate a bad object file even after errors. After an error message, `tricore-as` normally produces no output. If for some reason you are interested in object file output even after `tricore-as` gives an error message on your program, use the `-Z` option.
`-- | <files> ...`	Standard input, or source files to assemble.

If you are invoking `tricore-as` via the *GNU* compiler driver program (`tricore-gcc`), you may use the `-Wa` option to pass additional arguments to the assembler. The assembler arguments must be separated from each other (and the `-Wa`) by commas.

For example:

```
tricore-gcc -c -g -O -Wa,-alh,-L file.c
```

This passes two options to the assembler: `-alh` (emit a listing to standard output with high-level and assembly source) and `-L` (retain local symbols in the symbol table).

You can call the *GNU* compiler driver with the `-v` option to see precisely what options it passes to each compilation pass, including the assembler.

19.1.2 TriCore specific Command-Line Options

The following options are available for `tricore-as`:

`-mtc10`	Assemble for the TC1v1.0 instruction set; such modules are incompatible with modules assembled for any other version of the TriCore architecture.

> **Note:**
>
> This option is deprecated; support for TC1v1.0 will be dropped in a future version.

`-mtc12`	Assemble for the TC1v1.2 instruction set; such modules are compatible with modules assembled for the TC1v1.3 or TC2.
`-mtc13`	Assemble for the TC1v1.3 instruction set. Such modules are compatible with modules assembled for the TC1v1.2 or TC2.
`-mtc2`	Assemble for the TC2; such modules are compatible with modules assembled for the TC1v1.2 or TC1v1.3.
`-mcpu009`	Automatically insert two nop instructions after each dsync instruction as a software workaround for the hardware bug documented in TriCore errata sheets as CPU.9 or COR17.
`-mcpu034`	Generate code for the CPU-Bug CPU_TC.034: Insert an isnyc after each dsync. Also note that the options −mcpu009 and −mcpu034 are mutually exclusive; if both are specified, the last option specified wins
`-mcpu048`	This option enables a workaround for the CPU bug CPU_TC.048, which requires inserting a nop after a ld .[a,da] instruction and an immediately following indirect jump or call instruction that uses the just loaded address register. If your particular TriCore chip is affected by this bug, you should use this option for both manually written and compiler-generated assembler files.
`-mcpu050`	Generate code for the CPU-Bug CPU_TC.050: Insert a nop instruction between a multicycle integer instruction and load instruction.

> **Note:**
>
> Use this option only for manually written assembler files, not for assembler files generated by `tricore-gcc`, which already implements its own workaround for this bug.

`-mcpu060`	This option enables a workaround for the CPU bug CPU_TC.060, which requires inserting a nop after a ld .[a,da] instruction and an immediately following ld .[d,w] instruction that uses the just loaded address register. If your particular TriCore chip is affected by this bug, you should use this option for both manually written and compiler-generated assembler files.

`-mcpu069` This option inserts a NOP after a RSLCX instruction.

`-mcpu070` This option enables a workaround for the CPU bug CPU_TC.070, which requires inserting one or two nops after a conditional jump instruction and an immediately following loop instruction. If your particular TriCore chip is affected by this bug, you should use this option for both manually written and compiler-generated assembler files.

`-mcpu072` This option enables a workaround for the CPU bug CPU_TC.072, which requires inserting a nop after a ld .[a,da] instruction and an immediately following loop instruction that uses the just loaded address register. If your particular TriCore chip is affected by this bug, you should use this option for both manually written and compiler-generated assembler files.

`-mcpu081` Forbid loading of `%a10` from memory.

`-mcpu082` Data corruption may occur when a context store operation, STUCX or STLCX, is immediately followed by a memory load operation which reads from the last double-word address written by the context store. This option inserts a NOP between these instructions.

`-mcpu083` This option inserts a NOP after a DISABLE instruction.

`-mcpu094` The assembler inserts a NOP instruction between the IP jump and the CSA list instruction. For details see Infineon's Errata Sheet for this bug.

`-mcpu095` The assembler inserts a single NOP instruction between any SAT.B/SAT.H instruction and a following Load-Store instruction with a DGPR source operand addsc.a, addsc.at, mov.a, mtcr. For details see Infineon's Errata Sheet for this bug.

Note:

The compiler avoids generating that kind of loops, which are affected by CPU_TC.096, but this errata is not treated by the assembler. If you are using inline assembler or assembler files please inspect the relevant blocks of your source code. For details see CPU erratas for TriCore.

`-mdmi12` Enables a software workaround for TriCore TC1130 chips that are affected by the DMI_TC.012 hardware cache bug. This workaround will be enabled if the option `-mcpu=tc1130` is passed to the compiler. If you are compiling C/C++ source files using the compiler driver `tricore-gcc` and your TriCore chip is affected by this bug, you should add the option `-Wa,-mdmi12` which tells the driver program to enable the new workaround in the assembler. When enabled, the assembler auto-inserts an additional nop instruction whenever it detects a code sequence that is potentially affected by the data cache misbehaviour as described in Infineon's Errata Sheet for this bug. Note that code assembled with this workaround enabled will also run on non-affected TriCore derivatives, as the additional `nop` instructions merely have a negligible impact on the code size and execution time, but otherwise will not cause any incompatibilities whatsoever.

`-V` Show the version of the assembler before processing the given files (or standard input).

`-Y` This is useful for debugging `tricore-as`, as it shows its internal state, how mnemonics and operands are parsed, and which opcode is chosen for a given mnemonic instruction.

`--dont-optimize | --insn32-only | --insn32-preferred`
If none these mutually-exclusive options is specified, `tricore-as` will try to find the shortest possible opcode that fits a given mnemonic instruction. If an instruction is available in both 16-bit and 32-bit formats (e.g., `nop`), then the 16-bit instruction is chosen in favor of the 32-bit instruction. If a mnemonic instruction results in a 32-bit instruction, `tricore-as` tries to optimize it by finding an equivalent 16-bit instruction.

Example

- `add %dn,%dn,%dm` will be replaced by the 16-bit instruction `add %dn,%dm`
- `sel %dn, %d15, %dn,` **`const`** will be replaced by the 16-bit instruction `cmovn %dn, %d15,` **`const`**
- Under certain circumstances these jump equal instruciton `jeq`, `jne`, `jlt` etc. may be replaced by jump zero instruction `jz`, `jnz`, `jltz` etc.

This means that neither a compiler nor an assembler programmer needs to care about emitting or writing down the shortest possible instructions, as this is done automatically by `tricore-as`.

`--dont-optimize`
Don't try to find the shortest matching opcode (`.optim`, `.code16` and `.code32` are honored). Use this option to prevent the assembler from optimizing mnemonic instructions. However, you can use the `.optim` pseudo-opcode to enable optimization for the next instruction following `.optim`, thus temporarily overriding this option.

`--insn32-only` This directs the assembler to exclusively emit 32-bit opcodes. 16-bit instructions will be replaced with their 32-bit equivalent, and the pseudo-opcodes `.code16` and `.optim` will be ignored.

`--insn32-preferred`
This option is similar to `−−insn32−only` in that it prevents optimizing and replaces 16-bit instructions with their 32-bit equivalent, but the pseudo-opcode `.code16` will be honored.

`--enforce-aligned-data`
: When enabled, N-byte constants generated by `.short`, `.word`, and related pseudo-opcodes are automatically aligned to a N-byte boundary within their respective section. This is not the default, as compilers (notably `tricore-gcc`) may take care of alignment themselves and omit alignment statements deliberately in order to initialize `packed` structures.

19.2 Assembler Directives

The primary purpose of an assembler is to translate mnemonic opcodes into binary opcodes that can be executed by the hardware or used as data storage locations. In addition, the assembler understands and acts on assembler directives, which can be used to align code, define macro expansions, divide the code into named sections, declare named constants, provide conditional assembly, or simply be a shorthand method for defining character data. All assembler directives have names that begin with a period '.'; the rest of the name usually consists of small letters.

This chapter discusses directives that are available regardless of the target machine configuration for the *GNU* assembler. For machine specific directives see subsection 19.2.2 on page 211.

19.2.1 Common Assembler Direcitves

`.abort`
: This directive stops the assembly immediately. It is for compatibility with other assemblers. The original idea was that the assembly language source would be piped into the assembler. If the sender of the source quit, it could use this directive to tell `tricore-as` to quit also. One day `.abort` will no longer be supported.

`.align <boundary>, <abs-expr>, <maximum>`
: Pad the location counter (in the current subsection) to a particular storage boundary. All three values are absolute expressions. The first expression is the alignment required, as described below.

  The second expression gives the fill value to be stored in the padding bytes. It (and the comma) may be omitted. If it is omitted, the padding bytes are normally zero. However, on some systems, if the section is marked as containing code and the fill value is omitted, the space is filled with no-op instructions.

  The third expression is also optional. If it is present, it is the maximum number of bytes that should be skipped by this alignment directive. If doing the alignment would require skipping more bytes than the specified maximum, then the alignment is not done at all. You can omit the fill value (the second argument) entirely by simply using two commas after the required alignment; this can be useful if you want the alignment to be filled with no-op instructions when appropriate.

  The first expression is the alignment request in bytes. For example `.align 8` advances the location counter until it is a multiple of 8. If the location counter is already a multiple of 8, no change is needed.

For other systems, including the i386 using a.out format, and the arm and strongarm, it is the number of low-order zero bits the location counter must have after advancement. For example `.align 3` advances the location counter until it is a multiple of $2^3 \rightarrow 8$. If the location counter is already a multiple of 8, no change is needed.

This inconsistency is due to the different behaviors of the various native assemblers for these systems which gnu assembler (GAS) must emulate. GAS also provides `.balign` and `.p2align` directives, described later, which have a consistent behavior across all architectures (but are specific to GAS).

`.ascii "string" ...` `.ascii` expects zero or more string literals separated by commas. It assembles each string (with no automatic trailing zero byte) into consecutive addresses.

`.asciz "string" ...` `.asciz` is just like `.ascii`, but each string is followed by a zero byte. The "z" in `.asciz` stands for "zero".

`.balign[wl] <abs-expr>, <abs-expr>, <abs-expr>`

Pad the location counter (in the current subsection) to a particular storage boundary. The first expression (which must be absolute) is the alignment request in bytes. For example `.balign 8` advances the location counter until it is a multiple of 8. If the location counter is already a multiple of 8, no change is needed.

The second expression (also absolute) gives the fill value to be stored in the padding bytes. It (and the comma) may be omitted. If it is omitted, the padding bytes are normally zero. However, on some systems, if the section is marked as containing code and the fill value is omitted, the space is filled with no-op instructions.

The third expression is also absolute, and is also optional. If it is present, it is the maximum number of bytes that should be skipped by this alignment directive. If doing the alignment would require skipping more bytes than the specified maximum, then the alignment is not done at all. You can omit the fill value (the second argument) entirely by simply using two commas after the required alignment; this can be useful if you want the alignment to be filled with no-op instructions when appropriate.

The `.balignw` and `.balignl` directives are variants of the `.balign` directive. The `.balignw` directive treats the fill pattern as a two byte word value. The `.balignl` directives treats the fill pattern as a four byte longword value. For example, `.balignw 4,0x368d` will align to a multiple of 4. If it skips two bytes, they will be filled in with the value 0x368d (the exact placement of the bytes depends upon the endianness of the processor). If it skips 1 or 3 bytes, the fill value is undefined.

`.byte <expressions>`
: `.byte` expects zero or more expressions, separated by commas. Each expression is assembled into the next byte of the current section.

`.comm <symbol>, <length>, <alignment>`
: `.comm` declares a common symbol named <symbol>. An uninitialized memory location of <length> bytes is declared and tagged with the name <symbol>. When linking, a common symbol in one object file may be merged with a defined or common symbol of the same name in another object file. If no alignment is specified, `tricore-as` will set the alignment to the largest power of two less than or equal to the size of the symbol, up to a maximum of 16.

  When using ELF, the `.comm` directive takes an optional third argument <alignment>. If `tricore-ld` does not see a definition for the symbol–just one or more common symbols–then it will allocate <length> bytes of uninitialized memory. <length> must be an absolute expression. If `tricore-ld` finds multiple common symbols with the same name, and they do not all have the same size, it will allocate space using the largest size.

`.data <subsection>`
: `.data` tells `tricore-as` to assemble the following statements onto the end of the data subsection numbered <subsection> (which is an absolute expression). If <subsection> is omitted, it defaults to zero.

`.def <name>`
: Begins a block of debugging information, tagged by a symbol <name>, for insertion into COFF formatted object. The block continues until `.endef` directive terminates it.

`.desc <symbol>, <abs-expression>`
: The symbol is defined as having the specified value. The <value> must be an absolute expression. This directive produces no output for the COFF format.

`.dim`
: This directive is generated by compilers to include auxiliary debugging information in the symbol table. It is only permitted inside `.def`/`.endef` pairs. `.dim` is only valid for the COFF object format

`.double <value>`
: **.double** expects zero or more floating-point numbers <value>, separated by commas. The exact kind of floating point numbers emitted depends on how ***GNU* as** is configured (see subsection 19.2.2 on page 211).

`.eject`
: `eject` directive forces a page break at this point, when generating assembly listings.

`.else`
: **.else** is part of the `tricore-as` support for conditional assembly; See **.if**.

`.elseif`
: `. elseif` is part of the `tricore-as` support for conditional assembly; See **.if**. It is shorthand for beginning a new **.if** block that would otherwise fill the entire **.else** section.

`.end` .end marks the end of the assembly file. `tricore-as` does not process anything in the file past the .end directive.

`.endef` See .def.

`.endif` .endif is part of the `tricore-as` support for conditional assembly; it marks the end of a block of code that is only assembled conditionally. See .if.

`.endm` Mark the end of a macro definition. Also see .macro.

`.equ <symbol>, <value>`
This directive sets the value of <symbol> to <value>. The <value> can be either an absolute or relative expression. The .equ can be used multiple times on the same symbol, changing the value each time. It is synonymous with .set; See .set.

`.equiv <symbol>, <value>`
The .equiv directive is like .equ and .set, except that the assembler will signal an error if <symbol> is already defined.

```
.ifdef SYM
.err
.endif
.equ SYM,VAL
```

`.err` If `tricore-as` assembles a .err directive, it will print an error message and, unless the `-Z` option was used, it will not generate an object file. This can be used to signal error an conditionally compiled code.

`.exitm` Exit early from the current macro definition. See .macro.

`.extern` .extern is accepted in the source program—for compatibility with other assemblers—but it is ignored. `tricore-as` treats all undefined symbols as external.

`.fail <value>` Generates an error or a warning. If the <value> is 500 or more, `tricore-as` will print a warning message. If the value is less than 500, `tricore-as` will print an error message. The message will include the value of <value>. This can occasionally be useful inside complex nested macros or conditional assembly.

`.file <string>` . file tells `tricore-as` that we are about to start a new logical file. <string> is the new file name. In general, the filename is recognized whether or not it is surrounded by quotes "; but if you wish to specify an empty file name, you must give the quotes–`""`. This statement may go away in future: it is only recognized to be compatible with old `tricore-as` programs.

`.fill <repeat> , <size> , <value>`
<repeat>, <size> and <value> are absolute expressions. This emits <repeat> copies of <size> bytes. <repeat> may be zero or more. <size> may be zero or more, but if it is more than 8, then it is deemed to have the value 8, compatible with other people's assemblers. The

contents of each <repeat> bytes is taken from an 8-byte number. The highest order 4 bytes are zero. The lowest order 4 bytes are <value> rendered in the byte-order of an integer on the computer `tricore-as` is assembling for. Each <size> bytes in a repetition is taken from the lowest order <size> bytes of this number. Again, this bizarre behavior is compatible with other people's assemblers.

<size> and <value> are optional. If the second comma and <value> are absent, <value> is assumed zero. If the first comma and following tokens are absent, <size> is assumed to be 1. Also see .org and .p2algin.

`.float`
: For each value specified (separated by commas), a floating-point number is assembled and stored into memory. The internal representation of floating-point numbers, including size and range, varies depending on the platform. Also see **.double**. It has the same effect as .single.

`.global <symbol>`
: .global makes the symbol visible to `tricore-ld`. If you define <symbol> in your partial program, its value is made available to other partial programs that are linked with it. Otherwise, <symbol> takes its attributes from a symbol of the same name from another file linked into the same program.

`.globl`
: An alternate spelling of .global.

`.hidden <names>`
: .hidden is an ELF visibility directive. The other two are .internal and .protected (See assembler directive .internal and .protected). This directive overrides the named symbols default visibility (which is set by their binding: local, global or weak). The directive sets the visibility to hidden which means that the symbols are not visible to other components. Such symbols are always considered to be .protected as well.

`.hword <value>`
: .hword directive expects zero or more <value>, and emits a 16 bit number for each. This directive is a synonym for **.short**; depending on the target architecture, it may also be a synonym for .word.

`.ident`
: This directive is used by some assemblers to place tags in object files. `tricore-as` simply accepts the directive for source-file compatibility with such assemblers, but does not actually emit anything for it.

`.if <absolute expression>`
: **.if** marks the beginning of a section of code which is only considered part of the source program being assembled if the argument is non-zero. The end of the conditional section of code must be marked by .endif (See .endif); optionally, you may include code for the alternative condition, flagged by **.else** (See **.else**). If you have several conditions to check, .elseif may be used to avoid nesting blocks if/else within each subsequent **.else** block.

`.ifc <string1>,<string2>`
: Assembles the following section of code if the two

strings are the same. The strings may be optionally quoted with single quotes. If they are not quoted, the first string stops at the first comma, and the second string stops at the end of the line. Strings which contain whitespace should be quoted. The string comparison is case sensitive.

`.ifdef <symbol>` The conditional assembly occurs only if the symbol has been defined.

`.ifeq <absolute expression>`
Assembles the following section of code if the argument is zero.

`.ifeqs <string1>,<string2>`
Another form of . ifc . The strings must be quoted using double quotes.

`.ifge <absolute expression>`
Assembles the following section of code if the argument is greater than or equal to zero.

`.ifgt <absolute expression>`
Assembles the following section of code if the argument is greater than zero.

`.ifle <absolute expression>`
Assembles the following section of code if the argument is less than or equal to zero.

`.iflt <absolute expression>`
Assembles the following section of code if the argument is less than zero.

`.ifnc <string1>,<string2>`
Like . ifc , but the sense of the test is reversed: this assembles the following section of code if the two strings are not the same.

`.ifndef <symbol>` Assembles the following section of code if the specified <symbol> has not been defined.

`.ifnotdef <symbol>`
Synonym for . ifndef .

`.ifne <absolute expression>`
Assembles the following section of code if the argument is not equal to zero (in other words, this is equivalent to . if).

`.ifnes <string1>,<string2>`
Like . ifeqs , but the sense of the test is reversed: this assembles the following section of code if the two strings are not the same.

`.incbin "<file>"[,<skip>[,<count>]]`

The `.incbin` directive includes <file> verbatim at the current location. You can control the search paths used with the `-I` command-line option (See subsection 19.1.1 on page 187). Quotation marks are required around <file>.

The <skip> argument skips a number of bytes from the start of the <file>. The <count> argument indicates the maximum number of bytes to read.

Note:

The data is not aligned in any way, so it is the user's responsibility to make sure that proper alignment is provided both before and after the `incbin` directive.

`.include "<file>"` This directive provides a way to include supporting files at specified points in your source program. The code from <file> is assembled as if it followed the point of the `.include`; when the end of the included file is reached, assembly of the original file continues. You can control the search paths used with the `-I` command-line option (See subsection 19.1.1 on page 187). Quotation marks are required around <file>.

`.int <expressions>`

For each expression, emit a number that, at run time, is the value of that expression. The byte order and bit size of the number depends on what kind of target the assembly is for.

`.internal <names>`

`.internal` is an ELF visibility directive. The other two are `.hidden` and `.protected` (see `.hidden` and `.protected`). This directive overrides the named symbols default visibility (which is set by their binding: local, global or weak). The directive sets the visibility to `.internal` which means that the symbols are considered to be `.hidden` (ie not visible to other components), and that some extra, processor specific processing must also be performed upon the symbols as well.

`.irp <symbol>,<values> ...`

Evaluate a sequence of statements assigning different values to <symbol>. The sequence of statements starts at the `.irp` directive, and is terminated by an `.endr` directive. For each <value>, <symbol> is set to <value>, and the sequence of statements is assembled. If no <value> is listed, the sequence of statements is assembled once, with <symbol> set to the null string. To refer to <symbol> within the sequence of statements, use <symbol>.

```
.irp param,1,2,3
move d\param,sp\-
.endr
```

is equivalent to assembling

```
move d1,sp\-
move d2,sp\-
move d3,sp\-
```

`.irpc <symbol>,<values> ...`

Evaluate a sequence of statements assigning different values to <symbol>. The sequence of statements starts at the .irpc directive, and is terminated by an .endr directive. For each character in <value>, <symbol> is set to the character, and the sequence of statements is assembled. If no <value> is listed, the sequence of statements is assembled once, with <symbol> set to the null string. To refer to <symbol> within the sequence of statements, use <symbol>.

For example, assembling

```
.irpc param,123
 move d\param,sp\-
.endr
```

is equivalent to assembling

```
move d1,sp\-
move d2,sp\-
move d3,sp\-
```

Also see .macro, .rept, and .irp.

`.lcomm <symbol> , <length>`

Reserves the number of bytes specified by <length>, an absolute expression, as a local common block of data in .bss section. So at run-time the bytes start off zeroed. The <symbol> is local so it is unknown to outside the current module.

`.lflags` Is ignored by `tricore-as`.

`.line <line-number>`

Changes the current line number of the following line to the absolute expression <line-number>. One day `tricore-as` will no longer support this directive: it is recognized only for compatibility with existing assembler programs.

`.linkonce [<type>]`

Mark the current section so that the linker only includes a single copy of it. This may be used to include the same section in several different object files, but ensure that the linker will only include it once in the final output file. The .linkonce pseudo-op must be used for each instance of the section. Duplicate sections are detected based on the section name, so it should be unique.

Note:

This directive is only supported by a few object file formats; as of this writing, the only object file format which supports it is the Portable Executable format used on Windows NT.

The <type> argument is optional. If specified, it must be one of the following strings. For example:

```
.linkonce same_size
```

Not all types may be supported on all object file formats.

`discard`	Silently discard duplicate sections. This is the default.
`one_only`	Warn if there are duplicate sections, but still keep only one copy.
`same_size`	Warn if any of the duplicates have different sizes.
`same_contents`	Warn if any of the duplicates do not have exactly the same contents.

`.list`

Control (in conjunction with the .nolist directive) whether or not assembly listings are generated. These two directives maintain an internal counter (which is zero initially). .list increments the counter, and .nolist decrements it. Assembly listings are generated whenever the counter is greater than zero. By default, listings are disabled. When you enable them (with the -a command line option; see subsection 19.1.1 on page 187), the initial value of the listing counter is one.

`.ln <line-number>`

.ln is a synonym for .line.

`.long <expressions>`

This directive is a synonym of .int

`.macro <macname> <macargs> ...`

The commands .macro and .endm allow you to define macros that generate assembly output. For example, this definition specifies a macro sum that puts a sequence of numbers into memory:

```
.macro sum from=0, to=5
.long \from
.if \to-\from
sum "(\from+1)",\to
.endif
.endm
```

With that definition, SUM 0,5 is equivalent to this assembly input:

```
.long 0
.long 1
.long 2
.long 3
.long 4
.long 5
```

Begin the definition of a macro called <macname>. If your macro definition requires arguments, specify their names after the macro name, separated by commas or spaces. You can supply a default value for any macro argument by following the name with `=<deflt>`. For example, these are all valid .macro statements:

`.macro comm` Begin the definition of a macro called comm, which takes no arguments.

`.macro plus1 p, p1`
See .macro plus1 p p1.

`.macro plus1 p p1`
Either statement begins the definition of a macro called plus1, which takes two arguments; within the macro definition, write p or p1 to evaluate the arguments.

`.macro reserve_str p1=0 p2`
Begin the definition of a macro called reserve_str, with two arguments. The first argument has a default value, but not the second. After the definition is complete, you can call the macro either as reserve_str a,b (with p1 evaluating to <a> and p2 evaluating to <b>), or as reserve_str, b (with p1 evaluating as the default, in this case 0, and p2 evaluating to <b>).

When you call a macro, you can specify the argument values either by position, or by keyword. For example, sum 9,17 and sum to=17, from=9 are equivalent. The .macro directive can be used recursively and can accept arguments. (Also see .endm and .exitm)

`.nolist` Control (in conjunction with the . list directive) whether or not assembly listings are generated. These two directives maintain an internal counter (which is zero initially). . list increments the counter, and . nolist decrements it. Assembly listings are generated whenever the counter is greater than zero.

`.octa <bignums>` This directive expects zero or more bignums, separated by commas. For each bignum, it emits a 16-byte integer. Also see .quad

`.org <new-lc> , <fill>`
Advance the location counter of the current section to <new-lc>. new-lc is either an absolute expression or an expression with the same section as the current subsection. That is, you can't use .org to cross sections: if new-lc has the wrong section, the .org directive is ignored. To be compatible with former assemblers, if the section of <new-lc> is absolute, `tricore-as` issues a warning, then pretends the section of <new-lc> is the same as the current subsection. .org may only increase the location counter, or leave it unchanged; you can use .org to move the location counter forward, but not backward. The inserted bytes, if any, are initialized to the value <fill>.

Note:

Beware that the origin is relative to the start of the section, not to the start of the subsection. This is compatible with other people's assemblers.

When the location counter (of the current subsection) is advanced, the intervening bytes are filled with <fill> which should be an absolute expression. If the comma and <fill> are omitted, <fill> defaults to zero. Also see `.fill`, `.skip`, and `.p2align`.

`.p2align[wl] <abs-expr>, <abs-expr>, <abs-expr>`
Pad the location counter in the current subsection to a particular storage boundary. The first expression, which must be absolute, is the number of low-order zero bits the location counter must have after advancement. For example `.p2align 3` advances the location counter until it is a multiple of $2^3 \rightarrow 8$. If the location counter is already a multiple of 8, no change is needed.

The second expression (also absolute) gives the fill value to be stored in the padding bytes. It (and the comma) may be omitted. If it is omitted, the padding bytes are normally zero. However, on some systems, if the section is marked as containing code and the fill value is omitted, the space is filled with no-op instructions.

The third expression is also absolute, and is also optional. If it is present, it is the maximum number of bytes that should be skipped by this alignment directive. If doing the alignment would require skipping more bytes than the specified maximum, then the alignment is not done at all. You can omit the fill value (the second argument) entirely by simply using two commas after the required alignment; this can be useful if you want the alignment to be filled with no-op instructions when appropriate.

The `.p2alignw` and `.p2alignl` directives are variants of the `.p2align` directive. The `.p2alignw` directive treats the fill pattern as a two byte word value. The `.p2alignl` directives treats the fill pattern as a four byte longword value. For example, `.p2alignw 2,0x368d` will align to a multiple of 4. If it skips two bytes, they will be filled in with the value 0x368d (the exact placement of the bytes depends upon the endianness of the processor). If it skips 1 or 3 bytes, the fill value is undefined.

`.previous` This is one of the ELF section stack manipulation directives. The others are `.section`, `.subsection`. Also see `.pushsection` and `.popsection`.

This directive swaps the current section (and subsection) with most recently referenced section (and subsection) prior to this one. Multiple `.previous` directives in a row will flip between two sections (and their subsections).

In terms of the section stack, this directive swaps the current section with the top section on the section stack.

`.popsection` This is one of the ELF section stack manipulation directives. The others are `.section` and `.subsection` `.pushsection`.

This directive replaces the current section (and subsection) with the top section (and subsection) on the section stack. It is popped off the stack.

`.print "string"` `tricore-as` will print <string> on the standard output during assembly. You must put <string> in double quotes.

`.protected <names>`

This is one of the ELF visibility directives. The other two are `.hidden` and `.internal` (see `.hidden` and `.internal`).

This directive overrides the named symbols default visibility (which is set by their binding: local, global or weak). The directive sets the visibility to `.protected` which means that any references to the symbols from within the components that defines them must be resolved to the definition in that component, even if a definition in another component would normally preempt this.

`.psize <lines> , <columns>`

Use this directive to declare the number of lines—and, optionally, the number of columns—to use for each page, when generating listings.

If you do not use `.psize`, listings use a default line-count of 60. You may omit the comma and <columns> specification; the default width is 200 columns.

`tricore-as` generates formfeeds whenever the specified number of lines is exceeded (or whenever you explicitly request one, using `.eject`).

If you specify <lines> as `0`, no formfeeds are generated save those explicitly specified with `.eject`.

`.purgem <name>`

Undefine the macro <name>, so that later uses of the string will not be expanded.

`.pushsection <name> , <subsection>`

This directive is a synonym for `.section` (also see `.section` and `.previous`).

`.quad <bignums>`

Each bignum value is declared as an 8-byte value. Also see `.octa`.

`.rept <count>` Repeat the sequence of lines between the `.rept` directive and the next `.endr` directive <count> times.

For example, assembling

```
.rept 3
.long 0
.endr
```

is equivalent to assembling

```
.long 0
.long 0
.long 0
```

Also see .macro, .irp, and .irpc.

`.sbttl <subheading>`
Use <subheading> as the title (third line, immediately after the title line) when generating assembly listings. This directive affects subsequent pages, as well as the current page if it appears within ten lines of the top of a page.

`.scl <class>` This directive can be used inside a .def and .endef pair to specify the storag class value for a symbol. Storage class may flag whether a symbol is static or external, or it may record further symbolic debugging information.

`.section <name>` (COFF version) This form of the .section is valid for any object format that supports arbitrarily named sections. It assembles the following code into a section of the specified name.

`.section <name>[, "<flags>"]`
This form of the .section directive is valid for the COFF object format. Each flag is a single character in the "flags" string. The following flags are recognized:

`b`	A section containing uninitialized data (bss section)
`n`	section is not loaded when the program is executed.
`w`	This section can be written to during execution
`d`	A data section, as opposed to an executable section.
`r`	This section read-only section.
`x`	This is an executable section, as opposed to a data section.

If no flags are specified, the default flags depend upon the section name. If the section name is not recognized, the default will be for the section to be loaded and writable.

Note:

The n and w flags remove attributes from the section, rather than adding them, so if they are used on their own it will be as if no flags had been specified at all.

If the optional argument to the .section directive is not quoted, it is taken as a subsegment number.

`.section <name>` (ELF version) This is one of the ELF section stack manipulation directives. The others are .subsection, .pushsection, .popsection, and .previous.

For ELF targets, the `.section` directive is used like this:

```
.section <name> [, "<flags>"[, \<type>[, \<entsize>]]]
```

The optional <flags> argument is a quoted string which may contain any combination of the following characters:

`a`	The section is allocatable.
`w`	The section is writable.
`x`	The section is executable.
`M`	The section can be merged.
`S`	The section contains zero terminated strings

The optional <type> argument may contain one of the following constants:

`@progbits`	The section contains data.
`@nobits`	The section does not contain data (i.e., section only occupies space).

If <flags> contains M flag, <type> argument must be specified as well as <entsize> argument. Sections with M flag but not S flag must contain fixed size constants, each <entsize> octets long. Sections with both M and S must contain zero terminated strings where each character is <entsize> bytes long. The linker may remove duplicates within sections with the same name, same entity size and same flags.

If no flags are specified, the default flags depend upon the section name. If the section name is not recognized, the default will be for the section to have none of the above flags: it will not be allocated in memory, nor writable, nor executable. The section will contain data.

`.set <symbol>, <expression>`

Set the value of <symbol> to <expression>. This changes <symbol> value and type to conform to <expression>. If <symbol> was flagged as external, it remains flagged. You may `.set` a symbol many times in the same assembly. If you `.set` a global symbol, the value stored in the object file is the last value stored into it. This directive is the same as `.equ`. Also see `.equiv`.

