# Table of Contents

1. C Language ................................................................................................................................. 1
   1.1. Data Types .......................................................................................................................... 1
       1.1.1. Bit Data Type ........................................................................................................... 2
       1.1.2. Fractional Types ........................................................................................................ 3
       1.1.3. Packed Data Types .................................................................................................... 4
       1.1.4. Changing the Alignment: __align() .......................................................................... 6
   1.2. Accessing Memory .................................................................................................................. 6
       1.2.1. Memory Qualifiers ..................................................................................................... 6
       1.2.2. Placing an Object at an Absolute Address: __at() and __atbit() ........................... 8
   1.3. Data Type Qualifiers ............................................................................................................ 10
       1.3.1. Circular Buffers: __circ .......................................................................................... 10
       1.3.2. Accessing Hardware from C ................................................................................ 11
       1.3.3. Saturation: __sat ................................................................................................. 12
   1.4. Using Assembly in the C Source: __asm() .......................................................................... 13
   1.5. Pragmas to Control the Compiler ....................................................................................... 17
   1.6. Predefined Preprocessor Macros ....................................................................................... 22
   1.7. Switch Statement ............................................................................................................... 23
   1.8. Functions ............................................................................................................................ 24
       1.8.1. Calling Convention ................................................................................................. 24
       1.8.2. Register Usage ......................................................................................................... 26
       1.8.3. Inlining Functions: inline ....................................................................................... 27
       1.8.4. Interrupt and Trap Functions .................................................................................. 28
       1.8.5. Intrinsic Functions ................................................................................................. 33
   1.9. Compiler Generated Sections ............................................................................................. 42
       1.9.1. Rename Sections ..................................................................................................... 43
       1.9.2. Influence Section Definition ................................................................................... 44
       1.9.3. Change Section Alignment ...................................................................................... 44
2. C++ Language ............................................................................................................................... 47
   2.1. C++ Language Extension Keywords .................................................................................. 47
   2.2. C++ Dialect Accepted ....................................................................................................... 47
       2.2.1. Standard Language Features Accepted ................................................................. 48
       2.2.2. C++0x Language Features Accepted ..................................................................... 51
       2.2.3. Anachronisms Accepted ....................................................................................... 52
       2.2.4. Extensions Accepted in Normal C++ Mode ......................................................... 53
   2.3. GNU Extensions ................................................................................................................. 54
   2.4. Namespace Support ......................................................................................................... 65
   2.5. Template Instantiation ...................................................................................................... 67
       2.5.1. Automatic Instantiation ......................................................................................... 68
       2.5.2. Instantiation Modes ............................................................................................... 70
       2.5.3. Instantiation #pragma Directives .......................................................................... 71
       2.5.4. Implicit Inclusion ................................................................................................. 73
       2.5.5. Exported Templates .............................................................................................. 73
   2.6. Extern Inline Functions ..................................................................................................... 76
   2.7. Pragmas to Control the C++ Compiler .............................................................................. 77
   2.8. Predefined Macros ............................................................................................................ 78
   2.9. Precompiled Headers ....................................................................................................... 81
       2.9.1. Automatic Precompiled Header Processing ......................................................... 81
       2.9.2. Manual Precompiled Header Processing ............................................................. 84
2.9.3. Other Ways to Control Precompiled Headers ................................................. 85
2.9.4. Performance Issues ...................................................................................... 85
3. Assembly Language .......................................................................................... 87
   3.1. Assembly Syntax ......................................................................................... 87
   3.2. Assembler Significant Characters ............................................................... 88
   3.3. Operands of an Assembly Instruction .......................................................... 88
   3.4. Symbol Names ........................................................................................... 89
      3.4.1. Predefined Preprocessor Symbols ......................................................... 89
   3.5. Registers .................................................................................................. 90
      3.5.1. Special Function Registers .................................................................... 91
   3.6. Assembly Expressions ............................................................................... 91
      3.6.1. Numeric Constants ................................................................................ 92
      3.6.2. Strings ................................................................................................. 92
      3.6.3. Expression Operators .......................................................................... 93
   3.7. Working with Sections ............................................................................... 94
   3.8. Built-in Assembly Functions .................................................................... 96
   3.9. Assembler Directives and Controls ............................................................. 105
      3.9.1. Assembler Directives .......................................................................... 106
      3.9.2. Assembler Controls ........................................................................... 152
   3.10. Macro Operations .................................................................................... 170
      3.10.1. Defining a Macro ................................................................................ 170
      3.10.2. Calling a Macro ................................................................................ 170
      3.10.3. Using Operators for Macro Arguments ............................................. 171
4. Using the C Compiler ...................................................................................... 177
   4.1. Compilation Process ................................................................................... 177
   4.2. Calling the C Compiler ............................................................................. 178
   4.3. The C Startup Code ................................................................................... 180
   4.4. How the Compiler Searches Include Files ............................................... 182
   4.5. Compiling for Debugging ........................................................................ 183
   4.6. Compiler Optimizations .......................................................................... 184
      4.6.1. Generic Optimizations (frontend) ......................................................... 185
      4.6.2. Core Specific Optimizations (backend) .............................................. 186
      4.6.3. Optimize for Size or Speed ................................................................. 187
   4.7. C Code Checking: MISRA-C .................................................................... 190
   4.8. C Compiler Error Messages .................................................................... 191
5. Using the C++ Compiler ................................................................................. 193
   5.1. C++ Development Structure .................................................................... 193
      5.1.1. The Prelinker Phase ........................................................................... 194
      5.1.2. The Muncher Phase ........................................................................... 196
   5.2. Calling the C++ Compiler .......................................................................... 196
   5.3. How the C++ Compiler Searches Include Files ...................................... 198
   5.4. C++ Compiler Error Messages ................................................................. 199
6. Profiling ........................................................................................................ 203
   6.1. What is Profiling? .................................................................................... 203
      6.1.1. Four Methods of Profiling ................................................................. 203
   6.2. Profiling using Code Instrumentation (Dynamic Profiling) ..................... 205
      6.2.1. Step 1: Build your Application for Profiling ....................................... 206
      6.2.2. Step 2: Execute the Application ......................................................... 207
      6.2.3. Step 3: Displaying Profiling Results .................................................. 209
   6.3. Profiling at Compile Time (Static Profiling) ............................................. 212
6.3.1. Step 1: Build your Application with Static Profiling .................................................. 212
6.3.2. Step 2: Displaying Static Profiling Results ................................................................. 213
7. Using the Assembler ........................................................................................................ 215
  7.1. Assembly Process ........................................................................................................ 215
  7.2. Calling the Assembler ............................................................................................... 216
  7.3. How the Assembler Searches Include Files ................................................................. 217
  7.4. Assembler Optimizations .......................................................................................... 218
  7.5. Generating a List File ............................................................................................... 218
  7.6. Assembler Error Messages ....................................................................................... 219
8. Using the Linker ............................................................................................................ 221
  8.1. Linking Process .......................................................................................................... 221
    8.1.1. Phase 1: Linking .................................................................................................. 223
    8.1.2. Phase 2: Locating .............................................................................................. 224
  8.2. Calling the Linker ...................................................................................................... 225
  8.3. Linking with Libraries ............................................................................................... 226
    8.3.1. How the Linker Searches Libraries .................................................................. 229
    8.3.2. How the Linker Extracts Objects from Libraries .............................................. 229
  8.4. Incremental Linking .................................................................................................. 230
  8.5. Importing Binary Files ............................................................................................. 230
  8.6. Linker Optimizations ............................................................................................... 231
  8.7. Controlling the Linker with a Script ......................................................................... 232
    8.7.1. Purpose of the Linker Script Language ......................................................... 232
    8.7.2. Eclipse and LSL ............................................................................................... 233
    8.7.3. Structure of a Linker Script File ...................................................................... 234
    8.7.4. The Architecture Definition .......................................................................... 237
    8.7.5. The Derivative Definition .............................................................................. 240
    8.7.6. The Processor Definition .............................................................................. 242
    8.7.7. The Memory Definition ................................................................................... 242
    8.7.8. The Section Layout Definition: Locating Sections ......................................... 244
  8.8. Linker Labels ........................................................................................................... 246
  8.9. Generating a Map File ............................................................................................. 248
  8.10. Linker Error Messages .......................................................................................... 248
9. Using the Utilities ......................................................................................................... 251
  9.1. Control Program ........................................................................................................ 251
  9.2. Make Utility ................................................................................................................ 252
    9.2.1. Calling the Make Utility ................................................................................... 254
    9.2.2. Writing a Makefile ......................................................................................... 254
  9.3. Archiver .................................................................................................................... 262
    9.3.1. Calling the Archiver ....................................................................................... 263
    9.3.2. Archiver Examples ......................................................................................... 264
  9.4. C++ Prelinker .......................................................................................................... 265
10. Using the Debugger ..................................................................................................... 267
  10.1. Reading the Eclipse Documentation ....................................................................... 267
  10.2. Creating a Customized Debug Configuration ...................................................... 267
  10.3. Troubleshooting .................................................................................................... 273
  10.4. The TASKING Debugger Perspective ................................................................. 274
    10.4.1. Debug View .................................................................................................... 275
    10.4.2. Breakpoints View .......................................................................................... 277
    10.4.3. File System Simulation (FSS) View ............................................................... 278
    10.4.4. Disassembly View .......................................................................................... 279
10.4.5. Expressions View .................................................................................. 279
10.4.6. Memory View ...................................................................................... 280
10.4.7. Compare Application View ..................................................................... 281
10.4.8. Heap View ........................................................................................... 281
10.4.9. Logging View ....................................................................................... 281
10.4.10. RTOS View ........................................................................................ 281
10.4.11. TASKING Registers View .................................................................... 282
10.4.12. Trace View ........................................................................................ 283

11. Tool Options .................................................................................................. 285
  11.1. C Compiler Options .................................................................................. 285
  11.2. C++ Compiler Options .............................................................................. 363
  11.3. Assembler Options .................................................................................... 470
  11.4. Linker Options ........................................................................................ 511
  11.5. Control Program Options ........................................................................ 556
  11.6. Make Utility Options ................................................................................ 623
  11.7. Archiver Options ..................................................................................... 651
  11.8. C++ Prelinker Options ............................................................................. 664

12. Libraries ......................................................................................................... 693
  12.1. Library Functions ...................................................................................... 694
    12.1.1. assert.h .............................................................................................. 694
    12.1.2. complex.h .......................................................................................... 695
    12.1.3. cstart.h ............................................................................................ 696
    12.1.4. ctype.h and wctype.h ....................................................................... 696
    12.1.5. dbg.h ............................................................................................... 697
    12.1.6. errno.h .............................................................................................. 697
    12.1.7. fcntl.h .............................................................................................. 698
    12.1.8. fenv.h .............................................................................................. 698
    12.1.9. float.h .............................................................................................. 699
    12.1.10. inttypes.h and stdint.h ................................................................. 700
    12.1.11. io.h ................................................................................................. 700
    12.1.12. iso646.h .......................................................................................... 701
    12.1.13. limits.h ........................................................................................... 701
    12.1.14. locale.h ........................................................................................... 701
    12.1.15. malloc.h ........................................................................................ 701
    12.1.16. math.h and tgmath.h ...................................................................... 702
    12.1.17. setjmp.h ......................................................................................... 706
    12.1.18. signal.h ........................................................................................... 706
    12.1.19. stdarg.h ......................................................................................... 707
    12.1.20. stdbool.h ....................................................................................... 707
    12.1.21. stddef.h ........................................................................................ 708
    12.1.22. stdint.h ......................................................................................... 708
    12.1.23. stdio.h and wchar.h ...................................................................... 708
    12.1.24. stdlib.h and wchar.h .................................................................... 716
    12.1.25. string.h and wchar.h ................................................................. 719
    12.1.26. time.h and wchar.h ...................................................................... 720
    12.1.27. unistd.h ......................................................................................... 723
    12.1.28. wchar.h ......................................................................................... 724
    12.1.29. wctype.h ....................................................................................... 725

  12.2. C Library Reentrancy ................................................................................ 725

13. List File Formats ............................................................................................. 737
Chapter 1. C Language

This chapter describes the target specific features of the C language, including language extensions that are not standard in ISO-C. For example, pragmas are a way to control the compiler from within the C source.

The TASKING VX-toolset for TriCore® C compiler fully supports the ISO-C standard and adds extra possibilities to program the special functions of the target.

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

- extra data types, like __bit, __fract, __laccum and __packb
- keywords to specify memory types for data and functions
- attribute to specify alignment and absolute addresses
- intrinsic (built-in) functions that result in target specific assembly instructions
- pragmas to control the compiler from within the C source
- predefined macros
- the possibility to use assembly instructions in the C source
- keywords for inlining functions and programming interrupt routines
- libraries

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

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

1.1. Data Types

The C compiler supports the ISO C99 defined data types, and additionally the bit data type, fractional types and packed data types. The sizes of these types are shown in the following table.

<table>
<thead>
<tr>
<th>C Type</th>
<th>Size</th>
<th>Align</th>
<th>Limits</th>
</tr>
</thead>
<tbody>
<tr>
<td>__bit</td>
<td>1</td>
<td>8</td>
<td>0 or 1</td>
</tr>
<tr>
<td>_Bool</td>
<td>1</td>
<td>8</td>
<td>0 or 1</td>
</tr>
<tr>
<td>signed char</td>
<td>8</td>
<td>8</td>
<td>[-2^7, 2^7-1]</td>
</tr>
<tr>
<td>unsigned char</td>
<td>8</td>
<td>8</td>
<td>[0, 2^8-1]</td>
</tr>
<tr>
<td>short</td>
<td>16</td>
<td>16</td>
<td>[-2^15, 2^15-1]</td>
</tr>
<tr>
<td>unsigned short</td>
<td>16</td>
<td>16</td>
<td>[0, 2^16-1]</td>
</tr>
<tr>
<td>int</td>
<td>32</td>
<td>16</td>
<td>[-2^31, 2^31-1]</td>
</tr>
<tr>
<td>unsigned int</td>
<td>32</td>
<td>16</td>
<td>[0, 2^32-1]</td>
</tr>
</tbody>
</table>
### 1.1.1. Bit Data Type

The TASKING C compiler for the TriCore supports the data type `__bit`. The TriCore instruction set supports some operations of the `__bit` type directly.

The following rules apply to `__bit` type variables:

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

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

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

• Pointer to __bit is not allowed when it has the __atbit() qualifier.

• The __bit data type is allowed as a struct/union member.

• A __bit type variable is allowed as a parameter of a function.

• A __bit type variable is allowed as a return type of a function.

• A __bit typed expression is allowed as switch expression.

• The sizeof of a __bit type is 1.

• Global or static __bit type variables can be initialized.

• A __bit type variable can be declared absolute using the __atbit attribute. See

• A __bit type variable can be declared volatile.

Promotion rules

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

1.1.2. Fractional Types

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

```c
__fract f, f1, f2;  /* Declaration of fractional variables */
f1 = 0.5;           /* Assignment of a fractional constants */
f2 = 0.242;
f = f1 * f2;        /* Multiplication of two fractionals */
```

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

The __accum type is only included for compatibility reasons and is mapped to __laccum.
The TriCore instruction set supports most basic operations on fractional types directly. To obtain more portable code, you can use several intrinsic functions that use fractional types. Fractional values are automatically saturated.

Section 1.8.5, *Intrinsic Functions* explains intrinsic functions. Section 1.8.5.2, *Fractional Arithmetic Support* lists the intrinsic functions.

**Promotion rules**

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

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

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

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

<table>
<thead>
<tr>
<th>left</th>
<th>operand</th>
<th>right</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>fractional</td>
<td>*</td>
<td>integer</td>
<td>fractional</td>
</tr>
<tr>
<td>integer</td>
<td>*</td>
<td>fractional</td>
<td>fractional</td>
</tr>
<tr>
<td>fractional</td>
<td>/</td>
<td>integer</td>
<td>fractional</td>
</tr>
<tr>
<td>fractional</td>
<td>&lt;&lt;</td>
<td>integer</td>
<td>fractional</td>
</tr>
<tr>
<td>fractional</td>
<td>&gt;&gt;</td>
<td>integer</td>
<td>fractional</td>
</tr>
<tr>
<td>fractional: <code>__sfract</code>, <code>__fract</code></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>integer: <code>char</code>, <code>short</code>, <code>int</code>, <code>long</code>, <code>long long</code></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**1.1.3. Packed Data Types**

The TASKING C compiler additionally supports the packed types `__packb` and `__packhw`.

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

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

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

results into the following assembly code:
add4:
  add.b d2,d4,d5
  ret16

Section 1.8.5, Intrinsic Functions explains intrinsic functions. Section 1.8.5.3, Packed Data Type Support lists the intrinsic functions.

Halfword packed unions and structures

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

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

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

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

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

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

#pragma pack 0
packed_struct pstruct;

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

By default the TriCore compiler aligns objects to the minimum alignment required by the architecture. With the attribute __align() you can change the object alignment.

**Caution:** Use __align() with extreme care! You should know exactly how to use it properly, otherwise it may result in incorrect (trapping) code, as shown in the following example. Use it only to increase the alignment; and the alignment must always be a multiple of the original required alignment!

Example:

```c
#include <stdio.h>
short int i[3] = {1,2,3};
__align(2) char *hello="Hello World\n"; /* trap */

int main(void)
{
    printf("%s",hello);
}
```

The pointer 'hello' is given an alignment of 2 and is actually stored at a non-word aligned address. The compiler however does not take into account that pointers are aligned at 2 and uses a load address instruction. This will lead to a trap at run-time.

Instead of the attribute __align() you can also use #pragma align.

1.2. Accessing Memory

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

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

1.2.1. Memory Qualifiers

In the C language you can specify that a variable must lie in a specific part of memory. You can do this with a memory qualifier.