`.short <expressions>`

This may be a synonym for `.hword` or `.word`, depending on the platform. Also see **.int**.

`.single <flonums>`

This directive assembles zero or more flonums, separated by commas. It has the same effect as **.float**.

`.size <name> , <expression>`

This directive is used to set the size associated with a symbol <name>. The size in bytes is computed from <expression> which can make use of label arithmetic. This directive is typically used to set the size of function symbols.

`.skip <size> , <fill>`
: This directive emits <size> bytes, each of value <fill>. Both <size> and <fill> are absolute expressions. If the comma and <fill> are omitted, <fill> is assumed to be zero. This is the same as `.space`. Also see `. fill` , `.org`, and `. p2align`.

`.space <size> , <fill>`
: This is a synonym for `.skip`.

`.stabd, .stabn, .stabs`
: The `tricore-gcc` adds STABS (symbol table) debugging information to the assembly language code it generates, and this information is then included with the object code produced by the assembler. The assembler adds STABS information to the symbol table and string table appended to the end of each `.o` file. The linker combines the `.o` files into an executable file, combining the tables into a single symbol table, which is used by the debugger to identify sections of executable code. There are three directives that begin `.stab`. The symbols are not entered in the `tricore-as` hash table: they cannot be referenced elsewhere in the source file.

Up to five fields are required:

`<string>`	This is the symbol's name. It may contain any character except "000", so is more general than ordinary symbol names. Some debuggers used to code arbitrarily complex structures into symbol names using this field.
`<type>`	An absolute expression. The symbol's type is set to the low 8 bits of this expression. Any bit pattern is permitted, but `tricore-ld` and debuggers choke on silly bit patterns.
`<other>`	An absolute expression. The symbol's "other" attribute is set to the low 8 bits of this expression.
`<desc>`	An absolute expression. The symbol's descriptor is set to the low 16 bits of this expression.
`<value>`	An absolute expression which becomes the symbol's value.

If a warning is detected while reading a `.stabd`, `.stabn`, or `.stabs` statement, the symbol has probably already been created; you get a half-formed symbol in your object file. This is compatible with earlier assemblers!

`.stabd <type>, <other>, <desc>`
: The "name" of the symbol generated is not even an

empty string. It is a null pointer, for compatibility. Older assemblers used a null pointer so they didn't waste space in object files with empty strings.

The symbol's value is set to the location counter, relocatably. When your program is linked, the value of this symbol is the address of the location counter when the `.stabd` was assembled.

`.stabn <type>, <other>, <desc>, <value>`
The name of the symbol is set to the empty string "".

`.stabs <string>, <type>, <other>, <desc>, <value>`
All fields are specified.

`.string "<str>"` Copy the characters in <str> to the object file. You may specify more than one string to copy, separated by commas. Unless otherwise specified for a particular machine, the assembler marks the end of each string with a 0 byte. The backslash escape sequences defined for C can be used in the string.

`.struct <expression>`
Switch to the absolute section, and set the section offset to <expression>, which must be an absolute expression. You might use this as follows:

```
        .struct 0
field1:
        .struct field1 + 4
field2:
        .struct field2 + 4
field3:
```

This would define the symbol `field1` to have the value 0, the symbol `field2` to have the value 4, and the symbol `field3` to have the value 8. Assembly would be left in the absolute section, and you would need to use a `.section` directive of some sort to change to some other section before further assembly.

`.subsection <name>`
This is one of the ELF section stack manipulation directives. The others are `.section`, `.pushsection`, `.popsection`, and `.previous`

`.symver <name>, <name2@nodename>`
For the ELF object format, this directive binds the symbol to specific version nodes and is used when assembling code with a shared library. The symbol `name2@nodename` is created by this directive as an alias of name, which has been defined elsewhere in the same source file. The name2 portion of the alias is the actual external reference name to be resolved. The nodesname portion is the name of a node supplied to the linker on the command line.

`.tag <structname>`
This directive is generated by compilers to include auxiliary debugging

information in the symbol table. It is only permitted inside .def/.endef pairs. Tags are used to link structure definitions in the symbol table with instances of those structures. It is only valid for the COFF object format.

`.text <subsection>`
: Tells `tricore-as` to assemble the following statements onto the end of the text subsection numbered <subsection>, which is an absolute expression. If <subsection> is omitted, subsection number zero is used.

`.title "<heading>"`
: Use <heading> as the title (second line, immediately after the source file name and page number) when generating assembly listings. This directive affects subsequent pages, as well as the current page if it appears within ten lines of the top of a page.

`.type <name> , <type description>`
: (ELF version) This directive is used to set the type of symbol <name> to be either a function symbol or an object symbol. There are five different syntaxes supported for the <type description> field, in order to provide compatibility with various other assemblers. The syntaxes supported are:

```
.type <name>,#function
.type <name>,#object

.type <name>,\function
.type <name>,\object

.type <name>,%function
.type <name>,%object

.type <name>,"function"
.type <name>,"object"

.type <name> STT_FUNCTION
.type <name> STT_OBJECT
```

`.version "<string>"`
: This directive creates a .note section and places into it an ELF formatted note of type NT_VERSION. The note's name is set to string.

`.vtable_entry <table>, <offset>`
: This directive finds or creates a symbol table and a VTABLE_ENTRY relocation for it with an addend of offset.

`.vtable_inherit <child>, <parent>`
: This directive finds the symbol child and finds or creates the symbol parent and then creates a VTABLE_INHERIT relocation for the parent whose addend is the value of the child symbol. As a special case the parent name of 0 is treated as refering the *ABS* section.

`.weak <names>`
: This directive sets the weak attribute on the comma separated list of symbol names. If the symbols do not already exist, they will be created.

`.word <expressions>`
This directive expects zero or more <expressions>, of any section, separated by commas. This may be a synonym for `.hword` or `.short` depending on the platform.

19.2.2 Additional Pseudo-opcodes for TriCore

The TriCore version of `tricore-as` supports the following additional pseudo-opcodes:

`.code16` Directs the assembler to emit a 16-bit opcode for the next instruction following this directive. If it is not possible to find a 16-bit opcode, an error message is issued.

`.code32` Directs the assembler to emit a 32-bit opcode for the next instruction following this directive. With TriCore's current instruction sets, it is always possible to find a 32-bit equivalent for 16-bit opcodes.

`.optim` Directs the assembler to try to optimize the next instruction following this directive. This pseudo-opcode is honored even if the `--dont-optimize` option was specified.

`.noopt` Prevents the assembler from trying to optimize the next instruction following this directive.

`.word <expr>[,<expr>...]`
Evaluates each expression expr and appends their 32-bit results to the current section; these words are not auto-aligned unless the option −−enforce−aligned−data was specified. If an expr references an unknown symbol, a relocation entry will be generated for the linker to resolve.

`.blcomm` .blcomm name,size,align creates a symbol <name> in the local bss section, aligns it to align bytes (must be a power of two) and allocates size bytes for it in that section.

`.bit <bname>[,bexpr]`
Creates a global bit variable named <bname> and, if specified, initializes it with the value of the absolute expression bexpr, whose result must be a value of 0 or 1. If bexpr is omitted, <bname> will be initialized with the value zero. A bit variable consists of two parts: a byte address and a bit position within this byte that denotes where the contents of the variable is actually stored. `tricore-as` uses the special sections .bdata and .boffs to maintain these two parts, so do *not* use these sections for any other purposes. To access a bit variable, use its symbolic name to specify the byte address, and the prefix bpos: to specify the bit position (e.g., .bit foo; st.t foo,bpos:foo,1).

Note:

`tricore-as` reserves one byte of memory for each bit variable; when linking the final application, you may pass the option `--relax-bdata` (or `--relax`) to `tricore-ld` in order to compress bit data sections such that up to eight bit variables are allocated in a single byte. Note also that the `st.t` instruction requires that a bit variable is absolutely addressable, so you need to make sure that all `.bdata` sections are allocated in an absolutely addressable memory area.

`.lbit <bname>[,bexpr]`
This does the same as `.bit` (described above), except that the bit variable that is being created will have local scope, i.e., it will not be visible outside the current module.

Note:

It is not possible to use the `.global` pseudo-opcode to later change the scope of a bit variable created by `.lbit` from local to global.

`.bpos, .bposb, .bposh, .bposw`
They take the name of a bit variable as argument and output the bit position of that variable as a 1-, 2- or 4-byte value to the current section; this is mainly needed by the compiler in order to emit debug information for bit variables.

`.toc` Enters the `.toc` section; if this section doesn't exist already, it will be created automatically. The `.toc` section was used to store data that can be quickly addressed using a function's toc pointer (`%a12`). This directive still exists for historical reasons.

`.rodata` Enters the `.rodata` section; if this section doesn't exist already, it will be created automatically. The ".rodata" section is used to store read-only data (e.g., constants).

`.sdata` Enters the `.sdata` section; if this section doesn't exist already, it will be created automatically. The ".sdata" section is used to store initialized data that can be quickly addressed using the small data area pointer (%a0).

`.sbss` Enters the `.sbss` section; if this section doesn't exist already, it will be created automatically. The ".sbss" section is used to store uninitialized data that can be quickly addressed using the small data area pointer (%a0).

`.zdata` Enters the `.zdata` section; if this section doesn't exist already, it will be created automatically. The ".zdata" section is used to store initialized data that can be quickly addressed using TriCore's absolute addressing mode.

`.zbss` Enters the `.zbss` section; if this section doesn't exist already, it will be created automatically. The ".zbss" section is used to store uninitialized data that can be quickly addressed using TriCore's absolute addressing mode.

`.pcptext` Enters the PCP (Peripheral Control Processor) text section; if this section doesn't exist already, it will be created automatically. The PCP text section is used to store PCP code. Upon entering this section, `tricore-as` expects PCP mnemonics instead of TriCore instructions; you can return to the normal behavior simply by leaving the PCP text section, e.g. with `.text`, which (re-) enters the TriCore text section. This makes it possible to embed PCP programs as inline assembler in C or C++ modules, provided you're using the TriCore port of GCC, the GNU compiler collection.

`.pcpdata` Enters the PCP (Peripheral Control Processor) data section; if this section doesn't exist already, it will be created automatically. The PCP data section is used to store contexts and parameters for the PCP.

Note:

The PCP can access this section only word-wise (i.e., in quantities of 32 bits), so you should only use `.word` pseudo-ops to fill it, even though `tricore-as` also accepts `.byte`, `.half` and other pseudo-ops that can be used to define memory values of various sizes.

`.pcpinitword` `.pcpinitword initPC, initDPTR, initFLAGS` produces a 32-bit value that can be used to initialize PCP register `R7` from a channel's context save area if the PCP is operated in 'Channel Resume Mode'. initPC is typically a label in a PCP code section that points the first instruction the PCP should execute when the respective channel is triggered. initPC may also be specified as an absolute expression that results in a value between 0 and 65535. initDPTR is typically a label in a PCP data section that points to the base address of a channel's parameter and data storage area. The linker will issue a warning message if the label is not aligned to a 256-byte boundary; this is because the PCP accesses the PRAM indirectly using a base-plus-offset addressing mode that can address 64 32-bit words (256 bytes) starting at base, and because the base address is always forced to a 256-byte boundary by hardware, if the label is not aligned to a 256-byte boundary, only the word starting at this label is guaranteed to be accessible. initDPTR may also be specified as an absolute expression that results in a value between 0 and 255. initFLAGS are the initial status flags to be loaded into the least significant eight bits of register `R7`; they may be specified either as an absolute or symbolic expression that results in a value between 0 and 255. Note: because of the special semantics of this pseudo-opcode, there is no need (actually, it is even an error) to use any PCP-specific operand prefixes for labels used in symbolic expressions for initPC and initDPTR.

Example

The pseudoopcode .pcpinitword is used in `file1.s` to initialize the PC and the DPTR to the labels start_chan0 and start_page0, which are global symbols in `file2.s`

```
# file1.s
# initialize the PC and the DPTR

.pcpdata
#channel 0
.pcpinitword start_chan0, start_page0, 0

# file2.s

.pcptext

.globl start_chan0
start_chan0:

#code...

.pcpdata
.globl start_page0
start_page0:

#data...
```

`.uahalf <expr>` Evaluates the expression expr and appends the 16-bit result to the current section without any alignment. Used for storing DWARF debugging information.

`.uaword <expr>` Evaluates the expression expr and appends the 32-bit result to the current section without any alignment. Used for storing DWARF debugging information.

`.uaxword <expr>` Evaluates the expression expr and appends the 64-bit result to the current section without any alignment. Used for storing DWARF debugging information.

19.2.2.1 TriCore Assembler Syntax

19.2.2.2 Special Characters

`#` is the line comment character; i.e., all characters following # to the end of the line are ignored.

`;` can be used instead of a new line to separate statements.

Floating point numbers can be identified by a `0f` or `0F` prefix (single precision, 4 bytes), and by a `0d` or `0D` prefix (double precision, 8 bytes). Examples: `0f12.456`, `0D1.2345e12`. In addition, you can use the **.float** and **.double** pseudo-opcodes to generate IEEE-compliant floating point numbers.

19.2.2.3 Register Names

The TriCore architecture has 16 general purpose data registers which can be referred to as %d0-%d15 or %D0-%D15. A pair of two consecutive data registers (starting with an even-numbered register) can form an extended data register. Eight such extended registers exist on the TriCore, and their names are %e0, %e2, %e4, . . . , %e14. You can also use a capital 'E' to denote an extended register (e.g., %E6). Alternatively, you may also use the notation %dn+ or %Dn+ to refer to the extended data register %en (n = 0, 2, 4, ..., 14). Several TriCore instructions perform different operations depending on whether an operand is a single or an extended data register. For instance, mul %e0,%d3,%d4 computes the 64-bit product of registers %d3 and %d4 and assigns it to %d0 and %d1, while mul %d0,%d3,%d4 computes the 32-bit product of %d3 and %d4 and assigns it to register %d0.

The 16 general purpose address registers can be specified as %a0-%a15 or %A0-%A15. The stack pointer (%a10) may also be specified as %sp or %SP. Some TriCore instructions and address modes require an address register pair. However, unlike the data registers, there is no special notation for such register pairs and you simply need to specify the name of the pair's first (even-numbered) address register.

The special function registers (SFRs) of the TriCore architecture must be prefixed with a "$" character; their names are those defined in the 'TriCore Architecture Manual', and they are parsed case-insensitivly. For instance, mfcr %d8,$psw is identical to mfcr %d8,$PSW. Core SFRs are resolved to an unsigned 16-bit number representing their offset from TriCore's SFR base address, so you can use them as an operand for all TriCore instructions that allow 16-bit constants (in other words, their use is not restricted to mfcr and mtcr instructions).

Core v1.0 $dbiten.

Core ≥ v1.2 $dpr2_0l, $dpr2_0u, $dpr2_1l, $dpr2_1u, $dpr2_2l, $dpr2_2u, $dpr2_3l, $dpr2_3u, $dpr3_0l, $dpr3_0u, $dpr3_1l, $dpr3_1u, $dpr3_2l, $dpr3_2u, $dpr3_3l, $dpr3_3u, $cpr0_2l, $cpr0_2u, $cpr0_3l, $cpr0_3u, $cpr1_2l, $cpr1_2u, $cpr1_3l, $cpr1_3u, $cpr2_0l, $cpr2_0u, $cpr2_1l, $cpr2_1u, $cpr2_2l, $cpr2_2u, $cpr2_3l, $cpr2_3u, $cpr3_0l, $cpr3_0u, $cpr3_1l, $cpr3_1u, $cpr3_2l, $cpr3_2u, $cpr3_3l, $cpr3_3u, $cpuid v1.2, $cpu_id v1.2.

Core v1.0 or ≤ v1.3 $d0, $d1, $d2, $d3, $d4, $d5, $d6, $d7, $d8, $d9, $d10, $d11, $d12, $d13, $d14, $d15, $a0, $a1, $a2, $a3, $a4, $a5, $a6, $a7, $a8, $a9, $a10, $a11, $a12, $a13, $a14, $a15.

Core ≥ v1.3 $mmucon, $mmu_con, $asi, $mmu_asi, $mmuid, $mmu_id, $tva, $mmu_tva, $tpa, $mmu_tpa, $tpx, $mmu_tpx, $tfa, $mmu_tfa.

Core ≥ v2.0 $mmu_lpma, $mmu_tfas, $dspr, $dcache, $memtr, $datr, $dttar, $pspr, $pcache, $pcon, $pstr.

All cores $dpr0_0l, $dpr0_0u, $dpr0_1l, $dpr0_1u, $dpr0_2l, $dpr0_2u, $dpr0_3l, $dpr0_3u, $dpr1_0l, $dpr1_0u, $dpr1_1l, $dpr1_1u, $dpr1_2l, $dpr1_2u, $dpr1_3l, $dpr1_3u, $cpr0_0l, $cpr0_0u, $cpr0_1l, $cpr0_1u, $cpr1_0l, $cpr1_0u, $cpr1_1l, $cpr1_1u, $dpm0_0, $dpm0_1, $dpm0_2, $dpm0_3, $dpm1_0, $dpm1_1, $dpm1_2, $dpm1_3, $cpm0_0, $cpm0_1, $cpm1_0, $cpm1_1, $dbgsr, $gprwb, $exevt, $crevt, $swevt, $tr0evt, $tr1evt, $pcxi, $psw, $pc, $biv, $btv, $isp, $icr, $fcx, $lcx, $syscon.

In addition, all non-core SFRs that can be accessed using TriCore's 18-bit absolute addressing mode are known to `tricore-as` which allows instructions such as `st.w $buscon0,%d13`. The value of these symbols is the 32-bit address of the corresponding SFR.

$pwrclc, $pwrid, $rstreq, $rstsr, $wdtcon0, $wdtcon1, $wdtsr, $nmisr, $pmcon, $pmcsr, $pllclc, $eckclc, $icuclc, $stmclc, $stmid, $systim0, $systim1, $systim2, $systim3, $systim4, $systim5, $systim6, $systim7, $jdpid, $comdata, $iosr, $ebucon, $drmcon, $drmstat, $addsel0, $addsel1, $addsel2, $addsel3, $addsel4, $addsel5, $addsel6, $addsel7, $buscon0, $buscon1, $buscon2, $buscon3, $buscon4, $buscon5, $buscon6, $buscon7, $gtclc, $gtid, $t01irs, $t01ots, $t2con, $t2rccon, $t2ais, $t2bis, $t2es, $gtosel, $gtout, $t0dcba, $t0cba, $t0rdcba, $t0rcba, $t1dcba, $t1cba, $t1rdcba, $t1rcba, $t2, $t2rc0, $t2rc1, $t012run, $gtsrsel, $gtsrc0, $gtsrc1, $gtsrc2, $gtsrc3, $gtsrc4, $gtsrc5, $gtsrc6, $gtsrc7, $pcpclc, $pcpid, $pcpcs, $pcpes, $pcpicr, $pcpsrc3, $pcpsrc2, $pcpsrc1, $pcpsrc0.

19.2.2.4 Operand Prefixes

The following operand prefixes are recognized by the `tricore-as`:

`hi:<symbol_or_expression>`
Returns the high part (16 bits) of <symbol_or_expression> by adding 0x8000 to the 32-bit value of <symbol_or_expression> and then right-shifting the result by 16 bits. If <symbol_or_expression> isn't known at assemble time, a special relocation entry is generated for the liner to resolve.

```
movh.a %a15,hi:foo
```

`lo:<symbol_or_expression>`
Returns the lower 16 bits of <symbol_or_expression>. If the 32-bit value of <symbol_or_expression> isn't known at assemble time, a special relocation entry is generated for the linker to resolve.

```
lea %a15,[%a15]lo:foo
```

`sm:<symbol>`
Returns the 10-bit offset of <symbol> within the small data area; since the assembler cannot know this offset, it generates a special relocation entry for the linker to resolve.

```
ld.w %d4,[%a0]sm:foo
```

`up:<symbol_or_expression>`
Returns the upper 16 bits of <symbol_or_expression>. If the value of <symbol_or_expression> isn't known at assemble time, a special relocation entry is generated for the linker to resolve.

```
mov %d12,up:foo
```

`bpos:<symbol>`
Returns the 3-bit bit position of bit variable <symbol>. (Actually, this computation is done by the linker; the assembler merely generates a special relocation entry for the current instruction.) It is an error if <symbol> has not been created by a `.bit` or `.lbit` pseudo-opcode.

```
ld.b %d4,mybit; extr.u %d4,%d4,bpos:mybit,1}
```

When operating in PCP mode, `tricore-as` accepts the following prefixes:

`cptr:<symbol>` Returns the 16-bit PCP code address of the label <symbol>, which must be defined in some PCP code section. (Actually, this computation is done by the linker; the assembler merely generates a special relocation entry for the current instruction.)

```
ldl.il r2,cptr:return_address
```

Note:

It is not required to use this prefix when specifying symbolic target addresses for the jump instructions jc, jc.a and jl, because it is clear in these cases that <symbol> should resolve to its PCP code address.

`dptr:<symbol>` Returns the 8-bit PRAM data page number, left-shifted by 8 bits, of the label <symbol>, which must be defined in some PCP data section; the actual value returned is thus 16 bits wide, with the low byte cleared. (Actually, this computation is done by the linker; the assembler merely generates a special relocation entry for the current instruction).

```
ldl.il r7,dptr:chan4_vars
```

Note:

<symbol> must be at least word-aligned, and should be aligned to a 256-byte boundary. For an explanation, see .pcpinitword.

`doff:<symbol>` Returns the 6-bit PRAM data page offset of the label <symbol>, which must be defined in some PCP data section. (Actually, this computation is done by the linker; the assembler merely generates a special relocation entry for the current instruction).

```
ld.pi r3,[doff:X_val]; ldl.il r5,doff:foo
```

Note:

<symbol> must be word-aligned. Also, it is not required to use this prefix when specifying symbolic offsets for instructions that indirectly access the PRAM (i.e., all PCP instructions carrying the suffix .PI), because it is clear in these cases that <symbol> should resolve to its PRAM data page offset.

19.2.2.5 TriCore Opcodes

The opcodes (mnemonics) that are recognized by `tricore-as` are exactly those described in the 'TriCore Architecture Manual'.

No special suffix is available to identify 16-bit instructions, but you can always use the .code16 and .code32 pseudo-opcodes to explicitly select the 16-bit or 32-bit version of an instruction.

There is an additional opcode, not, that is recognized and automatically converted into a nor instruction. So, you can write not %dn instead of nor %dn or nor %dn,%dn,0.

The syntax of the opcodes, operands and addressing modes is also exactly as described in the manual mentioned above, except that registers must be prefixed with a `%` character, and SFRs must be prefixed with a `$` character. This is necessary in order to distinguish between register names and user-defined symbols, and between registers and SFRs, respectively (note that `%d0` is different from `$d0`: the former represents the first data register, while the latter represents the offset of this register to the SFR base address).

All PC-relative jump and call instructions are subject to relaxation in order to find the smallest possible instruction (or instruction sequence) that can reach the specified target. The maximum allowed displacement for local branches is +/-16 MB. `call`, `fcall`, `j` and `jl` instructions branching to global functions can be relaxed by the linker to reach any valid code address within TriCore's entire 32-bit address space; you need to pass the option `--relax-24rel` (or `--relax`) to `tricore-ld` in order to enable this feature.

19.2.2.6 TriCore PCP support

The `tricore-as` recognizes PCP (Peripheral Control Processor) instructions up to V2.0. All you need to do in order to specify PCP code or data is to use the pseudo-opcodes `.pcptext` and `.pcpdata`, respectively, which enter the relevant sections. Alternatively, you may also define arbitrarily named PCP code and data sections by adding the letter 'p' to the section flags for these sections. For instance, `.section .pcode,"axp",@progbits` defines (and enters) a PCP code section named `.pcode`; likewise, use the section flags `"awp"` to define a PCP data section.

The startup code (crt0.o, part of the TriCore port of GCC) takes care of copying the PCP code and data sections to TriCore's internal PCODE and PRAM memory areas (as defined in a linker script), so if your C or C++ program that makes use of the PCP reaches its `main()` function, the PCP code and data sections are already in place. Of course, you still need to initialize all relevant hardware registers in order to activate the PCP; see the TriCore and PCP manuals for details.

There are no special provisions in `tricore-as` to specify for which particular version of PCP you want to assemble code. If you want to assemble for PCP 1.x, just don't use instructions or features that are only available in PCP 2.0. Reversely, if PCP 2.0 introduced new operands for certain instructions, just specify them with a value of zero, should you need compatibility with PCP 1.x.

You can write PCP instructions and operands in small or capital letters (e.g., 'NeG cC_Uc,R4,r5' and 'neg cc_uc,r4,r5' are identical), except for symbols and labels, which must be written exactly as you defined them. If an instruction takes several operands, they need to be separated by commas (e.g. 'comp.i r3,32'); there can be any number of spaces and tabs between the mnemonic and the first operand, as well as between an operand and the comma preceding or following it (unless it's the first or last operand, where no preceding or trailing comma is allowed).

Please note that there are no special prefixes to distinguish keywords reserved by the PCP assembler (e.g. unlike with TriCore instructions, registers are not prefixed with a percent character ("%")), so you need to make sure that the names of labels and symbols don't clash with such PCP keywords: it's not wise to call a label or symbol 'cc_uc', as it would interfere with PCP's condition codes, or 'r6', which `tricore-as` would recognize as PCP register R6.

Upon entering a PCP code or data section, `tricore-as` recognizes PCP instructions exactly as described in the 'PCP 2.0 Assembler Mnemonics Specification, Mar. 2000', with the following additions and exceptions:

`DEBUG`	The debug instruction takes a parameter that describes whether the program, when resumed, should be continued at the same or the next instruction. This parameter can be either <RTA> or <RNA>.
`MSTEP`	You may write either mstep.l or mstep32, as well as either mstep.u or mstep64.
`LDL`	According to the specification, ldl.iu and ldl.il take a 16-bit expression as their second operand. as, however, expects a 32-bit expression; in case of ldl.iu, the upper 16 bits of the expression's value are taken, while for ldl.il the lower 16 bits are taken. If you need to port existing PCP code to GNU as, make sure to left-shift the expression for ldl.iu instructions by 16 bits (e.g. 'ldl.iu r4,1234 $\ll$ 16').
`JC`	If this jump instruction cannot reach its target (due to its limited 6-bit offset), `tricore-as` automatically expands it into a jc.a instruction that can reach any address within the PCP code space. In addition, if the target address of an unconditional jc instruction (e.g, jc label,cc_uc) cannot be reached with a 6-bit offset, but with a 10-bit offset, it will be replaced with a jl instruction.
`JL`	If this jump instruction cannot reach its target (due to its limited 10-bit offset), `tricore-as` automatically expands it into a jc.a instruction that can reach any address within the PCP code space.
`EXB, EXIB`	In accordance with PCP Assembler Mnemonics Specification, two additional opcodes are supported by `tricore-as` for extracting (inverted) bits: 'exb ra,n' is equivalent to 'chkb ra,n,set', and 'exib ra,n' is equivalent to 'chkb ra,n,clr' (where 'ra' is one of the registers r0−r7, and 'n' denotes the number of the bit to be extracted).
`Indirection`	If an operand (register or constant) is used to access memory indirectly, you may, at your option, wrap it in square brackets (e.g. '[r4]'). This is completely in compliance with the specification mentioned above; however, there are no options which let you specify if the use of such indirection specifiers (read: square brackets) is illegal, optional, or mandatory. This means you can't change the default, which is "optional". Of course, if you use indirection specifiers at places where they're not allowed, you'll get an error message, which again is compliant with the Assembler Mnemonics Specification.

With the exception of the three 32-bit instructions ldl.iu, ldl.il and jc.a, the two PC-relative jump instructions jl and jc, and instructions carrying a .PI suffix, all symbols used in operand expressions must be known at the time the assembler evaluates these expressions in order to finalize the instruction's opcode.

It seems that in some PCP manuals issued by Infineon different names are used for the operand `CNT0` that is part of the `copy` and `bcopy` instructions. Sometimes it is referred to as `RC0`, sometimes as `BRST`, and maybe there are even other variations mentioned. Please use the operand name `CNT0` to set the reload value for counter 0 in the above instructions.

20 The TriCore Linker

`tricore-ld` combines a number of object and archive files, relocates their data and ties up symbol references. Usually the last step in compiling a program is to run `tricore-ld`. (see Figure 1.1 on page 7)

`tricore-ld` accepts Linker Command Language files written in a superset of AT&T's Link Editor Command Language syntax, to provide explicit and total control over the linking process.

This version of `tricore-ld` uses the general purpose BFD libraries to operate on object files. This allows `tricore-ld` to read, combine, and write object files in many different formats—for example, COFF or `a.out`. Different formats may be linked together to produce any available kind of object file.

Aside from its flexibility, the `tricore-ld` is more helpful than other linkers in providing diagnostic information. Many linkers abandon execution immediately upon encountering an error; whenever possible, `tricore-ld` continues executing, allowing you to identify other errors or, in some cases, to get an output file in spite of the error.

20.1 Command Line Options

The linker supports a lot of command-line options, but in practice few of them are used in any particular context.

Instance, a frequent use of ld is to link Unix object files on an Unix system. On such a system, to link a file `hello.o`:

```
tricore-ld -o <output> /lib/crt0.o hello.o -lc
```

This tells tricore-ld to produce a file called <output> as the result of linking the file `/lib/crt0.o` with `hello.o` and the library `libc.a`, which will come from the standard search directories. (See the discussion of the `-l` option below.)

Some of the command-line options to tricore-ld may be specified at any point in the command line. However, options which refer to files, such as `-l` or `-T`, cause the file to be read at the point at which the option appears in the command line, relative to the object files and other file options. Repeating non-file options with a different argument will either have no further effect, or override prior occurrences (those further to the left on the command line) of that option. Options which may be meaningfully specified more than once are noted in the descriptions below.

Non-option arguments are object files or archives which are to be linked together. They may follow, precede, or be mixed in with command-line options, except that an object file argument may not be placed between an option and its argument.

Usually the linker is invoked with at least one object file, but you can specify other forms of binary input files using `-l`, `-R`, and the script command language. If *no* binary input files at all are specified, the linker does not produce any output, and issues the message 'No input files'.

If the linker can not recognize the format of an object file, it will assume that it is a linker script. A script specified in this way augments the main linker script used for the link (either the default linker script or the one specified by using `-T`). This feature permits the linker to link against a file which appears to be an object or an archive, but actually merely defines some symbol values, or uses `INPUT` or `GROUP` to load other objects.

Note:

Specifying a script in this way merely augments the main linker script; use the `-T` option to replace the default linker script entirely.

For options whose names are a single letter, option arguments must either follow the option letter without intervening whitespace, or be given as separate arguments immediately following the option that requires them.

For options whose names are multiple letters, either one dash or two can precede the option name; for example, `-trace-symbol` and `--trace-symbol` are equivalent.

Note:

There is one exception to this rule. Multiple letter options that start with a lower case 'o' can only be preceded by two dashes. This is to reduce confusion with the `-o` option. So for example `-omagic` sets the output file name to `magic` whereas `--omagic` sets the `NMAGIC` flag on the output.

Arguments to multiple-letter options must either be separated from the option name by an equals sign, or be given as separate arguments immediately following the option that requires them. For example, `--trace-symbol foo` and `--trace-symbol=foo` are equivalent. Unique abbreviations of the names of multiple-letter options are accepted.

If the linker is being invoked indirectly, via a compiler driver (e.g. `tricore-gcc`) then all the linker command line options should be prefixed by `-Wl,` (or whatever is appropriate for the particular compiler driver) like this:

```
tricore-gcc -Wl,--startgroup foo.o bar.o -Wl,--endgroup
```

This is important, because otherwise the compiler driver program may silently drop the linker options, resulting in a bad link.

`-(, --start-group <archives>`

The <archives> should be a list of archive files. They may be either explicit file names, or `-l` options.

The specified archives are searched repeatedly until no new undefined references are created. Normally, an archive is searched only once in the order that it is specified on the command line. If a symbol in that archive is needed to resolve an undefined symbol referred to by an object in an archive that appears later on the command line, the linker would not be able to resolve that reference. By grouping the archives, they all be searched repeatedly until all possible references are resolved.

Using this option has a significant performance cost. It is best to use it only when there are unavoidable circular references between two or more archives.

`-), --end-group`
: End a group.

`-A <architecture>, --architecture <architecture>`
: Set architecture

`-a KEYWORD`
: Shared library control for HP/UX compatibility

`--allow-multiple-definition`
: Normally when a symbol is defined multiple times, the linker will report a fatal error. This option allows multiple definitions and the first definition will be used.

`--allow-shlib-undefined`
: Allow undefined symbols in shared objects even if `--no-undefined` is set. The net result will be that undefined symbols in regular objects will still trigger an error, but undefined symbols in shared objects will be ignored. The implementation of `--no-undefined` makes the assumption that the runtime linker will choke on undefined symbols. However there is at least one system (BeOS) where undefined symbols in shared libraries is normal since the kernel patches them at load time to select which function is most appropriate for the current architecture. This means dynamically select an appropriate memset function.

`-assert <keyword>`
: This option is ignored for SunOS compatibility.

`-b <target>, --format <target>`
: Specify target for following input files

`-Bdynamic, -dy, -call_shared`
: Link against dynamic libraries. This is only meaningful on platforms for which shared libraries are supported. This option is normally the default on such platforms. The different variants of this option are for compatibility with various systems. You may use this option multiple times on the command line: it affects library searching for `-l` options which follow it.

`-Bstatic, -dn, -non_shared, -static`
: Do not link against shared libraries. This is only meaningful on platforms for which shared libraries are supported. The different variants of this option are for compatibility with various systems. You may use this option multiple times on the command line: it affects library searching for `-l` options which follow it.

`-Bsymbolic` When creating a shared library, bind references to global symbols to the definition within the shared library, if any. Normally, it is possible for a program linked against a shared library to override the definition within the shared library. This option is only meaningful on ELF platforms which support shared libraries.

`-c` The `-c` is a synonym of the `-T` option.

`--check-sections`
Asks the linker *not* to check section addresses after they have been assigned to see if there any overlaps. Normally the linker will perform this check, and if it finds any overlaps it will produce suitable error messages. The linker does know about, and does make allowances for sections in overlays. The default behavior can be restored by using the command line switch `--check-sections`.

v3.4 `-check-sdata` The linker will issue a warning if a variable that is defined within a small data section or small addressing attribute and is referenced in a diffrent module with a different addressing mode. With the linker option `--fatal-warnings` the generation of an executable `a.out` will be disabled.

```
/* Module 1 */
#pragma section .sdata aws
int var;
#pragma section

extern int func(void);

int
main (void)
 {
  var = func();
 }

/* Module 2 */
extern int var;

int
func (void)
 {
  return var;
 }
```

The TriCore Development Platform compiler support the small addressing mode for TriCore. The small data sections are prefixed by `.sdata`. The TriCore registers `a0`, `a1`, `a8`, `a9` are available for small addressing mode. The user can add these output sdata sections in the linker script (see section 14.2 on page 74 for details). This check is relevant if position independent data is required. One goal of stable software is to use it on different hardware without modification. The position independent brings this advantage to you, this means the location of your data may vary, but this has no effect for the software, because all relevant data is addressed small/relative. However the startup code must be modified to configure the register (e.g `a9`) for relative addressing mode.

Note:

The debug info contains always absolute addresses, so if the register for relative addressing mode is modified without a rebuilt of your project, the debug info will not be valid anymore.

The option `-check-sdata` checks if the small data is always accessed using the small addressing mode, otherwise these data will not be position independent.

`--mcpu=<core>` Option to check linking of objects with different cores (tc12, tc13, tc131, tc2).

`--cref` Output a cross reference table. If a linker map file is being generated, the cross reference table is printed to the map file. Otherwise, it is printed on the standard output.

The format of the table is intentionally simple, so that it may be easily processed by a script if necessary. The symbols are printed out, sorted by name. For each symbol, a list of file names is given. If the symbol is defined, the first file listed is the location of the definition. The remaining files contain references to the symbol.

`-d, -dc, -dp` These three options are equivalent; multiple forms are supported for compatibility with other linkers. They assign space to common symbols even if a relocatable output file is specified (with `-r`). The script command `FORCE_COMMON_ALLOCATION` has the same effect.

`--defsym <symbol>=<expression>`

Create a global symbol in the output file, containing the absolute address given by <expression>. You may use this option as many times as necessary to define multiple symbols in the command line. A limited form of arithmetic is supported for the <expression> in this context: you may give a hexadecimal constant or the name of an existing symbol, or use `+` and `-` to add or subtract hexadecimal constants or symbols.

`--demangle [=<style>]`

These options control whether to demangle symbol names in error messages and other output. When the linker is told to demangle, it tries to present symbol names in a readable fashion: it strips leading underscores if they are used by the object file format, and converts C++ mangled symbol names into user readable names. Different compilers have different mangling styles. The optional demangling style argument can be used to choose an appropriate demangling style for your compiler. The linker will demangle by default unless the environment variable `COLLECT_NO_DEMANGLE` is set. These options may be used to override the default.

`--discard-none` Don't discard any local symbols

`-e <address>, --entry <address>`
: Use <address> as the explicit symbol for beginning execution of your program, rather than the default entry point. If there is no symbol named <address>, the linker will try to parse <address> as a number, and use that as the entry address (the number will be interpreted in base 10; you may use a leading `0x` for base 16, or a leading `0` for base 8).

`-E, --export-dynamic`
: When creating a dynamically linked executable, add all symbols to the dynamic symbol table. The dynamic symbol table is the set of symbols which are visible from dynamic objects at run time.

  If you do not use this option, the dynamic symbol table will normally contain only those symbols which are referenced by some dynamic object mentioned in the link.

  If you use `dlopen` to load a dynamic object which needs to refer back to the symbols defined by the program, rather than some other dynamic object, then you will probably need to use this option when linking the program itself.

  You can also use the version script to control what symbols should be added to the dynamic symbol table if the output format supports it. See the description of `--version-script`.

`-EB` Link big-endian objects. This affects the default output format.

`-EL` Link little-endian objects. This affects the default output format.

`--embedded-relocs`
: This option is only meaningful when linking MIPS embedded position independent code, generated by the `-membedded-pic` option to the `tricore-gcc` and `tricore-as`. It causes the linker to create a table which may be used at runtime to relocate any data which was statically initialized to pointer values.

`--enable-new-dtags, --disable-new-dtags`
: This linker can create the new dynamic tags in ELF. But the older ELF systems may not understand them. If you specify the option `--enable-new-dtags`, the dynamic tags will be created as needed. If you specify `--disable-new-dtags`, no new dynamic tags will be created. By default, the new dynamic tags are not created.

Note:

Those options are only available for ELF systems with shared library suppport.

`--extmap=<output-option>`

Generates an extended map file with additional information. This option must be used together with either `--Map` or `-M`. Valid output-options for `--extmap` are:

option	effect
h	print header (version, date, ...)
L	list global symbols sorted by name
l	list all symbols sorted by name
N	list global symbols sorted by address
n	list all symbols sorted by address
m	list memory segments
a	all of the above.

For a detailed description of the extended map file see section 20.4 on page 282.

`--pcpmap[=type]`

Use this option if your particular TriCore MCU requires an address mapping between PCP and TriCore memory accesses. This is currently only the case for the TC1796, where memory accesses performed by the PCP to addresses in TriCore's memory segments 12 and 13 are mapped to segment 14 by the LFI bus bridge.

The "type" argument to the `--pcpmap` option may be omitted and defaults to "0" in this case; if specified, however, it must be "0" or "tc1796". When the `--pcpmap` option is given, the linker changes its behavior as follows:

- all symbolic data accesses by PCP data and code sections are subject to the following memory mapping:

  ```
  Source address range: Offset: Target address range:
  ---------------------------------------------------------
  0xc0000000..0xc03fffff 0x28000000 0xe8000000..0xe83fffff
  0xd0000000..0xd00fffff 0x18400000 0xe8400000..0xe84fffff
  0xd4000000..0xd40fffff 0x14500000 0xe8500000..0xe85fffff
  ```

  So, for instance, if a PCP code or data section references a variable called `c_var` that is located at address 0xc0000004, the linker will automatically add the appropriate offset (0x28000000) to the address, such that the PCP will access it at address 0xe8000004, where the LFI bus bridge will mirror it at runtime.

- the same mapping applies to all sections whose name is either `.pcp_c_ptr_init`, or starts with the string `.pcp_c_ptr_init.`; you can use this feature for C/C++ functions that exclusively assign the addresses of C/C++ variables to PCP variables located in PRAM.

> **Note:**
>
> The option `--pcpmap` can only map variables, no functions. Use this option only while final linking, not together with the option `-r`

`-f SHLIB, --auxiliary <name>`

When creating an ELF shared object, set the internal `DT_AUXILIARY` field to the specified name. This tells the dynamic linker that the symbol table of the shared object should be used as an auxiliary filter on the symbol table of the shared object <name>.

If you later link a program against this filter object, then, when you run the program, the dynamic linker will see the `DT_AUXILIARY` field. If the dynamic linker resolves any symbols from the filter object, it will first check whether there is a definition in the shared object <name>. If there is one, it will be used instead of the definition in the filter object. The shared object <name> need not exist. Thus the shared object <name> may be used to provide an alternative implementation of certain functions, perhaps for debugging or for machine specific performance.

This option may be specified more than once. The `DT_AUXILIARY` entries will be created in the order in which they appear on the command line.

`-F <name>, --filter <name>`

When creating an ELF shared object, set the internal `DT_FILTER` field to the specified name. This tells the dynamic linker that the symbol table of the shared object which is being created should be used as a filter on the symbol table of the shared object <name>.

If you later link a program against this filter object, then, when you run the program, the dynamic linker will see the `DT_FILTER` field. The dynamic linker will resolve symbols according to the symbol table of the filter object as usual, but it will actually link to the definitions found in the shared object <name>. Thus the filter object can be used to select a subset of the symbols provided by the object <name>.

The `tricore-ld` uses other mechanisms for this purpose: the `--format`, `--oformat`, `-b` options, the `TARGET` command in linker scripts, and the `GNUTARGET` environment variable. The `tricore-ld` will ignore the `-F` option when not creating an ELF shared object.

`--fatal-warnings`

Treat all warnings as errors.

`-fini <symbol>` When creating an ELF executable or shared object, call <symbol> when the executable or shared object is unloaded, by setting `DT_FINI` to the address of the function. By default, the linker uses `_fini` as the function to call.

`--force-exe-suffix`

Force generation of file with `.exe` suffix. If a successfully built fully linked output file does not have a `.exe` or `.dll` suffix, this option forces the linker to copy the output file to one of the same name with a `.exe` suffix. This option is useful when using unmodified Unix makefiles on a Microsoft Windows host, since some versions of Windows won't run an image unless it ends in a `.exe` suffix.

`--gc-sections` Enable garbage collection of unused input sections. It is ignored on targets that do not support this option. This option is not compatible with `-r`, nor should it be used with dynamic linking. The default behavior (of not performing this garbage collection) can be restored by specifying `--no-gc-sections` on the command line.

`-h <filename>, -soname <filename>`
When creating an ELF shared object, set the internal `DT_SONAME` field to the specified name. When an executable is linked with a shared object which has a `DT_SONAME` field, then when the executable is run the dynamic linker will attempt to load the shared object specified by the `DT_SONAME` field rather than using the file name given to the linker.

`--help` Print a summary of the command-line options on the standard output and exit.

`-I <program>, --dynamic-linker <program>`
Set the name of the dynamic linker to <program>. This is only meaningful when generating dynamically linked ELF executables. The default dynamic linker is normally correct; don't use this unless you know what you are doing.

`-init <symbol>` When creating an ELF executable or shared object, call <symbol> when the executable or shared object is loaded, by setting `DT_INIT` to the address of the function. By default, the linker uses `_init` as the function to call.

`-L <directory>, --library-path <directory>`
Add path <directory> to the list of paths that `tricore-ld` will search for archive libraries and `tricore-ld` control scripts. You may use this option any number of times. The directories are searched in the order in which they are specified on the command line. Directories specified on the command line are searched before the default directories. All `-L` options apply to all `-l` options, regardless of the order in which the options appear.

Note:

The paths can also be specified in a link script with the `SEARCH_DIR` command. Directories specified this way are searched at the point in which the linker script appears in the command line.

`-l <library>, --library <library>`
Add archive file <library> to the list of files to link. This option may be used any number of times. `tricore-ld` will search its path-list for

occurrences of `lib<library>.a` for every <library> specified.

If the archive defines a symbol which was undefined in some object which appeared before the archive on the command line, the linker will include the appropriate file(s) from the archive. However, an undefined symbol in an object appearing later on the command line will not cause the linker to search the archive again.

See the `-(` option for a way to force the linker to search archives multiple times.

`-m <emulation>` Emulate the <emulation> linker. You can list the available emulations with the `--verbose` or `-V` options. If the `-m` option is not used, the emulation is taken from the `LDEMULATION` environment variable, if that is defined. Otherwise, the default emulation depends upon how the linker was configured.

`-M, --print-map`
Print map file on standard output. A link map provides information about the link, including the following:

- Where object files and symbols are mapped into memory.
- How common symbols are allocated.
- All archive members included in the link, with a mention of the symbol which caused the archive member to be brought in.

`-Map <file>` Print a link map to the file <file>. See the description of the `-M` option.

`-n, --nmagic` Turn off page alignment of sections, and mark the output as `NMAGIC` if possible.

`-N, --omagic` Set the text and data sections to be readable and writable. Also, do not page-align the data segment. If the output format supports Unix style magic numbers, mark the output as `OMAGIC`.

`--no-check-sections`
Do not check section addresses for overlaps

`--no-define-common`
This option inhibits the assignment of addresses to common symbols. The script command `INHIBIT_COMMON_ALLOCATION` has the same effect.

The `--no-define-common` option allows decoupling the decision to assign addresses to Common symbols from the choice of the output file type; otherwise a non-relocatable output type forces assigning addresses to Common symbols.

Using `--no-define-common` allows Common symbols that are referenced from a shared library to be assigned addresses only in the main program. This eliminates the unused duplicate space in the shared library, and also prevents any possible confusion over resolving to the wrong duplicate when there are many dynamic modules with specialized search paths for runtime symbol resolution.

`--no-demangle` Do not demangle symbol names

`--no-gc-sections`
: Don't remove unused sections (default)

`--noinhibit-exec`
: Retain the executable output file whenever it is still usable. Normally, the linker will not produce an output file if it encounters errors during the link process; it exits without writing an output file when it issues any error whatsoever.

`--no-keep-memory`
: The linker normally optimizes for speed over memory usage by caching the symbol tables of input files in memory. The option tells `tricore-ld` to instead optimize for memory usage, by rereading the symbol tables as necessary. This may be required if `tricore-ld` runs out of memory space while linking a large executable.

`--no-undefined` Normally when creating a non-symbolic shared library, undefined symbols are allowed and left to be resolved by the runtime loader. These options disallows such undefined symbols.

`--no-warn-mismatch`
: Normally `tricore-ld` will give an error if you try to link together input files that are mismatched for some reason, perhaps because they have been compiled for different processors or for different endiannesses. This option tells `tricore-ld` that it should silently permit such possible errors. This option should only be used with care, in cases when you have taken some special action that ensures that the linker errors are inappropriate.

`--no-whole-archive`
: Turn off the effect of the `--whole-archive` option for subsequent archive files.

`-o <file>, --output <file>`
: Use <file> as the name for the program produced by `tricore-ld`; if this option is not specified, the name `a.out` is used by default. The script command `OUTPUT` can also specify the output file name.

`--oformat <target>`
: Specify target of output file. `tricore-ld` may be configured to support more than one kind of object file. If your `tricore-ld` is configured this way, you can use the `--oformat` option to specify the binary format for the output object file. <target> is a text string, the name of a particular format supported by the BFD libraries. (You can list the available binary formats with `tricore-objdump -i`.) The script command `OUTPUT_FORMAT` can also specify the output format, but this option overrides it.

`-q, --emit-relocs`
: Leave relocation sections and contents in fully linked executables. Post link analysis and optimization tools may need this information in order to perform correct modifications of executables. This results in larger executables.

> **Note:**
>
> This option is currently only supported on ELF platforms.

Generate relocations in final output

`-qmagic` Ignored for Linux compatibility.

`-Qy` Ignored for SVR4 compatibility

`-R <file>, --just-symbols <file>`
: Read from <file> symbol names and their addresses, but do not relocate it or include it in the output. This allows your output file to refer symbolically to absolute locations of memory defined in other programs. You may use this option more than once.

For compatibility with other ELF linkers, if the `-R` option is followed by a directory name, rather than a file name, it is treated as the `-rpath` option.

`-r, -i, --relocateable`
: Perform an incremental link (generate relocatable output). When an input file does not have the same format as the output file, partial linking is only supported if that input file does not contain any relocations. Different output formats can have further restrictions; for example some `a.out`-based formats do not support partial linking with input files in other formats at all.

This option does the same thing as `-i`.

`--relax` Relax branches on certain targets. On some platforms these link time global optimizations may make symbolic debugging of the resulting executable impossible. On platforms where this is not supported, `--relax` is accepted, but ignored.

`--relax-24rel` Relax call and jump instructions whose target address cannot be reached with a PC-relative offset, nor by switching to instructions using TriCore's absolute addressing mode. This option only takes effect in final (i.e., non-relocateable) link runs, and is also enabled implicitly by `--relax`.

> **Note:**
>
> The target of such `call` and `jump` instructions must be a global symbol. Also, the `call` or `jump` instruction must be located in a code section (section attribute 'x').

`--relax-bdata` Compress bit objects contained in `.bdata` input sections. This option only takes effect in final (i.e., non-relocateable) link runs, and is also enabled implicitly by `--relax`.

`--retain-symbols-file <file>`
Retain *only* the symbols listed in the file <file>, discarding all others. <file> is simply a flat file, with one symbol name per line. `--retain-symbols-file` does *not* discard undefined symbols, or symbols needed for relocations.

You may only specify `--retain-symbols-file` once in the command line. It overrides `-s` and `-S`.

`-rpath <path>` Add a directory to the runtime library search path. This is used when linking an ELF executable with shared objects. All `-rpath` arguments are concatenated and passed to the runtime linker, which uses them to locate shared objects at runtime. The `-rpath` option is also used when locating shared objects which are needed by shared objects explicitly included in the link; see the description of the `-rpath-link` option. If `-rpath` is not used when linking an ELF executable, the contents of the environment variable `LD_RUN_PATH` will be used if it is defined.

`-rpath-link <path>`
Set link time shared library search path.

`-s, --strip-all`
Strip all symbols information from the output file.

`-S, --strip-debug`
Omit debugger symbol information (but not all symbols) from the output file.

`--section-start <section>=<address>`
Set address of named section

`-shared, -Bshareable`
Create a shared library. This is currently only supported on ELF, XCOFF and SunOS platforms. On SunOS, the linker will automatically create a shared library if the `-e` option is not used and there are undefined symbols in the link.

`--sort-common` This option tells `tricore-ld` to sort the common symbols by size when it places them in the appropriate output sections. First come all the one byte symbols, then all the two byte, then all the four byte, and then everything else. This is to prevent gaps between symbols due to alignment constraints.

`--spare-dynamic-tags <count>`
How many tags to reserve in .dynamic section

`--split-by-file [=<size>]`
Similar to `--split-by-reloc` but creates a new output section for each input file when <size> is reached.

`--split-by-reloc [=<size>]`
<size> defaults to a size of 1 if not given. Tries to creates extra sections

in the output file so that no single output section in the file contains more than <size> relocations. This is useful when generating huge relocatable files for downloading into certain real time kernels with the COFF object file format; since COFF cannot represent more than 65535 relocations in a single section.

Note:

This will fail to work with object file formats which do not support arbitrary sections. The linker will not split up individual input sections for redistribution, so if a single input section contains more than <size> relocations one output section will contain that many relocations. <size> defaults to a value of 32768.

`--stats`
Compute and display statistics about the operation of the linker, such as execution time and memory usage.

`-T <file>, --script <file>`
Use <file> as the linker script. and replace default linker script (rather than adding to it), so <file> must specify everything necessary to describe the output file. If <file> does not exist in the current directory, `tricore-ld` looks for it in the directories specified by any preceding `-L` options. Multiple `-T` options accumulate.

`-t, --trace`
Print the names of the input files as `tricore-ld` processes them.

`--target-help`
Print a summary of all target specific options on the standard output and exit.

`--task-link <symbol>`
Do task level linking

`-Tbss <address>, -Tdata <address>, -Ttext <address>`
Use <address> as the starting address of `.bss`, `.data` or `.text` section.

Note:

Be careful to use these options when the linker script contains MEMORY statements.

`-u <symbol>, --undefined <symbol>`
Force <symbol> to be entered in the output file as an undefined symbol. Doing this may, for example, trigger linking of additional modules from standard libraries. `-u` may be repeated with different option arguments to enter additional undefined symbols. This option is equivalent to the `EXTERN` linker script command.

`--unique [=<section>]`
Creates a separate output section for every input section matching <section>, or if the optional wildcard <section> argument is missing, for every orphan input section. An orphan section is one not specifically mentioned in a linker script. You may use this option multiple times

on the command line; It prevents the normal merging of input sections with the same name, overriding output section assignments in a linker script.

`-Ur` For anything other than C++ programs, this option is equivalent to `-r`: it generates relocatable output—i.e., an output file that can in turn serve as input to `tricore-ld`. When linking C++ programs, `-Ur` *does* resolve references to constructors, unlike `-r`. It does not work to use `-Ur` on files that were themselves linked with `-Ur`; once the constructor table has been built, it cannot be added to. Use `-Ur` only for the last partial link, and `-r` for the others.

`-v, --version` Print version information.

`-V` Print version and also list the supported emulations.

`--verbose` Output lots of information during link

`--version-exports-section <symbol>`
Take export symbols list from .exports, using <symbol> as the version.

`--version-script <file>`
Specify the name of a version script to the linker. This is typically used when creating shared libraries to specify additional information about the version hierarchy for the library being created. This option is only meaningful on ELF platforms which support shared libraries.

`--warn-common` Warn about duplicate common symbols. This option allows you to find potential problems from combining global symbols. Unfortunately, some C libraries use this practice, so you may get some warnings about symbols in the libraries as well as in your programs.

`--warn-once` Only warn once for each undefined symbol, rather than once per module which refers to it.

`--warn-orphan` Issue a warning message if there is no dedicated mapping between an input section and an output section.

`--warn-section-align`
Warn if the address of an output section is changed because of alignment. Typically, the alignment will be set by an input section. The address will only be changed if it not explicitly specified; that is, if the `SECTIONS` command does not specify a start address for the section (see subsection 20.2.8 on page 252).

`--whole-archive`
For each archive mentioned after the `--whole-archive` option on the command line, include every object file in the archive in the link, rather than searching the archive for the required object files. This is normally used to turn an archive file into a shared library, forcing every object to be included in the resulting shared library. This option may be used more than once.

Note:

When using this option from `tricore-gcc`: First, you have to use `-Wl,-whole-archive`, because `tricore-gcc` doesn't know about this option. Second, don't forget to use `-Wl,-no-whole-archive` after your list of archives, because `tricore-gcc` will add its own list of archives to your link and you may not want this flag to affect those as well.

`--wrap <symbol>`

Use wrapper functions for <symbol>. Any undefined reference to <symbol> will be resolved to __wrap_<symbol>. Any undefined reference to __real_<symbol> will be resolved to <symbol>.

This can be used to provide a wrapper for a system function. The wrapper function should be called __wrap_<symbol>. If it wishes to call the system function, it should call __real_<symbol>.

Here is a trivial example:

```
void *
__wrap_malloc (int c)
{
  printf ("malloc called with %ld\n", c);
  return __real_malloc (c);
}
```

If you link other code with this file using `--wrap malloc`, then all calls to malloc will call the function __wrap_malloc instead. The call to __real_malloc in __wrap_malloc will call the real malloc function.

You may wish to provide a __real_malloc function as well, so that links without the `--wrap` option will succeed. If you do this, you should not put the definition of __real_malloc in the same file as __wrap_malloc; if you do, the assembler may resolve the call before the linker has a chance to wrap it to malloc.

`-x, --discard-all`

Discard all local symbols.

`-X, --discard-locals`

Delete all temporary local symbols. For most targets, this is all local symbols whose names begin with `L`.

`-y <symbol>, --trace-symbol <symbol>`

Print the name of each linked file in which <symbol> appears. This option may be given any number of times. On many systems it is necessary to prepend an underscore. This option is useful when you have an undefined symbol in your link but don't know where the reference is coming from.

20.1.1 Default options

The `tricore-ld` looks for library inputs in the following directories:

- `<install-dir>\bin\..\lib\gcc-lib\tricore\<gcc-version>\crt0.o`
- `-L<install-dir>\bin\..\lib\gcc-lib\tricore\<gcc-version>`
- `-L<install-dir>\bin\..\lib\gcc-lib`
- `-L<install-dir>\bin\..\lib\gcc-lib\tricore\<gcc-version>\ ..\..\..\..\tricore\lib`
- `-L<install-dir>\bin\..\lib\gcc-lib\tricore\<gcc-version>\..\..\..`
- -lgcc
- -lc
- -los
- -lc
- -lgcc

E.g. if you `-lgcc` the library `libgcc.a` is used.

20.2 The Linker Script File

The `tricore-ld` is controlled by a scripting language. If you do not specify a script, the one that was compiled into the linker when it is installed is used by default.

20.2.1 Sections and Relocation

Roughly, a section is a range of addresses, with no gaps; all data "in" those addresses is treated the same for some particular purpose. For example there may be a "read only" section.

The linker `tricore-ld` reads many object files (partial programs) and combines their contents to form a runable program. When `tricore-as` emits an object file, the partial program is assumed to start at address 0. `tricore-ld` assigns the final addresses for the partial program, so that different partial programs do not overlap. This is actually an oversimplification, but it suffices to explain how `tricore-as` uses sections.

`tricore-ld` moves blocks of bytes of your program to their run-time addresses. These blocks slide to their run-time addresses as rigid units; their length does not change and neither does the order of bytes within them. Such a rigid unit is called a *section*. Assigning run-time addresses to sections is called *relocation*. It includes the task of adjusting mentions of object-file addresses so they refer to the proper run-time addresses.

An object file written by `tricore-as` has at least three sections, any of which may be empty. These are named `.text`, `.data` and `.bss` sections. The `.bss` section is used for local common variable storage. You may allocate address space in the `.bss` section, but you may not dictate data to load into it before your program executes. When your program starts running, all the contents of the `.bss` section are zeroed bytes.

To let tricore-ld know which data changes when the sections are relocated, and how to change that data, `tricore-as` also writes to the object file details of the relocation needed. To perform relocation `tricore-ld` must know, each time an address in the object file is mentioned:

In this manual we use the notation <secname> <N> to mean "offset <N> into section <secname>.

Apart from text, data and bss sections you need to know about the *absolute* section. When `tricore-ld` mixes partial programs, addresses in the absolute section remain unchanged. For example, address `0` is "relocated" to run-time address 0 by `tricore-ld`. Although the linker never arranges two partial programs' data sections with overlapping addresses after linking, *by definition* their absolute sections must overlap. Address `absolute 239` in one art of a program is always the same address when the program is running as address `absolute 239` in any other part of the program.

The idea of sections is extended to the *undefined* section. Any address whose section is unknown at assembly time is by definition rendered undefined <U>—where <U> is filled in later. Since numbers are always defined, the only way to generate an undefined address is to mention an undefined symbol. A reference to a named common block would be such a symbol: its value is unknown at assembly time so it has section *undefined*.

By analogy the word *section* is used to describe groups of sections in the linked program. `tricore-ld` puts all partial programs' text sections in contiguous addresses in the linked program. It is customary to refer to the *text section* of a program, meaning all the addresses of all partial programs' text sections. Likewise for data and `.bss` sections.

Some sections are manipulated by `tricore-ld`; others are invented for use of `tricore-as` and have no meaning except during assembly.

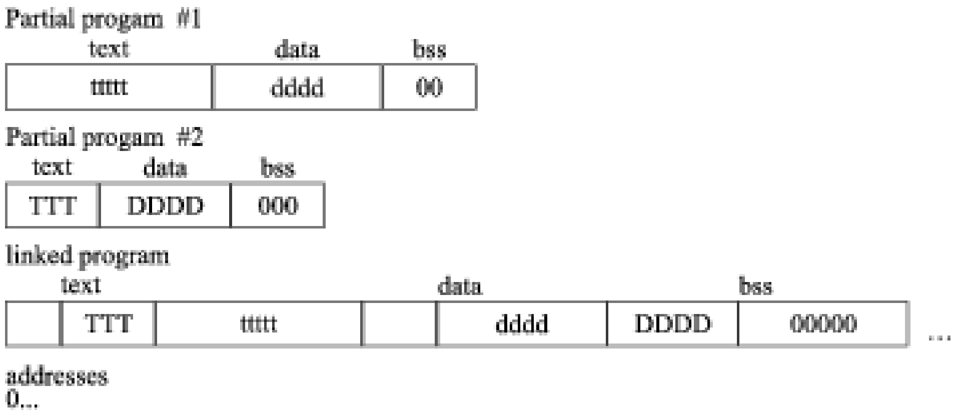

Figure 20.1: Sections

Note:

If you call the linker from the command-line, using the default linker description file, the sections of the input files are put together in the order they are passed to the linker. If you use an own linker description the order is indifferent.

`named sections` These sections hold your program. `tricore-as` and `tricore-ld` treat them as separate but equal sections. Anything you can say of one section is true for another.

`.bss` This section contains zeroed bytes when your program begins running. It is used to hold uninitialized variables or common storage. The length of each partial program's `.bss` section is important, but because it starts out containing zeroed bytes there is no need to store explicit zero bytes in the object file. The `.bss` section was invented to eliminate those explicit zeros from object files.

`absolute section` Address 0 of this section is always "relocated" to runtime address 0. This is useful if you want to refer to an address that `tricore-ld` must not change when relocating. In this sense we speak of absolute addresses being "unrelocatable": they do not change during relocation.

`undefined section` This "section" is a catch-all for address references to objects not in the preceding sections.

TriCore Sections

`.startup_code` The startup code for TriCore is stored in an own section named `.startup_code`. Per default the code of `crt0` is stored at the address `0xa0000000`.

`.init, .fini, .jcr` The input sections `.init` and `.fini` are located in the output section `.init`. The function `__main` will no longer be generated. The section `.init` contains the calls of all constructors, `.fini` contains the calls of all destructors and `.jcr` contains java constructors. The section `.jcr` is located at the `.rodata` section. In the file `crti.o`, the symbol `_init` is defined in the `.init` section, and the symbol `_fini` is defined in the `.fini` section. The file `crtn.o` contains the definition of the function epilog with the return instruction of the `.init` and `.fini` section. The symbol `_init` is called as a function before main from the startup file `crt0.o`. The section `.fini` will be generated to hold the code which has to be executed before `_exit` (see chapter 13 on page 67).

`.traptab, .inttab` The traptable and the interrupt tables have now their own sections which will be allocated by default after the startup code.

`.csa` The CSA memory is defined in an own section to let the linker known about the occupied memory. The size of the CSA is calculated dynamically from the size of the internal RAM (defined by `__INT_DATA_RAM_SIZE`) and the sizes of the absolute addressable section `.zdata` and `.zbss`.