You can specify the following memory qualifiers:

<table>
<thead>
<tr>
<th>Qualifier</th>
<th>Description</th>
<th>Location</th>
<th>Maximum object size</th>
<th>Pointer size</th>
<th>Section types</th>
</tr>
</thead>
<tbody>
<tr>
<td>__near</td>
<td>Near data, direct addressable</td>
<td>First 16 kB of a 256 MB block</td>
<td>16 kB</td>
<td>32-bit</td>
<td>neardata, nearrom, nearbss, nearnoclear</td>
</tr>
</tbody>
</table>
### C Language

<table>
<thead>
<tr>
<th>Qualifier</th>
<th>Description</th>
<th>Location</th>
<th>Maximum object size</th>
<th>Pointer size</th>
<th>Section types</th>
</tr>
</thead>
<tbody>
<tr>
<td>__far *</td>
<td>Far data, indirect addressable</td>
<td>Anywhere</td>
<td>no limit</td>
<td>32-bit</td>
<td>far data, far rom, far bss, far nolclear</td>
</tr>
<tr>
<td>__a0</td>
<td>Small data</td>
<td>Sign-extended 16-bit offset from address register A0.</td>
<td>64 kB</td>
<td>32-bit</td>
<td>a0 data, a0 bss</td>
</tr>
<tr>
<td>__a1</td>
<td>Literal data, read-only</td>
<td>Sign-extended 16-bit offset from address register A1.</td>
<td>64 kB</td>
<td>32-bit</td>
<td>a1 rom</td>
</tr>
<tr>
<td>__a8</td>
<td>Data, reserved for OS</td>
<td>Sign-extended 16-bit offset from address register A8.</td>
<td>64 kB</td>
<td>32-bit</td>
<td>a8 data, a8 rom, a8 bss</td>
</tr>
<tr>
<td>__a9</td>
<td>Data, reserved for OS</td>
<td>Sign-extended 16-bit offset from address register A9.</td>
<td>64 kB</td>
<td>32-bit</td>
<td>a9 data, a9 rom, a9 bss</td>
</tr>
</tbody>
</table>

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

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

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

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

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

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

### Examples using memory qualifiers

To declare a fast accessible integer in directly addressable memory:

```c
int __near Var_in_near;
```

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

```c
__far int *Ptr_in_far;
```
To declare and initialize a string in A0 memory:

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

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

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

Some examples of using pointers with memory qualifiers:

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

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

You cannot use memory qualifiers in structure declarations:

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

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

```c
extern int __near foo;  /* extern int in near memory*/
int __far foo;          /* int in far memory */
```

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

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

The generated assembly would be:

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

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

### 1.2.2. Placing an Object at an Absolute Address: __at() and __atbit()

Just like you can declare a variable in a specific part of memory (using memory qualifiers), you can also place an object at an absolute address in memory.
With the attribute `__at()` you can specify an absolute address. The address is a 32-bit linear address. If you use this keyword on `__bit` objects, the address is a bit address.

**Examples**

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

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

```c
int i __at(0x1000) = 1;
```

**Restrictions**

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

- The argument of the `__at()` attribute must be a constant address expression.
- You can place only global variables at absolute addresses. Parameters of functions, or automatic variables within functions cannot be placed at absolute addresses.
- You cannot place functions at absolute addresses.
- A variable that is declared `extern`, is not allocated by the compiler in the current module. Hence it is not possible to use the keyword `__at()` on an external variable. Use `__at()` at the definition of the variable.
- You cannot place structure members at an absolute address.
- Absolute variables cannot overlap each other. If you declare two absolute variables at the same address, the assembler and/or linker issues an error. The compiler does not check this.

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

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

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

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

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

The following restrictions apply:

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

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

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

### 1.3. Data Type Qualifiers

#### 1.3.1. Circular Buffers: `__circ`

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

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

**Example:**

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

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

**Circular pointers**

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

**Example:**

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

**Indexing**

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

**Example:**

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

The index is not calculated modulo; indexing outside the array boundaries yields undefined results.
**Intrinsic function __initcirc()**

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

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

### 1.3.2. Accessing Hardware from C

**Using Special Function Registers**

It is easy to access Special Function Registers (SFRs) that relate to peripherals from C. The SFRs are defined in a special function register file (*.sfr) as symbol names for use with the compiler. An SFR file contains the names of the SFRs and the bits in the SFRs.

**Example use in C (SFRs from regtc1165.sfr):**

```c
void set_sfr(void)
{
    LBCU_SRC.I |= 0xb32a;  /* access LBCU Service Request
        Control register as a whole */

    LBCU_SRC.B.SRE = 0x1;  /* access SRE bit-field of LBCU
        Service Request Control register */
}
```

You can find a list of defined SFRs and defined bits by inspecting the SFR file for a specific processor. The files are named `regcpu.sfr`, where `cpu` is the CPU specified with the C compiler option `--cpu` (you can use the compiler option `--cpu=tc1165` to compile the example above). The compiler automatically includes this register file, unless you specify option `--no-tasking-sfr`. The files are located in the standard include directory.

**Defining Special Function Registers: __sfrbit16, __sfrbit32**

SFRs are defined in SFR files and are written in C. With the data type qualifiers __sfrbit16 and __sfrbit32 you can declare bit-fields in special function registers.

According to the *TriCore Embedded Applications Binary Interface*, "normal" bit-fields are accessed as `char`, `short` or `int`. Bit-fields are aligned according to the table in [Section 1.1, Data Types](#).

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

When the SFR contains fields, the layout of the SFR is defined by a typedef-ed union. The next example is part of an SFR file and illustrates the declaration of a special function register using the data type qualifier __sfrbit32:
typedef volatile union
{
    struct
    {
        unsigned __sfrbit32 SRPN : 8;  /* Service Priority Number */
        unsigned __sfrbit32      : 2;
        unsigned __sfrbit32 TOS  : 2;  /* Type-of-Service Control */
        unsigned __sfrbit32 SRE  : 1;  /* Service Request Enable Control */
        unsigned __sfrbit32 SRR  : 1;  /* Service Request Flag */
        unsigned __sfrbit32 CLRR : 1;  /* Request Flag Clear Bit */
        unsigned __sfrbit32 SETR : 1;  /* Request Flag Set Bit */
        unsigned __sfrbit32      : 16;
    } B;
    int I;
    unsigned int U;
} LBCU_SRC_type;

Read-only fields can be marked by using the const keyword.

The SFR is defined by a cast to a ‘typedef-ed union’ pointer. The SFR address is given in parenthesis. Read-only SFRs are marked by using the const keyword in the macro definition.

#define LBCU_SRC (*(LBCU_SRC_type*)(0xF87FFEFCu))
    /* LBCU Service Control Register */

Restrictions

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

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

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

1.3.3. Saturation: __sat

When a variable is declared with the type qualifier __sat, all operations on that variable will be performed using saturating arithmetic. When an operation is performed on a plain variable and a __sat variable, the __sat takes precedence, and the operation is done using saturating arithmetic. The type of the result of such an operation also includes the qualifier __sat, so that another operation on the result will also be saturated. In this respect, the behavior of the type qualifier __sat is comparable to the unsigned keyword. You can overrule this behavior by inserting type casts with or without the type qualifier __sat in an expression.

You can only use the type qualifier __sat on type int (fractional types are always saturated).
Care should be taken when combining signed and unsigned integers, since no saturation between signed and unsigned is done.

Examples:

```c
__sat int si         = 0x7FFFFFFF;
int i               = 0x12345;
unsigned int ui     = 0xFFFFFFFF;
```

si + i  // a saturated addition is performed,
// yielding a saturated int

si + ui // a saturated unsigned addition is performed,
// yielding a saturated unsigned int

i + ui  // a normal unsigned addition is performed,
// yielding an unsigned int

### 1.4. Using Assembly in the C Source: __asm()

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

The compiler does not interpret assembly blocks but passes the assembly code to the assembly source file; they are regarded as a black box. So, it is your responsibility to make sure that the assembly block is syntactically correct. Possible errors can only be detected by the assembler.

#### General syntax of the __asm keyword

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

- `instruction_template` Assembly instructions that may contain parameters from the input list or output list in the form: `%parm_nr[.regnum]`
- `%parm_nr[.regnum]` Parameter number in the range 0 .. 9. With the optional `.regnum` you can access an individual register from a register pair or register quad. For example, with register pair d0/d1, .0 selects register d0.
- `output_param_list` [ [ "=&constraint_char"(C_expression)],....]` Says that an output operand is written to before the inputs are read, so this output must not be the same register as any input.
- `input_param_list` [ [ "constraint_char"(C_expression)],....] `&`
- `constraint_char` Constraint character: the type of register to be used for the `C_expression`. See the table below.
**Specifying registers for C variables**

With a *constraint character* you specify the register type for a parameter.

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

<table>
<thead>
<tr>
<th>Constraint character</th>
<th>Type</th>
<th>Operand</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>address register</td>
<td>a0 .. a15</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>address register pair</td>
<td>b2, b4, b6, b12, b14</td>
<td>b2 = pair a2/a3, b4 = a4/a5, ...</td>
</tr>
<tr>
<td>d</td>
<td>data register</td>
<td>d0 .. d15</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>date register pair</td>
<td>e0, e2, ..., e14</td>
<td>e0 = pair d0/d1, e2 = d2/d3, ...</td>
</tr>
<tr>
<td>m</td>
<td>memory</td>
<td>variable</td>
<td>stack or memory operand</td>
</tr>
<tr>
<td>i</td>
<td>immediate value</td>
<td>value</td>
<td></td>
</tr>
<tr>
<td>number</td>
<td>type of operand it is</td>
<td>same as %number</td>
<td>Input constraint only. The <em>number</em> must refer to an output parameter. Indicates that %<em>number</em> and <em>number</em> are the same register.</td>
</tr>
</tbody>
</table>

**Loops and conditional jumps**

The compiler does not detect loops with multiple **asm()** statements or (conditional) jumps across **asm()** statements and will generate incorrect code for the registers involved.

If you want to create a loop with **asm()**, the whole loop must be contained in a single **asm()** statement. The same counts for (conditional) jumps. As a rule of thumb, all references to a label in an **asm()** statement must be in that same statement.

**Example 1: no input or output**

A simple example without input or output parameters. You can use any instruction or label. When it is required that a sequence of **asm()** statements generates a contiguous sequence of instructions, then they can be best combined to a single **asm()** statement. Compiler optimizations can insert instruction(s) in between **asm()** statements. Use newline characters `\n` to continue on a new line in a **asm()** statement. For multi-line output, use tab characters `\t` to indent instructions.

```c
__asm( "nop\n" 
     "\tnop" );
```
**Example 2: using output parameters**

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

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

Generated assembly code:

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

**Example 3: using input parameters**

Assign a variable to a data register. A data register is chosen for the parameter because of the constraint \(d\); the compiler decides which register is best to use. The \%0 in the instruction template is replaced with the name of this register. The compiler generates code to move the input variable to the input register. Because there are no output parameters, the output parameter list is empty. Only the colon has to be present.

```c
int in;
void initreg( void )
{
    __asm( "MOV  D0,%0"
            : "d" (in) );
}
```

Generated assembly code:

```
ld.w  d15,in
MOV    D0,d15
```

**Example 4: using input and output parameters**

Multiply two C variables and assign the result to a third C variable. Data type registers are necessary for the input and output parameters (constraint \(d\), \%0 for `out`, \%1 for `in1`, \%2 for `in2` in the instruction template). The compiler generates code to move the input expressions into the input registers and to assign the result to the output variable.

```c
int in1, in2, out;

void multiply32( void )
{
    __asm( "mul %0, %1, %2"
            : "=d" (out) );
}
```
Example 5: reserving registers

Sometimes an instruction knocks out certain specific registers. The most common example of this is a function call, where the called function is allowed to do whatever it likes with some registers. If this is the case, you can list specific registers that get clobbered by an operation after the inputs.

Same as Example 4, but now register d0 is a reserved register. You can do this by adding a reserved register list (: "d0"). As you can see in the generated assembly code, register d0 is not used (the first register used is d1).

```c
int in1, in2, out;

void multiply32( void )
{
    __asm ( "mul %0, %1, %2"
           : "=d" (out)
             : "d" (in1), "d" (in2)
             : "d0" );
}
```

Generated assembly code:

```
ld.w   d15,in1
ld.w   d1,in2
mul    d15, d15, d1
st.w   out,d15
```

Example 6: input and output are the same

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

```c
int outvar;

void invert(int invar)
{
    __asm ("not %0": "=d"(outvar): "0"(invar) );
}
```
void main(void)
{
    invert(255);
}

Generated assembly code:

invert:
    not    d4
    st.w   ovar,d4

main:
    mov    d4,#255
    j invert

Example 7: accessing individual registers in a register pair

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

int out1, out2;

void foo(double din)
{
    __asm ("ld.w %0, %2.0\n"
            "\tlld.w %1, %2.1":"=d"(out1),"=d"(out2):"e"(din) );
}

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

    ld.w   d15, d4
    ld.w   d0, e4,1 ; note that e4,1 corresponds to d5
    st.w   out1,d15
    st.w   out2,d0

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

1.5. Pragmas to Control the Compiler

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

The syntax is:

#pragma pragma-spec [ON | OFF | DEFAULT | RESTORE]
or:

`_Pragma( "pragma-spec [ON | OFF | DEFAULT | RESTORE]" )`

on switches the pragma on, this is the same as without arguments. off switches the pragma off. default sets the pragma to the initial value. restore restores the previous value of the pragma.

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

**alias symbol=defined_symbol**

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

**align value | restore**

Increase the alignment of object of four bytes or longer.

See C compiler option **--align**.

**clear / noclear**

By default, uninitialized global or static variables are cleared to zero on startup. With pragma noclear, this step is skipped. Pragma clear resumes normal behavior. This pragma applies to constant data as well as non-constant data.

**CPU_functional_problem / DMU_functional_problem**

Use software workarounds for the specified functional problem.

See C compiler option **--silicon-bug**.

**default_a0_size [value] [default | restore]**

With this pragma you can specify a threshold value for `__a0` allocation.

See C compiler option **--default-a0-size (-Z)**.

**default_a1_size [value] [default | restore]**

With this pragma you can specify a threshold value for `__a1` allocation.

See C compiler option **--default-a1-size (-Y)**.

**default_near_size [value] [default | restore]**

With this pragma you can specify a threshold value for `__near` allocation.

See C compiler option **--default-near-size (-N)**.
extension isuffix [on | off | default | restore]

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

float 0.5i

extern symbol

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

With this pragma you can force an external reference (.EXTERN assembler directive), even when the symbol is not used in the module.

for_constant_data_use_memory memory
forExtern_data_use_memory memory
for_initialized_data_use_memory memory
for_uninitialized_data_use_memory memory

Use the specified memory for the type of data mentioned in the pragma name. You can specify the following memories: near, far, a0, a8 or a9. For pragma for_constant_data_use_memory you can also specify the a1 memory.

This pragma overrules the pragmas default_a0_size, default_a1_size, default_near_size, and the memory qualifiers __near and __far.

immediate_in_code

With this pragma you force the compiler to encode all immediate values into instructions.

See C compiler option --immediate-in-code.

indirect

Generates code for indirect function calling.

See C compiler option --indirect.

indirect-runtime

Generates code for indirect calls to run-time functions.

See C compiler option --indirect-runtime.

inline / noinline / smartinline

See Section 1.8.3, Inlining Functions: inline.
macro / nomacro

Turns macro expansion on or off. By default, macro expansion is enabled.

message "message" ...

Print the message string(s) on standard output.

object_comment "string" ...

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

See C compiler option --object-comment.

optimize [flags | default | restore] / endoptimize

You can overrule the C compiler option --optimize for the code between the pragmas optimize and endoptimize. The pragma works the same as C compiler option --optimize.

See Section 4.6, Compiler Optimizations.

pack 0 | 2

Specifies packing of structures. See Section 1.1.3, Packed Data Types.

profiling [on | off | default | restore]

If profiling is enabled on the command line (C compiler option --profile), you can disable part of your source code for profiling with the pragmas profiling off and profiling.

protect [on | off | default | restore] / endprotect

With these pragmas you can protect sections against linker optimizations. This excludes a section from unreferenced section removal and duplicate section removal by the linker. endprotect restores the default section protection.

runtime [flags | default | restore]

With this pragma you can control the generation of additional code to check for a number of errors at run-time. The pragma argument syntax is the same as for the arguments of the C compiler option --runtime. You can use this pragma to control the run-time checks for individual statements. In addition, objects declared when the "bounds" sub-option is disabled are not bounds checked. The "malloc" sub-option cannot be controlled at statement level, as it only extracts an alternative malloc implementation from the library.

section all "name"
section type "name"
section_name_with_module
section_name_with_symbol
section_per_data_object

Changes section names. See Section 1.9, Compiler Generated Sections and C compiler option --rename-sections for more information.

section code_init | const_init | vector_init

At startup copies corresponding sections to RAM for initialization.

section data_overlay

Allow overlaying data sections.

source [on | off | default | restore] / nosource

With these pragmas you can choose which C source lines must be listed as comments in assembly output.

See C compiler option --source.

stdinc [on | off | default | restore]

This pragma changes the behavior of the #include directive. When set, the C compiler options --include-directory and --no-stdinc are ignored.

switch auto | jump_tab | linear | lookup | restore

With these pragmas you can overrule the C compiler chosen switch method.

See Section 1.7, Switch Statement and C compiler option --switch.

tradeoff level [default | restore]

Specify tradeoff between speed (0) and size (4). See C compiler option --tradeoff

user_mode user-0 | user-1 | kernel | default | restore

With this pragma you specify the user mode (I/O privilege mode) the TriCore runs in.