`.toc`	The ".toc" section was used to store data that can be quickly addressed using a function's toc pointer (`%a12`). The compiler was changed to not create the `.toc`-Section any more.
`.text`	All code in this section will be situated in μC ROM.
`.pcptext`	Enters the PCP (Peripheral Control Processor) text section; if this section doesn't exist already, it will be created automatically. The PCP text section is used to store PCP code. Upon entering this section, `tricore-as` expects PCP mnemonics instead of TriCore instructions; you can return to the normal behavior simply by leaving the PCP text section, e.g. with `.text`, which (re-) enters the TriCore text section. This makes it possible to embed PCP programs as inline assembler in C or C++ modules, provided you're using the TriCore port of GCC, the GNU compiler collection.
`.rodata`	The ".rodata" section is used to store read-only data (e.g., constants).
`.sdata.rodata`	The section `.srodata` is renamed to `.sdata.rodata`. This section is used to store read-only data (e.g., constants) that can be quickly addressed using the small data area pointer (%a0).
`.sdata`	The ".sdata" section is used to store initialized data that can be quickly addressed using the small data area pointer (%a0).
`.pcpdata`	The PCP data section is used to store contexts and parameters for the PCP.

Note:

The PCP can access this section only word-wise (i.e., in quantities of 32 bits), so you should only use `.word` pseudo-ops to fill it, even though `tricore-as` also accepts `.byte`, `.half` and other pseudo-ops that can be used to define memory values of various sizes.

`.zdata`	The ".zdata" section is used to store initialized data that can be quickly addressed using TriCore's absolute addressing mode.

Note:

If you define your own sections with data addressed in the absolute addressing mode, these sections must have the prefix `.zdata`.

`.sbss`	The ".sbss" section is used to store uninitialized data that can be quickly addressed using the small data area pointer (%a0).
`.bbss`	The ".bbss" section is used to store uninitialized bit data.
`.zbss`	The ".zbss" section is used to store uninitialized data that can be quickly addressed using TriCore's absolute addressing mode.

> **Note:**
>
> If you define your own sections with data addressed in the absolute addressing mode, these sections must have the prefix `.zbss`.

`.eh_frame`	Exception handling frame used for C++ exceptions.

> **Note:**
>
> This section is renamed in `gcc_except_table` in new version of `tricore-gcc`

`.ctors`	Used for constructors.
`.dtors`	Used for destructors.
`.bdata`	This section is used for bit variables.

`.version_info` This section will be generated if the option `-mversion-info` is set. Every file that is compiled with this option contains the version information of the compiler, which was used to compile the file, in the section `.version_info` in the object file. In this section this configuration info is available:

- Filename of the source file
- Options passed to the compiler
- Version Information of the compiler

See chapter 23 on page 323 for details.

Per default the option `-mversion-info` is set. To compile a file use the option `-c`:

```
tricore-gcc -c <filename>.c
```

With the tool tricore-objdump and the option `-h` the section headers of an object file are displayed:

```
tricore-objdump -h <filename>.o
```

The contents of section the section `.version_info` is shown by using tricore-objdump with following options:

```
tricore-objdump -s --section=.version_info <<filename>.o>
```

If you compile a file with the compileroption `-mno-version-info` the section `.version_info` does not exist.

Note:

All compiler libraries are compiled with `-mversion-info` and have a section `.version_info`. Therefore a linked file will always contain a section `.version_info`, which contains the version info of the libraries that were linkd to the executable file, even if the option `-mno-version-info` is passed to the compiler.

`.pictext` Section for position independent code.

Note:

If the option `-mcode-pic` is set the code will be put in an own section called `.pictext` instead of the standard section `.text`.

The `tricore-ld` is controlled by a scripting language. If you do not specify a script, the one that was compiled into the linker when it is installed is used by default. You can override this default and provide your own script.

```
tricore-ld -T sprig.link start.o loop.o -o sprig
```

The `-T` option specifies the name of your own script file `sprig.link`. The `-c` is a synonym of the `-T` option.

The primary reason for using a special script is the addressing scheme. Normally the linker produces an executable file with adjustable addresses that can be set at the time the module is loaded into memory.

Note:

You can use the `--verbose` command line option to display the default linker script. Certain command line options, such as `-r` or `-N`, will affect the default linker script.

With an embedded module, all addresses are resolved by the linker into absolute locations so that every addressing reference is completely resolved and immovable. This process of locking down the address is known as *locating* the module. Some systems have a separate utility that processes the relocatable output from the linker into an absolute module, but the `tricore-ld` has the locator built in.

The linker reads object files produced by the compiler and combines them into a new object file (also called an executable file) as its output. An object file is divided into sections. Each section has a name and a size. The linker combines the input sections of the same name into a single output section. Some sections contain executable code, some contain data with initial values, and others contain uninitialized data. A section with uninitialized data usually has nothing other than a name and a size. You may also use linker scripts implicitly by naming them as input files to the linker, as though they were files to be linked (see subsection 20.2.22 on page 276).

20.2.2 Basic Linker Script Concepts

We need to define some basic concepts and vocabulary in order to describe the linker script language.

The linker combines input files into a single output file. The output file and each input file are in a special data format known as an *object file format*. Each file is called an *object file*. The output file is often called an *executable*, but for our purposes we will also call it an object file. Each object file has, among other things, a list of *sections*. We sometimes refer to a section in an input file as an *input section*; similarly, a section in the output file is an *output section*.

Each section in an object file has a name and a size. Most sections also have an associated block of data, known as the *section contents*. A section may be marked as *loadable*, which mean that the contents should be loaded into memory when the output file is run. A section with no contents may be *allocatable*, which means that an area in memory should be set aside, but nothing in particular should be loaded there (in some cases this memory must be zeroed out). A section which is neither loadable nor allocatable typically contains some sort of debugging information.

Every loadable or allocatable output section has two addresses:

VMA Virtual memory address used internally by the module when it is run.

LMA Loadable memory address specifying where the section is to be loaded.

In most cases the two addresses will be the same. An example of when they might be different is when a data section is loaded into ROM, and then copied into RAM when the program starts up (this technique is often used to initialize global variables in a ROM based system). In this case the ROM address would be the LMA, and the RAM address would be the VMA.

Note:

You can see the sections in an object file by using the `-h` option with `tricore-objdump` program.

Every object file also has a list of *symbols*, known as the *symbol table*. A symbol may be defined or undefined. Each symbol has a name, and each defined symbol has an address, among other information. If you compile a C or C++ program into an object file, you will get a defined symbol for every defined function and global or static variable. Every undefined function or global variable which is referenced in the input file will become an undefined symbol.

Note:

You can see the symbols in an object file by using the `tricore-nm` program, or by using the `tricore-objdump` program with the `-t` option.

20.2.3 Linker Script Format

Linker scripts are text files.

You write a linker script as a series of commands. Each command is either a *keyword*, possibly followed by arguments, or an assignment to a symbol. You may separate commands using semicolons. Whitespace is generally ignored.

Strings such as file or format names can normally be entered directly. If the file name contains a character such as a comma which would otherwise serve to separate file names, you may put the file name in double quotes. There is no way to use a double quote character in a file name.

You may include comments in linker scripts just as in C, delimited by `/*` and `*/`. As in C, comments are syntactically equivalent to whitespace.

20.2.4 Simple Linker Script Example

Many linker scripts are fairly simple.

The simplest possible linker script has just one command: `SECTIONS`. You use the command `SECTIONS` to describe the memory layout of the output file.

The `SECTIONS` command is a powerful command. Here we will describe a simple use of it. Let's assume your program consists only of code, initialized data, and uninitialized data. These will be in the `.text`, `.data`, and `.bss` sections, respectively. Let's assume further that these are the only sections which appear in your input files.

For this example, let's say that the code should be loaded at address 0x10000, and that the data should start at address 0x8000000. Here is a linker script which will do that:

```
SECTIONS
{
  . = 0x10000;
  .text : {
        *(.text)
  }
  . = 0x8000000;
  .data : {
        *(.data)
  }
  .bss : {
        *(.bss)
  }
}
```

You write the `SECTIONS` command as the keyword `SECTIONS`, followed by a series of symbol assignments and output section descriptions enclosed in curly braces.

The first entry inside the `SECTIONS` command of the above example sets the value of the special symbol ., which is the location counter. If you do not specify the address of an output section in some other way (other ways are described later), the address is set from the current value of the location counter. The location counter is then incremented by the size of the output section. At the start of the SECTIONS command, the location counter has the value 0.

The second entry defines an output section, .text. The colon is required syntax which may be ignored for now. Within the curly braces after the output section name, you list the names of the input sections which should be placed into this output section. The * is a wildcard which matches any file name. The expression *(.text) means all .text input sections in all input files.

Since the location counter is 0x10000 when the output section .text is defined, the linker will set the address of the .text section in the output file to be 0x10000.

The remaining lines define the .data and .bss sections in the output file. The linker will place the .data output section at address 0x8000000. After the linker places the .data output section, the value of the location counter will be 0x8000000 plus the size of the .data output section. The effect is that the linker will place the .bss output section immediately after the .data output section in memory

The linker will ensure that each output section has the required alignment, by increasing the location counter if necessary. In this example, the specified addresses for the .text and .data sections will probably satisfy any alignment constraints, but the linker may have to create a small gap between the .data and .bss sections.

Note:

That's it! That's a simple and complete linker script.

20.2.5 Call interface to the linker

If the linker can not recognize the format of an object file, it will assume that it is a linker script. A script specified in this way augments the main linker script used for the link. The compiler passes a file `memory.x` to the linker right after the startup file `crt0.o`. The file `memory.x` contains the default memory description of the CPU derivate. Specifying the derivate by the option `-mcpu=tc1796` the corresponding `memory.x` for e.g. TC1796 is selected. The `memory.x` file looks like:

```
/*
 * memory.x -- The default memory description
 *
 * Copyright (C) 2005 HighTec EDV-Systeme GmbH.
 *
 */
/* __TC1796__ __TC13__ with Core TC1.3 */
__TRICORE_DERIVATE_MEMORY_MAP__ = 0x1796;
/* the external RAM description */
__EXT_CODE_RAM_BEGIN = 0xa1000000;
__EXT_CODE_RAM_SIZE = 512K ;
__EXT_DATA_RAM_BEGIN = 0xa1080000;
__EXT_DATA_RAM_SIZE = 512K;
__RAM_END = __EXT_DATA_RAM_BEGIN + __EXT_DATA_RAM_SIZE;
/* the internal ram description */
__INT_CODE_RAM_BEGIN = 0xd4000000;
```

```
__INT_CODE_RAM_SIZE = 48K;
__INT_DATA_RAM_BEGIN = 0xd0000000;
__INT_DATA_RAM_SIZE = 56K;
/* the pcp memory description */
__PCP_CODE_RAM_BEGIN = 0xf0060000;
__PCP_CODE_RAM_SIZE = 32K;
__PCP_DATA_RAM_BEGIN = 0xf0050000;
__PCP_DATA_RAM_SIZE = 16K;

MEMORY
{
  ext_cram (arx): org = 0xa1000000, len = 512K
  ext_dram (aw!x): org = 0xa1080000, len = 512K
  int_cram (arx): org = 0xd4000000, len = 48K
  int_dram (aw!x): org = 0xd0000000, len = 56K
  pcp_data (aw!x): org = 0xf0050000, len = 16K
  pcp_text (arx): org = 0xf0060000, len = 32K
}

/* the symbol __TRICORE_DERIVATE_NAME__ will be defined in the crt0.S and is
 * tested here to confirm that this memory map and the startup file will
 * fit together
*/
_. = ASSERT ((__TRICORE_DERIVATE_MEMORY_MAP__ == __TRICORE_DERIVATE_NAME__),
    "Using wrong Memory Map. This Map is for TC1796");
```

The `crt0.o` file defines a new Symbol __TRICORE_DERIVATE_NAME__. The value of this symbol is derived from the derivate name

-mtc13	$\rightarrow$ 0x13
-mcpu=tc1130	$\rightarrow$ 0x1130
-mcpu=tc1796	$\rightarrow$ 0x1796

This symbol can be used to check the compability of several object files. Right now it is used to check the compability of `crt0.o` and `memroy.x`

Linker script

The MEMORY regions are defined by symbols like __EXT_CODE_RAM_BEGIN etc. (see `elf32tricore.x` for detailed description). This definitions can be overidden by passing a new description in a normal text file to the linker (see `memory.x` for an example)

```
tricore-gcc -Wl,memory.x <list of objects>
```

Sections

The file `memory.x` is passed to linker as additional script. It is not passed as linker description file (see option `-Wl,T<target.ld>`.

Note:

If a linker description file `target.ld` is passed to the compiler driver `tricore-gcc` with the option `-Ttarget.ld` the memory layout `memory.x` is ignored. With the option `-Wl,-Ttarget.ld` the memory layout of `memory.x` is still active, because this file is treated as startup file. To disable startup files (`crt0.o` and `memory.x`) use the option `-nostartfiles`.

`.startup_code` The startup code from `crt0.o` is defined in an extra section .startup_code.

`.traptab and .inttab`
The traptable and the interrupt tables have now their own sections which will be allocated by default after the startup code.

`.csa` The CSA memory is defined in an own section to let the linker known about the occupied memory. The size of the CSA is calculated dynamically from the size of the internal RAM (defined by __INT_DATA_RAM_SIZE) and the sizes of the absolute addressable section .zdata and .zbss.

Adding New Sections

For adding new sections you can edit a new file and save the script e.g. <`memory.x`>. If this script it is passed the linker your sections are added to the default linker script. In this manner you can also modify the defaults of the default linker description file.

```
SECTIONS {
mytext :
  {
   . = ALIGN(4) ;
   MySectionBegin = .;
    *(.mytext)
   . = ALIGN(4) ;
   MySectionEnd = .;
} > ext_cram
}
```

20.2.6 Simple Linker Script Commands

In this section we describe the simple linker script commands.

20.2.6.1 Setting the entry point

The first instruction to execute in a program is called the *entry point*. You can use the ENTRY linker script command to set the entry point. The argument is a symbol name:

```
ENTRY(<symbol>)
```

There are several ways to set the entry point. The linker will set the entry point by trying each of the following methods in order, and stopping when one of them succeeds:

- the '-e' <entry> command-line option;
- the ENTRY(<symbol>) command in a linker script;

- the value of the symbol `start`, if defined;
- the address of the first byte of the `.text` section, if present;
- The address `0`.

20.2.6.2 Commands dealing with files

Several linker script commands deal with files.

`INCLUDE <filename>`
: Include the linker script <filename> at this point. The file will be searched for in the current directory, and in any directory specified with the `-L` option. You can nest calls to `INCLUDE` up to 10 levels deep.

`INPUT(<file>, <file>, ...), INPUT(<file> <file> ...)`
: The `INPUT` command directs the linker to include the named files in the link, as though they were named on the command line.

For example, if you always want to include `subr.o` any time you do a link, but you can't be bothered to put it on every link command line, then you can put `INPUT (subr.o)` in your linker script.

In fact, if you like, you can list all of your input files in the linker script, and then invoke the linker with nothing but a `-T` option.

The linker will first try to open the file in the current directory. If it is not found, the linker will search through the archive library search path. See the description of `-L` in (see section 20.1 on page 221).

If you use `INPUT (–l<file>)`, `tricore-ld` will transform the name to `lib <file>.a`, as with the command line argument `-l`.

When you use the `INPUT` command in an implicit linker script, the files will be included in the link at the point at which the linker script file is included. This can affect archive searching.

`GROUP(<file>, <file>, ...), GROUP(<file> <file> ...)`
: The `GROUP` command is like `INPUT`, except that the named files should all be archives, and they are searched repeatedly until no new undefined references are created. See the description of `-(` in section 20.1 on page 221.

`OUTPUT(<filename>)`
: `OUTPUT` command names the output file. The `OUTPUT(<filename>)` entry in the linker script is exactly like using `-o <filename>` on the command line (see section 20.1 on page 221). If both are used, the command line option takes precedence.

You can use the `OUTPUT` command to define a default name for the output file other than the usual default of `a.out`.

`SEARCH_DIR(<path>)`

The SEARCH_DIR command adds <path> to the list of paths where `tricore-ld` looks for archive libraries. Using SEARCH_DIR(<path>) is exactly like using `-L <path>` on the command line (see section 20.1 on page 221). If both are used, then the linker will search both paths. Paths specified using the command line option are searched first.

`STARTUP(<filename>)`

The STARTUP command is just like the INPUT command, except that <filename> will become the first input file to be linked, as though it were specified first on the command line. This may be useful when using a system in which the entry point is always the start of the first file.

20.2.6.3 Commands dealing with object file formats

A couple of linker script commands deal with object file formats.

`OUTPUT_FORMAT(<bfdname>), OUTPUT_FORMAT(<default>, <big>, <little>)`

The OUTPUT_FORMAT command names the BFD format to use for the output file. Using OUTPUT_FORMAT(<bfdname>) is exactly like using `--oformat <bfdname>` on the command line (see section 20.1 on page 221). If both are used, the command line option takes precedence.

You can use OUTPUT_FORMAT with three arguments to use different formats based on the `-EB` and `-EL` command line options. This permits the linker script to set the output format based on the desired endianness.

If neither `-EB` nor `-EL` are used, then the output format will be the first argument, `default`. If `-EB` is used, the output format will be the second argument, <big>. If '-EL' is used, the output format will be the third argument, <little>.

For example, the default linker script for the TriCore ELF target uses this

```
OUTPUT_FORMAT(elf32-bigmips, elf32-bigmips, elf32-littlemips)
```

This says that the default format for the output file is 'elf32-bigmips', but if the user uses the `-EL` command line option, the output file will be created in the 'elf32-littlemips' format.

`TARGET(<bfdname>)`

The TARGET command names the BFD format to use when reading input files. It affects subsequent INPUT and GROUP commands. This command is like using `-b <bfdname>` on the command line (see section 20.1 on page 221). If the TARGET command is used but OUTPUT_FORMAT is not, then the last TARGET command is also used to set the format for the output file.

20.2.6.4 Other linker script commands

There are a few other linker scripts commands.

`ASSERT(<exp>, <message>)`
Ensure that <exp> is non-zero. If it is zero, then exit the linker with an error code, and print <message>.

`EXTERN(<symbol> <symbol> ...)`
Force <symbol> to be entered in the output file as an undefined symbol. Doing this may, for example, trigger linking of additional modules from standard libraries. You may list several <symbol>s for each EXTERN, and you may use EXTERN multiple times. This command has the same effect as the `-u` command-line option.

`FORCE_COMMON_ALLOCATION`
This command has the same effect as the `-d` command-line option: to make `tricore-ld` assign space to common symbols even if a relocatable output file is specified (`-r`).

`INHIBIT_COMMON_ALLOCATION`
This command has the same effect as the option `--no-define-common`: to make `tricore-ld` omit the assignment of addresses to common symbols even for a non-relocatable output file.

`NOCROSSREFS(<section> <section> ...)`
This command may be used to tell `tricore-ld` to issue an error about any references among certain output sections.

In certain types of programs, particularly on embedded systems when using overlays, when one section is loaded into memory, another section will not be. Any direct references between the two sections would be errors. For example, it would be an error if code in one section called a function defined in the other section.

The NOCROSSREFS command takes a list of output section names. If `tricore-ld` detects any cross references between the sections, it reports an error and returns a non-zero exit status.

Note:
The NOCROSSREFS command uses output section names, not input section names.

`OUTPUT_ARCH(<bfdarch>)`
Specify a particular output machine architecture. The argument is one of the names used by the BFD library. You can see the architecture of an object file by using the `tricore-objdump` program with the `-f` option.

20.2.7 Assigning Values to Symbols

You may assign a value to a symbol in a linker script. This will define the symbol as a global symbol.

```
.sb_code :
{
    *(.text)
} > sb_code
__TEST = <address>;
```

A global symbol can also be defined with the linker option `-Wl,-defsymb,name=<address>`.

If the symbol of a call contains an absolute value, then the call instruction is always absolute. If the address of the symbol is too far away you need the linker option `--relax-24rel` or `--relax`.

The definition of a symbol

```
.sb_code :
{
    *(.text)
    __TEST = .;
} > sb_code
```

__TEST = .; within a section with "." puts the symbol relative to the start address of the output section.

20.2.7.1 Simple Assignments

You may assign to a symbol using any of the C assignment operators:

```
<symbol> = <expression> ;
<symbol> += <expression> ;
<symbol> -= <expression> ;
<symbol> *= <expression> ;
<symbol> /= <expression> ;
<symbol> <<= <expression> ;
<symbol> >>= <expression> ;
<symbol> &= <expression> ;
<symbol> |= <expression> ;
```

The first case will define <symbol> to the value of <expression>. In the other cases, <symbol> must already be defined, and the value will be adjusted accordingly.

The special symbol name '.' indicates the location counter. You may only use this within a SECTIONS command.

The semicolon after <expression> is required.

Expressions are defined below; see subsection 20.2.14 on page 270.

You may write symbol assignments as commands in their own right, or as statements within a SECTIONS command, or as part of an output section description in a SECTIONS command.

The section of the symbol will be set from the section of the expression; for more information, see subsection 20.2.14 on page 270.

Here is an example showing the three different places that symbol assignments may be used:

```
floating_point = 0;
SECTIONS
{
  .text :
    {
      *(.text)
      _etext = .;
    }
  _bdata = (. + 3) & ~ 3;
  .data : { *(.data) }
}
```

In this example, the symbol floating_point will be defined as zero. The symbol _etext will be defined as the address following the last .text input section. The symbol _bdata will be defined as the address following the .text output section aligned upward to a 4 byte boundary.

20.2.7.2 PROVIDE

In some cases, it is desirable for a linker script to define a symbol only if it is referenced and is not defined by any object included in the link. For example, traditional linkers defined the symbol etext. However, ANSI C requires that the user be able to use etext as a function name without encountering an error. The PROVIDE keyword may be used to define a symbol, such as etext, only if it is referenced but not defined. The syntax is PROVIDE(<symbol> = <expression>).

Here is an example of using PROVIDE to define etext:

```
SECTIONS
{
  .text :
    {
      *(.text)
      _etext = .;
      PROVIDE(etext = .);
    }
}
```

In this example, if the program defines _etext (with a leading underscore), the linker will give a multiple definition error. If, on the other hand, the program defines etext (with no leading underscore), the linker will silently use the definition in the program. If the program references etext but does not define it, the linker will use the definition in the linker script.

20.2.8 SECTIONS command

The SECTIONS command tells the linker how to map input sections into output sections, and how to place the output sections in memory.

The format of the SECTIONS command is:

```
SECTIONS
{
  <sections-command>
  <sections-command>
  ...
}
```

Each <sections-command> may of be one of the following:

- an ENTRY command (see subsubsection 20.2.6.1 on page 247)
- a symbol assignment (see subsection 20.2.7 on page 251)
- an output section description
- an overlay description

The ENTRY command and symbol assignments are permitted inside the SECTIONS command for convenience in using the location counter in those commands. This can also make the linker script easier to understand because you can use those commands at meaningful points in the layout of the output file.

Output section descriptions and overlay descriptions are described below.

If you do not use a SECTIONS command in your linker script, the linker will place each input section into an identically named output section in the order that the sections are first encountered in the input files. If all input sections are present in the first file, for example, the order of sections in the output file will match the order in the first input file. The first section will be at address zero.

20.2.8.1 Output section description

The full description of an output section looks like this:

```
<section> [<address>] [(<type>)] : [AT(<lma>)]
  {
    <output-section-command>
    <var{output-section-command>
    ...
  } [><region>] [AT><lma_region>] [:<phdr> :<phdr> ...] [=<fillexp>]
```

Most output sections do not use most of the optional section attributes.

The whitespace around <section> is required, so that the section name is unambiguous. The colon and the curly braces are also required. The line breaks and other white space are optional.

Each <output-section-command> may be one of the following:

- a symbol assignment (see subsection 20.2.7 on page 251)
- an input section description (see subsubsection 20.2.8.4 on page 254)
- data values to include directly (see subsubsection 20.2.8.10 on page 258)

- a special output section keyword (see subsubsection 20.2.8.11 on page 259)

20.2.8.2 Output section name

The name of the output section is <section>. <section> must meet the constraints of your output format. In formats which only support a limited number of sections, such as `a.out`, the name must be one of the names supported by the format (`a.out`, for example, allows only `.text`, `.data` or `.bss`). If the output format supports any number of sections, but with numbers and not names (as is the case for Oasys), the name should be supplied as a quoted numeric string. A section name may consist of any sequence of characters, but a name which contains any unusual characters such as commas must be quoted.

The output section name '/DISCARD/' is special. (See also subsubsection 20.2.8.12 on page 260).

20.2.8.3 Output section address

The <address> is an expression for the VMA (the virtual memory address) of the output section. If you do not provide <address>, the linker will set it based on <region> if present, or otherwise based on the current value of the location counter.

If you provide <address>, the address of the output section will be set to precisely that. If you provide neither <address> nor <region>, then the address of the output section will be set to the current value of the location counter aligned to the alignment requirements of the output section. The alignment requirement of the output section is the strictest alignment of any input section contained within the output section.

```
.text . : { *(.text) }
```

and

```
.text : { *(.text) }
```

are subtly different. The first will set the address of the `.text` output section to the current value of the location counter. The second will set it to the current value of the location counter aligned to the strictest alignment of a `.text` input section.

The <address> may be an arbitrary expression; see subsection 20.2.14 on page 270. For example, if you want to align the section on a 0x10 byte boundary, so that the lowest four bits of the section address are zero, you could do something like this:

```
.text ALIGN(0x10) : { *(.text) }
```

This works because `ALIGN` returns the current location counter aligned upward to the specified value.

Specifying <address> for a section will change the value of the location counter.

20.2.8.4 Input section description

The most common output section command is an input section description.

The input section description is the most basic linker script operation. You use output sections to tell the linker how to lay out your program in memory. You use input section descriptions to tell the linker how to map the input files into your memory layout.

20.2.8.5 Input section basics

An input section description consists of a file name optionally followed by a list of section names in parentheses.

The file name and the section name may be wildcard patterns, which we describe further below (see subsubsection 20.2.8.6 on page 255).

The most common input section description is to include all input sections with a particular name in the output section. For example, to include all input .text sections, you would write:

```
*(.text)
```

Here the * is a wildcard which matches any file name. To exclude a list of files from matching the file name wildcard, EXCLUDE_FILE may be used to match all files except the ones specified in the EXCLUDE_FILE list. For example:

```
*(EXCLUDE_FILE (*crtend.o *otherfile.o) .ctors)
```

will cause all .ctors sections from all files except `crtend.o` and `otherfile.o` to be included.

There are two ways to include more than one section:

```
*(.text .rdata)
*(.text) *(.rdata)
```

The difference between these is the order in which the .text and .rdata input sections will appear in the output section. In the first example, they will be intermingled, appearing in the same order as they are found in the linker input. In the second example, all .text input sections will appear first, followed by all .rdata input sections.

You can specify a file name to include sections from a particular file. You would do this if one or more of your files contain special data that needs to be at a particular location in memory. For example:

```
data.o(.data)
```

If you use a file name without a list of sections, then all sections in the input file will be included in the output section. This is not commonly done, but it may by useful on occasion. For example:

```
data.o
```

When you use a file name which does not contain any wild card characters, the linker will first see if you also specified the file name on the linker command line or in an INPUT command. If you did not, the linker will attempt to open the file as an input file, as though it appeared on the command line. Note that this differs from an INPUT command, because the linker will not search for the file in the archive search path.

20.2.8.6 Input section wildcard patterns

In an input section description, either the file name or the section name or both may be wildcard patterns.

The file name of '*' seen in many examples is a simple wildcard pattern for the file name.

The wildcard patterns are:

`'*'`	matches any number of characters
`'?'`	matches any single character
`[<chars>]`	matches a single instance of any of the <chars>; the '-' character may be used to specify a range of characters, as in '[a-z]' to match any lower case letter
\	quotes the following character

When a file name is matched with a wildcard, the wildcard characters will not match a '/' character (used to separate directory names on Unix). A pattern consisting of a single '*' character is an exception; it will always match any file name, whether it contains a '/' or not. In a section name, the wildcard characters will match a '/' character.

File name wildcard patterns only match files which are explicitly specified on the command line or in an `INPUT` command. The linker does not search directories to expand wildcards.

If a file name matches more than one wildcard pattern, or if a file name appears explicitly and is also matched by a wildcard pattern, the linker will use the first match in the linker script. For example, this sequence of input section descriptions is probably in error, because the `data.o` rule will not be used:

```
.data : { *(.data) }
.data1 : { data.o(.data) }
```

Normally, the linker will place files and sections matched by wildcards in the order in which they are seen during the link. You can change this by using the `SORT` keyword, which appears before a wildcard pattern in parentheses (e.g., `SORT(.text*)`). When the `SORT` keyword is used, the linker will sort the files or sections into ascending order by name before placing them in the output file.

If you ever get confused about where input sections are going, use the `-M` linker option to generate a map file. The map file shows precisely how input sections are mapped to output sections.

This example shows how wildcard patterns might be used to partition files. This linker script directs the linker to place all `.text` sections in `.text` and all `.bss` sections in `.bss`. The linker will place the `.data` section from all files beginning with an upper case character in `.DATA`; for all other files, the linker will place the `.data` section in `.data`.

```
SECTIONS {
  .text : { *(.text) }
  .DATA : { [A-Z]*(.data) }
  .data : { *(.data) }
  .bss : { *(.bss) }
}
```

20.2.8.7 Input section for common symbols

A special notation is needed for common symbols, because in many object file formats common symbols do not have a particular input section. The linker treats common symbols as though they are in an input section named COMMON.

You may use file names with the COMMON section just as with any other input sections. You can use this to place common symbols from a particular input file in one section while common symbols from other input files are placed in another section.

In most cases, common symbols in input files will be placed in the .bss section in the output file. For example:

```
.bss { *(.bss) *(COMMON) }
```

Some object file formats have more than one type of common symbol. For example, the MIPS ELF object file format distinguishes standard common symbols and small common symbols. In this case, the linker will use a different special section name for other types of common symbols. In the case of MIPS ELF, the linker uses COMMON for standard common symbols and .scommon for small common symbols. This permits you to map the different types of common symbols into memory at different locations.

You will sometimes see [COMMON] in old linker scripts. This notation is now considered obsolete. It is equivalent to *(COMMON).

20.2.8.8 Input section and garbage collection

When link-time garbage collection is in use (`--gc-sections`), it is often useful to mark sections that should not be eliminated. This is accomplished by surrounding an input section's wildcard entry with KEEP(), as in KEEP(*(.init)) or KEEP(SORT(*)(.ctors)).

20.2.8.9 Input section example

The following example is a complete linker script. It tells the linker to read all of the sections from file `all.o` and place them at the start of output section 'outputa' which starts at location '0x10000'. All of section .input1 from file `foo.o` follows immediately, in the same output section. All of section .input2 from `foo.o` goes into output section 'outputb', followed by section .input1 from `foo1.o`. All of the remaining .input1 and .input2 sections from any files are written to output section 'outputc'.

```
SECTIONS {
  outputa 0x10000 :
    {
    all.o
    foo.o (.input1)
    }
  outputb :
    {
    foo.o (.input2)
    foo1.o (.input1)
    }
  outputc :
    {
```

```
    *(.input1)
    *(.input2)
    }
}
```

20.2.8.10 Output section data

You can include explicit bytes of data in an output section by using BYTE, SHORT, LONG, QUAD, or SQUAD as an output section command. Each keyword is followed by an expression in parentheses providing the value to store (subsection 20.2.14 on page 270). The value of the expression is stored at the current value of the location counter.

The BYTE, SHORT, LONG, and QUAD commands store one, two, four, and eight bytes (respectively). After storing the bytes, the location counter is incremented by the number of bytes stored.

For example, this will store the byte 1 followed by the four byte value of the symbol 'addr':

```
BYTE(1)
LONG(addr)
```

When using a 64 bit host or target, QUAD and SQUAD are the same; they both store an 8 byte, or 64 bit, value. When both host and target are 32 bits, an expression is computed as 32 bits. In this case QUAD stores a 32 bit value zero extended to 64 bits, and SQUAD stores a 32 bit value sign extended to 64 bits.

If the object file format of the output file has an explicit endianness, which is the normal case, the value will be stored in that endianness. When the object file format does not have an explicit endianness, as is true of, for example, S-records, the value will be stored in the endianness of the first input object file.

Note:

These commands only work inside a section description and not between them, so the following will produce an error from the linker:

```
SECTIONS { .text : { *(.text) } LONG(1) .data : { *(.data) } }
```

whereas this will work:

```
SECTIONS { .text : { *(.text) ; LONG(1) } .data : { *(.data) } }
```

You may use the FILL command to set the fill pattern for the current section. It is followed by an expression in parentheses. Any otherwise unspecified regions of memory within the section (for example, gaps left due to the required alignment of input sections) are filled with the value of the expression, repeated as necessary. A FILL statement covers memory locations after the point at which it occurs in the section definition; by including more than one FILL statement, you can have different fill patterns in different parts of an output section.

This example shows how to fill unspecified regions of memory with the value '0x90':

```
FILL(0x90909090)
```

The FILL command is similar to the '=<fillexp>' output section attribute, but it only affects the part of the section following the FILL command, rather than the entire section. If both are used, the FILL command takes precedence. (see subsubsection 20.2.9.5 on page 262, for details on the fill expression.

20.2.8.11 Output section keywords

There are a couple of keywords which can appear as output section commands.

CREATE_OBJECT_SYMBOLS
: The command tells the linker to create a symbol for each input file. The name of each symbol will be the name of the corresponding input file. The section of each symbol will be the output section in which the CREATE_OBJECT_SYMBOLS command appears.

  This is conventional for the a.out object file format. It is not normally used for any other object file format.

CONSTRUCTORS
: When linking using the a.out object file format, the linker uses an unusual set construct to support C++ global constructors and destructors. When linking object file formats which do not support arbitrary sections, such as ECOFF and XCOFF, the linker will automatically recognize C++ global constructors and destructors by name. For these object file formats, the CONSTRUCTORS command tells the linker to place constructor information in the output section where the CONSTRUCTORS command appears. The CONSTRUCTORS command is ignored for other object file formats.

  The symbol __CTOR_LIST__ marks the start of the global constructors, and the symbol __DTOR_LIST marks the end. The first word in the list is the number of entries, followed by the address of each constructor or destructor, followed by a zero word. The compiler must arrange to actually run the code. For these object file formats *GNU* C++ normally calls constructors from a subroutine __main; a call to __main is automatically inserted into the startup code for main. *GNU* C++ normally runs destructors either by using atexit, or directly from the function exit.

  For object file formats such as COFF or ELF which support arbitrary section names, *GNU* C++ will normally arrange to put the addresses of global constructors and destructors into the .ctors and .dtors sections. Placing the following sequence into your linker script will build the sort of table which the *GNU* C++ runtime code expects to see.

```
__CTOR_LIST__ = .;
LONG((__CTOR_END__ - __CTOR_LIST__) / 4 - 2)
*(.ctors)
LONG(0)
__CTOR_END__ = .;
__DTOR_LIST__ = .;
LONG((__DTOR_END__ - __DTOR_LIST__) / 4 - 2)
```

```
*(.dtors)
LONG(0)
__DTOR_END__ = .;
```

If you are using the *GNU* C++ support for initialization priority, which provides some control over the order in which global constructors are run, you must sort the constructors at link time to ensure that they are executed in the correct order. When using the CONSTRUCTORS command, use SORT(CONSTRUCTORS) instead. When using the .ctors and .dtors sections, use *(SORT(.ctors)) and *(SORT(.dtors)) instead of just *(.ctors) and *(.dtors).

Normally the compiler and linker will handle these issues automatically, and you will not need to concern yourself with them. However, you may need to consider this if you are using C++ and writing your own linker scripts.

20.2.8.12 Output section discarding

The linker will not create output sections which do not have any contents. This is for convenience when referring to input sections that may or may not be present in any of the input files. For example:

```
.foo { *(.foo) }
```

will only create a .foo section in the output file if there is a .foo section in at least one input file.

If you use anything other than an input section description as an output section command, such as a symbol assignment, then the output section will always be created, even if there are no matching input sections.

The special output section name /DISCARD/ may be used to discard input sections. Any input sections which are assigned to an output section named /DISCARD/ are not included in the output file.

20.2.9 Output section attributes

We showed above that the full description of an output section looked like this:

```
<section> [<address>] [(<type>)] : [AT(<lma>)]
  {
    <var{output-section-command>
    <output-section-command>
    ...
  } [><region>] [AT><lma_region>] [:<phdr> :<phdr> ...] [=<fillexp>]
```

We've already described <section>, <address>, and <output-section-command>. In this section we will describe the remaining section attributes.

20.2.9.1 Output section type

Each output section may have a type. The type is a keyword in parentheses. The following types are defined:

`NOLOAD` The section should be marked as not loadable, so that it will not be loaded into memory when the program is run.

`DSECT, COPY, INFO, OVERLAY`
These type names are supported for backward compatibility, and are rarely used. They all have the same effect: the section should be marked as not allocatable, so that no memory is allocated for the section when the program is run.

The linker normally sets the attributes of an output section based on the input sections which map into it. You can override this by using the section type. For example, in the script sample below, the 'ROM' section is addressed at memory location '0' and does not need to be loaded when the program is run. The contents of the 'ROM' section will appear in the linker output file as usual.

```
SECTIONS {
  ROM 0 (NOLOAD) : { ... }
  ...
}
```

20.2.9.2 Output section LMA

Every section has a virtual address (VMA) and a load address (LMA); see subsection 20.2.2 on page 243. The address expression which may appear in an output section description sets the VMA (see subsubsection 20.2.8.3 on page 254).

The linker will normally set the LMA equal to the VMA. You can change that by using the AT keyword. The expression <lma> that follows the AT keyword specifies the load address of the section. Alternatively, with AT<lma_region> expression, you may specify a memory region for the section's load address.

This feature is designed to make it easy to build a ROM image. For example, the following linker script creates three output sections: one called .text, which starts at 0x1000, one called .mdata, which is loaded at the end of the .text section even though its VMA is 0x2000, and one called .bss to hold uninitialized data at address 0x3000. The symbol _data is defined with the value 0x2000, which shows that the location counter holds the VMA value, not the LMA value.

```
SECTIONS
  {
  .text 0x1000 : { *(.text) _etext = . ; }
  .mdata 0x2000 :
    AT ( ADDR (.text) + SIZEOF (.text) )
    { _data = . ; *(.data); _edata = . ; }
  .bss 0x3000 :
    { _bstart = . ; *(.bss) *(COMMON) ; _bend = . ;}
}
```

The run-time initialization code for use with a program generated with this linker script would include something like the following, to copy the initialized data from the ROM image to its runtime address. Notice how this code takes advantage of the symbols defined by the linker script.

```
extern char _etext, _data, _edata, _bstart, _bend;
char *src = &_etext;
char *dst = &_data;

%/* ROM has data at end of text; copy it. */
while (dst < &_edata) {
  *dst++ = *src++;
}

%/* Zero bss */
for (dst = &_bstart; dst< &_bend; dst++)
  *dst = 0;
```

20.2.9.3 Output section region

You can assign a section to a previously defined region of memory by using ><region>. (see subsection 20.2.11 on page 264).

Here is a simple example:

```
MEMORY { rom : ORIGIN = 0x1000, LENGTH = 0x1000 }
SECTIONS { ROM : { *(.text) } >rom }
```

20.2.9.4 Output section phdr

You can assign a section to a previously defined program segment by using :<phdr>. If a section is assigned to one or more segments, then all subsequent allocated sections will be assigned to those segments as well, unless they use an explicitly :<phdr> modifier. You can use :NONE to tell the linker to not put the section in any segment at all.

Here is a simple example:

```
PHDRS { text PT_LOAD ; }
SECTIONS { .text : { *(.text) } :text }
```

20.2.9.5 Output section fill

You can set the fill pattern for an entire section by using =<fillexp>. <fillexp> is an expression. Any otherwise unspecified regions of memory within the output section (for example, gaps left due to the required alignment of input sections) will be filled with the value, repeated as necessary. If the fill expression is a simple hex number, ie. a string of hex digit starting with 0x and without a trailing k or M, then an arbitrarily long sequence of hex digits can be used to specify the fill pattern; Leading zeros become part of the pattern too. For all other cases, including extra parentheses or a unary +, the fill pattern is the four least significant bytes of the value of the expression. In all cases, the number is big-endian.

You can also change the fill value with a FILL command in the output section commands; (see subsubsection 20.2.8.10 on page 258).

Here is a simple example:

```
SECTIONS { .text : { *(.text) } =0x90909090 }
```

20.2.10 Overlay description

An overlay description provides an easy way to describe sections which are to be loaded as part of a single memory image but are to be run at the same memory address. At run time, some sort of overlay manager will copy the overlaid sections in and out of the runtime memory address as required, perhaps by simply manipulating addressing bits. This approach can be useful, for example, when a certain region of memory is faster than another.

Overlays are described using the OVERLAY command. The OVERLAY command is used within a SECTIONS command, like an output section description. The full syntax of the OVERLAY command is as follows:

```
OVERLAY [<start>] : [NOCROSSREFS] [AT ( <ldaddr> )]
  {
    <secname1>
      {
        <output-section-command>
        <output-section-command>
        ...
      } [:<phdr>...] [=<fill>]
    <secname2>
      {
        <output-section-command>
        <output-section-command>
        ...
      } [:<phdr>...] [=<fill>]
        ...
  } [><region>] [:<phdr>...] [=<fill>]
```

Everything is optional except OVERLAY (a keyword), and each section must have a name (<secname1> and <secname2> above). The section definitions within the OVERLAY construct are identical to those within the general SECTIONS construct (subsection 20.2.8 on page 252), except that no addresses and no memory regions may be defined for sections within an OVERLAY.

The sections are all defined with the same starting address. The load addresses of the sections are arranged such that they are consecutive in memory starting at the load address used for the OVERLAY as a whole (as with normal section definitions, the load address is optional, and defaults to the start address; the start address is also optional, and defaults to the current value of the location counter).

If the NOCROSSREFS keyword is used, and there are any references among the sections, the linker will report an error. Since the sections all run at the same address, it normally does not make sense for one section to refer directly to another. (see subsubsection 20.2.6.4 on page 250.

For each section within the OVERLAY, the linker automatically defines two symbols. The symbol __load_start_<secname> is defined as the starting load address of the section. The symbol __load_stop_<secname> is defined as the final load address of the section. Any characters within <secname> which are not legal within C identifiers are removed. C (or assembler) code may use these symbols to move the overlaid sections around as necessary.

At the end of the overlay, the value of the location counter is set to the start address of the overlay plus the size of the largest section.

Here is an example. Remember that this would appear inside a SECTIONS construct.

```
OVERLAY 0x1000 : AT (0x4000)
 {
   .text0 { o1/*.o(.text) }
   .text1 { o2/*.o(.text) }
 }
```

This will define both .text0 and .text1 to start at address 0x1000. .text0 will be loaded at address 0x4000, and .text1 will be loaded immediately after .text0. The following symbols will be defined: __load_start_text0, __load_stop_text0, __load_start_text1, __load_stop_text1.

C code to copy overlay .text1 into the overlay area might look like the following.

```
extern char __load_start_text1, __load_stop_text1;
memcpy ((char *) 0x1000, &__load_start_text1,
        &__load_stop_text1 - &__load_start_text1);
```

Note:

The OVERLAY command is just syntactic sugar, since everything it does can be done using the more basic commands. The above example could have been written identically as follows.

```
.text0 0x1000 : AT (0x4000) { o1/*.o(.text) }
__load_start_text0 = LOADADDR (.text0);
__load_stop_text0 = LOADADDR (.text0) + SIZEOF (.text0);
.text1 0x1000 : AT (0x4000 + SIZEOF (.text0)) { o2/*.o(.text) }
__load_start_text1 = LOADADDR (.text1);
__load_stop_text1 = LOADADDR (.text1) + SIZEOF (.text1);
. = 0x1000 + MAX (SIZEOF (.text0), SIZEOF (.text1));
```

20.2.11 MEMORY command

The linker's default configuration permits allocation of all available memory. You can override this by using the MEMORY command.

The MEMORY command describes the location and size of blocks of memory in the target. You can use it to describe which memory regions may be used by the linker, and which memory regions it must avoid. You can then assign sections to particular memory regions. The linker will set section addresses based on the memory regions, and will warn about regions that become too full. The linker will not shuffle sections around to fit into the available regions.

A linker script may contain at most one use of the MEMORY command. However, you can define as many blocks of memory within it as you wish. The syntax is:

```
MEMORY
  {
    <name> [(<attr>)] : ORIGIN = <origin>, LENGTH = <len>
    ...
  }
```

The <name> is a name used in the linker script to refer to the region. The region name has no meaning outside of the linker script. Region names are stored in a separate name space, and will not conflict with symbol names, file names, or section names. Each memory region must have a distinct name.

The <attr> string is an optional list of attributes that specify whether to use a particular memory region for an input section which is not explicitly mapped in the linker script. As described in subsection 20.2.8 on page 252, if you do not specify an output section for some input section, the linker will create an output section with the same name as the input section. If you define region attributes, the linker will use them to select the memory region for the output section that it creates.

The <attr> string must consist only of the following characters:

R Read-only section.

W Read/write section.

X Executable section.

A Allocatable section.

I Initialized section.

L Same as `I`.

P Section contains PCP code or data.

! Invert the sense of any of the preceding attributes.

If a unmapped section matches any of the listed attributes other than !, it will be placed in the memory region. The ! attribute reverses this test, so that an unmapped section will be placed in the memory region only if it does not match any of the listed attributes.

The <origin> is an expression for the start address of the memory region. The expression must evaluate to a constant before memory allocation is performed, which means that you may not use any section relative symbols. The keyword ORIGIN may be abbreviated to org or o (but not, for example, ORG).

The <len> is an expression for the size in bytes of the memory region. As with the <origin> expression, the expression must evaluate to a constant before memory allocation is performed. The keyword LENGTH may be abbreviated to len or l.

In the following example, we specify that there are two memory regions available for allocation: one starting at '0' for 256 kilobytes, and the other starting at '0x40000000' for four megabytes. The linker will place into the 'rom' memory region every section which is not explicitly mapped into a memory region, and is either read-only or executable. The linker will place other sections which are not explicitly mapped into a memory region into the 'ram' memory region.

```
MEMORY
  {
    rom (rx) : ORIGIN = 0, LENGTH = 256K
    ram (!rx) : org = 0x40000000, l = 4M
  }
```

Once you define a memory region, you can direct the linker to place specific output sections into that memory region by using the '>region' output section attribute. For example, if you have a memory region named 'mem', you would use '>mem' in the output section definition. (see subsubsection 20.2.9.3 on page 262). If no address was specified for the output section, the linker will set the address to the next available address within the memory region. If the combined output sections directed to a memory region are too large for the region, the linker will issue an error message.

20.2.12 PHDRS Command

The ELF object file format uses *program headers*, also known as *segments*. The program headers describe how the program should be loaded into memory. You can print them out by using the `tricore-objdump` program with the `-p` option.

When you run an ELF program on a native ELF system, the system loader reads the program headers in order to figure out how to load the program. This will only work if the program headers are set correctly. This manual does not describe the details of how the system loader interprets program headers; for more information, see the ELF ABI.

The linker will create reasonable program headers by default. However, in some cases, you may need to specify the program headers more precisely. You may use the PHDRS command for this purpose. When the linker sees the PHDRS command in the linker script, it will not create any program headers other than the ones specified.

The linker only pays attention to the PHDRS command when generating an ELF output file. In other cases, the linker will simply ignore PHDRS.

This is the syntax of the PHDRS command. The words PHDRS, FILEHDR, AT, and FLAGS are keywords.

```
PHDRS
{
  <name> <type> [ FILEHDR ] [ PHDRS ] [ AT ( <address> ) ]
         [ FLAGS ( <flags> ) ] ;
}
```

The <name> is used only for reference in the SECTIONS command of the linker script. It is not put into the output file. Program header names are stored in a separate name space, and will not conflict with symbol names, file names, or section names. Each program header must have a distinct name.

Certain program header types describe segments of memory which the system loader will load from the file. In the linker script, you specify the contents of these segments by placing allocatable output sections in the segments. You use the ':<phdr>' output section attribute to place a section in a particular segment. See subsubsection 20.2.9.4 on page 262.

It is normal to put certain sections in more than one segment. This merely implies that one segment of memory contains another. You may repeat ':<phdr>', using it once for each segment which should contain the section.

If you place a section in one or more segments using ':<phdr>', then the linker will place all subsequent allocatable sections which do not specify ':<phdr>' in the same segments. This is for convenience, since generally a whole set of contiguous sections will be placed in a single segment. You can use :NONE to override the default segment and tell the linker to not put the section in any segment at all.

You may use the FILEHDR and PHDRS keywords appear after the program header type to further describe the contents of the segment. The FILEHDR keyword means that the segment should include the ELF file header. The PHDRS keyword means that the segment should include the ELF program headers themselves.

The <type> may be one of the following. The numbers indicate the value of the keyword.

`PT_NULL (0)`	Indicates an unused program header.
`PT_LOAD (1)`	Indicates that this program header describes a segment to be loaded from the file.
`PT_DYNAMIC (2)`	Indicates a segment where dynamic linking information can be found.
`PT_INTERP (3)`	Indicates a segment where the name of the program interpreter may be found.
`PT_NOTE (4)`	Indicates a segment holding note information.
`PT_SHLIB (5)`	A reserved program header type, defined but not specified by the ELF ABI.
`PT_PHDR (6)`	Indicates a segment where the program headers may be found.
`<expression>`	An expression giving the numeric type of the program header. This may be used for types not defined above.

You can specify that a segment should be loaded at a particular address in memory by using an AT expression. This is identical to the AT command used as an output section attribute (see subsubsection 20.2.9.2 on page 261). The AT command for a program header overrides the output section attribute.

The linker will normally set the segment flags based on the sections which comprise the segment. You may use the FLAGS keyword to explicitly specify the segment flags. The value of <flags> must be an integer. It is used to set the p_flags field of the program header.

Here is an example of PHDRS. This shows a typical set of program headers used on a native ELF system.

```
PHDRS
{
  headers PT_PHDR PHDRS ;
  interp PT_INTERP ;
  text PT_LOAD FILEHDR PHDRS ;
  data PT_LOAD ;
  dynamic PT_DYNAMIC ;
}

SECTIONS
{
  . = SIZEOF_HEADERS;
  .interp : { *(.interp) } :text :interp
  .text : { *(.text) } :text
  .rodata : { *(.rodata) } /* defaults to :text */
  ...
  . = . + 0x1000; /* move to a new page in memory */
  .data : { *(.data) } :data
  .dynamic : { *(.dynamic) } :data :dynamic
  ...
}
```

20.2.13 VERSION Command

The linker supports symbol versions when using ELF. Symbol versions are only useful when using shared libraries. The dynamic linker can use symbol versions to select a specific version of a function when it runs a program that may have been linked against an earlier version of the shared library.

You can include a version script directly in the main linker script, or you can supply the version script as an implicit linker script. You can also use the `--version-script` linker option.

The syntax of the VERSION command is simply

```
VERSION { version-script-commands }
```

The format of the version script commands is identical to that used by Sun's linker in Solaris 2.5. The version script defines a tree of version nodes. You specify the node names and interdependencies in the version script. You can specify which symbols are bound to which version nodes, and you can reduce a specified set of symbols to local scope so that they are not globally visible outside of the shared library.

The easiest way to demonstrate the version script language is with a few examples.

```
VERS_1.1 {
     global:
          foo1;
     local:
          old*;
          original*;
          new*;
};

VERS_1.2 {
          foo2;
} VERS_1.1;

VERS_2.0 {
          bar1; bar2;
} VERS_1.2;
```

This example version script defines three version nodes. The first version node defined is VERS_1.1; it has no other dependencies. The script binds the symbol foo1 to VERS_1.1. It reduces a number of symbols to local scope so that they are not visible outside of the shared library; this is done using wildcard patterns, so that any symbol whose name begins with old, original , or new is matched. The wildcard patterns available are the same as those used in the shell when matching filenames (also known as "globbing").

Next, the version script defines node VERS_1.2. This node depends upon VERS_1.1. The script binds the symbol foo2 to the version node VERS_1.2.

Finally, the version script defines node VERS_2.0. This node depends upon VERS_1.2. The scripts binds the symbols bar1 and bar2 to the version node VERS_2.0.

When the linker finds a symbol defined in a library which is not specifically bound to a version node, it will effectively bind it to an unspecified base version of the library. You can bind all otherwise unspecified symbols to a given version node by using global : * somewhere in the version script.

The names of the version nodes have no specific meaning other than what they might suggest to the person reading them. The '2.0' version could just as well have appeared in between '1.1' and '1.2'. However, this would be a confusing way to write a version script.

Node name can be omited, provided it is the only version node in the version script. Such version script doesn't assign any versions to symbols, only selects which symbols will be globally visible out and which won't.

```
{ global: foo; bar; local: *; }
```

When you link an application against a shared library that has versioned symbols, the application itself knows which version of each symbol it requires, and it also knows which version nodes it needs from each shared library it is linked against. Thus at runtime, the dynamic loader can make a quick check to make sure that the libraries you have linked against do in fact supply all of the version nodes that the application will need to resolve all of the dynamic symbols. In this way it is possible for the dynamic linker to know with certainty that all external symbols that it needs will be resolvable without having to search for each symbol reference.

The symbol versioning is in effect a much more sophisticated way of doing minor version checking that SunOS does. The fundamental problem that is being addressed here is that typically references to external functions are bound on an as-needed basis, and are not all bound when the application starts up. If a shared library is out of date, a required interface may be missing; when the application tries to use that interface, it may suddenly and unexpectedly fail. With symbol versioning, the user will get a warning when they start their program if the libraries being used with the application are too old.

There are several GNU extensions to Sun's versioning approach. The first of these is the ability to bind a symbol to a version node in the source file where the symbol is defined instead of in the versioning script. This was done mainly to reduce the burden on the library maintainer. You can do this by putting something like:

```
__asm__(".symver original_foo,foo@VERS_1.1");
```

in the C source file. This renames the function original_foo to be an alias for foo bound to the version node VERS_1.1. The local : directive can be used to prevent the symbol original_foo from being exported. A .symver directive takes precedence over a version script.

The second GNU extension is to allow multiple versions of the same function to appear in a given shared library. In this way you can make an incompatible change to an interface without increasing the major version number of the shared library, while still allowing applications linked against the old interface to continue to function.

To do this, you must use multiple .symver directives in the source file. Here is an example:

```
__asm__(".symver original_foo,foo@");
__asm__(".symver old_foo,foo@VERS_1.1");
__asm__(".symver old_foo1,foo@VERS_1.2");
__asm__(".symver new_foo,foo@@VERS_2.0");
```

In this example, foo@ represents the symbol foo bound to the unspecified base version of the symbol. The source file that contains this example would define 4 C functions: original_foo, old_foo, old_foo1, and new_foo.

When you have multiple definitions of a given symbol, there needs to be some way to specify a default version to which external references to this symbol will be bound. You can do this with the foo@@VERS_2.0 type of .symver directive. You can only declare one version of a symbol as the default in this manner; otherwise you would effectively have multiple definitions of the same symbol.

If you wish to bind a reference to a specific version of the symbol within the shared library, you can use the aliases of convenience (i.e. old_foo), or you can use the .symver directive to specifically bind to an external version of the function in question.

You can also specify the language in the version script:

```
VERSION extern "lang" { version-script-commands }
```

The supported langs are 'C', 'C++', and 'Java'. The linker will iterate over the list of symbols at the link time and demangle them according to 'lang' before matching them to the patterns specified in 'version-script-commands'.

20.2.14 Expressions in Linker Scripts

The syntax for expressions in the linker script language is identical to that of C expressions. All expressions are evaluated as integers. All expressions are evaluated in the same size, which is 32 bits if both the host and target are 32 bits, and is otherwise 64 bits.

You can use and set symbol values in expressions.

The linker defines several special purpose builtin functions for use in expressions.

20.2.15 Constants

All constants are integers.

As in C, the linker considers an integer beginning with '0' to be octal, and an integer beginning with '0x' or '0X' to be hexadecimal. The linker considers other integers to be decimal.

In addition, you can use the suffixes K and M to scale a constant by 1024 or 1024^2 respectively. For example, the following all refer to the same quantity:

```
  _fourk_1 = 4K;
  _fourk_2 = 4096;
  _fourk_3 = 0x1000;
```

20.2.16 Symbol Names

Unless quoted, symbol names start with a letter, underscore, or period and may include letters, digits, underscores, periods, and hyphens. Unquoted symbol names must not conflict with any keywords. You can specify a symbol which contains odd characters or has the same name as a keyword by surrounding the symbol name in double quotes:

```
  "SECTION" = 9;
  "with a space" = "also with a space" + 10;
```

Since symbols can contain many non-alphabetic characters, it is safest to delimit symbols with spaces. For example, 'A-B' is one symbol, whereas 'A - B' is an expression involving subtraction.

20.2.17 The Location Counter

The special linker variable *dot* '.' always contains the current output location counter. Since the . always refers to a location in an output section, it may only appear in an expression within a SECTIONS command. The . symbol may appear anywhere that an ordinary symbol is allowed in an expression.

Assigning a value to . will cause the location counter to be moved. This may be used to create holes in the output section. The location counter may never be moved backwards.

```
SECTIONS
{
  output :
    {
      file1(.text)
      . = . + 1000;
      file2(.text)
      . += 1000;
      file3(.text)
    } = 0x12345678;
}
```

In the previous example, the .text section from file1 is located at the beginning of the output section output. It is followed by a 1000 byte gap. Then the .text section from file2 appears, also with a 1000 byte gap following before the .text section from file3. The notation = 0x12345678 specifies what data to write in the gaps (see subsubsection 20.2.9.5 on page 262).

Note:

. actually refers to the byte offset from the start of the current containing object. Normally this is the SECTIONS statement, who's start address is 0, hence . can be used as an absolute address. If . is used inside a section description however, it refers to the byte offset from the start of that section, not an absolute address.

Thus in a script like this:

```
SECTIONS
{
    . = 0x100
    .text: {
      *(.text)
      . = 0x200
    }
    . = 0x500
    .data: {
      *(.data)
      . += 0x600
    }
}
```

The .text section will be assigned a starting address of 0x100 and a size of exactly 0x200 bytes, even if there is not enough data in the .text input sections to fill this area. (If there is too much data, an error will be produced because this would be an attempt to move . backwards). The .data section will start at 0x500 and it will have an extra 0x600 bytes worth of space after the end of the values from the .data input sections and before the end of the .data output section itself.

20.2.18 Operators

The linker recognizes the standard C set of arithmetic operators, with the standard bindings and precedence levels:

precedence	associativity	Operators
(highest)		
1	left	! - ˜
2	left	* / %
3	left	+ -
4	left	≫ ≪
5	left	== != > < <= >=
6	left	&
7	left	|
8	left	&&
9	left	||
10	right	? :
11	right	&= += -= *= /=

20.2.19 Evaluation

The linker evaluates expressions lazily. It only computes the value of an expression when absolutely necessary.

The linker needs some information, such as the value of the start address of the first section, and the origins and lengths of memory regions, in order to do any linking at all. These values are computed as soon as possible when the linker reads in the linker script.

However, other values (such as symbol values) are not known or needed until after storage allocation. Such values are evaluated later, when other information (such as the sizes of output sections) is available for use in the symbol assignment expression.

The sizes of sections cannot be known until after allocation, so assignments dependent upon these are not performed until after allocation.

Some expressions, such as those depending upon the location counter ., must be evaluated during section allocation.

If the result of an expression is required, but the value is not available, then an error results. For example, a script like the following

```
SECTIONS
  {
    .text 9+this_isnt_constant :
      { *(.text) }
  }
```

will cause the error message `non constant expression for initial address`.

20.2.20 The Section of an Expression

When the linker evaluates an expression, the result is either absolute or relative to some section. A relative expression is expressed as a fixed offset from the base of a section.

The position of the expression within the linker script determines whether it is absolute or relative. An expression which appears within an output section definition is relative to the base of the output section. An expression which appears elsewhere will be absolute.

A symbol set to a relative expression will be relocatable if you request relocatable output using the −r option. That means that a further link operation may change the value of the symbol. The symbol's section will be the section of the relative expression.

A symbol set to an absolute expression will retain the same value through any further link operation. The symbol will be absolute, and will not have any particular associated section.

You can use the builtin function ABSOLUTE to force an expression to be absolute when it would otherwise be relative. For example, to create an absolute symbol set to the address of the end of the output section .data:

```
SECTIONS
  {
    .data : { *(.data) _edata = ABSOLUTE(.); }
  }
```

If ABSOLUTE were not used, _edata would be relative to the .data section.

20.2.21 Builtin Functions

The linker script language includes a number of builtin functions for use in linker script expressions.

`ABSOLUTE(<exp>)`
: Return the absolute (non-relocatable, as opposed to non-negative) value of the expression <exp>. Primarily useful to assign an absolute value to a symbol within a section definition, where symbol values are normally section relative. (see subsection 20.2.20 on page 273).

`ADDR(<section>)`
: Return the absolute address (the VMA) of the named <section>. Your script must previously have defined the location of that section. In the following example, symbol_1 and symbol_2 are assigned identical values:

```
SECTIONS { ...
  .output1 :
    {
    start_of_output_1 = ABSOLUTE(.);
    ...
    }
  .output :
    {
    symbol_1 = ADDR(.output1);
    symbol_2 = start_of_output_1;
    }
... }
```

`ALIGN(<exp>)`
: Return the location counter (.) aligned to the next <exp> boundary. <exp> must be an expression whose value is a power of two. This is equivalent to

```
(. + <exp> - 1) & ~(<exp> - 1)
```

ALIGN doesn't change the value of the location counter—it just does arithmetic on it. Here is an example which aligns the output .data section to the next 0x2000 byte boundary after the preceding section and sets a variable within the section to the next 0x8000 boundary after the input sections:

```
SECTIONS { ...
  .data ALIGN(0x2000): {
    *(.data)
    variable = ALIGN(0x8000);
  }
... }
```

The first use of ALIGN in this example specifies the location of a section because it is used as the optional <address> attribute of a section definition (see subsubsection 20.2.8.3 on page 254). The second use of ALIGN is used to define the value of a symbol.

The builtin function NEXT is closely related to ALIGN.

`BLOCK(<exp>)` This is a synonym for ALIGN, for compatibility with older linker scripts. It is most often seen when setting the address of an output section.

`DATA_SEGMENT_ALIGN(<maxpagesize>,<commonpagesize>)`
This is equivalent to either

```
(ALIGN(<maxpagesize>) + (. & (<maxpagesize> - 1)))
```

or

```
(ALIGN(<maxpagesize>) + (. & (<maxpagesize> - <commonpagesize>)))
```

depending on whether the latter uses fewer <commonpagesize> sized pages for the data segment (area between the result of this expression and DATA_SEGMENT_END) than the former or not. If the latter form is used, it means <commonpagesize> bytes of runtime memory will be saved at the expense of up to <commonpagesize> wasted bytes in the on-disk file.

This expression can only be used directly in SECTIONS commands, not in any output section descriptions and only once in the linker script. <commonpagesize> should be less or equal to <maxpagesize> and should be the system page size the object wants to be optimized for (while still working on system page sizes up to <maxpagesize>).