See C compiler option --user-mode.

warning [number,...] [default | restore]

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

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

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

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

1.6. Predefined Preprocessor Macros

The TASKING C compiler supports the predefined macros as defined in the table below. The macros are useful to create conditional C code.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BUILD</strong></td>
<td>Identifies the build number of the compiler, composed of decimal digits for the build number, three digits for the major branch number and three digits for the minor branch number. For example, if you use build 1.22.1 of the compiler, <strong>BUILD</strong> expands to 1022001. If there is no branch number, the branch digits expand to zero. For example, build 127 results in 127000000.</td>
</tr>
<tr>
<td><strong>CTC</strong></td>
<td>Identifies the compiler. You can use this symbol to flag parts of the source which must be recognized by the TASKING ctc compiler only. It expands to 1.</td>
</tr>
<tr>
<td><strong>CPU</strong></td>
<td>Expands to a string with the CPU supplied with the option --cpu. When no --cpu is supplied, this symbol is not defined.</td>
</tr>
<tr>
<td><strong>DATE</strong></td>
<td>Expands to the compilation date: &quot;mmm dd yyyy&quot;.</td>
</tr>
<tr>
<td><strong>DOUBLE_FP</strong></td>
<td>Expands to 1 if you did not use option --no-double (Treat ‘double’ as ‘float’), otherwise unrecognized as macro.</td>
</tr>
<tr>
<td><strong>DSPC</strong></td>
<td>Indicates conformation to the DSP-C standard. It expands to 1.</td>
</tr>
<tr>
<td><strong>DSPC_VERSION</strong></td>
<td>Expands to the decimal constant 200001L.</td>
</tr>
<tr>
<td><strong>FILE</strong></td>
<td>Expands to the current source file name.</td>
</tr>
<tr>
<td><strong>FPU</strong></td>
<td>Expands to 1 if you used option --fpu-present (Use hardware floating-point instructions), otherwise unrecognized as macro.</td>
</tr>
<tr>
<td><strong>LINE</strong></td>
<td>Expands to the line number of the line where this macro is called.</td>
</tr>
<tr>
<td><strong>PROF_ENABLE</strong></td>
<td>Expands to 1 if profiling is enabled, otherwise expands to 0.</td>
</tr>
<tr>
<td><strong>REVISION</strong></td>
<td>Expands to the revision number of the compiler. Digits are represented as they are; characters (for prototypes, alphas, betas) are represented by -1. Examples: v1.0r1 -&gt; 1, v1.0rb -&gt; -1</td>
</tr>
</tbody>
</table>
### Macros

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__SFRFILE__(cpu)</code></td>
<td>This macro expands to the filename of the used SFR file, including the <code>&lt; &gt;</code>. The <code>cpu</code> is the argument of the macro. For example, if <code>--cpu=tc1165</code> is specified, the macro <code>__SFRFILE__(__CPU__)</code> expands to <code>__SFRFILE__(tc1165)</code>, which expands to <code>&lt;regtc1165.sfr&gt;</code>.</td>
</tr>
<tr>
<td><code>__SFRFILE_WANTED__</code></td>
<td>Expands to 1 if TASKING .sfr files are used. Not defined when option <code>--no-tasking-sfr</code> is used.</td>
</tr>
<tr>
<td><code>__SINGLE_FP__</code></td>
<td>Expands to 1 if you used option <code>--no-double</code> (Treat ‘double’ as ‘float’), otherwise unrecognized as macro.</td>
</tr>
<tr>
<td><code>__STDC__</code></td>
<td>Identifies the level of ANSI standard. The macro expands to 1 if you set option <code>--language</code> (Control language extensions), otherwise expands to 0.</td>
</tr>
<tr>
<td><code>__STDC_HOSTED__</code></td>
<td>Always expands to 0, indicating the implementation is not a hosted implementation.</td>
</tr>
<tr>
<td><code>__STDC_VERSION__</code></td>
<td>Identifies the ISO-C version number. Expands to 199901L for ISO C99 or 199409L for ISO C90.</td>
</tr>
<tr>
<td><code>__TASKING__</code></td>
<td>Identifies the compiler as a TASKING compiler. Expands to 1 if a TASKING compiler is used.</td>
</tr>
<tr>
<td><code>__TC131__</code></td>
<td>Expands to 1 if the TriCore 1.3.1 architecture is selected (option <code>--core=tc1.3.1</code>).</td>
</tr>
<tr>
<td><code>__TC2__</code></td>
<td>Expands to 1 if the TriCore 2 architecture is selected (option <code>--core=tc2</code>).</td>
</tr>
<tr>
<td><code>__TIME__</code></td>
<td>Expands to the compilation time: “hh:mm:ss”</td>
</tr>
<tr>
<td><code>__VERSION__</code></td>
<td>Identifies the version number of the compiler. For example, if you use version 2.1r1 of the compiler, <code>__VERSION__</code> expands to 2001 (dot and revision number are omitted, minor version number in 3 digits).</td>
</tr>
</tbody>
</table>

---

### Example

```c
#define __FPU__
/* this part is only valid if an FPU is present */
...
#endif
```

### 1.7. Switch Statement

The TASKING C compiler supports three ways of code generation for a switch statement: a jump chain (linear switch), a jump table or a lookup table.

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

By default, the compiler will automatically choose the most efficient switch implementation based on code and data size and execution speed. With the **C compiler option --tradeoff** you can tell the compiler to emphasis more on speed than on ROM size.
Especially for large switch statements, the jump table approach executes faster than the lookup table approach. Also the jump table has a predictable behavior in execution speed: independent of the switch argument, every case is reached in the same execution time. However, when the case labels are distributed far apart, the jump table becomes sparse, wasting code memory. The compiler will not use the jump table method when the waste becomes excessive.

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

**How to overrule the default switch method**

You can overrule the compiler chosen switch method by using a pragma:

```c
#pragma switch linear force jump chain code
#pragma switch jumptab force jump table code
#pragma switch lookup force lookup table code
#pragma switch auto let the compiler decide the switch method used (this is the default)
#pragma switch restore restore previous switch method
```

The switch pragmas must be placed before the `switch` statement. Nested `switch` statements use the same switch method, unless the nested `switch` is implemented in a separate function which is preceded by a different switch pragma.

Example:

```c
/* place pragma before function body */

#pragma switch jumptab

void test(unsigned char val)
{
    /* function containing the switch */
    switch (val)
    {
        /* use jump table */
    }
}
```

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

### 1.8. Functions

#### 1.8.1. Calling Convention

**Parameter Passing**

A lot of execution time of an application is spent transferring parameters between functions. The fastest parameter transport is via registers. Therefore, function parameters are first passed via registers. If no more registers are available for a parameter, the compiler pushes parameters on the stack.
Registers available for parameter passing are D4, D5, E4, D6, D7, E6, A4, A5, A6, A7. Up to 4 arithmetic types and 4 pointers can be passed this way. A 64-bit argument is passed in an even/odd data register pair. Parameter registers skipped because of alignment for a 64-bit argument are used by subsequent 32-bit arguments. Any remaining function arguments are passed on the stack. Stack arguments are pushed in reversed order, so that the first one is at the lowest address. On function entry, the first stack parameter is at the address (SP+0).

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

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

Examples:

```c
void func1( int i, char *p, char c );      /* D4 A4 D5 */
void func2( int i1, double d, int i2 );    /* D4 E6 D5 */
void func3( char c1, char c2, char c3[] ); /* D4 D5 A4 */
void func4( double d1, int i1, double d2, int i2 );
    /* E4 D6 stack D7 */
```

**Function Return Values**

The C compiler uses registers to store C function return values, depending on the function return types.

<table>
<thead>
<tr>
<th>Return Type</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic 32 bit</td>
<td>D2</td>
</tr>
<tr>
<td>Arithmetic 64 bit</td>
<td>D2/D3 (E2)</td>
</tr>
<tr>
<td>Pointer</td>
<td>A2</td>
</tr>
<tr>
<td>Circular pointer</td>
<td>A2/A3</td>
</tr>
</tbody>
</table>

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

**Stack model: __stackparm**

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

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

Example:

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

    SFP     fp_stack;
    RFP     fp_reg;

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

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

**Function Calling Modes: __indirect**

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

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

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

**1.8.2. Register Usage**

The C compiler uses the data registers and address registers according to the convention given in the following table.

<table>
<thead>
<tr>
<th>Register</th>
<th>Usage</th>
<th>Register</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>D0</td>
<td>E0</td>
<td>D1</td>
<td>scratch</td>
</tr>
<tr>
<td></td>
<td>scratch</td>
<td>A0</td>
<td>global</td>
</tr>
<tr>
<td>A0</td>
<td></td>
<td>A1</td>
<td>global</td>
</tr>
</tbody>
</table>
1.8.3. Inlining Functions: inline

With the C compiler option `--optimize=+inline`, the C compiler automatically inlines small functions in order to reduce execution time (smart inlining). The compiler inserts the function body at the place the function is called. The C compiler decides which functions will be inlined. You can overrule this behavior with the two keywords `inline` (ISO-C) and `__noinline`.

With the `inline` keyword you force the compiler to inline the specified function, regardless of the optimization strategy of the compiler itself:

```c
inline unsigned int abs(int val)
{
    unsigned int abs_val = val;
    if (val < 0) abs_val = -val;
    return abs_val;
}
```

If a function with the keyword `inline` is not called at all, the compiler does not generate code for it.

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

With the `__noinline` keyword, you prevent a function from being inlined:

```c
__noinline unsigned int abs(int val)
{
    unsigned int abs_val = val;
```
Using pragmas: inline, noinline, smartinline

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

```c
#pragma inline
unsigned int abs(int val) {
    unsigned int abs_val = val;
    if (val < 0) abs_val = -val;
    return abs_val;
}
```

```c
#pragma noinline
void main( void ) {
    int i;
    i = abs(-1);
}
```

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

With the #pragma noinline/#pragma smartinline you can temporarily disable the default behavior that the C compiler automatically inlines small functions when you turn on the C compiler option --optimize=+inline.

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

Combining inline with __asm to create intrinsic functions

With the keyword __asm it is possible to use assembly instructions in the body of an inline function. Because the compiler inserts the (assembly) body at the place the function is called, you can create your own intrinsic function. See Section 1.8.5, Intrinsic Functions.

1.8.4. Interrupt and Trap Functions

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

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

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

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

Overview of function qualifiers

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

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

There is one special type of trap function which you can call manually, the system call exception (trap class 6). See Section 1.8.4.3, Defining a Trap Service Routine Class 6: __syscallfunc().

```c
__syscallfunc()
```

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

```c
__enable_      __bisr_()
```

1.8.4.1. Defining an Interrupt Service Routine: __interrupt(), __interrupt_fast()

With the function type qualifier __interrupt() you can declare a function as an interrupt service routine. The function type qualifier __interrupt() takes one interrupt vector (0..255) as argument.

Interrupt functions cannot return anything and must have a void argument type list:

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

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

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

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

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

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

```c
int c;

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

**Fast interrupts**

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

**1.8.4.2. Defining a Trap Service Routine: __trap(), __trap_fast()**

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

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

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

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

The next table shows the classes supported by TriCore.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class 0</td>
<td>Reset</td>
</tr>
<tr>
<td>Class 1</td>
<td>Internal Protection Traps</td>
</tr>
<tr>
<td>Class 2</td>
<td>Instruction Errors</td>
</tr>
<tr>
<td>Class 3</td>
<td>Context Management</td>
</tr>
<tr>
<td>Class 4</td>
<td>System Bus and Peripheral Errors</td>
</tr>
<tr>
<td>Class 5</td>
<td>Assertion Traps</td>
</tr>
<tr>
<td>Class</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>Class 6</td>
<td>System Call</td>
</tr>
<tr>
<td>Class 7</td>
<td>Non-Maskable Interrupt</td>
</tr>
</tbody>
</table>

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

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

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

**Fast traps**

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

### 1.8.4.3. Defining a Trap Service Routine Class 6: `__syscallfunc()`

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

```
__syscallfunc(TIN)
```

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

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

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

**Example**

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

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

int x;
```
TASKING VX-toolset for TriCore User Guide
void main( void )
{
x = syscall_a(1,2);
x = syscall_b(4,3);
}

// causes a trap class 6 with TIN = 1
// causes a trap class 6 with TIN = 2

int __trap( 6 ) trap6( int a, int b ) // trap class 6 handler
{
int tin;
__asm("mov %0,d15" : "=d"(tin)); // put d15 in C variable tin
switch( tin )
{
case 1:
a += b;
break;
case 2:
a -= b;
break;
default:
break;
}
return a;
}

1.8.4.4. Enabling Interrupt Requests: __enable_, __bisr_()
Enabling interrupt service requests
During the execution of an interrupt service routine or trap service routine, the system blocks the CPU
from taking further interrupt requests. You can immediately re-enable the system to accept interrupt
requests:
__interrupt(vector) __enable_ isr( void )
__trap(class) __enable_ tsr( void )
The compiler generates an enable instruction as first instruction in the routine. The enable instruction
sets the interrupt enable bit (ICR.IE) in the interrupt control register.
You can also generate the enable instruction with the intrinsic function __enable(), but it is not
guaranteed that it will be the first instruction in the routine.
Enabling interrupt service requests and setting CPU priority number
The function qualifier __bisr_() also re-enables the system to accept interrupt requests. In addition,
the current CPU priority number (CCPN) in the interrupt control register is set:
__interrupt(vector) __bisr_(CCPN) isr( void )
__trap(class) __bisr_(CCPN) tsr( void )
The argument CCPN is a number between 0 and 255. The system accepts all interrupt requests that
have a higher pending interrupt priority number (PIPN) than the current CPU priority number. So, if the

32


CPU priority number is set to 0, the system accepts all interrupts. If it is set to 255, no interrupts are accepted.

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

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

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

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

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

### 1.8.5. Intrinsic Functions

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

The compiler always inlines the corresponding assembly instructions in the assembly source (rather than calling it as a function). This avoids parameter passing and register saving instructions which are normally necessary during function calls.

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

You can use intrinsic functions in C as if they were ordinary C (library) functions. All intrinsics begin with a double underscore character (`__`).

The following example illustrates the use of an intrinsic function and its resulting assembly code.

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

The resulting assembly code is inlined rather than being called:

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

The intrinsics cover the following subjects:

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

Writing your own intrinsic function

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

1. First write a function with assembly in the body using the keyword `__asm()`. See Section 1.4, Using Assembly in the C Source: `__asm()`