Example:

```
  . = DATA_SEGMENT_ALIGN(0x10000, 0x2000);
```

`DATA_SEGMENT_END(<exp>)`
This defines the end of data segment for DATA_SEGMENT_ALIGN evaluation purposes.

```
  . = DATA_SEGMENT_END(.);
```

`DEFINED(<symbol>)`
Return 1 if <symbol> is in the linker global symbol table and is defined, otherwise return 0. You can use this function to provide default values for symbols. For example, the following script fragment shows how to set a global symbol begin to the first location in the .text section—but if a symbol called begin already existed, its value is preserved:

```
SECTIONS { ...
  .text : {
    begin = DEFINED(begin) ? begin : . ;
    ...
  }
  ...
}
```

`LOADADDR(<section>)`
: Return the absolute LMA of the named <section>. This is normally the same as ADDR, but it may be different if the AT attribute is used in the output section definition (subsubsection 20.2.9.2 on page 261).

`MAX(<exp1>, <exp2>)`
: Returns the maximum of <exp1> and <exp2>.

`MIN(<exp1>, <exp2>)`
: Returns the minimum of <exp1> and <exp2>.

`NEXT(<exp>)`
: Return the next unallocated address that is a multiple of <exp>. This function is closely related to ALIGN(<exp>); unless you use the MEMORY command to define discontinuous memory for the output file, the two functions are equivalent.

`SIZEOF(<section>)`
: Return the size in bytes of the named <section>, if that section has been allocated. If the section has not been allocated when this is evaluated, the linker will report an error. In the following example, symbol_1 and symbol_2 are assigned identical values:

```
SECTIONS{ ...
  .output {
    .start = . ;
    ...
    .end = . ;
    }
  symbol_1 = .end - .start ;
  symbol_2 = SIZEOF(.output);
... }
```

`SIZEOF_HEADERS, sizeof_headers`
: Return the size in bytes of the output file's headers. This is information which appears at the start of the output file. You can use this number when setting the start address of the first section, if you choose, to facilitate paging. When producing an ELF output file, if the linker script uses the SIZEOF_HEADERS builtin function, the linker must compute the number of program headers before it has determined all the section addresses and sizes. If the linker later discovers that it needs additional program headers, it will report an error `not enough room for program headers`. To avoid this error, you must avoid using the SIZEOF_HEADERS function, or you must rework your linker script to avoid forcing the linker to use additional program headers, or you must define the program headers yourself using the PHDRS command (see subsection 20.2.12 on page 266).

20.2.22 Implicit Linker Scripts

If you specify a linker input file which the linker can not recognize as an object file or an archive file, it will try to read the file as a linker script. If the file can not be parsed as a linker script, the linker will report an error.

An implicit linker script will not replace the default linker script.

Typically an implicit linker script would contain only symbol assignments, or the `INPUT`, `GROUP`, or `VERSION` commands.

Any input files read because of an implicit linker script will be read at the position in the command line where the implicit linker script was read. This can affect archive searching.

20.3 The Mapfile

The linker has a built-in function which outputs detailed information about the generated executable. This information is written to the link map. The link map gives the user detailed information of the used sections and symbols and their address locations. Further the dependency of required libraries for the object files is listed. The common symbols and the memory configuration is also available. In addition to that you can build a cross reference table, which lists all global symbols.

The link map is generated as output on the command line per default. The linker can however generate a separate output file in which the link map is written, the mapfile. Since the content of the link map is useful for error tracking it is recommendable to write the link map in the mapfile, which may examined after the compilation process. So this manual only speaks of the mapfile.

The Mapfile is divided in different sections:

Archive Members List of linked object files from archives or libraries.

Allocating common symbols List of common symbols.

Memory configuration List of available memory region.

Linker script and Memory map Information from the linker script about the allocation of memory addresses and symbols.

Cross Reference Table (optional) List of files, where the symbols are defined and referenced.

20.3.1 Generating the Mapfile

The `Mapfile` and the cross reference table are generated by the linker, if the required options are set. these options may either be set directly at the invocation of the linker, if the linker is invoked without `tricore-gcc` (this is not recommended) or via the option `-Wl` of the control program.

These options are available for the linker to generate the link map and the cross reference table:

```
tricore-ld -M
tricore-gcc -Wl,-M
```

The option `-M` generates a Link Map at the command-line.

```
tricore-ld -Map <mapfile>
tricore-gcc -Wl,-Map,<mapfile>
```

The option `-Map <mapfile>` saves the link map in the file <mapfile>.

```
tricore-ld --cref
tricore-gcc -Wl,--cref
```

The option `--cref` is required for generating a cross reference table by the linker. By default the output is generated in the command-line. The combination of `--cref` and `-Map <mapfile>` appends a own section in the `Mapfile` containing the cross reference table:

```
tricore-ld -Map <mapfile> --cref
tricore-gcc -Wl,-Map,<mapfile>,--cref
```

20.3.2 Archive Members

The Archive Members section of the mapfile lists the object files that are taken from libraries to be added to the executable. For all added object files the name of the object file, the name and path to the library it is taken from, the name of the referenced symbol and the name of the file where the symbol is referenced are listed.

The first two lines of the following output tells the user that in `main.o` the symbol __main is referenced. This is the reason why __main.o is added from the library `libgcc.a`.

```
Archive member included because of file (symbol)

<install-dir>\lib\gcc-lib\tricore\<gcc-version>\libgcc.a(__main.o)
                              main.o (__main)
<install-dir>\tricore\lib\libos.a(cint.o)
                              uart.o (_install_int_handler)
...
```

Similar for `cint.o` from `libos.a`. This object file is required, because in `uart.o` exists a reference to the symbol _install_int_handler.

20.3.3 Allocating bit symbols

If the program contains _bit variables these variables are collected in a common section in the Mapfile. In this section all _bit variables are collected with their byte addresses and the bit offsets which defines the bit in the byte. The address of the byte and the bit offset is separated by a dot. As additional information the scope of the variables and the file in which a variable is defined are listed.

```
Bit symbol          bit address  l/g  file
foobit              0xa0000000.0  g   uart.o
```

20.3.4 Allocating common symbols

Common Symbols are global variables which are not initialized. The name of the object file where the symbols are defined and the size of the symbol is listed in the Mapfile.

The linker scans all object files for common symbols, because these symbols are written in a separate section. Because common symbols do not have an own input section the variables must be collected before the allocation.

> **Note:**
>
> The keyword COMMON in an output section in the linker description file tells the linker to put the common symbols in this output section.

```
Common symbol       size      file
Tdisptab            0x20      <install-dir>\tricore\lib\libos.a(cint.o)
uarts               0x4       uart.o
...
```

The size of the common symbol Tdisptab is 0x20. It is defined in `cint.o` in the library `libos.a`. The same for the common symbol uarts: it has the size of 0x4 and is defined in `uart.o`.

20.3.5 Memory configuration

The memory configuration lists the available memory regions and their names. In addition the start address, the length and attributes of the section is listed. These information is part of the linker script and is written to the Mapfile.

```
Name             Origin             Length             Attributes
ext_cram         0xa0000000         0x00080000
ext_dram         0xa0080000         0x00100000
...
```

The memory ext_cram starts at 0xa0000000 and the length is 0x00080000. For these sections no attribute is specified in the linker description files.

20.3.6 Linker script and Memory map

A main part of the Mapfile is the Memory Map section. The Memory Map gives a overview how the input sections that are defined in the object files are combined to output sections. The linker script controls this mechanism. Both the allocation of the symbols and the control info of the linker script is listed.

The Memory Map starts with the list of object files and the archive files, that are linked by `tricore-ld`.

```
LOAD <install-dir>\lib\gcc-lib\tricore\<gcc-version>\crt0.o
LOAD main.o
LOAD uart.o
LOAD <install-dir>\lib\gcc-lib\tricore\<gcc-version>\libgcc.a
...
```

Next you find a list which looks like this but a little bit longer:

```
.text           0xa0000000     0x5000
                0xa0000000               . = ALIGN (0x4)
 *(.text)
 .text          0xa0000000      0x1ec <install-dir>\lib\gcc-lib\tricore\
                                                  <gcc-version>\crt0.o
                0xa0000000               _start
 .text          0xa00001ec        0xc main.o
                0xa00001ec               main
 .text          0xa00001f8       0xe4 uart.o
                0xa00002ac                asc0senddata
                0xa000029a                asc0getdata
                0xa00001f8                inituart
                0xa00002ce                rxint

   ...

 *fill*         0xa0002072       0x8e
 .traptab       0xa0002100      0x100 <install-dir>\tricore\lib\
                                                 libos.a(cint.o)
                0xa00021a0                __trap_5
                0xa0002140                __trap_2
                0xa00021e0                __trap_7
                0xa0002180                __trap_4
                0xa0002120                __trap_1
                0xa00021c0                ___trap_6
                0xa0002100                TriCore_trap_table

   ...

.rodata         0xa0005000      0x104
                0xa0005000                . = ALIGN (0x8)
 *(.rodata)

   ...
```

This list is an excerpt of a 'real' mapfile and shall be the example to describe the content of the memory map.

The Memory Map is organized in blocks. Each of this blocks describes an output section. In these blocks not only the input sections are listed but also the symbols and linker information from the linker script for the output sections.

Each block for an output section starts with the name of the section.

```
.text           0xa0000000     0x5000
```

This line has no leading space. The output section `.text` starts at address `0xa0000000`. The length of the output section is `0x5000`.

The next line contains the linker information of the linker script.

```
                0xa0000000                . =  ALIGN (0x4)
```

All input sections that are linked to an output are listed below the output section. To distinguish input and output sections, the input sections have a leading space in the Mapfile.

The rules how the input sections are linked in output sections is part of the linker script. Here a small cutoff from a linker description file.

```
(*(EXCLUDE_FILE (*foo.o) .text))
```

This entry in the linker description file advises the linker to exclude all files with the extension *`foo.o`. Then the <object file> and the (<input section>) is specified. Instead of the name of the object file also a regular expression is possible. In one output section several of such rules can be defined, so different input sections can be merged to one output section.

For a detailed description see section 20.2 on page 237.

```
 *(.text)
```

The linker script defines that all `.text` input sections of all object files are linked in this output section.

Then the input sections of the assigned object files from the linker script are listed. Each input section starts with its name. The start address and the length of the object files is in the same line. The input section has a leading space, so you can distinguish it from a output section.

```
.text          0xa00001f8       0xe4 uart.o
```

The section `.text` of the input file `uart.o` is input in this output section. The start address of this input section in the executable is 0xa00001f8 and its length is 0xe4 bytes.

Each global symbol of the input sections is listed with its name and address in a separated line.

```
                0xa00002ac                asc0senddata
                0xa000029a                asc0getdata
                0xa00001f8                inituart
                0xa00002ce                rxint
```

For each input section you can find such an entry in the `Mapfile`.

In this example the output section `.text` does *not* only include input sections `.text` but also an input section `.traptab`. The input section has an alignment of 2^8, this means the start is a multiple of this unit. For this requirement the block between the allocated address of the first symbol of the input section and the address of alignment is filled. The keyword ∗ fill ∗ at the beginning of the line indicates that.

```
 *fill*         0xa0002072       0x8e
```

At address 0xa0002072, 0x8e bytes are filled because of alignment.

After all input sections are merged in the output section `.text`, a new output section is started:

```
.rodata         0xa0005000      0x104
```

20.3.7 Cross reference table

The last part of the Mapfile is the cross reference table.

Note:

This table is generated if you call `tricore-gcc` or `tricore-ld` with the `--cref` option.

The cross reference table contains each symbol and the files where they are defined and referenced. For each symbol one or more lines can exist. The first line specifies the file where the symbol is defined. The next lines contain the names of the object files where the symbol is referenced.

```
Symbol              File
Cdisptab            <install-dir>\tricore\lib\libos.a(int1.o)
                    <install-dir>\tricore\lib\libos.a(cint.o)
Tdisptab            <install-dir>\tricore\lib\libos.a(trap6.o)
                    <install-dir>\tricore\lib\libos.a(cint.o)
inituart            uart.o
                    main.o
     ...
```

The symbol inituart is defined in the object file `uart.o` and referenced in `main.o`.

20.4 The Extended Mapfile

In the default mapfile the listing of the symbols is not that clear. So the default mapfile is extended by additional sections.

20.4.1 Generating the extended mapfile

The extended map file is generated by passing the option `--extmap=<output>` to the linker. The kind of output included in the extended mapfile may be configured by the output-option. These output-options are available:

option	effect
h	print header (version, date, ...)
L	list global symbols sorted by name
l	list all symbols sorted by name
N	list global symbols sorted by address
n	list all symbols sorted by address
m	list memory segments
a	all of the above.

Table 20.1: Parameters for `--extmap`

To generate an extended map file both options `--Map` or `-M` and `--extmap` must be passed to the linker.

20.4.2 Linker information

The section 'Linker and link run information' contains information about the used linker, the path to the linker executable, the date and time of the link run and the name of the generated map file. This section is generated if the option `--extmap` has at least the parameter `h` set.

20.4.3 Symbols sorted by address

The section 'Symbols; sorted by address' contains the symbols, that are included in the linkers output file, sorted by the symbols names. A table of these symbols is output with a line for each symbol. The table contains the following columns:

Start The start address of the symbol

End Its end address

Size The size of the symbol

S This column has the value `g` for global symbols and `l` for static symbols

Name The name of the symbol

Memory The memory segment the symbol is output to. See subsection 20.4.5 on page 283 for a list of the memory segments

O-Sec The section the symbols is output to

I-Sec The input section the linker read the symbol from

Input object The input object that contained the symbol

This section is generated if the option `--extmap` has at least the parameters `N` or `n` set. `N` lists all global symbols, `n` lists all symbols

Note:

The size of the columns is calculated every time the map file is generated and depends on the size of the biggest entry.

20.4.4 Symbols sorted by name

This section also contains a list of the symbols. This list is sorted by the name of the symbol rather than by their addresses. The table in this section has the same columns as in the Section 'Symbols; sorted by address'. This section is generated if the option `--extmap` has at least the parameters `L` or `l` set. `L` lists all global symbols, `l` lists all symbols.

20.4.5 Memory segments

The section 'Memory segments' contains the memory segments sorted by name. These memory segments are displayed in a table with the following columns:

Name The name of the memory segment

Start The start address of the memory segment

End Its end address

Used The bytes used in the memory segment

Free The free space in the memory segment

This section is generated if the option `--extmap` has at least the parameter `m` set.

Part IV

More BinUtils and Tools

21 Binutils

Introduction

This brief manual contains documentation for the *GNU* binary utilities:

`tricore-addr2line`	Convert addresses into file names and line numbers.
`tricore-ar`	Create, modify, and extract from archives.
`tricore-nm`	List symbols from object files.
`tricore-objcopy`	Copy and translate object files.
`tricore-objdump`	Display information from object files.
`tricore-ranlib`	Generate index to archive contents.
`tricore-readelf`	Display the contents of ELF format files.
`tricore-size`	List file section sizes and total size.
`tricore-strings`	List printable strings from files.
`tricore-strip`	Discard symbols.

21.1 tricore-addr2line

21.1.1 Description

```
addr2line [options] [address address ...]
```

`tricore-addr2line` translates program addresses into file names and line numbers. Given an address and an executable, it uses the debugging information in the executable to figure out which file name and line number are associated with a given address.

The executable to use is specified with the `-e` option. The default output file is `a.out`.

`tricore-addr2line` has two operation modes.

In the first, hexadecimal addresses are specified on the command line, and line number for each address are displayed by `tricore-addr2line`.

In the second, `tricore-addr2line` reads hexadecimal addresses from standard input, and prints the file name and line number for each address on standard output. In this mode, `tricore-addr2line` may be used in a pipe to convert dynamically chosen addresses.

The format of the output is <filename>:<lineno>. The file name and line number for each address is printed on a separate line. If the `-f` option is used, then each <filename>:<lineno> line is preceded by a <functionname> line which is the name of the function containing the address.

If the file name or function name can not be determined, `tricore-addr2line` will print two question marks in their place. `tricore-addr2line` will print 0, if the line number can not be determined.

21.1.2 Invocation

The long and short forms of options, shown here as alternatives, are equivalent.

`-b <bfdname>`
`--target=<bfdname>`
	Set the binary file format

`-C [style]`
`--demangle[=style]`
	Demangle function names. Decode *demangle* low-level symbol names into user-level names. Besides removing any initial _ prepended by the system, this makes C++ function names readable. Different compilers have different mangling styles. The optional demangling style argument can be used to choose an appropriate demangling style for your compiler.

`-e <executable>`
`--exe=<executable>`
	Specify the name of the executable for which addresses should be translated. The default file is `a.out`.

`-f --functions`	Display function names as well as file and line number information.
`-s --basenames`	Strip directory names and display only the base of each file name.
`-h --help`	Display help information
`-v --version`	Display the program's version

21.1.3 Example

In this example the code at addresses `0xa000031c` and `0xa00001f0` in the executable `main` are of interest. `Addr2line` is used to find out from which function in which file the code at these addresses are taken. Only the name of the files, not the complete paths are to be output by `addr2line`

```
tricore-addr2line --exe=main -s -f 0xa00001f0 0xa000031c
```

```
main
main.c:27
asc0senddata
uart.c:94
```

21.2 tricore-ar

21.2.1 Description

`tricore-ar [-]<option>[<modifier>] <archive> [<member> ...]`

The `tricore-ar` program creates, modifies, and extracts from archives. An *archive* is a single file holding a collection of other files in a structure that makes it possible to retrieve the original individual files (called 'members' of the archive).

The original files contents, mode (permissions), timestamp, owner, and group are preserved in the archive, and can be restored on extraction.

`tricore-ar` can maintain archives whose members have names of any length; however, depending on how `tricore-ar` is configured on your system, a limit on member-name length may be imposed for compatibility with archive formats maintained with other tools. If it exists, the limit is often 15 characters (typical of formats related to a.out) or 16 characters (typical of formats related to coff).

`tricore-ar` is considered a binary utility because archives created with `tricore-ar` are most often used as *libraries* holding commonly needed object files.

`tricore-ar` creates an index to the symbols defined in relocatable object modules in the archive when you specify the modifier 's'. Once created, this index is updated in the archive whenever `tricore-ar` makes a change to its contents (save for the 'q' update operation). An archive with such an index speeds up linking to the library, and allows routines in the library to call each other without regard to their placement in the archive.

You may use `tricore-nm -s` or `tricore-nm --print-armap` to list this index table. If an archive lacks the table `tricore-ranlib` can be used to add just the table.

21.2.2 Invocation

`tricore-ar [-]<option>[<modifier>] <archive> [<member> ...]`

`tricore-ar` is controlled by command line options. These command line options are always one letter which specifies the operation that is to be done and one ore more optional modifiers to modify the default behavior of the operation. `tricore-ar` expects one option which defines what to do (optionally with a modifier) and one archive that is modified.

Most of the UNIX-tools expect a dash preceding the options. This dash is optional.

The available options for `tricore-ar` are.

`d`	Delete modules from the archive. Specify the names of modules to be deleted as <member>. The archive is untouched if you specify no files to delete. If you specify the 'v' modifier, `tricore-ar` lists each module as it is deleted.
`m`	Use this operation to move members in an archive. The ordering of members in an archive can make a difference in how programs are

linked using the library, if a symbol is defined in more than one member. If no modifiers are used with `m`, any members you name in the <member> arguments are moved to the end of the archive. You may use the modifiers `a`, `b` or `i` to move them to a specified place instead.

`p` Print the specified members of the archive, to the standard output file. If the modifier `v` is specified, show the member name before copying its contents to standard output. If you specify no <member> arguments, all the files in the archive are printed.

Note:

This option prints the binary contents of the archive member to stdout! This will cause weird characters, beeps and line feeds.

`q` Quick add the files <member> to the end of <archive> without checking for replacement.

The modifiers `a`, `b` and `i` do not affect this operation; new members are always placed at the end of the archive.

The modifier `v` makes `tricore-ar` list each file as it is appended.

Since the point of this operation is speed, the archive's symbol table index is not updated, even if it already existed. You can use `tricore-ar s` or `tricore-ranlib` explicitly to update the symbol table index.

Note:

However, too many different systems assume quick append rebuilds the index, so GNU ar implements `q` as a synonym for `r`.

`r` Insert the files <member> ... into <archive> with replacement. This operation differs from 'q' in that any previously existing members are deleted if their names match those being added.

If one of the files named in <member> ... does not exist, `tricore-ar` displays an error message, and leaves undisturbed any existing members of the archive matching that name.

By default, new members are added at the end of the file. But you may use one of the modifiers `a`, `b` or `i` to request placement relative to some existing member.

The modifier `v` used with this operation elicits a line of output for each file inserted, along with one of the letters `a` or `r` to indicate whether the file was appended (no old member deleted) or replaced.

To only update the files in the archive, use the modifier `u`.

`t` Display a table listing the contents of <archive>, or those of the files listed in <member> ... that are present in the archive. Normally only the member name is shown. If you also want to see the modes (permissions), timestamp, owner, group, and size, you can request that by also specifying the modifier `v`.

`x[o]` Extract members (named <member>) from the archive. You can use the modifier 'v' with this operation, to request that `tricore-ar` list each name as it extracts it.

If you do not specify a <member>, all files in the archive are extracted.

A number of modifiers may immediately follow the option to specify variations on an operation's behavior:

`a <relpos>` Add new files after an existing member of the archive. If you use the modifier `a`, the name of an existing archive member must be present as the <relpos> argument, before the <archive> specification.

`b <relpos>` Add new files before an existing member of the archive. If you use the modifier `b`, the name of an existing archive member must be present as the <relpos> argument, before the <archive> specification. This modifier is equivalent to `i`.

`c` Create the archive. The specified <archive> is always created if it did not exist, when you request an update. But a warning is issued unless you specify in advance that you expect to create it, by using this modifier.

`i` Same as 'b'.

`N <count>` Uses the <count> parameter. This is used if there are multiple entries in the archive with the same name. Extract or delete instance <count> of the given name from the archive.

`o` Preserve the original dates of members when extracting them. If you do not specify this modifier, files extracted from the archive are stamped with the time of extraction.

`P` Use the full path name when matching names in an archive. `tricore-ar` can not create an archive with a full path name (such archives are not POSIX compliant), but other archive creators can. This option will cause `tricore-ar` to match file names using a complete path name, which can be convenient when extracting a single file from an archive created by another tool.

`s` Write an object-file index into the archive, or update an existing one, even if no other change is made to the archive. You may use this modifier flag either with any operation, or alone. Running `tricore-ar s` on an archive is equivalent to running `tricore-ranlib` on it.

`S` Do not generate an archive symbol table. This can speed up building a large library in several steps. The resulting archive can not be used with the linker. In order to build a symbol table, you must omit

	the `S` modifier on the last execution of `tricore-ar` or you must run `tricore-ranlib` on the archive.
`u`	Normally, `tricore-ar r` inserts all files listed into the archive. If you would like to insert only those of the files you list that are newer than existing members of the same names, use this modifier. The 'u' modifier is allowed only for the operation `r` (replace). In particular, the combination `qu` is not allowed, since checking the timestamps would lose any speed advantage from the operation `q`.
`v`	This modifier requests the verbose version of an operation. Many operations display additional information, such as filenames processed, when the modifier `v` is appended.
`V`	This modifier shows the version number of `tricore-ar`.

21.2.3 Examples

The following examples deal with the object files `uart.o`, `rs232.o` and `tty.o`. These object files shall be placed in a library called `libserial.a`.

First all objects are added to the library. To see what `tricore-ar` is doing, the modifier `v` is used to activate the verbose mode.

```
tricore-ar rv libserial.a *.o
```

```
a - rs232.o
a - tty.o
a - uart.o
```

The content of the archive is displayed. If the modifier `v` was not set, only the names of the objects would have been displayed.

```
tricore-ar tv libserial.a
```

```
rw-rw-rw- 0/0 7208 Mar 10 12:19 2004 rs232.o
rw-rw-rw- 0/0 4288 Mar 10 12:19 2004 tty.o
rw-rw-rw- 0/0 2095 Mar 10 12:19 2004 uart.o
```

Now the file `rs232.o` should be moved from the beginning of the archive to the location after the file `tty.o`:

```
tricore-ar mav tty.o libserial.a rs232.o
```

```
m - rs232.o
```

If one of the object files has been changed, it must be updated in the archive. To update all files in the archive the modifier `u` is used. In this example only the object `tty.o` has changed and is replaced in the archive.

```
tricore-ar ruv libserial.a *.o
```

```
r - tty.o
```

To remove objects from an archive use the option `d`:

```
tricore-ar dv libserial.a tty.o
```

```
d - tty.o
```

21.3 tricore-c++filt

21.3.1 Description

`tricore-c++filt [options] <symbol>`

The C++ language provides function overloading, which means that you can write many functions with the same name (providing each takes parameters of different types). All C++ function names are encoded into a low-level assembly label (this process is known as mangling). The tricore-c++filt program does the inverse mapping: it decodes (demangles) low-level names into user-level names so that the linker can keep these overloaded functions from clashing.

Every alphanumeric word (consisting of letters, digits, underscores, dollars, or periods) seen in the input is a potential label. If the label decodes into a C++ name, the C++ name replaces the low-level name in the output.

If no symbol arguments are given, tricore-c++filt reads symbol names from the standard input and writes the demangled names to the standard output. All results are printed on the standard output.

Warning: tricore-c++filt is a new utility, and the details of its user interface are subject to change in future releases. In particular, a command-line option may be required in the the future to decode a name passed as an argument on the command line; in other words `tricore-c++filt <symbol>` may in a future release become `tricore-c++filt <option> <symbol>`

21.3.2 Invocation

`-_, --strip-underscores`
: On some systems, both the C and C++ compilers put an underscore in front of every name. For example, the C name foo gets the low-level name _foo. This option removes the initial underscore. Whether tricore-c++filt removes the underscore by default is target dependent.

`-n --no-strip-underscores`
: Do not remove the initial underscore.

`--help` Print a summary of the options to tricore-c++filt and exit.

`--version` Print the version number of tricore-c++filt and exit.

21.3.3 Example

Assumed you have a C++-program with a class Serial_T, which as a method **int** init (**void**).
This program is compiled to an object file and the symbols in it are displayed by `tricore-objdump -t`:

```
tricore-objdump -t serial.o
```

```
  serial.o: file format elf32-tricore

SYMBOL TABLE:
00000000 l df *ABS* 00000000 text.cpp
00000000 l d .text 00000000
00000000 l d .data 00000000
00000000 l d .bss 00000000
00000000 *UND* 00000000 _ZN8Serial_T4initEv
```

The name of the method is output in the symbol table as `_ZN8Serial_T4initEv`. This is the compiler internal low-level format for C++ function. To demangle this function name use `tricore-c++filt`:

```
tricore-c++filt _ZN8Serial_T4initEv
```

```
Serial_T::init()
```

21.4 tricore-nm

21.4.1 Description

```
tricore-nm [options] [<filename>]
```

The `tricore-nm` utility can be used to list all the symbols defined in (or referenced from) an object file, static archive library or a shared library. If no file is named on the command line, the file name `a.out` is assumed. Using the command-line options, the symbols can be organized according to their address, size, or name, and the output can be formatted in number of ways. The symbols can also be demangled and presented in the same form as they appear in the original source code.

The type of symbol is coded in a letter. You will find the descriptions of these letters later in this chapter.

21.4.2 Invocation

`-a, --debug-syms`
: Displays the symbols intended for use by the debugger. Normally these are not listed.

`-A, --print-file-name`
: Print name of the input file before every symbol

`-B`
: Same as `--format=bsd`. This is the default.

`-C <style>`

`--demangle[=<style>]`
: Decode (*demangle*) low-level symbol names into user-level names. Besides removing any initial underscore prepended by the system, this makes C++ function names readable. Different compilers have different mangling styles. The optional demangling style argument can be used to choose an appropriate demangling style for your compiler. The

`style`, if specified, can be 'auto' (the default), 'gnu', 'lucid', 'arm', 'hp', 'edg' or 'gnu-new-abi'.

`--no-demangle` Do not demangle low-level symbol names. This is the default.

`-D, --dynamic` Display the dynamic symbols rather than the normal symbols. This is only meaningful for dynamic objects, such as certain types of shared libraries.

`--defined-only` Display only defined symbols for each object file.

`-e` (ignored)

`-f <format>`

`--format=<format>`
Use the output format <format>, which can be 'bsd', 'sysv', or 'posix'. The default is 'bsd'. Only the first character of <format> is significant; it can be either upper or lower case.

`-g, --extern-only`
Display only external symbols.

`-l, --line-numbers`
For each symbol, use debugging information to try to find a filename and line number. For a defined symbol, look for the line number of the address of the symbol. For an undefined symbol, look for the line number of a relocation entry which refers to the symbol. If line number information can be found, print it after the other symbol information.

`-n, --numeric-sort`
Sort symbols numerically by address.

`-o` Same as -A.

`-p, --no-sort` Do not sort the symbols; print them in the order encountered.

`-P, --portability`
Same as `--format=posix`.

`-r, --reverse-sort`
Reverse the sense of the sort whether numeric or alphabetic.

`-S, --print-size`
Print size of defined symbols for the 'bsd' output format.

`-s, --print-armap`
Include index for symbols from archive members.

`--size-sort` Sort symbols by size. The size is computed as the difference between the value of the symbol and the value of the symbol with the next higher value. The size of the symbol is printed, rather than the value.

`-t, --radix=<radix>`
Use <radix> for printing symbol values. It must be 'd' for decimal, 'o' for octal, or 'x' for hexadecimal.

`--target=<bfdname>`
: Specify an object code format <bfdname> other than your system's default format.

`-u, --undefined-only`
: Display only undefined symbols, those external to each object file.

`-h, --help`
: Display the helpscreen

`-V, --version`
: Display this program's version number

21.4.3 Symbol types

While displaying symbols `tricore-nm` outputs the type of the symbol coded with the following symbols. If lowercase, the symbol is local; if uppercase, the symbol is global (external).

`A`	The symbols value is absolute, and will not be changed by further linking.
`B`	The symbol is in the uninitialized data section (known as BSS).
`C`	The symbol is common. Common symbols are uninitialized data. When linking, multiple common symbols may appear with the same name. If the symbol is defined anywhere, the common symbols are treated as undefined references.
`D`	The symbol is in the initialized data section.
`G`	The symbol is in an initialized data section for small objects. Some object file formats permit more efficient access to small data objects, such as a global int variable as opposed to a large global array.
`I`	The symbol is an indirect reference to another symbol. This is a GNU extension to the a.out object file format which is rarely used.
`N`	The symbol is a debugging symbol.
`R`	The symbol is in a read only data section.
`S`	The symbol is in an uninitialized data section for small objects.
`T`	The symbol is in the text (code) section.
`U`	The symbol is undefined.
`V`	The symbol is a weak object. When a weak defined symbol is linked with a normal defined symbol, the normal defined symbol is used with no error. When a weak undefined symbol is linked and the symbol is not defined, the value of the weak symbol becomes zero with no error.
`W`	The symbol is a weak symbol that has not been specifically tagged as a weak object symbol. When a weak defined symbol is linked with a normal defined symbol, the normal defined symbol is used with no

error. When a weak undefined symbol is linked and the symbol is not defined, the value of the weak symbol becomes zero with no error.

?	The symbol type is unknown, or object file format specific.

21.4.4 Examples

`tricore-nm` is normally used to display the symbols in an object file or an executable. This example shows how to display the symbols of `uart.o`:

`tricore-nm uart.o`

```
         U _install_int_handler
00000000 d asc0 000000d0 T asc0getdata
000000e2 T asc0senddata
00000024 T inituart
         U lock_wdtcon
00000004 d port0
00000104 T rxint
00000004 C uarts
         U unlock_wdtcon
00000004 C val
```

To display the symbols in a library use the name of the library as parameter to `tricore-nm`. To additionally display the archive index use the option `-s`:

`tricore-nm libserial.a -s`

```
Archive index:
val in uart.o
inituart in uart.o
uarts in uart.o
rxint in uart.o
asc0getdata in uart.o
asc0senddlata in uart.o

uart.o: U _install_int_handler
00000000 d asc0 000000d0 T asc0getdata
000000e2 T asc0senddata
00000024 T inituart U lock_wdtcon
00000004 d port0
00000104 T rxint
00000004 C uarts
         U unlock_wdtcon
00000004 C val
```

There are many options to modify the way `tricore-nm` displays the symbol information. This example shows the functionality of three of these options.

The external symbols of the object `uart.o` shall be displayed in descending order of addresses:

`tricore-nm -n -r -g uart.o`

```
00000104 T rxint
000000e2 T asc0senddata
```

```
000000d0 T asc0getdata
00000024 T inituart
00000004 C val
00000004 C uarts
         U unlock_wdtcon
         U lock_wdtcon
         U _install_int_handler}
```

21.5 tricore-objcopy

21.5.1 Description

`tricore-objcopy [options] <infile> <outfile>`

The `tricore-objcopy` utility copies the contents of an object file to another. It can write the destination object file in a format different from that of the source object file. The exact behavior of `tricore-objcopy` is controlled by command-line options.

Note:

`tricore-objcopy` should be able to copy a fully linked file between any two formats. However, copying a relocatable object file between any two formats may not work as expected.

`tricore-objcopy` creates temporary files to do its translations and deletes them afterwards. `tricore-objcopy` uses the BFD, the Berkeley File Descriptor, to do all its translation work. It has access to all the formats described in BFD and thus is able to recognize most formats without being told explicitly.

`tricore-objcopy` can be used to generate S-records by using an output target of 'srec' (e.g., use '-O srec').

`tricore-objcopy` can be used to generate a raw binary file by using an output target of 'binary' (e.g., use `-O binary`). When `tricore-objcopy` generates a raw binary file, it will essentially produce a memory dump of the contents of the input object file. All symbols and relocation information will be discarded. The memory dump will start at the load address of the lowest section copied into the output file.

When generating an S-record or a raw binary file, it may be helpful to use `-S` to remove sections containing debugging information. In some cases `-R` will be useful to remove sections which contain information that is not needed by the binary file.

Note:

`tricore-objcopy` is not able to change the endianness of its input files. If the input format has an endianness, (some formats do not), it can only copy the inputs into file formats that have the same endianness or which have no endianness (e.g. 'srec').

If you do not specify <outfile>, `tricore-objcopy` creates a temporary file and destructively renames the result with the name of <infile>.

Supported targets (BFD-Names) are: elf32-tricore ieee elf32-little elf32-big srec symbolsrec tekhex binary ihex.

21.5.2 Invocation

`-I <bfdname>`

`--input-target <bfdname>`
: Consider the source file's object format to be <bfdname>, rather than attempting to deduce it.

`-O <bfdname>`

`--output-target <bfdname>`
: Create an output file in format <bfdname>. `tricore-objcopy` utility copies the contents of an object file to another.

`-B <arch>`

`--binary-architecture <arch>`
: Useful when transforming a raw binary input file into an object file. In this case the output architecture can be set to <arch>. This option will be ignored if the input file has a known <bfdarch>. You can access this binary data inside a program by referencing the special symbols that are created by the conversion process. These symbols are called:

- _binary_<objfile>_start
- _binary_<objfile>_end
- _binary_<objfile>_size

E.g. you can transform a picture file into an object file and then access it in your code using these symbols.

`-F <bfdname>`

`--target <bfdname>`
: Use <bfdname> as the object format for both the input and the output file; i.e., simply transfer data from source to destination with no translation.

`--debugging` Convert debugging information, if possible. This is not the default because only certain debugging formats are supported, and the conversion process can be time consuming.

`-p --preserve-dates`
: Copy timestamps of the input file and modify/access to the output file.

`-j <name>`

`--only-section <name>`
: Copy only the named section from the input file to the output file. This option may be given more than once.

> **Note:**
>
> Using this option inappropriately may make the output file unusable.

`-R <name>`

`--remove-section <name>`
: Remove any section named <sectionname> from the output file. This option may be given more than once.

> **Note:**
>
> Using this option inappropriately may make the output file unusable.

`-S --strip-all` Do not copy relocation and symbol information from the source file.

`-g --strip-debug`
: Do not copy debugging symbols from the source file.

`--strip-unneeded`
: Strip all symbols that are not needed for relocation processing.

`-N <symbolname>`

`--strip-symbol <symbolname>`
: Do not copy symbol <symbolname> from the source file. This option may be given more than once.

`-K <symbolname>`

`--keep-symbol <symbolname>`
: Copy only symbol <symbolname> from the source file. This option may be given more than once.

`-L <symbolname>`

`-L --localize-symbol <symbolname>`
: Make symbol <symbolname> local to the file, so that it is not visible externally. This option may be given more than once.

`-G <symbolname>`

`--keep-global-symbol <symbolname>`
: Keep only symbol <symbolname> global. Make all other symbols local to the file, so that they are not visible externally. This option may be given more than once.

`-W <symbolname>`

`--weaken-symbol <symbolname>`
: Make symbol <symbolname> weak. This option may be given more than once.

`-x --discard-all`
: Remove all non-global symbols.

`-X --discard-locals`
: Remove any compiler-generated symbols.

`-i <number>`

`--interleave <number>`
: Only copy one out of every <number> bytes. Select which byte to copy with the `-b` or `--byte` option. `tricore-objcopy` ignores this option if you do not specify either `-b` or `--byte`. The default is 4.

`-b <number>`

`--byte <number>`
: Keep only every <number>th byte of the input file (header data is not affected). <number> can be in the range from 0 to <number>-1, where <number> is given by the `-i` or `--interleave` option, or the default of 4. This option is useful for creating files to program ROM. It is typically used with an `srec` output target.

`--gap-fill <val>`
: Fill gaps between sections with <val>. This operation applies to the load address (LMA) of the sections. It is done by increasing the size of the section with the lower address, and filling in the extra space created with <val>.

`--pad-to <addr>`
: Pad the output file up to the load address <addr>. This is done by increasing the size of the last section. The extra space is filled in with the value specified by `--gap-fill` (default zero).

`--set-start <addr>`
: Set the start address of the new file to <addr>. Not all object file formats support setting the start address.

`--adjust-start <incr>`

`--change-start <incr>`
: Change the start address by adding <incr>. Not all object file formats support setting the start address.

`--adjust-vma <incr>`

`--change-addresses <incr>`
: Change the VMA and LMA addresses of all sections, as well as the start address, by adding <incr>. Some object file formats do not permit section addresses to be changed arbitrarily.

> **Note:**
>
> This does not relocate the sections; if the program expects sections to be loaded at a certain address, and this option is used to change the sections such that they are loaded at a different address, the program may fail.

`--adjust-section-vma <section>=|+|-<val>`

`--change-section-address <section>=|+|-<val>`
: The VMA address and the LMA address of the named <section> are set or changed both . If '=' is used, the section address is set to <val>. Otherwise, <val> is added to or subtracted from the section address. See the comments under `--change-addresses`, above. If <section> does not exist in the input file, a warning will be issued, unless `--no-change-warnings` is used.

`--change-section-lma <section>=|+|-<val>`
: Set or change the LMA address of the named <section>. The LMA address is the address where the section will be loaded into memory at program load time. Normally this is the same as the VMA address, which is the address of the section at program run time, but on some systems, especially those where a program is held in ROM, the two can be different. If '=' is used, the section address is set to <val>. Otherwise, <val> is added to or subtracted from the section address. See the comments under `--change-addresses`, above. If <section> does not exist in the input file, a warning will be issued, unless `--no-change-warnings` is used.

`--change-section-vma <section>=|+|-<val>`
: Set or change the VMA address of the named <section>. The VMA address is the address where the section will be located once the program has started executing. Normally this is the same as the LMA address, which is the address where the section will be loaded into memory, but on some systems, especially those where a program is held in ROM, the two can be different. If '=' is used, the section address is set to <val>. Otherwise, <val> is added to or subtracted from the section address. See the comments under `--change-addresses`, above. If <section> does not exist in the input file, a warning will be issued, unless `--no-change-warnings` is used.

`--[no-]change-warnings`

`--[no-]adjust-warnings`
: If `--change-section-address` or `--change-section-lma` or the option `--change-section-vma` is used, and the named section does not exist, a warning is issued. If these options are used in the 'no'-form (`--no-change-warnings` or `--no-adjust-warnings`) these warnings are suppressed. The warnings are enabled by default.

`--set-section-flags <section>=<flags>`
: Set the flags for the named section. The <flags> argument is a comma separated string of flag names. The recognized names are 'alloc', 'contents', 'load', 'noload', 'readonly', 'code', 'data', 'rom', 'share', and 'debug'. You can set the 'contents' flag for a section which does not have contents, but it is not meaningful to clear the 'contents' flag of a section which does have contents–just remove the section instead. Not all flags are meaningful for all object file formats.

`--add-section <sectionname>=<filename>`
: Add a new section named <sectionname> while copying the file. The contents of the new section are taken from the file <filename>. The size of the section will be the size of the file. This option only works on file formats which can support sections with arbitrary names.

`--rename-section <old>=<new>[,<flags>]`
: Rename a section from <old> to <new>, optionally changing the section's flags to <flags> in the process. This has the advantage over using a linker script to perform the rename in that the output stays as an object file and does not become a linked executable. This option is particularly helpful when the input format is binary, since this will always create a section called `.data`.

`--change-leading-char`
: Some object file formats use special characters at the start of symbols. The most common such character is underscore, which compilers often add before every symbol. This option tells `tricore-objcopy` to change the leading character of every symbol when it converts between object file formats. If the object file formats use the same leading character, this option has no effect. Otherwise, it will add a character, or remove a character, or change a character, as appropriate.

`--remove-leading-char`
: If the first character of a global symbol is a special symbol leading character used by the object file format, remove the character. The most common symbol leading character is underscore. This option will remove a leading underscore from all global symbols. This can be useful if you want to link together objects of different file formats with different conventions for symbol names. This is different from `--change-leading-char` because it always changes the symbol name when appropriate, regardless of the object file format of the output file.

`--redefine-sym <old>=<new>`
: Redefine symbol name <old> to <new>. This can be useful when one is trying link two things together for which you have no source, and there are name collisions.

`--srec-len <number>`
: Restrict the length of generated Srecords. This length covers both address, data and crc fields.

`--srec-forceS3` Restrict the type of generated Srecords to S3. Avoid generation of S1/S2 records, creating S3-only record format.

`--strip-symbols <filename>`
Apply `--strip-symbol` option to each symbol listed in the file <filename>. <filename> is simply a flat file, with one symbol name per line. Line comments may be introduced by the hash character. This option may be given more than once. `-N` for all symbols listed in <filename>.

`--keep-symbols <filename>`
Apply `--keep-symbol` option to each symbol listed in the file <filename>. <filename> is simply a flat file, with one symbol name per line. Line comments may be introduced by the hash character. This option may be given more than once. `-K` for all symbols listed in <filename>.

`--localize-symbols <filename>`
Apply `--localize-symbol` option to each symbol listed in the file <filename>. <filename> is simply a flat file, with one symbol name per line. Line comments may be introduced by the hash character. This option may be given more than once. `-L` for all symbols listed in <filename>.

`--keep-global-symbols <filename>`
Apply `--keep-global-symbol` option to each symbol listed in the file <filename>. <filename> is simply a flat file, with one symbol name per line. Line comments may be introduced by the hash character. This option may be given more than once. `-G` for all symbols listed in <filename>.

`--weaken-symbols <filename>`
`-W` for all symbols listed in <filename>. Apply `--weaken-symbol` option to each symbol listed in the file <filename>. <filename> is simply a flat file, with one symbol name per line. Line comments may be introduced by the hash character. This option may be given more than once.

`--alt-machine-code <index>`
If the output architecture has alternate machine codes, use the <index>th code instead of the default one. This is useful in case a machine is assigned an official code and the tool-chain adopts the new code, but other applications still depend on the original code being used.

`-v --verbose` Verbose output: list all object files modified.

`-V --version` Show the version number of `tricore-objcopy`.

`-h --help` Show a summary of the options to `tricore-objcopy`.

`--weaken` Change all global symbols in the file to be weak. This can be useful when building an object which will be linked against other objects using the `-R` option to the linker. This option is only effective when using an object file format which supports weak symbols.

21.5.3 Examples

`tricore-objcopy` is used to change the format of the object file or to deal with sections in object files.

To create a file in the IntelHex output type from an ELF-object type:

```
tricore-objcopy -O ihex Input.elf Output.hex
```

If you want to move the code of an ELF-file form its original address by <value> use:

```
tricore-objcopy --adjust-vma=±<value> Input.elf Output.elf
```

21.6 tricore-objdump

21.6.1 Description

```
tricore-objdump [options] <objfile>
```

The `tricore-objdump` utility can be used to extract information from object files, static libraries, and shared libraries and then list information in a human-readable form. It can be used to dump the information from several different formats of object files. This information is mostly useful to programmers who are working on the compilation tools, as opposed to programmers who just want their program to compile and work.

21.6.2 Invocation

At least one of the following switches must be given

`-a, --archive-headers`
: If any of the <objfile> files are archives, display the archive header information (in a format similar to 'ls -l'). Besides the information you could list with `tricore-ar tv`, `tricore-objdump -a` shows the object file format of each archive member.

`-d, --disassemble`
: Display the assembler mnemonics for the machine instructions from <objfile>. This option only disassembles those sections which are expected to contain instructions.

Note:

If you have compiled your program with `-mcode-pic` the compiler writes the original address of each function before its first instruction. If this function is disassembled, random mnemonics are displayed in the disassembly at the start of the function. This does not affect the correct working of the program!

`-D, --disassemble-all`
: Like `-d`, but disassemble the contents of all sections, not just those expected to contain instructions.

`-f, --file-headers`
: Display summary information from the overall header of each of the <objfile> files.

`-g, --debugging`
: Display debug information in object file. This is a generic option and not available for TriCore-targets, since it can not handle DWARF-2 debug information.

`-h, --section-headers`
: Display summary information from the section headers of the object file.

`-H, --help`
: Print a summary of the options of `tricore-objdump` and exit.

`-i, --info`
: List object formats and architectures supported.

`-p, --private-headers`
: File header contents that are specific to the object format.

`-r, --reloc`
: Print the relocation entries of the file. If used with `-d` or `-D`, the relocations are printed interspersed with the disassembly.

`-R, --dynamic-reloc`
: Print the dynamic relocation entries of the file. This is only meaningful for dynamic objects, such as certain types of shared libraries.

`-s, --full-contents`
: Display the full contents of all sections requested.

`-S, --source`
: Display source code intermixed with disassembly, if possible. Implies `-d`.

`-t, --syms`
: Print the symbol table entries of the file. This is similar to the information provided by the `tricore-nm` program.

`-T, --dynamic-syms`
: Print the dynamic symbol table entries of the file. This is only meaningful for dynamic objects, such as certain types of shared libraries. This is similar to the information provided by the 'nm' program when given the `-D` (`--dynamic`) option.

`-v, --version`
: Print the version number of `tricore-objdump` and exit.

`-x, --all-headers`
: Display all available header information, including the symbol table and relocation entries. Using `-x` is equivalent to specifying all of `-a -f -h -r -t`.

The following switches are optional:

`--adjust-vma=<offset>`
: Add <offset> to all displayed section addresses. This is useful if the section addresses do not correspond to the symbol table, which can happen when putting sections at particular addresses when using a format which can not represent section addresses, such as a.out.

`-b, --target=<bfdname>`
: Specify that the object-code format for the object files is <bfdname>. This option may not be necessary; `tricore-objdump` can automatically recognize many formats.

`-C [<style>]`

`--demangle[=<style>]`
: Decode (demangle) low-level symbol names into user-level names. Besides removing any initial underscore prepended by the system, this makes C++ function names readable. Different compilers have different mangling styles. The optional demangling style argument can be used to choose an appropriate demangling style for your compiler. The <style>, if specified, can be 'auto', 'gnu','lucid', 'arm', 'hp', 'edg', or 'gnu-new-abi'.

`-EB --endian=big`
: Assume big endian format when disassembling. Specifies the endianness of the object files. This only affects disassembly. This can be useful when disassembling a file format which does not describe endianness information, such as S-records.

`-EL --endian=little`
: Assume little endian format when disassembling.

`--file-start-context`
: Specify that when displaying interlisted source code/disassembly (assumes `-S`) from a file that has not yet been displayed, extend the context to the start of the file.

`-j <name>`

`--section=<name>`
: Display information only for section <name>.

`-l, --line-numbers`
: Label the display (using debugging information) with the filename and source line numbers corresponding to the object code or relocs shown. Only useful with `-d`, `-D`, or `-r`.

`-m <machine>`

`--architecture=<machine>`
: Specify the architecture to use when disassembling object files. This can be useful when disassembling object files which do not describe architecture information, such as S-records. You can list the available architectures with the `-i` option.

`-M, --disassembler-options=<option>`
: Pass target specific information to the disassembler.

Note:

Only supported on some targets.

`--prefix-addresses`
: When disassembling, print the complete address on each line.

`--[no-]show-raw-insn`
: Display hexadecimal opcodes along with the mnemonic assembly language instruction. This is the default except if `--prefix-addresses` is used. Using `--no-show-raw-insn` does not print the instruction bytes, when disassembling instructions.

`--start-address=<addr>`
: Only process data who's address is ≥ <addr>. This affects the output of the options `-d`, `-r` and `-s`.

`--stop-address=<addr>`
: Only process data who's address is ≥ <addr>. This affects the output of the options `-d`, `-r` and `-s`.

`-w, --wide`
: Format some lines for output devices that have more than 80 columns. Also do not truncate symbol names when they are displayed.

`-z, --disassemble-zeroes`
: Normally the disassembly output will skip blocks of zeroes. This option directs the disassembler to disassemble those blocks, just like any other data.

21.6.3 Examples

To view the sections in an object file or an executable type: `tricore-objdump -h main.o`

```
main.o: file format elf32-tricore

Sections:
Idx Name Size VMA LMA File off Algn
  0 .text 00000066 00000000 00000000 00000034 2**1
              CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data 00000000 00000000 00000000 000000a0 2**3
              CONTENTS, ALLOC, LOAD, DATA
  2 .bss 00000000 00000000 00000000 000000a0 2**3
              ALLOC
  3 .debug_abbrev 0000013f 00000000 00000000 000000a0 2**0
              CONTENTS, READONLY, DEBUGGING
  4 .debug_info 00000a6f 00000000 00000000 000001df 2**0
              CONTENTS, RELOC, READONLY, DEBUGGING
  5 .debug_line 000001e1 00000000 00000000 00000c4e 2**0
              CONTENTS, RELOC, READONLY, DEBUGGING
  6 .rodata 0000002c 00000000 00000000 00000e30 2**2
              CONTENTS, ALLOC, LOAD, READONLY, DATA
  7 .debug_frame 00000024 00000000 00000000 00000e5c 2**2
              CONTENTS, RELOC, READONLY, DEBUGGING
  8 .debug_pubnames 0000002d 00000000 00000000 00000e80 2**0
              CONTENTS, RELOC, READONLY, DEBUGGING
  9 .debug_aranges 00000020 00000000 00000000 00000ead 2**0
              CONTENTS, RELOC, READONLY, DEBUGGING
 10 .debug_str 00000012 00000000 00000000 00000ecd 2**0
              CONTENTS, READONLY, DEBUGGING
```

To disassemble the contents of the section `.text` use:

```
tricore-objdump -d -j .text main.o
```

If the option `-j` is not given, `tricore-objdump` will disassemble all sections that contain executable code. This is normally only the section `.text`.

21.7 tricore-ranlib

21.7.1 Description

```
tricore-ranlib <archive>
```

`tricore-ranlib` generates an index to the contents of an archive and stores it in the archive. The index lists each symbol defined by a member of an archive that is a relocatable object file. You may use `tricore-nm -s` or `tricore-nm --print-armap` to list this index.

An archive with such an index speeds up linking to the library and allows routines in the library to call each other without regard to their placement in the archive. Running `tricore-ranlib` is equivalent to executing `tricore-ar -s <archive>`.

21.7.2 Invocation

`-v, -V, --version`
Show the version number of `tricore-ranlib`.

21.7.3 Example

This example generates a archive index to the library file `libserial.a`:

```
tricore-ranlib libserial.a
```

21.8 tricore-readelf

21.8.1 Description

```
tricore-readelf [options] <elffile> ...
```

`tricore-readelf` displays information about one or more ELF format object files. The options control what particular information to display.

<elffile> ... are the object files to be examined.

Note:
At the moment, `tricore-readelf` does not support examining archives, nor does it support examining 64 bit ELF files.

21.8.2 Invocation

`--debug-dump[=line,=info,=abbrev,=pubnames,=ranges,=macro,=frames,=str,=loc]`
: Same as `-w`.

`-a --all`
: Equivalent to specifying `--file-header`, `--program-headers`, `--sections`, `--symbols`, `--relocs`, `--dynamic`, `--notes` and `--version-info`.

`-A --arch-specific`
: Display architecture specific information (if any).

`-d --dynamic`
: Displays the contents of the file's dynamic section, if it has one.

`-D --use-dynamic`
: When displaying symbols, this option makes `tricore-readelf` use the symbol table in the file's dynamic section, rather than the one in the symbols section.

`-e --headers`
: Display all the headers in the file. Equivalent to `-h -l -S`.

`-h --file-header`
: Displays the information contained in the ELF header at the start of the file.

`-H --help`
: Display the command line options understood by `tricore-readelf`.

`-I --histogram`
: Display a histogram of bucket list lengths when displaying the contents of the symbol tables.

`-l --program-headers`
: Display the program headers

`-n --notes`
: Displays the contents of the notes segment, if it exists.

`-r --relocs`
: Displays the contents of the file's relocation section, if it has one.

`-S --section-headers`
: Display the sections' header

`-s --syms`
: Displays the entries in symbol table section of the file, if it has one.

`--sections`
: An alias for `--section-headers`.

`--segments`
: An alias for `--program-headers`. Displays the information contained in the file's segment headers, if it has any.

`--symbols`
: An alias for `--syms`.

`-v --version`
: Display the version number of `tricore-readelf`.

`-V --version-info`
: Displays the contents of the version sections in the file, it they exist.

`-w[liaprmfFso]`
: Displays the contents of the debug sections in the file, if any are present. If one of the optional letters or words follows the switch then only data found in those specific sections will be dumped.

`-W --wide` Allow output width to exceed 80 characters.

`-x <section>`

`--hex-dump=<section>`
Displays the contents of the indicated <section> as a hexadecimal dump.

21.8.3 Example

To view the symbol table of an object use the option `-s`:

```
tricore-readelf -s main.o
```

```
Symbol table '.symtab' contains 16 entries:
   Num: Value Size Type Bind Vis Ndx Name
     0: 00000000 0 NOTYPE LOCAL DEFAULT UND
     1: 00000000 0 FILE LOCAL DEFAULT ABS main.c
     2: 00000000 0 SECTION LOCAL DEFAULT 1
     3: 00000000 0 SECTION LOCAL DEFAULT 3
     4: 00000000 0 SECTION LOCAL DEFAULT 4
     5: 00000000 0 SECTION LOCAL DEFAULT 5
     6: 00000000 112 FUNC GLOBAL DEFAULT 1 main
     7: 00000000 0 NOTYPE GLOBAL DEFAULT UND __main
     8: 00000004 4 OBJECT GLOBAL DEFAULT COM foo
     9: 00000001 65535 OBJECT GLOBAL DEFAULT COM foo1
    10: 00000000 0 NOTYPE GLOBAL DEFAULT UND read
    11: 00000001 65535 OBJECT GLOBAL DEFAULT COM foo2
    12: 00000000 0 NOTYPE GLOBAL DEFAULT UND unlink
    13: 00000000 0 NOTYPE GLOBAL DEFAULT UND write
    14: 00000000 0 NOTYPE GLOBAL DEFAULT UND close
    15: 00000000 0 NOTYPE GLOBAL DEFAULT UND printf
```

To output the sections of an object or an executable plus some information on the file use the option `-e`:

```
tricore-readelf -e main
```

```
ELF Header:
  Magic: 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class: ELF32
  Data: 2's complement, little endian
  Version: 1 (current)
  OS/ABI: UNIX - System V
  ABI Version: 0
  Type: EXEC (Executable file)
  Machine: Siemens Tricore
  Version: 0x1
  Entry point address: 0xa0000000
  Start of program headers: 52 (bytes into file)
  Start of section headers: 67476 (bytes into file)
  Flags: 0x2
  Size of this header: 52 (bytes)
  Size of program headers: 32 (bytes)
  Number of program headers: 3
  Size of section headers: 40 (bytes)
  Number of section headers: 16
  Section header string table index: 15

Section Headers:
  [Nr] Name Type Addr Off Size ES Flg Lk Inf Al
  [ 0] NULL 00000000 000000 000000 00 0 0 0
  [ 1] .startup PROGBITS a0000000 004000 0001ec 00 AX 0 0 2
  [ 2] .zbss PROGBITS a00001f0 010720 000000 00 W 0 0 1
  [ 3] .zdata PROGBITS a00001f0 010720 000000 00 W 0 0 1
  [ 4] .text PROGBITS a00001f0 0041f0 009e10 00 AX 0 0 8192
  [ 5] .rodata PROGBITS a000a000 00e000 0004f8 00 A 0 0 4
  [ 6] .eh_frame PROGBITS a000a4f8 010720 000000 00 W 0 0 1
  [ 7] .ctors PROGBITS a000a4f8 00e4f8 00000c 00 A 0 0 1
  [ 8] .dtors PROGBITS a000a504 00e504 000008 00 WA 0 0 1
  [ 9] .pcptext PROGBITS f0020000 010720 000000 00 W 0 0 1
  [10] .pcpdata PROGBITS f0010000 010720 000000 00 W 0 0 1
  [11] .data PROGBITS a0080000 010000 000720 00 WA 0 0 8
  [12] .sdata PROGBITS a0080720 010720 000000 00 WA 0 0 8
  [13] .sbss PROGBITS a0080720 010720 000000 00 W 0 0 1
  [14] .bss NOBITS a0080720 010720 0208b8 00 WA 0 0 8
  [15] .shstrtab STRTAB 00000000 010720 000071 00 0 0 1

Key to Flags:
 W (write), A (alloc), X (execute), M (merge), S (strings)
 I (info), L (link order), G (group), x (unknown)
 O (extra OS processing required) o (OS specific), p (processor specific)

Program Headers:
  Type Offset VirtAddr PhysAddr FileSiz MemSiz Flg Align
  LOAD 0x004000 0xa0000000 0xa0000000 0x0a50c 0x0a50c RWE 0x4000
  LOAD 0x010000 0xa0080000 0xa000a50c 0x00720 0x00720 RW 0x4000
  LOAD 0x010720 0xa0080720 0xa0080720 0x00000 0x208b8 RW 0x4000

  Section to Segment mapping:
   Segment Sections...
    00 .startup .text .rodata .ctors .dtors
    01 .data
    02 .bss
```

21.9 tricore-size

21.9.1 Description

`tricore-size [options] [<objfile> <objfile> ... ]`

The `tricore-size` utility lists the section sizes and the total size for each of the object or archive files <objfile> in its argument list.

By default, one line of output is generated for each object file or each module in an archive.

<objfile> . . . are the object files to be examined. If none are specified, the file `a.out` will be used.

21.9.2 Invocation

`-A`	Equal to `--format=sysv`

`-B`	Equal to `--format=berkeley`
`--format=<format>`	Select output style. The output style may be one of `sysv` or `berkeley`. Default is `berkeley`.
`-o`	Equal to `--radix=8`.
`-d`	Equal to `--radix=10`.
`-x`	Equal to `--radix=16`.
`--radix=<number>`	Display numbers in octal, decimal or hex. In `--radix=<number>`, only the three values 8, 10 or 16 are supported. The total size is always given in two radices; decimal and hexadecimal for `-d` or `-x` output, or octal and hexadecimal if you're using `-o`.
`-t --totals`	Show totals of all objects listed (Berkeley format listing mode only).
`--target=<bfdname>`	Specify that the object-code format for <objfile> is <bfdname> (binary file format). This option may not be necessary; `tricore-size` can automatically recognize many formats.
`-h --help`	Show a summary of acceptable arguments and options.
`-v --version`	Display the version number of `tricore-size`.

21.9.3 Example

`tricore-size` shows the sizes of the sections of all object in a library file. The sizes are displayed as hexadecimal values and the sums of the sizes for all objects are output:

```
tricore-size -x -t libserial.a
```

```
   text data bss dec hex filename
  0x112 0x8 0x0 282 11a uart.o (ex libserial.a)
  0x112 0x8 0x0 282 11a (TOTALS)
```

21.10 tricore-strip

21.10.1 Description

```
tricore-strip [options] <objfile>
```

The `tricore-strip` utility removes the debugging symbol table information from the object file or files named on the command line. The object file can be static library, a shared library, or an object file produced by the compiler. Depending on how much debugging information has been included in the file, stripping can dramatically reduce the size of the file.

Note:

The `tricore-strip` utility replaces the existing file with the stripped version, so if you want to be able to restore the original unstripped versions, you will need to save the files before stripping them or use the `-o` option to produce the output in a different file.

21.10.2 Invocation

`--strip-unneeded`
Remove all symbols that are not needed for relocation processing.

`-F <bfdname>`
`--target=<bfdname>`
Treat the original <objfile> as a file with the object code format <bfdname>, and rewrite it in the same format.

`-g -S -d --strip-debug`
Remove debugging symbols only.

`-h --help`
Show a summary of the options to `tricore-strip` and exit.

`-I --input-target=<bfdname>`
Treat the original <objfile> as a file with the object code format <bfdname>.

`-K <symbolname>`

`--keep-symbol=<symbolname>`
Keep only symbol <symbolname> from the source file. This option may be given more than once.

`-N <symbolname>`

`--strip-symbol=<symbolname>`
Do not copy symbol <symbolname>. This option may be given more than once, and may be combined with strip options other than `-K`.

`-o <outfile>`
Instead of overwriting the original file, the output is written to a new file named <outfile>. Using this option limits the command to operate on a single file.

`-O <bfdname>`
`--output-target=<bfdname>`
Create an output file in format <bfdname>

`-p --preserve-dates`
Copy modified/access timestamps to the output.

`-R <sectionname>`

`--remove-section=<sectionname>`
Remove any section named <sectionname> from the output file. This option may be given more than once.

Note:

Using this option inappropriately may make the output file unusable.

`-s --strip-all` Remove all symbol and relocation information.

`-v --verbose` Verbose output: list all object files modified. In the case of archives, `tricore-strip -v` lists all members of the archive.

`-V --version` Display this program's version number.

`-x --discard-all`
Remove all non-global symbols.

`-X --discard-locals`
Remove any compiler-generated symbols.

21.10.3 Example

As an example, the following command will strip all debugging information from the file `main.o`

```
tricore-strip main.o
```

21.11 tricore-strings

21.11.1 Description

```
tricore-strings [<options>] [<file(s)>]
```

For each <file> given, `tricore-strings` prints the printable character sequences that are at least 4 characters long (or the number given with the options below) and are followed by an unprintable character. By default, it only prints the strings from the initialized and loaded sections of object files. For other types of files, it prints the strings from the whole file. `tricore-strings` is mainly useful for determining the contents of non-text files.

21.11.2 Invocation

`-a --all` Scan the entire file, not just the data section.

`-e <encoding>`

`--encoding=<encoding>`
Select the character encoding of the strings that are to be found. Possible values for <encoding> are: 's' = single-byte characters (ASCII, ISO 8859, etc., default), 'b' = 16-bit Bigendian, 'l' = 16-bit Littleendian, 'B' = 32-bit Bigendian, 'L' = 32-bit Littleendian. Useful for finding wide character strings.

`-f --print-file-name`
Print the name of the file before each string.