2. Next make sure that the function is inlined rather than being called. You can do this with the function qualifier `inline`. This qualifier is discussed in more detail in Section 1.8.3, Inlining Functions: `inline`.

```c
int a, b, result;

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

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

Generated assembly code:

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

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

1.8.5.1. Minimum and Maximum of (Short) Integers

The next table provides an overview of the intrinsic functions that return the minimum or maximum of a signed integer, unsigned integer or short integer.
1.8.5.2. Fractional Arithmetic Support

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

Conversion of fractional values

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>long __mulfractlong( __fract,long )</td>
<td>Integer part of the multiplication of a __fract and a long</td>
</tr>
<tr>
<td>__sfract __round16( __fract )</td>
<td>Convert __fract to __sfract</td>
</tr>
<tr>
<td>__fract __getfract( __accum )</td>
<td>Convert __accum to __fract</td>
</tr>
<tr>
<td>short __clssf( __sfract )</td>
<td>Count the consecutive number of bits that have the same value as bit 15 of an __sfract</td>
</tr>
<tr>
<td>__sfract __shasfracts( __sfract, int )</td>
<td>Left/right shift of an __sfract</td>
</tr>
<tr>
<td>__fract __shafracts( __fract, int )</td>
<td>Left/right shift of an __fract</td>
</tr>
<tr>
<td>__laccum __shaaccum( __laccum, int )</td>
<td>Left/right shift of an __laccum</td>
</tr>
<tr>
<td>__sfract __mac_sf( __sfract a, __sfract b, __sfract c)</td>
<td>Multiply-add __sfract. Returns ( a + b * c )</td>
</tr>
<tr>
<td>__sfract __mac_r_sf( __sfract, __sfract, __sfract )</td>
<td>Multiply-add with rounding. Returns the rounded result of ( a + b * c )</td>
</tr>
<tr>
<td>__sfract __u16_to_sfract( unsigned short integer )</td>
<td>Convert unsigned short to __sfract</td>
</tr>
<tr>
<td>__sfract __s16_to_sfract( signed short integer )</td>
<td>Convert signed short to __sfract</td>
</tr>
<tr>
<td>unsigned short int __sfract_to_u16( __sfract )</td>
<td>Convert __sfract to unsigned short</td>
</tr>
<tr>
<td>signed short int __sfract_to_s16( __sfract )</td>
<td>Convert __sfract to signed short</td>
</tr>
</tbody>
</table>

1.8.5.3. Packed Data Type Support

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

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

Extract values from packed data types

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

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

<table>
<thead>
<tr>
<th>Description</th>
<th>Intrinsic Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idem, but unsigned</td>
<td>unsigned short __getuhw1( unsigned __packhw * )</td>
</tr>
<tr>
<td>Extract second short from a __packhw</td>
<td>short __gethw2( __packhw * )</td>
</tr>
<tr>
<td>Idem, but unsigned</td>
<td>unsigned short __getuhw2( unsigned __packhw * )</td>
</tr>
</tbody>
</table>

### Insert values into packed data types

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

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__packb __insertbyte1( __packb, char )</td>
<td>Inserting char into first byte of a __packb</td>
</tr>
<tr>
<td>unsigned __packb __insertbyte1( unsigned __packb, unsigned char )</td>
<td>Idem, but unsigned</td>
</tr>
<tr>
<td>__packb __insertbyte2( __packb, char )</td>
<td>Inserting char into second byte of a __packb</td>
</tr>
<tr>
<td>unsigned __packb __insertbyte2( unsigned __packb, unsigned char )</td>
<td>Idem, but unsigned</td>
</tr>
<tr>
<td>__packb __insertbyte3( __packb, char )</td>
<td>Inserting char into third byte of a __packb</td>
</tr>
<tr>
<td>unsigned __packb __insertbyte3( unsigned __packb, unsigned char )</td>
<td>Idem, but unsigned</td>
</tr>
<tr>
<td>__packb __insertbyte4( __packb, char )</td>
<td>Inserting char into fourth byte of a __packb</td>
</tr>
<tr>
<td>unsigned __packb __insertbyte4( unsigned __packb, unsigned char )</td>
<td>Idem, but unsigned</td>
</tr>
<tr>
<td>__packhw __inserth1( __packhw, short )</td>
<td>Inserting short into first halfword of a __packhw</td>
</tr>
<tr>
<td>unsigned __packhw __insertuhw1( unsigned __packhw, unsigned short )</td>
<td>Idem, but unsigned</td>
</tr>
<tr>
<td>__packhw __inserth2( __packhw, short )</td>
<td>Inserting short into second halfword of a __packhw</td>
</tr>
<tr>
<td>unsigned __packhw __insertuhw2( unsigned __packhw, unsigned short )</td>
<td>Idem, but unsigned</td>
</tr>
<tr>
<td>void __setbyte1( __packb *, char )</td>
<td>Inserting char into first byte of a __packb</td>
</tr>
<tr>
<td>void __setbyte1( unsigned __packb *, unsigned char )</td>
<td>Idem, but unsigned</td>
</tr>
<tr>
<td>void __setbyte2( __packb *, char )</td>
<td>Inserting char into second byte of a __packb</td>
</tr>
<tr>
<td>void __setbyte2( unsigned __packb *, unsigned char )</td>
<td>Idem, but unsigned</td>
</tr>
<tr>
<td>void __setbyte3( __packb *, char )</td>
<td>Inserting char into third byte of a __packb</td>
</tr>
<tr>
<td>void __setbyte3( unsigned __packb *, unsigned char )</td>
<td>Idem, but unsigned</td>
</tr>
<tr>
<td>void __setbyte4( __packb *, char )</td>
<td>Inserting char into fourth byte of a __packb</td>
</tr>
<tr>
<td>void __setbyte4( unsigned __packb *, unsigned char )</td>
<td>Idem, but unsigned</td>
</tr>
<tr>
<td>void __setuhw1( __packhw *, short )</td>
<td>Inserting short into first halfword of a __packhw</td>
</tr>
<tr>
<td>void __setuhw1( unsigned __packhw *, unsigned short )</td>
<td>Idem, but unsigned</td>
</tr>
</tbody>
</table>
Combine packed data types into a packed word

The next table provides an overview of the intrinsic functions to combine the value of packed data types into a packed word. The packed word (64-bit) is represented by the data type `long long`. You can combine two `__packb` (2 x 4 bytes) into a `long long` or two `__packhw` (2 x 2 halfwords) into a `long long`.

These intrinsics are only supported for the TriCore2 (`--core=tc2`).

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

Calculate absolute values of packed data type values

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

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

Calculate minimum packed data type values

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

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

1.8.5.4. Interrupt Handling

The next table provides an overview of the intrinsic functions to read or set interrupt handling.
Intrinsic Function                  Description
---                                ---
__enable ( void )                  Enable interrupts immediately at function entry
__disable ( void )                 Disable interrupts. Only supported for TriCore1.
__disable_and_save ( void )        Disable interrupts and return previous interrupt state (enabled or disabled). Only supported for TriCore2 (--core=tc2).
__restore ( int )                  Restore interrupt state. Only supported for TriCore2 (--core=tc2).
__bisr ( int )                     Set CPU priority number [0..512] and enable interrupts immediately at function entry
__syscall ( int )                  Call a system call function number

### 1.8.5.5. Insert Single Assembly Instruction

The next table provides an overview of the intrinsic functions that you can use to insert a single assembly instruction. You can also use inline assembly but these intrinsics provide a shorthand for frequently used assembly instructions.

See Section 1.4, *Using Assembly in the C Source: __asm()*.

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__debug( void )</td>
<td>Insert DEBUG instruction</td>
</tr>
<tr>
<td>__dsync( void )</td>
<td>Insert DSYNC instruction</td>
</tr>
<tr>
<td>__isync( void )</td>
<td>Insert ISYNC instruction</td>
</tr>
<tr>
<td>__svlcx( void )</td>
<td>Insert SVLCX instruction</td>
</tr>
<tr>
<td>__rslcx( void )</td>
<td>Insert RSLCX instruction</td>
</tr>
<tr>
<td>__nop( void )</td>
<td>Insert NOP instruction</td>
</tr>
<tr>
<td>volatile __ldmst( unsigned int *address, unsigned int mask, unsigned int value)</td>
<td>Insert LDMST instruction</td>
</tr>
<tr>
<td>volatile __swap( unsigned int *place, unsigned int value)</td>
<td>Insert SWAP instruction</td>
</tr>
</tbody>
</table>

### 1.8.5.6. Register Handling

#### Access control registers

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

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

#### Perform register value operations

The next table provides an overview of the intrinsic functions that operate on a register and return a value in another register.
### 1.8.5.7. Insert / Extract Bit-fields and Bits

#### Insert / extract bit-fields

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

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

#### Atomic load-modify-store

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

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

#### Store a single bit

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

This intrinsic is implemented as a macro definition which uses the __imaskldmst() intrinsic:
#define __putbit ( value, address, bitoffset ) __imaskldmst
( address, value, bitoffset, 1 )

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void __putbit( int value, int* address, int bitoffset )</td>
<td>Store a single bit</td>
</tr>
</tbody>
</table>

**Load a single bit**

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

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

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

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned int __getbit( int* address, int bitoffset )</td>
<td>Load a single bit</td>
</tr>
</tbody>
</table>

### 1.8.5.8. Miscellaneous Intrinsic Functions

**Multiply and scale back**

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int __mulsc( int a, int b, int offset )</td>
<td>Multiply two 32-bit numbers to an intermediate 64-bit result, and scale back the result to 32 bits. To scale back the result, 32 bits are extracted from the intermediate 64-bit result: bit 63-offset to bit 31-offset.</td>
</tr>
</tbody>
</table>

**Swap mask**

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void __swapmsk ( int value, int mask, int* memory )</td>
<td>Exchanges the values of value and memory, but only those bits that are allowed by mask. Before the swapmsk instruction is generated, the parameters value and mask are moved into a double register. Only supported for TriCore2 (--core=tc2).</td>
</tr>
</tbody>
</table>

**Initialize circular pointer**

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__circ void * __initcirc( void* buf, unsigned short bufsize, unsigned short byteindex )</td>
<td>Initialize a circular pointer with a dynamically allocated buffer at run-time. See also Section 1.3.1, Circular Buffers: __circ.</td>
</tr>
</tbody>
</table>

**Rotate left/right**

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__rol( unsigned int operand, unsigned in count)</td>
<td>Rotate operand left count times. The bits that are shifted out are inserted at the right side (bit 31 is shifted to bit 0).</td>
</tr>
</tbody>
</table>
Intrinsic Function | Description
--- | ---
__rol( unsigned int \( operand \), unsigned int \( count \) ) | Rotate \( operand \) right \( count \) times. The bits that are shifted out are inserted at the left side (bit 0 is shifted to bit 31).

### Intrinsic Function List

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void * volatile __alloc( __size_t ( size ) );</td>
<td>Allocate memory on the stack. Returns a pointer to memory of ( size ) bytes length. Returns NULL if there is not enough space left.</td>
</tr>
<tr>
<td>char * <strong>dotdotdot</strong>( void );</td>
<td>Variable argument ‘...’ operator. Used in library function ( \text{va}_\text{start}() ). Returns the stack offset to the variable argument list.</td>
</tr>
<tr>
<td>void volatile __free( void *p );</td>
<td>Deallocates the memory pointed to by ( p ). ( p ) must point to memory earlier allocated by a call to __alloc().</td>
</tr>
<tr>
<td>__codeptr __get_return_address( void );</td>
<td>Used by the compiler for profiling when you compile with the option --profile. Returns the return address of a function.</td>
</tr>
</tbody>
</table>

### 1.9. Compiler Generated Sections

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

[section_type_prefix/module_name/symbol_name]

The prefix depends on the type of the section and determines if the section is initialized, constant or uninitialized and which addressing mode is used. The symbol_name is either the name of an object or the name of a function.

The following table lists the section types and name prefixes.

<table>
<thead>
<tr>
<th>Section type</th>
<th>Name prefix</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>code</td>
<td>.text</td>
<td>program code</td>
</tr>
<tr>
<td>neardata</td>
<td>.zdata</td>
<td>initialized __near data</td>
</tr>
<tr>
<td>fardata</td>
<td>.data</td>
<td>initialized __far data</td>
</tr>
<tr>
<td>nearrom</td>
<td>.zrodata</td>
<td>constant __near data</td>
</tr>
<tr>
<td>farrom</td>
<td>.rodata</td>
<td>constant __far data</td>
</tr>
<tr>
<td>nearbss</td>
<td>.zbss</td>
<td>uninitialized __near data (cleared)</td>
</tr>
<tr>
<td>farbss</td>
<td>.bss</td>
<td>uninitialized __far data (cleared)</td>
</tr>
<tr>
<td>nearnoclear</td>
<td>.zbss</td>
<td>uninitialized __near data</td>
</tr>
<tr>
<td>farnoclear</td>
<td>.bss</td>
<td>uninitialized __far data</td>
</tr>
<tr>
<td>a0data</td>
<td>.sdata</td>
<td>initialized __a0 data</td>
</tr>
<tr>
<td>a0bss</td>
<td>.sbss</td>
<td>uninitialized __a0 data (cleared)</td>
</tr>
<tr>
<td>a1rom</td>
<td>.ldata</td>
<td>constant __a1 data</td>
</tr>
</tbody>
</table>
#pragma section type "name"

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

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

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

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

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

#pragma section all "name"

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

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

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

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

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

On the command line you can use the C compiler option --rename-sections[=name].

#pragma section_name_with_module

All section renaming pragmas will use a renaming scheme like:

```
section_type_prefix.module_namepragma_value
```

#pragma section_name_with_symbol

All section renaming pragmas will use a renaming scheme like:
section_type_prefix.symbol_name.pragma_value

See also C compiler option --section-name-with-symbol.

Examples

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

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

1.9.2. Influence Section Definition

The following pragmas also influence the section definition:

#pragma section code_init

Code sections are copied from ROM to RAM at program startup.

#pragma section const_init

Sections with constant data are copied from ROM to RAM at program startup.

#pragma section vector_init

Sections with interrupts and trap vectors are copied from ROM to RAM at program startup.

#pragma section data_overlay

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

#pragma section_per_data_object

All data objects get their own section, using a naming scheme like:

section_type_prefix.module_name.symbol_name

See also C compiler option --section-per-data-object.

1.9.3. Change Section Alignment

By default the compiler generates a 4-byte alignment for sections, this is done because the default initialization code does not know where code ends up and hardware could require a word aligned read/access to the memory. With the C compiler option --no-default-section-alignment sections are no longer forced to a 4-byte alignment.
Please note that this means that you will have to use your own initialization code! Furthermore you will have to remove the `copy_unit = 4` part of the `copytable` declaration within the LSL file.
Chapter 2. C++ Language

The TASKING C++ compiler (cptc) offers a new approach to high-level language programming for the TriCore family. The C++ compiler accepts the C++ language as defined by the ISO/IEC 14882:1998 standard and modified by TC1 for that standard. It also accepts the language extensions of the C compiler (see Chapter 1, C Language).

This chapter describes the C++ language implementation and some specific features.

Note that the C++ language itself is not described in this document. For more information on the C++ language, see


2.1. C++ Language Extension Keywords

The C++ compiler supports the same language extension keywords as the C compiler. When option --strict is used, the extensions will be disabled.

Additionally the following language extensions are supported:

**attributes**

Attributes, introduced by the keyword __attribute__, can be used on declarations of variables, functions, types, and fields. The alias, aligned, cdecl, const, constructor, deprecated, destructor, format, format_arg, init_priority, malloc, mode, naked, no_check_memory_usage, no_instrument_function, nocommon, noreturn, packed, pure, section, sentinel, stdcall, transparent_union, unused, used, visibility, volatile, and weak attributes are supported.

**pragmas**

The C++ compiler supports the same pragmas as the C compiler and some extra pragmas as explained in Section 2.7, Pragmas to Control the C++ Compiler. Pragmas give directions to the code generator of the compiler.

2.2. C++ Dialect Accepted

The C++ compiler accepts the C++ language as defined by the ISO/IEC 14882:1998 standard and modified by TC1 for that standard.

Command line options are also available to enable and disable anachronisms and strict standard-conformance checking.
2.2.1. Standard Language Features Accepted

The following features not in traditional C++ (the C++ language of "The Annotated C++ Reference Manual" by Ellis and Stroustrup (ARM)) but in the standard are implemented:

- The dependent statement of an if, while, do-while, or for is considered to be a scope, and the restriction on having such a dependent statement be a declaration is removed.

- The expression tested in an if, while, do-while, or for, as the first operand of a "?" operator, or as an operand of the "&&", "::", or "!!" operators may have a pointer-to-member type or a class type that can be converted to a pointer-to-member type in addition to the scalar cases permitted by the ARM.

- Qualified names are allowed in elaborated type specifiers.

- A global-scope qualifier is allowed in member references of the form x::A::B and p->::A::B.

- The precedence of the third operand of the "?" operator is changed.

- If control reaches the end of the main() routine, and main() has an integral return type, it is treated as if a return 0; statement were executed.

- Pointers to arrays with unknown bounds as parameter types are diagnosed as errors.

- A functional-notation cast of the form A() can be used even if A is a class without a (nontrivial) constructor. The temporary created gets the same default initialization to zero as a static object of the class type.

- A cast can be used to select one out of a set of overloaded functions when taking the address of a function.

- Template friend declarations and definitions are permitted in class definitions and class template definitions.

- Type template parameters are permitted to have default arguments.

- Function templates may have nontype template parameters.

- A reference to const volatile cannot be bound to an rvalue.

- Qualification conversions, such as conversion from T** to T const * const * are allowed.

- Digraphs are recognized.

- Operator keywords (e.g., not, and, bitand, etc.) are recognized.

- Static data member declarations can be used to declare member constants.

- When option --wchar_t-keyword is set, wchar_t is recognized as a keyword and a distinct type.

- bool is recognized.

- RTTI (run-time type identification), including dynamic_cast and the typeid operator, is implemented.
• Declarations in tested conditions (in if, switch, for, and while statements) are supported.
• Array new and delete are implemented.
• New-style casts (static_cast, reinterpret_cast, and const_cast) are implemented.
• Definition of a nested class outside its enclosing class is allowed.
• mutable is accepted on non-static data member declarations.
• Namespaces are implemented, including using declarations and directives. Access declarations are broadened to match the corresponding using declarations.
• Explicit instantiation of templates is implemented.
• The typename keyword is recognized.
• explicit is accepted to declare non-converting constructors.
• The scope of a variable declared in the for-init-statement of a for loop is the scope of the loop (not the surrounding scope).
• Member templates are implemented.
• The new specialization syntax (using "template <>") is implemented.
• Cv-qualifiers are retained on rvalues (in particular, on function return values).
• The distinction between trivial and nontrivial constructors has been implemented, as has the distinction between PODs and non-PODs with trivial constructors.
• The linkage specification is treated as part of the function type (affecting function overloading and implicit conversions).
• extern inline functions are supported, and the default linkage for inline functions is external.
• A typedef name may be used in an explicit destructor call.
• Placement delete is implemented.
• An array allocated via a placement new can be deallocated via delete.
• Covariant return types on overriding virtual functions are supported.
• enum types are considered to be non-integral types.
• Partial specialization of class templates is implemented.
• Partial ordering of function templates is implemented.
• Function declarations that match a function template are regarded as independent functions, not as "guiding declarations" that are instances of the template.
• It is possible to overload operators using functions that take enum types and no class types.
• Explicit specification of function template arguments is supported.
• Unnamed template parameters are supported.
• The new lookup rules for member references of the form $x.A::B$ and $p->A::B$ are supported.
• The notation $:: template$ (and $->template$, etc.) is supported.
• In a reference of the form $f()$-$g()$, with $g$ a static member function, $f()$ is evaluated. The ARM specifies that the left operand is not evaluated in such cases.
• $enum$ types can contain values larger than can be contained in an $int$.
• Default arguments of function templates and member functions of class templates are instantiated only when the default argument is used in a call.
• String literals and wide string literals have $const$ type.
• Class name injection is implemented.
• Argument-dependent (Koenig) lookup of function names is implemented.
• Class and function names declared only in unqualified friend declarations are not visible except for functions found by argument-dependent lookup.
• A $void$ expression can be specified on a return statement in a $void$ function.
• Function-try-blocks, i.e., try-blocks that are the top-level statements of functions, constructors, or destructors, are implemented.
• Universal character set escapes (e.g., $\uabcd$) are implemented.
• On a call in which the expression to the left of the opening parenthesis has class type, overload resolution looks for conversion functions that can convert the class object to pointer-to-function types, and each such pointed-to "surrogate function" type is evaluated alongside any other candidate functions.
• Dependent name lookup in templates is implemented. Nondependent names are looked up only in the context of the template definition. Dependent names are also looked up in the instantiation context, via argument-dependent lookup.
• Value-initialization is implemented. This form of initialization is indicated by an initializer of "()" and causes zeroing of certain POD-typed members, where the usual default-initialization would leave them uninitialized.
• A partial specialization of a class member template cannot be added outside of the class definition.
• Qualification conversions may be performed as part of the template argument deduction process.
• The $export$ keyword for templates is implemented.
2.2.2. C++0x Language Features Accepted

The following features added in the working paper for the next C++ standard (expected to be completed in 2009 or later) are enabled in C++0x mode (with option --c++0x). Several of these features are also enabled in default (nonstrict) C++ mode.

- A "right shift token" (>>) can be treated as two closing angle brackets. For example:

```cpp
template<typename T> struct S {};  
S<S<int>> s; // OK. No whitespace needed  
               // between closing angle brackets.
```

- The friend class syntax is extended to allow nonclass types as well as class types expressed through a typedef or without an elaborated type name. For example:

```cpp
typedef struct S ST;  
class C {
    friend S;       // OK (requires S to be in scope).  
    friend ST;      // OK (same as "friend S;").  
    friend int;     // OK (no effect).  
    friend S const; // Error: cv-qualifiers cannot  
                     // appear directly.
};
```

- Mixed string literal concatenations are accepted (a feature carried over from C99):

```cpp
wchar_t *str = "a" L"b"; // OK, same as L"ab".
```

- Variadic macros and empty macro arguments are accepted, as in C99.

- A trailing comma in the definition of an enumeration type is silently accepted (a feature carried over from C99):

```cpp
enum E { e, };  
```

- If the command line option --long-long is specified, the type long long is accepted. Unsuffixed integer literals that cannot be represented by type long, but could potentially be represented by type unsigned long, have type long long instead (this matches C99, but not the treatment of the long long extension in C89 or default C++ mode).

- The keyword typename followed by a qualified-id can appear outside a template declaration.

```cpp
struct S { struct N {}; };  
typename S::N *p; // Silently accepted  
                     // in C++0x mode
2.2.3. Anachronisms Accepted

The following anachronisms are accepted when anachronisms are enabled (with --anachronisms):

- overload is allowed in function declarations. It is accepted and ignored.

- Definitions are not required for static data members that can be initialized using default initialization. The anachronism does not apply to static data members of template classes; they must always be defined.

- The number of elements in an array may be specified in an array delete operation. The value is ignored.

- A single operator++() and operator--() function can be used to overload both prefix and postfix operations.

- The base class name may be omitted in a base class initializer if there is only one immediate base class.

- Assignment to this in constructors and destructors is allowed. This is allowed only if anachronisms are enabled and the "assignment to this" configuration parameter is enabled.

- A bound function pointer (a pointer to a member function for a given object) can be cast to a pointer to a function.

- A nested class name may be used as a non-nested class name provided no other class of that name has been declared. The anachronism is not applied to template classes.

- A reference to a non-const type may be initialized from a value of a different type. A temporary is created, it is initialized from the (converted) initial value, and the reference is set to the temporary.

- A reference to a non-const class type may be initialized from an rvalue of the class type or a derived class thereof. No (additional) temporary is used.

- A function with old-style parameter declarations is allowed and may participate in function overloading as though it were prototyped. Default argument promotion is not applied to parameter types of such functions when the check for compatibility is done, so that the following declares the overloading of two functions named f:

  ```
  int f(int);
  int f(x) char x; { return x; }
  ```

  Note that in C this code is legal but has a different meaning: a tentative declaration of f is followed by its definition.

- When option --nonconst-ref-anachronism is set, a reference to a non-const class can be bound to a class rvalue of the same type or a derived type thereof.

  ```
  struct A {
    A(int);
    A operator=(A&);
    A operator+(const A&);
  }
  ```
}.

```cpp
main () {
    A b(1);
    b = A(1) + A(2); // Allowed as anachronism
}
```

### 2.2.4. Extensions Accepted in Normal C++ Mode

The following extensions are accepted in all modes (except when strict ANSI/ISO violations are diagnosed as errors or were explicitly noted):

- A `friend` declaration for a class may omit the `class` keyword:
  ```cpp
class A {
    friend B; // Should be "friend class B"
}
```

- Constants of scalar type may be defined within classes:
  ```cpp
class A {
    const int size = 10;
    int a[size];
};
```

- In the declaration of a class member, a qualified name may be used:
  ```cpp
struct A {
    int A::f(); // Should be int f();
};
```

- The `restrict` keyword is allowed.

- Implicit type conversion between a pointer to an `extern "C"` function and a pointer to an `extern "C++"` function is permitted. Here's an example:
  ```cpp
extern "C" void f(); // f's type has extern "C" linkage
void (*pf)(); // pf points to an extern "C++" function
    = &f; // error unless implicit conversion is allowed
```

This extension is allowed in environments where C and C++ functions share the same calling conventions. It is enabled by default.