`-h --help` Print a summary of the program usage on the standard output and exit.

`-<min-len>`

`-n <min-len>`

`--bytes=<min-len>`
Print sequences of characters that are at least <min-len> characters long, instead of the default 4.

`-o` An alias for `--radix=o` (See `-t`).

`-t <radix>`

`--radix=<radix>`
Print the offset within the file before each string. The single character argument specifies the radix of the offset 'o' for octal, 'x' for hexadecimal, or 'd' for decimal.

`-T <bfdname>`

`--target=<bfdname>`
Specify the binary file format to be <bfdname>.

`-v --version` Print the program version number on the standard output and exit.

21.11.3 Examples

To display all strings in an object file that are longer than 4 characters `tricore-strings` is used:

```
tricore-strings main.o
```

```
hello world
gcc rules
foobar
```

To only display the strings longer than 7 characters and to additionally display the offset of the string within the object in hexadecimal format use the options `-n` and `-t`:

```
tricore-strings main.o -n 7 -t x
```

```
    e08 hello world
    e18 gcc rules
```

22 tricore-gcov

`-b, --branch-probabilities`
: Write branch frequencies to the output file, and write branch summary info to the standard output. This option allows you to see how often each branch in your program was taken.

`-c, --branch-counts`
: Write branch frequencies as the number of branches taken, rather than the percentage of branches taken.

`-f, --function-summaries`
: Output summaries for each function.

`-h, --help`
: Display help on the standard output, and exit without doing any further processing.

`-l, --long-file-names`
: Create long file names for included source files. For example, if the header file `x.h` contains code, and was included in the file `a.c`, then running `tricore-gcov` on the file `a.c` will produce an output file called `a.c.x.h.gcov` instead of `x.h.gcov`. This can be useful if `x.h` is included in multiple source files.

`-n, --no-output`
: Do not create an output file.

`-o, --object-directory OBJDIR`
: The directory where the object files live. `tricore-gcov` will search for `.bb`, `.bbg`, and `.da` files in this directory.

`-v, --version`
: Display version number (on the standard output), and exit without doing any further processing.

`tricore-gcov` is a tool you can use in conjunction with `tricore-gcc` to test code coverage in your programs.

22.1 Introduction to tricore-gcov

`tricore-gcov` is a test coverage program. Use it in concert with `tricore-gcc` to analyze your programs to help create more efficient, faster running code. You can use `tricore-gcov` as a profiling tool to help discover where your optimization efforts will best affect your code.

Note:

For using `tricore-gcov -fprofile-arcs`, `-ftest-coverage` must be set for `tricore-gcc`. Additional use the compiler option `-g` to generate debug information.

Profiling tools help you analyze your code's performance. Using `tricore-gcov`, you can find out some basic performance statistics, such as:

- how often each line of code executes
- what lines of code are actually executed
- how much computing time each section of code uses

Once you know these things about how your code works when compiled, you can look at each module to see which modules should be optimized. `tricore-gcov` helps you determine where to work on optimization.

Software developers also use coverage testing in concert with testsuites, to make sure software is actually good enough for a release. Testsuites can verify that a program works as expected; a coverage program tests to see how much of the program is exercised by the testsuite. Developers can then determine what kinds of test cases need to be added to the testsuites to create both better testing and a better final product.

You should compile your code without optimization if you plan to use `tricore-gcov` because the optimization, by combining some lines of code into one function, may not give you as much information as you need to look for 'hot spots' where the code is using a great deal of computer time. Likewise, because `tricore-gcov` accumulates statistics by line (at the lowest resolution), it works best with a programming style that places only one statement on each line. If you use complicated macros that expand to loops or to other control structures, the statistics are less helpful—they only report on the line where the macro call appears. If your complex macros behave like functions, you can replace them with inline functions to solve this problem.

`tricore-gcov` creates a logfile called `<sourcefile>.gcov` which indicates how many times each line of a source file `<sourcefile>.c` has executed.

`tricore-gcov` works only on code compiled with `tricore-gcc`. It is not compatible with any other profiling or test coverage mechanism.

22.2 Invoking tricore-gcov

```
tricore-gcov [options] <sourcefile>
```

Note:

When using `tricore-gcov`, you must first compile your program with two special `tricore-gcc` options: `-fprofile-arcs -ftest-coverage`. Additional use the compiler option `-g` to generate debug information.

This tells the compiler to generate additional information needed by `tricore-gcov` (basically a flow graph of the program) and also includes additional code in the object files for generating the extra profiling information needed by `tricore-gcov`. These additional files are placed in the directory where the source code is located.

Running the program will cause profile output to be generated. For each source file compiled with `-fprofile-arcs`, an accompanying `.da` file will be placed in the source directory. This feature is only available with some kind of stdio on the target platform. I.e. through simulated IO within the debugger and the included library `libos.a`.

Running `tricore-gcov` with your program's source file names as arguments will now produce a listing of the code along with frequency of execution for each line. For example, if your program is called `tmp.c`, this is what you see when you use the basic `tricore-gcov` facility:

```
tricore-gcc -fprofile-arcs -ftest-coverage tmp.c
```

```
tricore-gcov tmp.c
```

Note:

Use exit(0); in main, so the `.da` can be generated e.g. by the debugger at runtime.

The file `tmp.c.gcov` contains output `tricore-gcov`.

Here is a sample:

```
            main()
            {
       1      int i, total;

       1      total = 0;

      11      for (i = 0; i < 10; i++)
      10        total += i;

       1      if (total != 45)
  ######        printf ("Failure\n");
              else
       1        printf ("Success\n");
       1    }
```

When you use the `-b` option, your output looks like this:

```
tricore-gcov -b tmp.c
```

```
 87.50% of 8 source lines executed in file tmp.c
 80.00% of 5 branches executed in file tmp.c
 80.00% of 5 branches taken at least once in file tmp.c
 50.00% of 2 calls executed in file tmp.c
Creating tmp.c.gcov.
```

Here is a sample of a resulting `tmp.c.gcov` file:

```
                  main()
                  {
             1      int i, total;

             1      total = 0;
```

```
          11      for (i = 0; i < 10; i++)
branch 0 taken = 91%
branch 1 taken = 100%
branch 2 taken = 100%
          10        total += i;

           1      if (total != 45)
branch 0 taken = 100%
      ######        printf ("Failure\n");
call 0 never executed
branch 1 never executed
                  else
           1        printf ("Success\n");
call 0 returns = 100%
           1    }
```

For each basic block, a line is printed after the last line of the basic block describing the branch or call that ends the basic block. There can be multiple branches and calls listed for a single source line if there are multiple basic blocks that end on that line. In this case, the branches and calls are each given a number. There is no simple way to map these branches and calls back to source constructs. In general, though, the lowest numbered branch or call will correspond to the leftmost construct on the source line.

For a branch, if it was executed at least once, then a percentage indicating the number of times the branch was taken divided by the number of times the branch was executed will be printed. Otherwise, the message "never executed" is printed.

For a call, if it was executed at least once, then a percentage indicating the number of times the call returned divided by the number of times the call was executed will be printed. This will usually be 100%, but may be less for functions call `exit` or `longjmp`, and thus may not return every time they are called.

The execution counts are cumulative. If the example program were executed again without removing the `.da` file, the count for the number of times each line in the source was executed would be added to the results of the previous run(s). This is potentially useful in several ways. For example, it could be used to accumulate data over a number of program runs as part of a test verification suite, or to provide more accurate long-term information over a large number of program runs.

The data in the `.da` files is saved immediately before the program exits. For each source file compiled with `-fprofile-arcs`, the profiling code first attempts to read in an existing `.da` file; if the file doesn't match the executable (differing number of basic block counts) it will ignore the contents of the file. It then adds in the new execution counts and finally writes the data to the file.

22.3 Using tricore-gcov with tricore-gcc optimization

If you plan to use `tricore-gcov` to help optimize your code, you must first compile your program with two special `tricore-gcc` options `-fprofile-arcs -ftest-coverage`. Aside from that, you can use any other `tricore-gcc` options; but if you want to prove that every single line in your program was executed, you should not compile with optimization at the same time. On some machines the optimizer can eliminate some simple code lines by combining them with other lines. For example, code like this:

```
if (a != b)
  c = 1;
else
  c = 0;
```

can be compiled into one instruction on some machines. In this case, there is no way for `tricore-gcov` to calculate separate execution counts for each line because there isn't separate code for each line. Hence the `tricore-gcov` output looks like this if you compiled the program with optimization:

```
      100  if (a != b)
      100    c = 1;
      100  else
      100    c = 0;
```

The output shows that this block of code, combined by optimization, executed 100 times. In one sense this result is correct, because there was only one instruction representing all four of these lines. However, the output does not indicate how many times the result was 0 and how many times the result was 1.

22.4 Brief description of tricore-gcov data files

`tricore-gcov` uses three files for doing profiling. The names of these files are derived from the original *source* file by substituting the file suffix with either `.bb`, `.bbg`, or `.da`. All of these files are placed in the same directory as the source file, and contain data stored in a platform-independent method.

The `.bb` and `.bbg` files are generated when the source file is compiled with the option `-ftest-coverage` of `tricore-gcc`. The `.bb` file contains a list of source files (including headers), functions within those files, and line numbers corresponding to each basic block in the source file.

The `.bb` file format consists of several lists of 4-byte integers which correspond to the line numbers of each basic block in the file. Each list is terminated by a line number of 0. A line number of -1 is used to designate that the source file name (padded to a 4-byte boundary and followed by another -1) follows. In addition, a line number of -2 is used to designate that the name of a function (also padded to a 4-byte boundary and followed by a -2) follows.

The `.bbg` file is used to reconstruct the program flow graph for the source file. It contains a list of the program flow arcs (possible branches taken from one basic block to another) for each function which, in combination with the `.bb` file, enables `tricore-gcov` to reconstruct the program flow.

In the `.bbg` file, the format is:

```
        number of basic blocks for function #0 (4-byte number)
        total number of arcs for function #0 (4-byte number)
        count of arcs in basic block #0 (4-byte number)
```

```
destination basic block of arc #0 (4-byte number)
flag bits (4-byte number)
destination basic block of arc #1 (4-byte number)
flag bits (4-byte number)
...
destination basic block of arc #N (4-byte number)
flag bits (4-byte number)
count of arcs in basic block #1 (4-byte number)
destination basic block of arc #0 (4-byte number)
flag bits (4-byte number)
...
```

A -1 (stored as a 4-byte number) is used to separate each function's list of basic blocks, and to verify that the file has been read correctly.

The `.da` file is generated when a program containing object files built with the option `-fprofile-arcs` of `tricore-gcc`. A separate `.da` file is created for each source file compiled with this option, and the name of the `.da` file is stored as an absolute pathname in the resulting object file. This path name is derived from the source file name by substituting a `.da` suffix.

The format of the `.da` file is fairly simple. The first 8-byte number is the number of counts in the file, followed by the counts (stored as 8-byte numbers). Each count corresponds to the number of times each arc in the program is executed. The counts are cumulative; each time the program is executed, it attempts to combine the existing `.da` files with the new counts for this invocation of the program. It ignores the contents of any `.da` files whose number of arcs doesn't correspond to the current program, and merely overwrites them instead.

All three of these files use the functions in `gcov-io.h` to store integers; the functions in this header provide a machine-independent mechanism for storing and retrieving data from a stream.

23 tricore-version-info

23.1 Description

`tricore-version-info` <objfile>

`tricore-version-info` <option>

`tricore-version-info` displays the content of the section .version_info of the objects given as input parameters.

23.2 Invocation

`-h`	Print helpscreen
`-v`	Print version number

23.3 Example

```
tricore-version-info main.o uart.o
```

23.4 Tool Version of GCC

The tool version of the compiler is available as built-in, so it can include or exclude parts of the source for different tool versions.

```
#if (__TOOL_VERSION__ == 20)
/* this code will be included in tool version 2.0 */

#endif
```

24 Warningsfilter

24.1 Introduction

The warnings that are issued by the compiler are marked by an index number. The docu of the warnings is available in the files `list-of-errors-and-warnings.pdf` and `warningsfilter/Help/gcc.html`.

Normally the build process of a program is controlled by a makefile. To save the make output of stdout to a log-file, type:

```
make <options> > name.log 2>&1
```

The log file can be analyzed be the warningsfilter by invoking the following command.

```
perl WarnungsFilter.pl --input-file=name.log --out-dir=<directory>
```

The generated output is stored in the specified directory `--out-dir=<directory>`, if the output directory is not specified the files will be stored in the default directory `.auto_wf`. The script `WarnungsFilter.pl` generates a HTML file called `index.html`. This file gives you an overview which source `name.c` generates a warning with the corresponding line. Additionally the HTML file contains a link to the HMTL docu `gcc.html` of the warnings with its description.

The `WarnungsFilter.pl` supports several command line options.

24.2 Options

`--help`	Displays the help and exit. The displayed message can be found under `[Script's directory]/Doc/help.txt`. If specified, all the others arguments are ignored.
`--version`	Displays the version of the tool, and of each module used by the tool, and exit. If specified, all the others arguments are ignored (except `--help`, because it has the hightest priority).
`--history`	Displays the history of the tool, and exit. If specified, all the others arguments are ignored (except `--help` and `--version`, because they have the hightest priority).
`--silent`	Do not display anything on the output, except error message (when the script dies, or crashes for example) it means, nothing is displayed on STDOUT this does not apply for the tool's log file.
`--verbosity=n`	(default = 2) This is only partially implemented, some modules don't care about it Verbosity level. It applies for the output's log file (and STDOUT, except if `--silent` is turned on).

`0`	Does not say anything (in this case, no output's log file generated).
`1`	Only says what it's supposed to do.
`2`	Gives some statistics when processing, and all the name of the files generated.
`3`	Gives number of lines processed, and others details.
`4`	Gives details about intern routines, results of intermediate computes.
`5`	Very detailed information.

`--sort=n` (default = no sorting) Sort the categories into different ways:

`1`	Sorted by number of files.
`2`	Sorted by warnings.
`3`	Sorted by occurrences.
`4`	Sorted by names.
`other`	Sorted by the internal hash function of the perl interpreter.

`--input-file=File`
(no default: must be specified somewhere, here or in a Configuration's file) Specify the input file to process, with the extention. If the directory is ommited, the current work directory is used.

`--log-file=File`
(default = [Tool's Out Directory]/Wf.log) Gives the name of the tool's log file. If you don't want a log file, look at this option: `--verbosity`.

`--config-file=File`
(default = no configuration's file) Specifies the configuration's file to use. If the path is omitted, or relative, the script will try to find the specified configuration's file in the following directories, in this order:

current directory	/
home directory	/
home directory	/.WarnungFilter/
Script's directory	/templates/Config/

if the file is not found, the script dies with the message 'The configuration's file [name of the file] was not found in the following directories [list of directories]'.

`--OS=Operating_System`
(default = autodetected from the $Ô built-in variable) Please choose in the list (case sensitive!)

- VMS
- MSDOS (You shouldn't use this, because of the long names, i.e the extention `html` has to many characters... Don't report bug for this.)
- MacOS
- AmigaOS
- MSWin32 (applies for all the MS Windows versions, since the Windows 95)
- Unix
- linux (is identical to Unix)

If you are giving something else, Unix is assumed (Sorry, but the module used to choice the OS belongs to standard librairies of perl. Only MSWin32 and Unix (or linux) where tested. If you are interested in other operating systems, ask the authors. The script will probably work with the others system, but it's not sure. This option might be removed soon, because at the moment the name's gestion is problematic. This will be fixed in new versions.

`--out-dir=Directory`
(default = [Work Directory]/.auto_wf/ If this is a relative path, it will be take regarding the current work directory. Do not forget the last '/' (or '\' for MS style path)

`--warning-url-database`
(default = [Tool's directory]/Doc/W_url.pm) This mini perl-sript contains a hash, named %WF_Warning_Help_Link. It gives the tool the url to the html help files for the warning's categories. This is partially implemented, but this option is not activated yet.

25 JTAG Server for TriCore

The jtag interface of TriCore gives the user an convenient access mechanism for debugging an embedded systems. When JTAG is used as a debugging tool, an in-circuit emulator which in turn uses JTAG as the transport mechanism enables a programmer to access an on-chip debug module which is integrated into the CPU via JTAG.

The `tricore-jtagsrv` requires an environment variable `JIOBASE`, that is set to the directory where the `tricore/jtag_targets/*.ini` files for the derivatives are located.

Note:

For debugging via JTAG-Interface the BIOS settings for the printer port must not be configured for ECP-Mode. Select EPP or SPP mode.

For debugging your application do the following steps:

- Apply a power supply to your board
- Connect the target board with JTAG
- Start the JTAG-Server `tricore-jtagsrv`

The `tricore-jtagsrv` supports the following command line option.

`-n, --no-reset` Per default the `tricore-jtagsrv` performs a reset of target. Sometimes the user wants to keep the state of the application and the content of special function registers or RAM. With this option the reset is suppressed, otherwise the content of RAM and direction of the port pins are set to default values.

`-i, --init <target-ini-file>`
The inifile specified with the option `-i` contains information about derivative specific board configuration. This information is required for the initialization of the board and to get access via jtag.

```
[Init]
EBU_CON         = 0x0000FF44
EBU_ADDRSEL0    = 0xA4000853
...
```

If the environment variable `JIOBASE` is set, the use of option will not be necessary.

`-v, --version` Prints version information of the `tricore-jtagsrv`.

`-p, --port` Per default the port is set to 6785. With this option the port for debugging can be modified.

```
tricore-jtagsrv -p :3210
```

support flashtypes amd, intel

Part V

Libraries and Headers

26 Libraries of the Compiler

26.1 Functions of libgcc.a

The libgcc includes the implementation of implicit called library routines. For some instructions like long long arithmetic the compiler generates implicit library calls to subroutines like _muldi3, __divdi3 etc. Since the TriCore doesn't support floating point arithmetic, all floating point arithmetic is done by library calls to function in libgcc.a.

The `libgcc.a` includes the function _bb_init_func and __bb_exit_func to support the production of profiling information with the option `-fprofile-arcs` and `-ftest-coverage`. This feature is only available with some kind of stdio on the target platform. I.e. through simulated IO within the debugger and the included library `libos.a`.

26.2 Functions of libgccoptfp.a

If your TriCore version has a integrated floating point unit you can use `libgccoptfp.a`, so all floating point arithmetic are not emulated anymore by library calls to function in libgcc.a.

Note:

`libgccoptfp.a` link against a TriCore optimized floating point library; the single-float functions contained therein are very fast, but they don't provide any error checking facilities. The library is not IEEE conform.

26.3 Functions in libos.a

Interrupt interface in the module cint.c/cint.h. Functions to lock/unlock access to protected SFRs in wdtcon.c/wdtcon.h

Functions for the I/O-simulation in connection with a debugger GDB. All I/O-calls like open, read, write, close, create, unlink, lseek will generate a break condition which can be evaluated by the GDB to perform the I/O call for the target application.

The module `libos.c` contains some useful routines

`abort, exit`	Generate a debug trap.
`sbrk`	Allocate memory from the HEAP.
`fork, wait, fstat`	Standard system calls.
`stat`	Calling abort.
`times, usec`	Returns the contents of the system timer.

26.4 Function of libnosys.a

`libnosys.a` implements a dummy operating system interface. It includes routines for the standard system calls like open,read, fork etc. All this routines return -1 and set the global variable errno to ENOSYS to indicate the absence of an operating system to handle these calls.

26.5 Function of libm.a

`libm.a` implements the MATH-Library. See the appropriate manual for a full description.

26.6 Function of libc.a/libg.a

`libc.a/libg.a` is the implementation of a standard C-Library build from the so called newlib. See the appropriate manual for a full description.

26.7 Floating Point Support

The TriCore Development Platform supports the FPU of TriCore.

26.7.1 Treat Doubles as Floats

Example

The TriCore Development Platform compiler is ANSI conform, these means that a `1.0` is per default a **double** and not a **float**. In the case of the example will convert **double** to **float**. This code is not effective for using FPU of TriCore, therefore use the modifier `f` to mark the operands as float `1.0f`.

```
float x_f = 1.01

int
main (void)
 {
  float y_f:
  y_f = 1.0/x_f; /* divide double by float */
  return 0;
```

With the option `-fshort-double` the compiler assumes the same size for **float** and **double**. This option is not ANSI conform, because **double** have better precision as **float**, so the result has not the same precision as a double operation.

26.7.2 Floating Point Math Functions

```
#include <math.h>

int
main (void)
 {
  float x;
  sinf (x);
  return 0;
 }
```

To access the floating point functions of libraries, add the suffix `f`. Please use the option `-mcpu=<derivative>` to select the derivative specific floating point libraries. This option also sets all workaournds of relevant cpu erratas.

Note:

Add the math library `libm.a` to your linker option `-lm`.

26.8 Description of libgcc.a and libgccoptfp.a

These functions are part of `libgcc.a` and `libgccoptfp.a`:

`_fpadd_parts`	Adds two floating point values, which are split into sign, exponent and mantissa
`__absvdi2`	Returns the absolute value of a **long long** variable
`__absvsi2`	Returns the absolute value of a **int** variable
`__addvdi3`	Adds two **long long** variables and verifies signed overflow
`__addvsi3`	Adds two **int** variables and verifies signed overflow
`__bb_exit_func`	Outputs the results of a code coverage check generated by `--profile-arcs` to a `.da`-file
`__bb_fork_func`	This function is called before fork() or cmd() if code coverage checks are enabled. It writes out and resets the profile information gathered so far
`__bb_init_func`	Initializes the code coverage. This function is called automatically by __main
`__clear_cache`	Clears parts of an instruction cache
`__cmpdf2`	Compares two **double** variables
`__cmpsf2`	Compares two **float** variables
`__divdi3`	Performs a division of two **long long** variables
`__do_global_ctors`	Runs all the global constructors on entry to the program
`__do_global_dtors`	Runs all the global destructors on exit from the program
`__d_add`	Adds two **double** variables
`__d_div`	Divides two **double** variables
`__d_dtof`	Converts an **double** to a **float**

`__d_dtoi`	Converts an **double** to an **int**
`__d_dtoui`	Converts an **double** to an **unsigned int**
`__d_itod`	Converts an **int** to a **double**
`__d_mul`	Multiplies two **double** variables
`__d_neg`	Negates a **double**
`__d_sub`	Subtracts two **double** variables
`__d_uitod`	Converts an **unsigned int** to a **double**
`__eprintf`	This function is mapped to fprintf () and outputs a string to `stderr`
`__eqdf2`	Checks if a **double** is equal to another **double**
`__eqsf2`	Checks if a **float** is equal to another **float**
`__ffsdi2`	Finds first set bit in a **long long**
`__fixunsdfdi`	Performs an unsigned cast from a **double** variable to **long long** variable
`__fixunsdfsi`	Performs an unsigned cast from a **double** variable to **int** variable
`__fixunssfdi`	Performs an unsigned cast from a **float** variable to **long long** variable
`__fixunssfsi`	Performs an unsigned cast from a **float** variable to **int** variable
`__fixdfdi`	Performs a cast from a **double** variable to **long long** variable
`__fixsfdi`	Performs a cast from a **float** variable to **long long** variable
`__floatdidf`	Performs a cast from a **long long** variable to **double** variable
`__floatdisf`	Performs a cast from a **long long** variable to **float** variable
`__fpcmp_parts_d`	Compares two **double** variables which are split into sign, exponent and mantissa
`__fpcmp_parts_f`	Compares two **float** variables which are split into sign, exponent and mantissa
`__f_add`	Adds two **float** variables
`__f_div`	Divides two **float** variables
`__f_ftod`	Converts an **float** to a **double**
`__f_ftoi`	Converts an **float** to an **int**
`__f_ftoui`	Converts an **float** to an **unsigned int**
`__f_itof`	Converts an **int** to a **float**
`__f_mul`	Multiplies two **float** variables

`__f_neg`	Negates a **float**
`__f_sub`	Subtracts two **float** variables
`__f_uitof`	Converts an **unsigned int** to a **float**
`__gcc_bcmp`	Compares two strings
`__gedf2`	Checks if a **double** is greater or equal to another **double**
`__gesf2`	Checks if a **float** is greater or equal to another **float**
`__gtdf2`	Checks if a **double** is greater than another **double**
`__gtsf2`	Checks if a **float** is greater than another **float**
`__ledf2`	Checks if a **double** is less or equal to another **double**
`__lesf2`	Checks if a **float** is less or equal to another **float**
`__ltdf2`	Checks if a **double** is less than another **double**
`__ltsf2`	Checks if a **float** is less than another **float**
`__main`	Initializes the program by executing __do_global_ctors()
`__make_dp`	Generates a double precision float value from sign, exponent and mantissa
`__make_fp`	Generates a single precision float value from sign, exponent and mantissa
`__moddi3`	Performs a modulo operation of two **long long** variables
`__mulvdi3`	Multiplies and verifies signed overflow two **long long** variables
`__mulvsi3`	Multiplies and verifies signed overflow two **int** variables
`__nedf2`	Checks if a **double** is not equal to another **double**
`__negvdi2`	Negates and verifies signed overflow a **long long** variable
`__negvsi2`	Negates and verifies signed overflow a **int** variable
`__nesf2`	Checks if a **float** is not equal to another **float**
`__pack_d`	Generates a IEEE-**double** variable from sign, exponent and mantissa
`__pack_f`	Generates a IEEE-**float** variable from sign, exponent and mantissa
`__subvdi3`	Subtracts two **long long** variables
`__subvsi3`	Subtracts two **int** variables
`__udivdi3`	Performs an unsigned division of two **long long** variables
`__udivmoddi4`	Performs an unsigned division and modulo operation of two **long long** variables

`__udiv_w_sdiv`	unsigned division by using signed division
`__umoddi3`	Performs an unsigned modulo operation of two **long long** variables
`__unorddf2`	unordered compare of a **double**
`__unordsf2`	unordered compare of a **float**
`__unpack_d`	Splits a IEEE-**double** in sign, exponent and mantissa
`__unpack_f`	Splits a IEEE-**float** in sign, exponent and mantissa

27 TriCore Header

The TriCore header files are generated using the descriptions from the Infineon's Dave description of TriCore. In the headerfiles the hardware description for the derivatives is available. The docu of TriCore headers is in the `html` directory of your installation path.

If you open the file e.g. `html/<derivative>/index.html` (see option `-mcpu=<derivative>`) you find the docu of the generated header files.

You find the structs of the function units e.g. TC1130 Step A in the following directory relative to your installation path.

ASC_t	tc1130a/asc-struct.h	ASC0, ASC1, ASC2
PORT_t	tc1130a/port-struct.h	P0, P1, P2, P3
PORT4_t	tc1130a/port4-struct.h	P4
SSC_t	tc1130a/ssc-struct.h	SSC00, SSC01
CCU_t	tc1130a/ccu-struct.h	CCU60, CCU61
USB_t	tc1130a/usb-struct.h	USB
MLI_t	tc1130a/mli-struct.h	MLI0, MLI1
DMA_t	tc1130a/dma-struct.h	DMA
CAN_t	tc1130a/can-struct.h	MultiCAN

If you select for example P0 in the HTML-file, you will be redirected to the port P0 description. The special function register are listed with the following information.

- Register Name
- Description
- Address
- Type
- Reset
- Access (read)
- Access (write)

Example

Select for example the special function register `P0_OUT`. The read and write access is enabled in user mode (U) and supervisor mode (SV). The **struct** of the `P0_OUT` is defined in `Pn_OUT_t`. In this **struct** all port pins are accessable as bitfield by the name of the port pin.

```
  unsigned int P0:1;
  unsigned int P1:1;
...
```

E.g. you get access to pin 0 of port 0 and set it in your source by:

```
P0_OUT.bits.P0 = 1;
```

Part VI

Appendices

28 The GNU project

```
From CSvax:pur-ee:inuxc!ixn5c!ihnp4!houxm!mhuxi!eagle!mit-vax!mit-eddie!RMS@MIT-OZ
From: RMS%MIT-OZ@mit-eddie
Newsgroups: net.unix-wizards,net.usoft
Subject: new UNIX implementation
Date: Tue, 27-Sep-83 12:35:59 EST
Organization: MIT AI Lab, Cambridge, MA

Free Unix!

Starting this Thanksgiving I am going to write a complete
Unix-compatible software system called GNU (for Gnu's Not Unix), and
give it away free(1) to everyone who can use it.  Contributions of time,
money, programs and equipment are greatly needed.

To begin with, GNU will be a kernel plus all the utilities needed to
write and run C programs: editor, shell, C compiler, linker,
assembler, and a few other things.  After this we will add a text
formatter, a YACC, an Empire game, a spreadsheet, and hundreds of
other things.  We hope to supply, eventually, everything useful that
normally comes with a Unix system, and anything else useful, including
on-line and hardcopy documentation.

GNU will be able to run Unix programs, but will not be identical
to Unix.  We will make all improvements that are convenient, based
on our experience with other operating systems.  In particular,
we plan to have longer filenames, file version numbers, a crashproof
file system, filename completion perhaps, terminal-independent
display support, and eventually a Lisp-based window system through
which several Lisp programs and ordinary Unix programs can share a screen.
Both C and Lisp will be available as system programming languages.
We will have network software based on MIT's chaosnet protocol,
far superior to UUCP.  We may also have something compatible
with UUCP.


Who Am I?

I am Richard Stallman, inventor of the original much-imitated EMACS
editor, now at the Artificial Intelligence Lab at MIT.  I have worked
extensively on compilers, editors, debuggers, command interpreters, the
Incompatible Timesharing System and the Lisp Machine operating system.
I pioneered terminal-independent display support in ITS.  In addition I
have implemented one crashproof file system and two window systems for
Lisp machines.


Why I Must Write GNU

I consider that the golden rule requires that if I like a program I
must share it with other people who like it.  I cannot in good
conscience sign a nondisclosure agreement or a software license
```

agreement.

So that I can continue to use computers without violating my principles, I have decided to put together a sufficient body of free software so that I will be able to get along without any software that is not free.

How You Can Contribute

I am asking computer manufacturers for donations of machines and money. I'm asking individuals for donations of programs and work.

One computer manufacturer has already offered to provide a machine. But we could use more. One consequence you can expect if you donate machines is that GNU will run on them at an early date. The machine had better be able to operate in a residential area, and not require sophisticated cooling or power.

Individual programmers can contribute by writing a compatible duplicate of some Unix utility and giving it to me. For most projects, such part-time distributed work would be very hard to coordinate; the independently-written parts would not work together. But for the particular task of replacing Unix, this problem is absent. Most interface specifications are fixed by Unix compatibility. If each contribution works with the rest of Unix, it will probably work with the rest of GNU.

If I get donations of money, I may be able to hire a few people full or part time. The salary won't be high, but I'm looking for people for whom knowing they are helping humanity is as important as money. I view this as a way of enabling dedicated people to devote their full energies to working on GNU by sparing them the need to make a living in another way.

For more information, contact me.
Arpanet mail:
  RMS@MIT-MC.ARPA

Usenet:
  ...!mit-eddie!RMS@OZ
  ...!mit-vax!RMS@OZ

US Snail:
  Richard Stallman
  166 Prospect St
  Cambridge, MA 02139

URL catalog

[↪ARM] ARM homepage.
http://www.arm.com/

[↪C16x] Infineon c166 page.
http://www.infineon.com/c166

[↪GCC] GCC home page.
http://gcc.gnu.org/

[↪HIGHTEC] HighTec homepage.
http://www.hightec-rt.com/

[↪INTEL] Intel homepage.
http://www.intel.com

[↪MSP430] MSP430 homepage.
http://www.ti.com/sc/msp430

[↪PowerPC] Motorola homepage.
http://e-www.motorola.com/

[↪TriCore] TriCore homepage.
http://www.infineon.com/tricore/

List of changes

At this list of changes you will find all significant changes. The numbers behind the versions are the pages, where the changes are described. At the margins of these pages you will find corresponding version marks.

Index