- A `"?"` operator whose second and third operands are string literals or wide string literals can be implicitly converted to `"char *"` or `"wchar_t *"`. (Recall that in C++ string literals are `const`. There is a deprecated implicit conversion that allows conversion of a string literal to `"char *"`, dropping the `const`. That conversion, however, applies only to simple string literals. Allowing it for the result of a `"?"` operation is an extension.)
char *p = x ? "abc" : "def";

- Default arguments may be specified for function parameters other than those of a top-level function declaration (e.g., they are accepted on `typedef` declarations and on pointer-to-function and pointer-to-member-function declarations).

- Non-static local variables of an enclosing function can be referenced in a non-evaluated expression (e.g., a `sizeof` expression) inside a local class. A warning is issued.

- In default C++ mode, the friend class syntax is extended to allow nonclass types as well as class types expressed through a typedef or without an elaborated type name. For example:

```cpp
typedef struct S ST;
class C {
  friend S;       // OK (requires S to be in scope).
  friend ST;      // OK (same as "friend S;").
  friend int;     // OK (no effect).
  friend S const; // Error: cv-qualifiers cannot
  // appear directly.
};
```

- In default C++ mode, mixed string literal concatenations are accepted. (This is a feature carried over from C99 and also available in GNU modes).

```cpp
wchar_t *str = "a" L"b"; // OK, same as L"ab".
```

- In default C++ mode, variadic macros are accepted. (This is a feature carried over from C99 and also available in GNU modes.)

- In default C++ mode, empty macro arguments are accepted (a feature carried over from C99).

- A trailing comma in the definition of an enumeration type is silently accepted (a feature carried over from C99):

```cpp
enum E { e, };
```

### 2.3. GNU Extensions

The C++ compiler can be configured to support the GNU C++ mode (`command line option --g++`). In this mode, many extensions provided by the GNU C++ compiler are accepted. The following extensions are provided in GNU C++ mode.

- Extended designators are accepted
- Compound literals are accepted.
- Non-standard anonymous unions are accepted
• The `typeof` operator is supported. This operator can take an expression or a type (like the `sizeof` operator, but parentheses are always required) and expands to the type of the given entity. It can be used wherever a `typedef` name is allowed.

```c
typeof(2*2.3) d; // Declares a "double"
typeof(int) i;   // Declares an "int"
```

This can be useful in macro and template definitions.

• The `__extension__` keyword is accepted preceding declarations and certain expressions. It has no effect on the meaning of a program.

```c
__extension__ __inline__ int f(int a) {
    return a > 0 ? a/2 : f(__extension__ 1-a);
}
```

• In all GNU C modes and in GNU C++ modes with `gnu_version < 30400`, the type modifiers `signed`, `unsigned`, `long` and `short` can be used with `typedef` types if the specifier is valid with the underlying type of the typedef in ANSI C. E.g.:

```c
typedef int I;
unsigned I *pui; // OK in GNU C++ mode;
               // same as "unsigned int *pui"
```

• If the command line option `--long-long` is specified, the extensions for the `long long` and `unsigned long long` types are enabled.

• Zero-length array types (specified by `[0]`) are supported. These are complete types of size zero.

• C99-style flexible array members are accepted. In addition, the last field of a class type have a class type whose last field is a flexible array member. In GNU C++ mode, flexible array members are treated exactly like zero-length arrays, and can therefore appear anywhere in the class type.

• The C99 `__Pragma` operator is supported.

• The gcc built-in `<stdarg.h>` and `<varargs.h>` facilities (`__builtin_va_list`, `__builtin_va_arg`, ...) are accepted.

• The `sizeof` operator is applicable to `void` and to function types and evaluates to the value one.

• Variables can be redeclared with different top-level cv-qualifiers (the new qualification is merged into existing qualifiers). For example:

```c
extern int volatile x;
int const x = 32;      // x is now const volatile
```

• The "assembler name" of variables and routines can be specified. For example:

```c
int counter __asm__("counter_v1") = 0;
```
• Register variables can be mapped on specific registers using the asm keyword.

```c
register int i asm("eax");
    // Map "i" onto register eax.
```

• The keyword inline is ignored (with a warning) on variable declarations and on block-extern function declarations.

• Excess aggregate initializers are ignored with a warning.

```c
struct S { int a, b; };
struct S a1 = { 1, 2, 3 };
    // "3" ignored with a warning; no error
int a2[2] = { 7, 8, 9 };
    // "9" ignored with a warning; no error
```

• Expressions of types void*, void const*, void volatile* and void const volatile* can be dereferenced; the result is an lvalue.

• The __restrict__ keyword is accepted. It is identical to the C99 restrict keyword, except for its spelling.

• Out-of-range floating-point values are accepted without a diagnostic. When IEEE floating-point is being used, the "infinity" value is used.

• Extended variadic macros are supported.

• Dollar signs ($) are allowed in identifiers.

• Hexadecimal floating point constants are recognized.

• The __asm__ keyword is recognized and equivalent to the asm token. Extended syntax is supported to indicate how assembly operands map to C/C++ variables.

```c
asm("fsinx %1,%0" : "=f"(x) : "f"(a));
    // Map the output operand on "x",
    // and the input operand on "a".
```

• The \e escape sequence is recognized and stands for the ASCII "ESC" character.

• The address of a statement label can be taken by use of the prefix "&&" operator, e.g., void *a = &&L. A transfer to the address of a label can be done by the "goto *" statement, e.g., goto *a.

• Multi-line strings are supported, e.g.,

```c
char *p = "abc
def";
```

• ASCII "NULL" characters are accepted in source files.
• A source file can end with a backslash ("\") character.

• Case ranges (e.g., "case 'a' ... 'z':") are supported.

• A number of macros are predefined in GNU mode. See Section 2.8, Predefined Macros.

• A predefined macro can be undefined.

• A large number of special functions of the form __builtin_xyz (e.g., __builtinalloca) are predeclared.

• Some expressions are considered to be constant-expressions even though they are not so considered in standard C and C++. Examples include "((char *)&((struct S *)0)->c[0]) - (char *)0" and "(int)"Hello" & 0".

• The macro __GNUC__ is predefined to the major version number of the emulated GNU compiler. Similarly, the macros __GNUC_MINOR__ and __GNUC_PATCHLEVEL__ are predefined to the corresponding minor version number and patch level. Finally, __VERSION__ is predefined to a string describing the compiler version.

• The __thread specifier can be used to indicate that a variable should be placed in thread-local storage (requires gnu_version >= 30400).

• An extern inline function that is referenced but not defined is permitted (with a warning).

• Trigraphs are ignored (with a warning).

• Non-standard casts are allowed in null pointer constants, e.g., (int)(int *)0 is considered a null pointer constant in spite of the pointer cast in the middle.

• Statement expressions, e.g., ({int j; j = f(); j;}) are accepted. Branches into a statement expression are not allowed. In C++ mode, branches out are also not allowed. Variable-length arrays, destructible entities, try, catch, local non-POD class definitions, and dynamically-initialized local static variables are not allowed inside a statement expression.

• Labels can be declared to be local in statement expressions by introducing them with a __label__ declaration.

  ({ __label__ lab; int i = 4; lab: i = 2*i-1; if (!(i%17)) goto lab; i; })

• Not-evaluated parts of constant expressions can contain non-constant terms:

  int i;
  int a[ 1 || i ]; // Accepted in g++ mode

• Casts on an lvalue that don’t fall under the usual "lvalue cast" interpretation (e.g., because they cast to a type having a different size) are ignored, and the operand remains an lvalue. A warning is issued.

  int i;
  (short)i = 0; // Accepted, cast is ignored; entire int is set
• Variable length arrays (VLAs) are supported. GNU C also allows VLA types for fields of local structures, which can lead to run-time dependent sizes and offsets. The C++ compiler does not implement this, but instead treats such arrays as having length zero (with a warning); this enables some popular programming idioms involving fields with VLA types.

```c
void f(int n) {
    struct {
        int a[n]; // Warning: n ignored and // replaced by zero
    };
}
```

• Complex type extensions are supported (these are the same as the C99 complex type features, with the elimination of _Imaginary and the addition of __complex, __real, __imag, the use of "~" to denote complex conjugation, and complex literals such as "1.2i").

• If an explicit instantiation directive is preceded by the keyword extern, no (explicit or implicit) instantiation is for the indicated specialization.

• An explicit instantiation directive that names a class may omit the class keyword, and may refer to a typedef.

• An explicit instantiation or extern template directive that names a class is accepted in an invalid namespace.

• std::type_info does not need to be introduced with a special pragma.

• A special keyword __null expands to the same constant as the literal "0", but is expected to be used as a null pointer constant.

• When gnu_version < 30400, names from dependent base classes are ignored only if another name would be found by the lookup.

```c
const int n = 0;
template <class T> struct B {
    static const int m = 1; static const int n = 2;
};
template <class T> struct D : B<T> {
    int f() { return m + n; } // B::m + ::n in g++ mode
};
```

• A non-static data member from a dependent base class, which would usually be ignored as described above, is found if the lookup would have otherwise found a nonstatic data member of an enclosing class (when gnu_version is < 30400).

```c
template <class T> struct C {
    struct A { int i; };
    struct B: public A {
        void f() {
```
• A new operation in a template is always treated as dependent (when gnu_version >= 30400).

```cpp
template <class T> struct A {
    void f() {
        void *p = 0;
        new (&p) int(0); // calls operator new
                         // declared below
    }
};
void* operator new(size_t, void* p);
```

• When doing name lookup in a base class, the injected class name of a template class is ignored.

```cpp
namespace N {
    template <class T> struct A {};  // No class name
}  // N

struct A {
    int i;
};
struct B : N::A<int> {
    B() { A x; x.i = 1; } // g++ uses ::A, not N::A
};
```

• The injected class name is found in certain contexts in which the constructor should be found instead.

```cpp
struct A {
    A(int) {};
};
A::A a(1);
```

• In a constructor definition, what should be treated as a template argument list of the constructor is instead treated as the template argument list of the enclosing class.

```cpp
template <int u1> struct A { };
template <> struct A<1> {
    template<class T> A(T i, int j);
};
template <> A<1>::A<1>(int i, int j) {}  // accepted in g++ mode
```
A difference in calling convention is ignored when redeclaring a typedef.

typedef void F();

extern "C" {
    typedef void F(); // Accepted in GNU C++ mode
    // (error otherwise)
}

• The macro __GNUG__ is defined identically to __GNUC__ (i.e., the major version number of the GNU
  compiler version that is being emulated).

• The macro _GNU_SOURCE is defined as "1".

• Guiding declarations (a feature present in early drafts of the standard, but not in the final standard) are
  disabled.

• Namespace std is predeclared.

• No connection is made between declarations of identical names in different scopes even when these
  names are declared extern "C". E.g.,

typedef void F();

extern "C" { void f(int); }
namespace N {
    extern "C" {
        void f() {} // Warning (not error) in g++ mode
    }
}

int main() { f(1); }

This example is accepted by the C++ compiler, but it will emit two conflicting declarations for the function
f.

• When a using-directive lookup encounters more than one extern "C" declaration (created when
  more than one namespace declares an extern "C" function of a given name, as described above),
  only the first declaration encountered is considered for the lookup.

extern "C" int f(void);
extern "C" int g(void);
namespace N {
    extern "C" int f(void); // same type
    extern "C" void g(void); // different type
};
using namespace N;
int i = f(); // calls ::f
int j = g(); // calls ::f

• The definition of a member of a class template that appears outside of the class definition may declare
  a nontype template parameter with a type that is different than the type used in the definition of the
  class template. A warning is issued (GNU version 30300 and below).
template <int I> struct A { void f(); };  
template <unsigned int I> void A<I>::f(){}  

- A class template may be redeclared with a nontype template parameter that has a type that is different than the type used in the earlier declaration. A warning is issued.

  template <int I> class A;  
  template <unsigned int I> class A {};

- A friend declaration may refer to a member typedef.

  class A {
      class B {};
      typedef B my_b;
      friend class my_b;
  };

- When a friend class is declared with an unqualified name, the lookup of that name is not restricted to the nearest enclosing namespace scope.

  struct S;
  namespace N {
      class C {
          friend struct S; // ::S in g++ mode,  
                         // N::S in default mode  
      };
  }

- A friend class declaration can refer to names made visible by using-directives.

  namespace N { struct A { }; }  
  using namespace N;
  struct B {
      void f() { A a; }
      friend struct A; // in g++ mode N::A,  
                       // not a new declaration of ::A
  };

- An inherited type name can be used in a class definition and later redeclared as a typedef.

  struct A { typedef int I; };
  struct B : A {
      typedef I J;    // Refers to A::I
      typedef double I; // Accepted in g++ mode  
                       // (introduces B::I)
  };

- In a catch clause, an entity may be declared with the same name as the handler parameter.
try { } 
catch(int e) {
    char e;
}

• The diagnostic issued for an exception specification mismatch is reduced to a warning if the previous declaration was found in a system header.

• The exception specification for an explicit template specialization (for a function or member function) does not have to match the exception specification of the corresponding primary template.

• A template argument list may appear following a constructor name in constructor definition that appears outside of the class definition:

```cpp
template <class T> struct A {
    A();
};
template <class T> A<T>::A<T>{}{}
```

• When gnu_version < 30400, an incomplete type can be used as the type of a nonstatic data member of a class template.

```cpp
class B;
template <class T> struct A {
    B b;
};
```

• A constructor need not provide an initializer for every nonstatic const data member (but a warning is still issued if such an initializer is missing).

```cpp
struct S {
    int const ic;
    S() {} // Warning only in GNU C++ mode
    // (error otherwise).
};
```

• Exception specifications are ignored on function definitions when support for exception handling is disabled (normally, they are only ignored on function declarations that aren't definitions).

• A friend declaration in a class template may refer to an undeclared template.

```cpp
template <class T> struct A {
    friend void f<>(A<T>);
};
```

• When gnu_version is < 30400, the semantic analysis of a friend function defined in a class template is performed only if the function is actually used and is done at the end of the translation unit (instead of at the point of first use).
A function template default argument may be redeclared. A warning is issued and the default from the initial declaration is used.

```cpp
template<class T> void f(int i = 1);
template<class T> void f(int i = 2){}
int main() {
    f<void>();
}
```

A definition of a member function of a class template that appears outside of the class may specify a default argument.

```cpp
template <class T> struct A { void f(T); };
template <class T> void A<T>::f(T value = T() ) { }
```

Function declarations (that are not definitions) can have duplicate parameter names.

```cpp
void f(int i, int i); // Accepted in GNU C++ mode
```

Default arguments are retained as part of deduced function types.

A namespace member may be redeclared outside of its namespace.

A template may be redeclared outside of its class or namespace.

```cpp
namespace N {
    template< typename T > struct S {};
}
template< typename T > struct N::S;
```

The injected class name of a class template can be used as a template template argument.

```cpp
template <template <class> class T> struct A {}; 
template <class T> struct B {
    A<
```
struct A { enum { e1 = 1 }; 
int main () {
    A a;
    int x[a.e1]; // Accepted in GNU C++ mode
    return 0;
}

• Strong using-directives are supported.
    using namespace debug __attribute__((strong));

• Partial specializations that are unusable because of nondeducible template parameters are accepted
  and ignored.

    template<class T> struct A {class C { };};
    template<class T> struct B {enum {e = 1}; 
    template <class T> struct B<typename A<T>::C> {enum {e = 2}; 
    int main(int argc, char **argv) {
        printf("%d\n", B<int>::e);
        printf("%d\n", B<A<int>::C>::e);
    }

• Template parameters that are not used in the signature of a function template are not ignored for partial
  ordering purposes (i.e., the resolution of core language issue 214 is not implemented) when gnu_version
  is < 40100.

    template <class S, class T> void f(T t);
    template <class T> void f(T t);
    int main() {
        f<int>(3); // not ambiguous when gnu_version
        // is < 40100
    }

• Prototype instantiations of functions are deferred until the first actual instantiation of the function to
  allow the compilation of programs that contain definitions of unusable function templates (gnu_version
  30400 and above). The example below is accepted when prototype instantiations are deferred.

    class A {};
    template <class T> struct B {
        B () {}; // error: no initializer for
        // reference member "B<T>::a"
        A& a;
    };

• When doing nonclass prototype instantiations (e.g., gnu_version 30400 and above), the severity of the
  diagnostic issued if a const template static data member is defined without an initializer is reduced to
  a warning.
template <class T> struct A {
    static const int i;
};
template <class T> const int A<T>::i;

- When doing nonclass prototype instantiations (e.g., gnu_version 30400 and above), a template static
data member with an invalid aggregate initializer is accepted (the error is diagnosed if the static data
member is instantiated).

struct A {
    A(double val);
};
template <class T> struct B {
    static const A I[1];
};
template <class T> const A B<T>::I[1] = {
    {1.,0.,0.,0.}
};

The following GNU extensions are not currently supported:

- The forward declaration of function parameters (so they can participate in variable-length array
  parameters).
- GNU-style complex integral types (complex floating-point types are supported)
- Nested functions

2.4. Namespace Support

Namespaces are enabled by default. You can use the command line option --no-namespaces to disable
the features.

When doing name lookup in a template instantiation, some names must be found in the context of the
template definition while others may also be found in the context of the template instantiation. The C++
compiler implements two different instantiation lookup algorithms: the one mandated by the standard
(referred to as "dependent name lookup"), and the one that existed before dependent name lookup was
implemented.

Dependent name lookup is done in strict mode (unless explicitly disabled by another command line option)
or when dependent name processing is enabled by either a configuration flag or command line option.

Dependent Name Processing

When doing dependent name lookup, the C++ compiler implements the instantiation name lookup rules
specified in the standard. This processing requires that non-class prototype instantiations be done. This
in turn requires that the code be written using the typename and template keywords as required by
the standard.
Lookup Using the Referencing Context

When not using dependent name lookup, the C++ compiler uses a name lookup algorithm that approximates the two-phase lookup rule of the standard, but does so in such a way that is more compatible with existing code and existing compilers.

When a name is looked up as part of a template instantiation but is not found in the local context of the instantiation, it is looked up in a synthesized instantiation context that includes both names from the context of the template definition and names from the context of the instantiation. Here's an example:

```cpp
namespace N {
    int g(int);
    int x = 0;
    template <class T> struct A {
        T f(T t) { return g(t); }
        T f() { return x; }
    };
}
namespace M {
    int x = 99;
    double g(double);
    N::A<int> ai;
    int i = ai.f(0); // N::A<int>::f(int) calls
                    // N::g(int)
    int i2 = ai.f(); // N::A<int>::f() returns
                    // 0 (= N::x)
    N::A<double> ad;
    double d = ad.f(0); // N::A<double>::f(double)
                         // calls M::g(double)
    double d2 = ad.f(); // N::A<double>::f() also
                         // returns 0 (= N::x)
}
```

The lookup of names in template instantiations does not conform to the rules in the standard in the following respects:

- Although only names from the template definition context are considered for names that are not functions, the lookup is not limited to those names visible at the point at which the template was defined.

- Functions from the context in which the template was referenced are considered for all function calls in the template. Functions from the referencing context should only be visible for "dependent" function calls.

Argument Dependent Lookup

When argument-dependent lookup is enabled (this is the default), functions made visible using argument-dependent lookup overload with those made visible by normal lookup. The standard requires that this overloading occurs even when the name found by normal lookup is a block extern declaration. The C++ compiler does this overloading, but in default mode, argument-dependent lookup is suppressed when the normal lookup finds a block extern.
This means a program can have different behavior, depending on whether it is compiled with or without argument-dependent lookup --no-arg-dep-lookup, even if the program makes no use of namespaces. For example:

```cpp
struct A { }
A operator+(A, double);
void f() {
    A a1;
    A operator+(A, int);
    a1 + 1.0; // calls operator+(A, double)
    // with arg-dependent lookup enabled but
    // otherwise calls operator+(A, int);
}
```

### 2.5. Template Instantiation

The C++ language includes the concept of templates. A template is a description of a class or function that is a model for a family of related classes or functions. For example, one can write a template for a Stack class, and then use a stack of integers, a stack of floats, and a stack of some user-defined type. In the source, these might be written `Stack<int>`, `Stack<float>`, and `Stack<X>`. From a single source description of the template for a stack, the compiler can create instantiations of the template for each of the types required.

The instantiation of a class template is always done as soon as it is needed in a compilation. However, the instantiations of template functions, member functions of template classes, and static data members of template classes (hereafter referred to as template entities) are not necessarily done immediately, for several reasons:

- One would like to end up with only one copy of each instantiated entity across all the object files that make up a program. (This of course applies to entities with external linkage.)

- The language allows one to write a specialization of a template entity, i.e., a specific version to be used in place of a version generated from the template for a specific data type. (One could, for example, write a version of `Stack<int>`, or of just `Stack<int>::push`, that replaces the template-generated version; often, such a specialization provides a more efficient representation for a particular data type.) Since the compiler cannot know, when compiling a reference to a template entity, if a specialization for that entity will be provided in another compilation, it cannot do the instantiation automatically in any source file that references it.

- C++ templates can be exported (i.e., declared with the keyword `export`). Such templates can be used in a translation unit that does not contain the definition of the template to instantiate. The instantiation of such a template must be delayed until the template definition has been found.

- The language also dictates that template functions that are not referenced should not be compiled, that, in fact, such functions might contain semantic errors that would prevent them from being compiled. Therefore, a reference to a template class should not automatically instantiate all the member functions of that class.

---

1 Since templates are descriptions of entities (typically, classes) that are parameterizable according to the types they operate upon, they are sometimes called parameterized types.
(It should be noted that certain template entities are always instantiated when used, e.g., inline functions.)

From these requirements, one can see that if the compiler is responsible for doing all the instantiations automatically, it can only do so on a program-wide basis. That is, the compiler cannot make decisions about instantiation of template entities until it has seen all the source files that make up a complete program.

This C++ compiler provides an instantiation mechanism that does automatic instantiation at link time. For cases where you want more explicit control over instantiation, the C++ compiler also provides instantiation modes and instantiation pragmas, which can be used to exert fine-grained control over the instantiation process.

2.5.1. Automatic Instantiation

The goal of an automatic instantiation mode is to provide painless instantiation. You should be able to compile source files to object code, then link them and run the resulting program, and never have to worry about how the necessary instantiations get done.

In practice, this is hard for a compiler to do, and different compilers use different automatic instantiation schemes with different strengths and weaknesses:

• AT&T/USL/Novell’s cfront product saves information about each file it compiles in a special directory called ptrepository. It instantiates nothing during normal compilations. At link time, it looks for entities that are referenced but not defined, and whose mangled names indicate that they are template entities. For each such entity, it consults the ptrepository information to find the file containing the source for the entity, and it does a compilation of the source to generate an object file containing object code for that entity. This object code for instantiated objects is then combined with the "normal" object code in the link step.

If you are using cfront you must follow a particular coding convention: all templates must be declared in .h files, and for each such file there must be a corresponding .cc file containing the associated definitions. The compiler is never told about the .cc files explicitly; one does not, for example, compile them in the normal way. The link step looks for them when and if it needs them, and does so by taking the .h filename and replacing its suffix. ²

This scheme has the disadvantage that it does a separate compilation for each instantiated function (or, at best, one compilation for all the member functions of one class). Even though the function itself is often quite small, it must be compiled along with the declarations for the types on which the instantiation is based, and those declarations can easily run into many thousands of lines. For large systems, these compilations can take a very long time. The link step tries to be smart about recompiling instantiations only when necessary, but because it keeps no fine-grained dependency information, it is often forced to "recompile the world" for a minor change in a .h file. In addition, cfront has no way of ensuring that preprocessing symbols are set correctly when it does these instantiation compilations, if preprocessing symbols are set other than on the command line.

• Borland’s C++ compiler instantiates everything referenced in a compilation, then uses a special linker to remove duplicate definitions of instantiated functions.

If you are using Borland’s compiler you must make sure that every compilation sees all the source code it needs to instantiate all the template entities referenced in that compilation. That is, one cannot refer

²The actual implementation allows for several different suffixes and provides a command line option to change the suffixes sought.
to a template entity in a source file if a definition for that entity is not included by that source file. In practice, this means that either all the definition code is put directly in the .h files, or that each .h file includes an associated .cc (actually, .cpp) file.

This scheme is straightforward, and works well for small programs. For large systems, however, it tends to produce very large object files, because each object file must contain object code (and symbolic debugging information) for each template entity it references.

Our approach is a little different. It requires that, for each instantiation of a non-exported template, there is some (normal, top-level, explicitly-compiled) source file that contains the definition of the template entity, a reference that causes the instantiation, and the declarations of any types required for the instantiation. This requirement can be met in various ways:

- The Borland convention: each .h file that declares a template entity also contains either the definition of the entity or includes another file containing the definition.

- Implicit inclusion: when the compiler sees a template declaration in a .h file and discovers a need to instantiate that entity, it is given permission to go off looking for an associated definition file having the same base name and a different suffix, and it implicitly includes that file at the end of the compilation. This method allows most programs written using the cfront convention to be compiled with our approach. See Section 2.5.4, Implicit Inclusion.

- The ad hoc approach: you make sure that the files that define template entities also have the definitions of all the available types, and add code or pragmas in those files to request instantiation of the entities there.

Exported templates are also supported by our automatic instantiation method, but they require additional mechanisms explained further on.

Our C++ compiler's automatic instantiation method works as follows for non-exported templates:

1. The first time the source files of a program are compiled, template entities are instantiated wherever they are used. Template information files (with a .ti suffix) are generated and contain information about things that could have been instantiated in each compilation.

2. When the object files are linked together, a program called the prelinker, prelktc, is run. It examines the object files, looking for references and definitions of template entities, and for the added information about entities that could be instantiated.

3. If the prelinker finds a reference to a template entity for which there is no definition anywhere in the set of object files, it looks for a file that indicates that it could instantiate that template entity. When it finds such a file, it assigns the instantiation to it. The set of instantiations assigned to a given file is recorded in the associated instantiation request file (with, by default, a .ii suffix).

4. The prelinker then executes the compiler again to recompile each file for which the instantiation request file .ii file was changed. The original compilation command line options (saved in the template information file) are used for the recompilation.

Isn't this always the case? No. Suppose that file A contains a definition of class X and a reference to Stack<X>::push, and that file B contains the definition for the member function push. There would be no file containing both the definition of push and the definition of X.
5. When the compiler compiles a file, it reads the .ii file for that file and obeys the instantiation requests therein. It produces a new object file containing the requested template entities (and all the other things that were already in the object file).

6. The prelinker repeats steps 3-5 until there are no more instantiations to be adjusted.

7. The object files are linked together. The linker discards duplicate definitions of template entities.

Once the program has been linked correctly, the .ii files contain a complete set of instantiation assignments. From then on, whenever source files are recompiled, the compiler will consult the .ii files and do the indicated instantiations as it does the normal compilations. That means that, except in cases where the set of required instantiations changes, the prelink step from then on will find that all the necessary instantiations are present in the object files and no instantiation assignment adjustments need be done. That's true even if the entire program is recompiled.

If you provide a specialization of a template entity somewhere in the program, the specialization will be seen as a definition by the prelinker. Since that definition satisfies whatever references there might be to that entity, the prelinker will see no need to request an instantiation of the entity. If you add a specialization to a program that has previously been compiled, the prelinker will notice that too and remove the assignment of the instantiation from the proper .ii file.

The .ii files should not, in general, require any manual intervention. One exception: if a definition is changed in such a way that some instantiation no longer compiles (it gets errors), and at the same time a specialization is added in another file, and the first file is being recompiled before the specialization file and is getting errors, the .ii file for the file getting the errors must be deleted manually to allow the prelinker to regenerate it.

If you supplied the option -v to the control program cctc, and the prelinker changes an instantiation assignment, the prelinker will issue messages like:

```
C++ prelinker: A<int>::f() assigned to file test.o
C++ prelinker: executing: cctc -c test.cc
```

The automatic instantiation scheme can coexist with partial explicit control of instantiation by you through the use of pragmas or command line specification of the instantiation mode. See the following sections.

The automatic instantiation mode is enabled by default. It can be turned off by the command line option --no-auto-instantiation. If automatic instantiation is turned off, the extra information about template entities that could be instantiated in a file is not put into the object file.

### 2.5.2. Instantiation Modes

Normally, when a file is compiled, template entities are instantiated everywhere where they are used. The overall instantiation mode can, however, be changed by a command line option:

--instantiate=none

Do not automatically create instantiations of any template entities.

--instantiate=used
Instantiate those template entities that were used in the compilation. This will include all static data members for which there are template definitions. This is the default.

--instantiate=all

Instantiate all template entities declared or referenced in the compilation unit. For each fully instantiated template class, all of its member functions and static data members will be instantiated whether or not they were used. Non-member template functions will be instantiated even if the only reference was a declaration.

--instantiate=local

Similar to --instantiate=used except that the functions are given internal linkage. This is intended to provide a very simple mechanism for those getting started with templates. The compiler will instantiate the functions that are used in each compilation unit as local functions, and the program will link and run correctly (barring problems due to multiple copies of local static variables.) However, one may end up with many copies of the instantiated functions, so this is not suitable for production use. --instantiate=local cannot be used in conjunction with automatic template instantiation. If automatic instantiation is enabled by default, it will be disabled by the --instantiate=local option.

In the case where the cctc command is given a single file to compile and link, e.g.,

```
cctc test.cc
```

the compiler knows that all instantiations will have to be done in the single source file. Therefore, it uses the --instantiate=used mode and suppresses automatic instantiation.

### 2.5.3. Instantiation #pragma Directives

Instantiation pragmas can be used to control the instantiation of specific template entities or sets of template entities. There are three instantiation pragmas:

- The `instantiate` pragma causes a specified entity to be instantiated.

- The `do_not_instantiate` pragma suppresses the instantiation of a specified entity. It is typically used to suppress the instantiation of an entity for which a specific definition will be supplied.

- The `can_instantiate` pragma indicates that a specified entity can be instantiated in the current compilation, but need not be; it is used in conjunction with automatic instantiation, to indicate potential sites for instantiation if the template entity turns out to be required.

The argument to the instantiation pragma may be:

- a template class name `A<int>`

- a template class declaration `class A<int>`

- a member function name `A<int>::f`

- a static data member name `A<int>::i`

- a static data declaration `int A<int>::i`
• a member function declaration void A<int>::f(int, char)

• a template function declaration char* f(int, float)

A pragma in which the argument is a template class name (e.g., A<int> or class A<int>) is equivalent to repeating the pragma for each member function and static data member declared in the class. When instantiating an entire class a given member function or static data member may be excluded using the do_not_instantiate pragma. For example,

```c
#pragma instantiate A<int>
#pragma do_not_instantiate A<int>::f
```

The template definition of a template entity must be present in the compilation for an instantiation to occur. If an instantiation is explicitly requested by use of the instantiate pragma and no template definition is available or a specific definition is provided, an error is issued.

```c
template <class T> void f1(T);  // No body provided
template <class T> void g1(T);  // No body provided
void f1(int) {} // Specific definition
void main()
{
    int i;
    double d;
    f1(i);
    f1(d);
    g1(i);
    g1(d);
}
#pragma instantiate void f1(int) // error - specific
// definition
#pragma instantiate void g1(int) // error - no body
// provided
```

f1(double) and g1(double) will not be instantiated (because no bodies were supplied) but no errors will be produced during the compilation (if no bodies are supplied at link time, a linker error will be produced).

A member function name (e.g., A<int>::f) can only be used as a pragma argument if it refers to a single user defined member function (i.e., not an overloaded function). Compiler-generated functions are not considered, so a name may refer to a user defined constructor even if a compiler-generated copy constructor of the same name exists. Overloaded member functions can be instantiated by providing the complete member function declaration, as in

```c
#pragma instantiate char* A<int>::f(int, char*)
```

The argument to an instantiation pragma may not be a compiler-generated function, an inline function, or a pure virtual function.
2.5.4. Implicit Inclusion

When implicit inclusion is enabled, the C++ compiler is given permission to assume that if it needs a definition to instantiate a template entity declared in a .h file it can implicitly include the corresponding .cc file to get the source code for the definition. For example, if a template entity ABC::f is declared in file xyz.h, and an instantiation of ABC::f is required in a compilation but no definition of ABC::f appears in the source code processed by the compilation, the compiler will look to see if a file xyz.cc exists, and if so it will process it as if it were included at the end of the main source file.

To find the template definition file for a given template entity the C++ compiler needs to know the path name specified in the original include of the file in which the template was declared and whether the file was included using the system include syntax (e.g., #include <file.h>). This information is not available for preprocessed source containing #line directives. Consequently, the C++ compiler will not attempt implicit inclusion for source code containing #line directives.

The file to be implicitly included is found by replacing the file suffix with each of the suffixes specified in the instantiation file suffix list. The normal include search path mechanism is then used to look for the file to be implicitly included.

By default, the list of definition-file suffixes tried is .c, .cc, .cpp, and .cxx.

Implicit inclusion works well alongside automatic instantiation, but the two are independent. They can be enabled or disabled independently, and implicit inclusion is still useful when automatic instantiation is not done.

The implicit inclusion mode can be turned on by the command line option --implicit-include.

Implicit inclusions are only performed during the normal compilation of a file, (i.e., not when doing only preprocessing). A common means of investigating certain kinds of problems is to produce a preprocessed source file that can be inspected. When using implicit inclusion it is sometimes desirable for the preprocessed source file to include any implicitly included files. This may be done using the command line option --no-preprocessing-only. This causes the preprocessed output to be generated as part of a normal compilation. When implicit inclusion is being used, the implicitly included files will appear as part of the preprocessed output in the precise location at which they were included in the compilation.

2.5.5. Exported Templates

Exported templates are templates declared with the keyword export. Exporting a class template is equivalent to exporting each of its static data members and each of its non-inline member functions. An exported template is special because its definition does not need to be present in a translation unit that uses that template. In other words, the definition of an exported (non-class) template does not need to be explicitly or implicitly included in a translation unit that instantiates that template. For example, the following is a valid C++ program consisting of two separate translation units:

```cpp
// File 1:
#include <stdio.h>
static void trace() { printf("File 1\n"); }

export template<class T> T const& min(T const&, T const&);
int main()
{
```
// File 2:
#include <stdio.h>
static void trace() { printf("File 2\n"); }

export template<class T> T const& min(T const &a, T const &b) {
    trace();
    return a<b? a: b;
}

Note that these two files are separate translation units: one is not included in the other. That allows the
two functions trace() to coexist (with internal linkage).

Support for exported templates is enabled by default, but you can turn it off with command line option
--no-export.

2.5.5.1. Finding the Exported Template Definition

The automatic instantiation of exported templates is somewhat similar (from a user's perspective) to that
of regular (included) templates. However, an instantiation of an exported template involves at least two
translation units: one which requires the instantiation, and one which contains the template definition.

When a file containing definitions of exported templates is compiled, a file with a .et suffix is created
and some extra information is included in the associated .ti file. The .et files are used later by the C++
compiler to find the translation unit that defines a given exported template.

When a file that potentially makes use of exported templates is compiled, the compiler must be told where
to look for .et files for exported templates used by a given translation unit. By default, the compiler looks
in the current directory. Other directories may be specified with the command line option
--template-directory. Strictly speaking, the .et files are only really needed when it comes time to generate
an instantiation. This means that code using exported templates can be compiled without having the
definitions of those templates available. Those definitions must be available by the time prelinking is done
(or when explicit instantiation is done).

The .et files only inform the C++ compiler about the location of exported template definitions; they do
not actually contain those definitions. The sources containing the exported template definitions must
therefore be made available at the time of instantiation (usually, when prelinking is done). In particular,
the export facility is not a mechanism for avoiding the publication of template definitions in source form.

2.5.5.2. Secondary Translation Units

An instantiation of an exported template can be triggered by the prelinker, by an explicit instantiation
directive, or by the command line option --instantiate=used. In each case, the translation unit that contains
the initial point of instantiation will be processed as the primary translation unit. Based on information it
finds in the .et files, the C++ compiler will then load and parse the translation unit containing the definition
of the template to instantiate. This is a secondary translation unit. The simultaneous processing of the
primary and secondary translation units enables the C++ compiler to create instantiations of the exported
templates (which can include entities from both translation units). This process may reveal the need for additional instantiations of exported templates, which in turn can cause additional secondary translation units to be loaded.

When secondary translation units are processed, the declarations they contain are checked for consistency. This process may report errors that would otherwise not be caught. Many these errors are so-called "ODR violations" (ODR stands for "one-definition rule"). For example:

```
// File 1:
struct X {
    int x;
};

int main() {
    return min(2, 3);
}

// File 2:
struct X {
    unsigned x; // Error: X::x declared differently // in File 1
};

export template<class T> T const& min(T const &a, T const &b) {
    return a<b? a: b;
}
```

If there are no errors, the instantiations are generated in the output associated with the primary translation unit. This may also require that entities with internal linkage in secondary translation units be "externalized" so they can be accessed from the instantiations in the primary translation unit.

### 2.5.5.3. Libraries with Exported Templates

Typically a (non-export) library consists of an include directory and a lib directory. The include directory contains the header files required by users of the library and the lib directory contains the object code libraries that client programs must use when linking programs.

With exported templates, users of the library must also have access to the source code of the exported templates and the information contained in the associated .et files. This information should be placed in a directory that is distributed along with the include and lib directories. This is the export directory. It must be specified using the command line option --template-directory when compiling client programs.

The recommended procedure to build the export directory is as follows:

1. For each .et file in the original source directory, copy the associated source file to the export directory.

---

4As a consequence, using exported templates may require considerably more memory that similar uses of regular (included) templates.
2. Concatenate all of the .et files into a single .et file (e.g., mylib.et) in the export directory. The individual .et files could be copied to the export directory, but having all of the .et information in one file will make use of the library more efficient.

3. Create an export_info file in the export directory. The export_info file specifies the include search paths to be used when recompiling files in the export directory. If no export_info file is provided, the include search path used when compiling the client program that uses the library will also be used to recompile the library exported template files.

The export_info file consists of a series of lines of the form

include=x

or

sys_include=x

where x is a path name to be placed on the include search path. The directories are searched in the order in which they are encountered in the export_info file. The file can also contain comments, which begin with a "#", and blank lines. Spaces are ignored but tabs are not currently permitted. For example:

# The include directories to be used for the xyz library

include = /disk1/xyz/include
sys_include = /disk2/abc/include
include=/disk3/jkl/include

The include search path specified for a client program is ignored by the C++ compiler when it processes the source in the export library, except when no export_info file is provided. Command line macro definitions specified for a client program are also ignored by the C++ compiler when processing a source file from the export library; the command line macros specified when the corresponding .et file was produced do apply. All other compilation options (other than the include search path and command line macro definitions) used when recompiling the exported templates will be used to compile the client program.

When a library is installed on a new system, it is likely that the export_info file will need to be adapted to reflect the location of the required headers on that system.

2.6. Extern Inline Functions

Depending on the way in which the C++ compiler is configured, out-of-line copies of extern inline functions are either implemented using static functions, or are instantiated using a mechanism like the template instantiation mechanism. Note that out-of-line copies of inline functions are only required in cases where the function cannot be inlined, or when the address of the function is taken (whether explicitly by the user, by implicitly generated functions, or by compiler-generated data structures such as virtual function tables or exception handling tables).

When static functions are used, local static variables of the functions are promoted to global variables with specially encoded names, so that even though there may be multiple copies of the code, there is only one copy of such global variables. This mechanism does not strictly conform to the standard because the address of an extern inline function is not constant across translation units.
When the instantiation mechanism is used, the address of an extern inline function is constant across translation units, but at the cost of requiring the use of one of the template instantiation mechanisms, even for programs that don't use templates. Definitions of extern inline functions can be provided either through use of the automatic instantiation mechanism or by use of the \texttt{--instantiate=used} or \texttt{--instantiate=all} instantiation modes. There is no mechanism to manually control the definition of extern inline function bodies.

\section*{2.7. Pragmas to Control the C++ Compiler}

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

The syntax is:

\begin{verbatim}
#pragma pragma-spec
\end{verbatim}

The C++ compiler supports the following pragmas and all C compiler pragmas that are described in Section 1.5, \textit{Pragmas to Control the Compiler}.

\subsection*{instantiate / do\_not\_instantiate / can\_instantiate}

These are template instantiation pragmas. They are described in detail in Section 2.5.3, \textit{Instantiation #pragma Directives}.

\subsection*{hdrstop / no\_pch}

These are precompiled header pragmas. They are described in detail in Section 2.9, \textit{Precompiled Headers}.

\subsection*{once}

When placed at the beginning of a header file, indicates that the file is written in such a way that including it several times has the same effect as including it once. Thus, if the C++ compiler sees \texttt{#pragma once} at the start of a header file, it will skip over it if the file is \texttt{#included} again.

A typical idiom is to place an \texttt{#ifndef} guard around the body of the file, with a \texttt{#define} of the guard variable after the \texttt{#ifndef}:

\begin{verbatim}
#pragma once    // optional
#ifndef FILE_H
#define FILE_H
... body of the header file ...
#endif
\end{verbatim}

The \texttt{#pragma once} is marked as optional in this example, because the C++ compiler recognizes the \texttt{#ifndef} idiom and does the optimization even in its absence. \texttt{#pragma once} is accepted for compatibility with other compilers and to allow the programmer to use other guard-code idioms.

\subsection*{ident}

This pragma is given in the form:
#pragma ident "string"

or

#ident "string"

## 2.8. Predefined Macros

The C++ compiler defines a number of preprocessing macros. Many of them are only defined under certain circumstances. This section describes the macros that are provided and the circumstances under which they are defined.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__ABI_COMPATIBILITY_VERSION</td>
<td>Defines the ABI compatibility version being used. This macro is set to 9999, which means the latest version. This macro is used when building the C++ library.</td>
</tr>
<tr>
<td>__ABI_CHANGES_FOR_RTTI</td>
<td>This macro is set to TRUE, meaning that the ABI changes for RTTI are implemented. This macro is used when building the C++ library.</td>
</tr>
<tr>
<td>__ABI_CHANGES_FOR_ARRAY_NEW_AND_DELETE</td>
<td>This macro is set to TRUE, meaning that the ABI changes for array new and delete are implemented. This macro is used when building the C++ library.</td>
</tr>
<tr>
<td>__ABI_CHANGES_FOR_PLACEMENT_DELETE</td>
<td>This macro is set to TRUE, meaning that the ABI changes for placement delete are implemented. This macro is used when building the C++ library.</td>
</tr>
<tr>
<td>__ARRAY_OPERATORS</td>
<td>Defined when array new and delete are enabled. This is the default.</td>
</tr>
<tr>
<td><strong>BASE_FILE</strong></td>
<td>Similar to <strong>FILE</strong> but indicates the primary source file rather than the current one (i.e., when the current file is an included file).</td>
</tr>
<tr>
<td>_BOOL</td>
<td>Defined when bool is a keyword. This is the default.</td>
</tr>
<tr>
<td><strong>BUILD</strong></td>
<td>Identifies the build number of the C++ compiler, composed of decimal digits for the build number, three digits for the major branch number and three digits for the minor branch number. For example, if you use build 1.22.1 of the compiler, <strong>BUILD</strong> expands to 1022001. If there is no branch number, the branch digits expand to zero. For example, build 127 results in 127000000.</td>
</tr>
<tr>
<td>Macro</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>-------------</td>
</tr>
<tr>
<td>__CHAR_MIN / __CHAR_MAX</td>
<td>Used in limits.h to define the minimum/maximum value of a plain char respectively.</td>
</tr>
<tr>
<td><strong>CPTC</strong></td>
<td>Identifies the C++ compiler. You can use this symbol to flag parts of the source which must be recognized by the cptc C++ compiler only. It expands to 1.</td>
</tr>
<tr>
<td>__cplusplus</td>
<td>Identifies the C++ compiler. You can use this symbol to flag parts of the source which must be recognized by the cptc C++ compiler only. It expands to 1.</td>
</tr>
<tr>
<td>__CPLUSPLUS</td>
<td>Always defined.</td>
</tr>
<tr>
<td><strong>CPU</strong></td>
<td>Expands to a string with the CPU supplied with the option --cpu. When no --cpu is supplied, this symbol is not defined.</td>
</tr>
<tr>
<td><strong>DATE</strong></td>
<td>Defined to the date of the compilation in the form &quot;Mmm dd yyyy&quot;.</td>
</tr>
<tr>
<td>__DELTA_TYPE</td>
<td>Defines the type of the offset field in the virtual function table. This macro is used when building the C++ library.</td>
</tr>
<tr>
<td><strong>DOUBLE_FP</strong></td>
<td>Expands to 1 if you did not use option --no-double (Treat ‘double’ as ‘float’), otherwise unrecognized as macro.</td>
</tr>
<tr>
<td>__embedded_cplusplus</td>
<td>Defined as 1 in Embedded C++ mode.</td>
</tr>
<tr>
<td>__EXCEPTIONS</td>
<td>Defined when exception handling is enabled (~exceptions).</td>
</tr>
<tr>
<td><strong>FILE</strong></td>
<td>Expands to the current source file name.</td>
</tr>
<tr>
<td><strong>FUNCTION</strong></td>
<td>Defined to the name of the current function. An error is issued if it is used outside of a function.</td>
</tr>
<tr>
<td><strong>func</strong></td>
<td>Same as <strong>FUNCTION</strong> in GNU mode.</td>
</tr>
<tr>
<td>__IMPLICIT_USING_STD</td>
<td>Defined when the standard header files should implicitly do a using-directive on the std namespace (~using-std).</td>
</tr>
<tr>
<td>__JMP_BUF_ELEMENT_TYPE</td>
<td>Specifies the type of an element of the setjmp buffer. This macro is used when building the C++ library.</td>
</tr>
<tr>
<td>__JMP_BUF_NUM_ELEMENTS</td>
<td>Defines the number of elements in the setjmp buffer. This macro is used when building the C++ library.</td>
</tr>
<tr>
<td><strong>LINE</strong></td>
<td>Expands to the line number of the line where this macro is called.</td>
</tr>
<tr>
<td>__NAMESPACES</td>
<td>Defined when namespaces are supported (this is the default, you can disable support for namespaces with --no-namespaces).</td>
</tr>
<tr>
<td>Macro</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>-------------</td>
</tr>
<tr>
<td>__NO_LONG_LONG</td>
<td>Defined when the long long type is not supported. This is the default.</td>
</tr>
<tr>
<td>__NULL_EH_REGION_NUMBER</td>
<td>Defines the value used as the null region number value in the exception handling tables. This macro is used when building the C++ library.</td>
</tr>
<tr>
<td>__PLACEMENT_DELETE</td>
<td>Defined when placement delete is enabled.</td>
</tr>
<tr>
<td><strong>PRETTY_FUNCTION</strong></td>
<td>Defined to the name of the current function. This includes the return type and parameter types of the function. An error is issued if it is used outside of a function.</td>
</tr>
<tr>
<td>__PTRDIFF_MIN / __PTRDIFF_MAX</td>
<td>Used in stdint.h to define the minimum/maximum value of a ptrdiff_t type respectively.</td>
</tr>
<tr>
<td>__REGION_NUMBER_TYPE</td>
<td>Defines the type of a region number field in the exception handling tables. This macro is used when building the C++ library.</td>
</tr>
<tr>
<td><strong>REVISION</strong></td>
<td>Expands to the revision number of the C++ compiler. Digits are represented as they are; characters (for prototypes, alphas, betas) are represented by -1. Examples: v1.0r1 -&gt; 1, v1.0rb -&gt; -1</td>
</tr>
<tr>
<td>__RTTI</td>
<td>Defined when RTTI is enabled (--rtti).</td>
</tr>
<tr>
<td>__RUNTIME_USES_NAMESPACES</td>
<td>Defined when the run-time uses namespaces.</td>
</tr>
<tr>
<td><strong>SFRFILE</strong>(cpu)</td>
<td>This macro expands to the filename of the used SFR file, including the &lt; &gt;. The cpu is the argument of the macro. For example, if --cpu=tc1165 is specified, the macro <strong>SFRFILE</strong>(<strong>CPU</strong>) expands to <strong>SFRFILE</strong>(tc1165), which expands to &lt;regtc1165.sfr&gt;.</td>
</tr>
<tr>
<td><strong>SIGNED_CHARS</strong></td>
<td>Defined when plain char is signed.</td>
</tr>
<tr>
<td><strong>SINGLE_FP</strong></td>
<td>Expands to 1 if you used option --no-double (Treat ‘double’ as ‘float’), otherwise unrecognized as macro.</td>
</tr>
<tr>
<td>__SIZE_MIN / __SIZE_MAX</td>
<td>Used in stdint.h to define the minimum/maximum value of a size_t type respectively.</td>
</tr>
<tr>
<td><strong>STDC</strong></td>
<td>Always defined, but the value may be redefined.</td>
</tr>
<tr>
<td><strong>STDC_VERSION</strong></td>
<td>Identifies the ISO-C version number. Expands to 199901L for ISO C99, but the value may be redefined.</td>
</tr>
</tbody>
</table>
### Macro Definitions

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_STLP_NO_IOSTREAMS</td>
<td>Defined when option <code>--io-streams</code> is not used. This disables I/O stream functions in the STLPport C++ library.</td>
</tr>
<tr>
<td><strong>TASKING</strong></td>
<td>Always defined for the TASKING C++ compiler.</td>
</tr>
<tr>
<td><strong>TIME</strong></td>
<td>Expands to the compilation time: “hh:mm:ss”</td>
</tr>
<tr>
<td>__TYPE_TRAITS_ENABLED</td>
<td>Defined when type traits pseudo-functions (to ease the implementation of ISO/IEC TR 19768; e.g., __is_union) are enabled. This is the default in C++ mode.</td>
</tr>
<tr>
<td>__VAR_HANDLE_TYPE</td>
<td>Defines the type of the variable-handle field in the exception handling tables. This macro is used when building the C++ library.</td>
</tr>
<tr>
<td><strong>VERSION</strong></td>
<td>Identifies the version number of the C++ compiler. For example, if you use version 2.1r1 of the compiler, <strong>VERSION</strong> expands to 2001 (dot and revision number are omitted, minor version number in 3 digits).</td>
</tr>
<tr>
<td>__VIRTUAL_FUNCTION_INDEX_TYPE</td>
<td>Defines the type of the virtual function index field of the virtual function table. This macro is used when building the C++ library.</td>
</tr>
<tr>
<td>__VIRTUAL_FUNCTION_TYPE</td>
<td>Defines the type of the virtual function field of the virtual function table. This macro is used when building the C++ library.</td>
</tr>
<tr>
<td>__WCHAR_MIN / __WCHAR_MAX</td>
<td>Used in stdint.h to define the minimum/maximum value of a wchar_t type respectively.</td>
</tr>
<tr>
<td>__WCHAR_T</td>
<td>Defined when wchar_t is a keyword.</td>
</tr>
</tbody>
</table>

### 2.9. Precompiled Headers

It is often desirable to avoid recompiling a set of header files, especially when they introduce many lines of code and the primary source files that include them are relatively small. The C++ compiler provides a mechanism for, in effect, taking a snapshot of the state of the compilation at a particular point and writing it to a disk file before completing the compilation; then, when recompiling the same source file or compiling another file with the same set of header files, it can recognize the "snapshot point", verify that the corresponding precompiled header (PCH) file is reusable, and read it back in. Under the right circumstances, this can produce a dramatic improvement in compilation time; the trade-off is that PCH files can take a lot of disk space.

#### 2.9.1. Automatic Precompiled Header Processing

When `--pch` appears on the command line, automatic precompiled header processing is enabled. This means the C++ compiler will automatically look for a qualifying precompiled header file to read in and/or will create one for use on a subsequent compilation.
The PCH file will contain a snapshot of all the code preceding the "header stop" point. The header stop point is typically the first token in the primary source file that does not belong to a preprocessing directive, but it can also be specified directly by \#pragma hdrstop (see below) if that comes first. For example:

```c
#include "xxx.h"
#include "yyy.h"
int i;
```

The header stop point is `int` (the first non-preprocessor token) and the PCH file will contain a snapshot reflecting the inclusion of `xxx.h` and `yyy.h`. If the first non-preprocessor token or the \#pragma hdrstop appears within a \#if block, the header stop point is the outermost enclosing \#if. To illustrate, here's a more complicated example:

```c
#include "xxx.h"
#ifndef YYY_H
#define YYY_H 1
#include "yyy.h"
#endif
#if TEST
int i;
#endif
```

Here, the first token that does not belong to a preprocessing directive is again `int`, but the header stop point is the start of the \#if block containing it. The PCH file will reflect the inclusion of `xxx.h` and conditionally the definition of `YYY_H` and inclusion of `yyy.h`; it will not contain the state produced by \#if TEST.

A PCH file will be produced only if the header stop point and the code preceding it (mainly, the header files themselves) meet certain requirements:

- The header stop point must appear at file scope -- it may not be within an unclosed scope established by a header file. For example, a PCH file will not be created in this case:

```c
// xxx.h
class A {

// xxx.C
#include "xxx.h"
int i; }
```

- The header stop point may not be inside a declaration started within a header file, nor (in C++) may it be part of a declaration list of a linkage specification. For example, in the following case the header stop point is `int`, but since it is not the start of a new declaration, no PCH file will be created:

```c
// yyy.h
static

// yyy.C
#include "yyy.h"
int i;
```
• Similarly, the header stop point may not be inside a `#if` block or a `#define` started within a header file.

• The processing preceding the header stop must not have produced any errors. (Note: warnings and other diagnostics will not be reproduced when the PCH file is reused.)

• No references to predefined macros `__DATE__` or `__TIME__` may have appeared.

• No use of the `#line` preprocessing directive may have appeared.

• `#pragma no_pch` (see below) must not have appeared.

• The code preceding the header stop point must have introduced a sufficient number of declarations to justify the overhead associated with precompiled headers. The minimum number of declarations required is 1.

When the host system does not support memory mapping, so that everything to be saved in the precompiled header file is assigned to preallocated memory (MS-Windows), two additional restrictions apply:

• The total memory needed at the header stop point cannot exceed the size of the block of preallocated memory.

• No single program entity saved can exceed 16384, the preallocation unit.

When a precompiled header file is produced, it contains, in addition to the snapshot of the compiler state, some information that can be checked to determine under what circumstances it can be reused. This includes:

• The compiler version, including the date and time the compiler was built.

• The current directory (i.e., the directory in which the compilation is occurring).

• The command line options.

• The initial sequence of preprocessing directives from the primary source file, including `#include` directives.

• The date and time of the header files specified in `#include` directives.

This information comprises the PCH prefix. The prefix information of a given source file can be compared to the prefix information of a PCH file to determine whether the latter is applicable to the current compilation.

As an illustration, consider two source files:

```c++
// a.cc
#include "xxx.h"
... // Start of code

// b.cc
#include "xxx.h"
... // Start of code
```
When a.cc is compiled with --pch, a precompiled header file named a.pch is created. Then, when b.cc is compiled (or when a.cc is recompiled), the prefix section of a.pch is read in for comparison with the current source file. If the command line options are identical, if xxx.h has not been modified, and so forth, then, instead of opening xxx.h and processing it line by line, the C++ compiler reads in the rest of a.pch and thereby establishes the state for the rest of the compilation.

It may be that more than one PCH file is applicable to a given compilation. If so, the largest (i.e., the one representing the most preprocessing directives from the primary source file) is used. For instance, consider a primary source file that begins with

```c
#include "xxx.h"
#include "yyy.h"
#include "zzz.h"
```

If there is one PCH file for xxx.h and a second for xxx.h and yyy.h, the latter will be selected (assuming both are applicable to the current compilation). Moreover, after the PCH file for the first two headers is read in and the third is compiled, a new PCH file for all three headers may be created.

When a precompiled header file is created, it takes the name of the primary source file, with the suffix replaced by an implementation-specified suffix (pch by default). Unless --pch-dir is specified (see below), it is created in the directory of the primary source file.

When a precompiled header file is created or used, a message such as

"test.cc": creating precompiled header file "test.pch"

is issued. The user may suppress the message by using the command line option --no-pch-messages.

When the option --pch-verbose is used the C++ compiler will display a message for each precompiled header file that is considered that cannot be used giving the reason that it cannot be used.

In automatic mode (i.e., when --pch is used) the C++ compiler will deem a precompiled header file obsolete and delete it under the following circumstances:

- if the precompiled header file is based on at least one out-of-date header file but is otherwise applicable for the current compilation; or

- if the precompiled header file has the same base name as the source file being compiled (e.g., xxx.pch and xxx.cc) but is not applicable for the current compilation (e.g., because of different command line options).

This handles some common cases; other PCH file clean-up must be dealt with by other means (e.g., by the user).

Support for precompiled header processing is not available when multiple source files are specified in a single compilation: an error will be issued and the compilation aborted if the command line includes a request for precompiled header processing and specifies more than one primary source file.

### 2.9.2. Manual Precompiled Header Processing

Command line option --create-pch=file-name specifies that a precompiled header file of the specified name should be created.
Command line option `--use-pch=file-name` specifies that the indicated precompiled header file should be used for this compilation; if it is invalid (i.e., if its prefix does not match the prefix for the current primary source file), a warning will be issued and the PCH file will not be used.

When either of these options is used in conjunction with `--pch-dir`, the indicated file name (which may be a path name) is tacked on to the directory name, unless the file name is an absolute path name.

The options `--create-pch`, `--use-pch`, and `--pch` may not be used together. If more than one of these options is specified, only the last one will apply. Nevertheless, most of the description of automatic PCH processing applies to one or the other of these modes -- header stop points are determined the same way, PCH file applicability is determined the same way, and so forth.

### 2.9.3. Other Ways to Control Precompiled Headers

There are several ways in which the user can control and/or tune how precompiled headers are created and used.

- **#pragma hdrstop** may be inserted in the primary source file at a point prior to the first token that does not belong to a preprocessing directive. It enables you to specify where the set of header files subject to precompilation ends. For example,

  ```
  #include "xxx.h"
  #include "yyy.h"
  #pragma hdrstop
  #include "zzz.h"
  ```

  Here, the precompiled header file will include processing state for `xxx.h` and `yyy.h` but not `zzz.h`. (This is useful if the user decides that the information added by what follows the `#pragma hdrstop` does not justify the creation of another PCH file.)

- **#pragma no_pch** may be used to suppress precompiled header processing for a given source file.

- Command line option `--pch-dir=directory-name` is used to specify the directory in which to search for and/or create a PCH file.

Moreover, when the host system does not support memory mapping and preallocated memory is used instead, then one of the command line options `--pch`, `--create-pch`, or `--use-pch`, if it appears at all, must be the *first* option on the command line.

### 2.9.4. Performance Issues

The relative overhead incurred in writing out and reading back in a precompiled header file is quite small for reasonably large header files.

In general, it does not cost much to write a precompiled header file out even if it does not end up being used, and if it *is* used it almost always produces a significant speedup in compilation. The problem is that the precompiled header files can be quite large (from a minimum of about 250K bytes to several megabytes or more), and so one probably does not want many of them sitting around.

Thus, despite the faster recompilations, precompiled header processing is not likely to be justified for an arbitrary set of files with nonuniform initial sequences of preprocessing directives. Rather, the greatest benefit occurs when a number of source files can share the same PCH file. The more sharing, the less
disk space is consumed. With sharing, the disadvantage of large precompiled header files can be minimized, without giving up the advantage of a significant speedup in compilation times.

Consequently, to take full advantage of header file precompilation, users should expect to reorder the #include sections of their source files and/or to group #include directives within a commonly used header file.

Below is an example of how this can be done. A common idiom is this:

```c
#include "comnfile.h"
#pragma hdrstop
#include ...
```

where `comnfile.h` pulls in, directly and indirectly, a few dozen header files; the #pragma hdrstop is inserted to get better sharing with fewer PCH files. The PCH file produced for `comnfile.h` can be a bit over a megabyte in size. Another idiom, used by the source files involved in declaration processing, is this:

```c
#include "comnfile.h"
#include "decl_hdrs.h"
#pragma hdrstop
#include ...
```

`decl_hdrs.h` pulls in another dozen header files, and a second, somewhat larger, PCH file is created. In all, the source files of a particular program can share just a few precompiled header files. If disk space were at a premium, you could decide to make `comnfile.h` pull in all the header files used -- then, a single PCH file could be used in building the program.

Different environments and different projects will have different needs, but in general, users should be aware that making the best use of the precompiled header support will require some experimentation and probably some minor changes to source code.
Chapter 3. Assembly Language

This chapter describes the most important aspects of the TASKING assembly language for TriCore. For a complete overview of the architecture you are using, refer to the target’s core Architecture Manual (for example the TriCore2 Architecture Manual [2002, Infineon]).

3.1. Assembly Syntax

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

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

The syntax of an assembly statement is:

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

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

Examples:

```
LAB1: ; This label is followed by a colon and can be prefixed by whitespace
LAB1 ; This label has to start at the beginning of a line
```

**instruction**
An instruction consists of a mnemonic and zero, one or more operands. It must not start in the first column.

Operands are described in Section 3.3, Operands of an Assembly Instruction. The instructions are described in the target’s core Architecture Manual.

**directive**
With directives you can control the assembler from within the assembly source. Except for preprocessing directives, these must not start in the first column. Directives are described in Section 3.9, Assembler Directives and Controls.

**macro_call**
A call to a previously defined macro. It must not start in the first column. See Section 3.10, Macro Operations.

**comment**
Comment, preceded by a ; (semicolon).

You can use empty lines or lines with only comments.
Apart from the assembly statements as described above, you can put a so-called 'control line' in your assembly source file. These lines start with a $ in the first column and alter the default behavior of the assembler.

```
$control
```

For more information on controls see Section 3.9, Assembler Directives and Controls.

### 3.2. Assembler Significant Characters

You can use all ASCII characters in the assembly source both in strings and in comments. Also the extended characters from the ISO 8859-1 (Latin-1) set are allowed.

Some characters have a special meaning to the assembler. Special characters associated with expression evaluation are described in Section 3.6.3, Expression Operators. Other special assembler characters are:

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>;</td>
<td>Start of a comment</td>
</tr>
<tr>
<td>\</td>
<td>Line continuation character or macro operator: argument concatenation</td>
</tr>
<tr>
<td>?</td>
<td>Macro operator: return decimal value of a symbol</td>
</tr>
<tr>
<td>%</td>
<td>Macro operator: return hex value of a symbol</td>
</tr>
<tr>
<td>^</td>
<td>Macro operator: override local label</td>
</tr>
<tr>
<td>&quot;</td>
<td>Macro string delimiter or quoted string .DEFINE expansion character</td>
</tr>
<tr>
<td>'</td>
<td>String constants delimiter</td>
</tr>
<tr>
<td>@</td>
<td>Start of a built-in assembly function</td>
</tr>
<tr>
<td>*</td>
<td>Location counter substitution</td>
</tr>
<tr>
<td>#</td>
<td>Constant number</td>
</tr>
<tr>
<td>++</td>
<td>String concatenation operator</td>
</tr>
<tr>
<td>[]</td>
<td>Substring delimiter</td>
</tr>
</tbody>
</table>

### 3.3. Operands of an Assembly Instruction

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

<table>
<thead>
<tr>
<th>Operand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>symbol</td>
<td>A symbolic name as described in Section 3.4, Symbol Names. Symbols can also occur in expressions.</td>
</tr>
<tr>
<td>register</td>
<td>Any valid register as listed in Section 3.5, Registers.</td>
</tr>
<tr>
<td>expression</td>
<td>Any valid expression as described in Section 3.6, Assembly Expressions.</td>
</tr>
<tr>
<td>address</td>
<td>A combination of expression, register and symbol.</td>
</tr>
</tbody>
</table>
Addressing modes

The TriCore assembly language has several addressing modes. These are described in detail in the target's core Architecture Manual.

3.4. Symbol Names

User-defined symbols

A user-defined symbol can consist of letters, digits and underscore characters (_). The first character cannot be a digit. The size of an identifier is only limited by the amount of available memory. The case of these characters is significant. You can define a symbol by means of a label declaration or an equate or set directive.

Predefined preprocessor symbols

These symbols start and end with two underscore characters, __symbol__, and you can use them in your assembly source to create conditional assembly. See Section 3.4.1, Predefined Preprocessor Symbols.

Labels

Symbols used for memory locations are referred to as labels. It is allowed to use reserved symbols as labels as long as the label is followed by a colon.

Reserved symbols

Symbol names and other identifiers beginning with a period (.) are reserved for the system (for example for directives or section names). Instructions and registers are also reserved. The case of these built-in symbols is insignificant.

Examples

Valid symbol names:

\texttt{loop\_1}  
\texttt{ENTRY}  
\texttt{a\_B\_c}  
\texttt{\_aBC}

Invalid symbol names:

\texttt{1\_loop}; starts with a number  
\texttt{d15}; reserved register name  
\texttt{.DEFINE}; reserved directive name

3.4.1. Predefined Preprocessor Symbols

The TASKING assembler knows the predefined symbols as defined in the table below. The symbols are useful to create conditional assembly.
### Example

```
.if @defined('__ASTC__')
   ; this part is only for the astc assembler
...
.endif
```

### 3.5. Registers

The following register names, either upper or lower case, should not be used for user-defined symbol names in an assembly language source file:
D0 .. D15   (data registers)
E0 .. E14   (data register pairs, only the even numbers)
A0 .. A15   (address registers)

3.5.1. Special Function Registers

It is easy to access Special Function Registers (SFRs) that relate to peripherals from assembly. The SFRs are defined in a special function register definition file (*.def) as symbol names for use assembler. The assembler reads the SFR definition file as defined by the selected derivative with the command line option --cpu (-C). SFRs are defined with .EQU directives.

For example (from regtc1165.def):

PSW      .equ  0xFE04

Example use in assembly:

```
  mfc r d0, #PSW
  and n d0, d0, #0x7f ; reset counter
  insert d0, d0, #1, #7, #1 ; enable
  insert d0, d0, #1, #8, #1 ; set GW bit
  mtcr #PSW, d0
  isync
```

Without an SFR file the assembler only knows the general purpose registers D0-D15 and A0-A15.

3.6. Assembly Expressions

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

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

Expressions follow the conventional rules of algebra and boolean arithmetic.

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

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

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

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

- numeric constant
- string
3.6.1. Numeric Constants

Numeric constants can be used in expressions. If there is no prefix, by default the assembler assumes the number is a decimal number. Prefixes can be used in either lower or upper case.

<table>
<thead>
<tr>
<th>Base</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>A 0b or 0B prefix followed by binary digits (0,1).</td>
<td>0B1101</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0b1101010</td>
</tr>
<tr>
<td>Hexadecimal</td>
<td>A 0x or 0X prefix followed by hexadecimal digits (0-9, A-F, a-f).</td>
<td>0X12FF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x45</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0xfa10</td>
</tr>
<tr>
<td>Decimal integer</td>
<td>Decimal digits (0-9).</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1245</td>
</tr>
<tr>
<td>Decimal floating-point</td>
<td>Decimal digits (0-9), includes a decimal point, or an 'E' or 'e' followed</td>
<td>6E10</td>
</tr>
<tr>
<td></td>
<td>by the exponent.</td>
<td>.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3.14</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2.7e10</td>
</tr>
</tbody>
</table>

3.6.2. Strings

ASCII characters, enclosed in single (') or double ("”) quotes constitute an ASCII string. Strings between double quotes allow symbol substitution by a .DEFINE directive, whereas strings between single quotes are always literal strings. Both types of strings can contain escape characters.

Strings constants in expressions are evaluated to a number (each character is replaced by its ASCII value). Strings in expressions can have a size of up to 4 characters or less depending on the operand of an instruction or directive; any subsequent characters in the string are ignored. In this case the assembler issues a warning. An exception to this rule is when a string is used in a .BYTE assembler directive; in that case all characters result in a constant value of the specified size. Null strings have a value of 0.

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

\[ string, offset, length \]
offset is the start position within string. length is the length of the desired substring. Both values may not exceed the size of string.

Examples

'ABCD'  ; (0x41424344)
'''79'  ; to enclose a quote double it
"A""BC" ; or to enclose a quote escape it
'AB'+1  ; (0x4143) string used in expression
''    ; null string
.word 'abcdef' ; (0x64636261) 'ef' are ignored

warning: string value truncated

'abc'++'de' ; you can concatenate
; two strings with the '++' operator.
; This results in 'abcde'
['TASKING',0,4] ; results in the substring 'TASK'

3.6.3. Expression Operators

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

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

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Operator</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( )</td>
<td>()</td>
<td>parenthesis</td>
<td>Expressions enclosed by parenthesis are evaluated first.</td>
</tr>
<tr>
<td>Unary</td>
<td>+</td>
<td>plus</td>
<td>Returns the value of its operand.</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>minus</td>
<td>Returns the negative of its operand.</td>
</tr>
<tr>
<td></td>
<td>~</td>
<td>one's complement</td>
<td>Integer only. Returns the one's complement of its operand. It cannot be used with a floating-point operand.</td>
</tr>
<tr>
<td></td>
<td>!</td>
<td>logical negate</td>
<td>Returns 1 if the operands' value is 0; otherwise 0. For example, if buf is 0 then !buf is 1. If buf has a value of 1000 then !buf is 0.</td>
</tr>
<tr>
<td>Arithmetic</td>
<td>*</td>
<td>multiplication</td>
<td>Yields the product of its operands.</td>
</tr>
<tr>
<td></td>
<td>/</td>
<td>division</td>
<td>Yields the quotient of the division of the first operand by the second. For integer operands, the divide operation produces a truncated integer result.</td>
</tr>
<tr>
<td></td>
<td>%</td>
<td>modulo</td>
<td>Integer only. This operator yields the remainder from the division of the first operand by the second.</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>addition</td>
<td>Yields the sum of its operands.</td>
</tr>
<tr>
<td>Type</td>
<td>Operator</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>----------</td>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
<td>Yields the difference of its operands.</td>
<td></td>
</tr>
<tr>
<td>Shift</td>
<td>&lt;&lt;</td>
<td>shift left</td>
<td>Integer only. Causes the left operand to be shifted to the left (and zero-filled) by the number of bits specified by the right operand.</td>
</tr>
<tr>
<td></td>
<td>&gt;&gt;</td>
<td>shift right</td>
<td>Integer only. Causes the left operand to be shifted to the right by the number of bits specified by the right operand. The sign bit will be extended.</td>
</tr>
</tbody>
</table>

| Relational | <        | less than     | Returns an integer 1 if the indicated condition is TRUE or an integer 0 if the indicated condition is FALSE. |
|            | <=       | less than or equal | For example, if D has a value of 3 and E has a value of 5, then the result of the expression D < E is 1, and the result of the expression D > E is 0. |
|            | >        | greater than  | Use tests for equality involving floating-point values with caution, since rounding errors could cause unexpected results. |
|            | >=       | greater than or equal | Use tests for equality involving floating-point values with caution, since rounding errors could cause unexpected results. |
|            | ==       | equal         | Use tests for equality involving floating-point values with caution, since rounding errors could cause unexpected results. |
|            | !=       | not equal     | Use tests for equality involving floating-point values with caution, since rounding errors could cause unexpected results. |

| Bit and Bitwise | :       | bit position  | Specify bit position (right operand) in a data register (left operand) for use in bit operations (instructions that have the .T data type modifier). |
| Logical        | &       | AND           | Integer only. Yields the bitwise AND function of its operand. |
| Logical        | | | Integer only. Yields the bitwise OR function of its operand. |
| Logical        | ^       | exclusive OR  | Integer only. Yields the bitwise exclusive OR function of its operands. |

The relational operators and logical operators are intended primarily for use with the conditional assembly . if directive, but can be used in any expression.

### 3.7. Working with Sections

Sections are absolute or relocatable blocks of contiguous memory that can contain code or data. Some sections contain code or data that your program declared and uses directly, while other sections are created by the compiler or linker and contain debug information or code or data to initialize your application. These sections can be named in such a way that different modules can implement different parts of these sections. These sections are located in memory by the linker (using the linker script language, LSL) so that concerns about memory placement are postponed until after the assembly process.
All instructions and directives which generate data or code must be within an active section. The assembler emits a warning if code or data starts without a section definition and activation. The compiler automatically generates sections. If you program in assembly you have to define sections yourself.

For more information about locating sections see Section 8.7.8, *The Section Layout Definition: Locating Sections*.

**Section definition**

Sections are defined with the `.SDECL` directive and have a name. A section may have attributes to instruct the linker to place it on a predefined starting address, or that it may be overlaid with another section.

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

See the description of the `.SDECL` directive for a complete description of all possible attributes.

**Section activation**

Sections are defined once and are activated with the `.SECT` directive.

```
.SECT "name"
```

The linker will check between different modules and emits an error message if the section attributes do not match. The linker will also concatenate all matching section definitions into one section. So, all "code" sections generated by the compiler will be linked into one big "code" chunk which will be located in one piece. A `.SECT` directive referring to an earlier defined section is called a *continuation*. Only the name can be specified.

**Examples**

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

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

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

Defines and activates an absolute section named `.CONST` starting at address 0x1000.

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

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

```
.SECT   ".fardata"
```
3.8. Built-in Assembly Functions

The TASKING assembler has several built-in functions to support data conversion, string comparison, and math computations. You can use functions as terms in any expression.

Syntax of an assembly function

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

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

@ABS(expression)

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

Example:

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

@ACS(expression)

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

Example:

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

@ARG('symbol | expression)

Returns integer 1 if the macro argument represented by symbol or expression is present, 0 otherwise.

You can specify the argument with a symbol name (the name of a macro argument enclosed in single quotes) or with expression (the ordinal number of the argument in the macro formal argument list). If you use this function when macro expansion is not active, the assembler issues a warning.

Example:

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

@ASN(expression)

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

Example:

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

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

Example:

ANAME: .byte @ASTC() ;ANAME = 'astc'

@AT2(expression1,expression2)

Returns the arc tangent of expression1/expression2 as a floating-point value in the range -pi to pi. expression1 and expression2 must be separated by a comma.

Example:

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

@ATN(expression)

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

Example:

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

@CAT(string1,string2)

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

Example:

.DEFIN ED ID "@CAT('TASK','ING')" ;ID = 'TASKING'

@CEL(expression)

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

Example:

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

@CNT()

Returns the number of macro arguments of the current macro expansion as an integer. If you use this function when macro expansion is not active, the assembler issues a warning.

Example:

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

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

Example:

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

@COS(expression)

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

Example:

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

@CPU(string)

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

Example:

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

@CVF(expression)

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

Example:

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

@CVI(expression)

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

Example:

INT .SET @CVI(-1.05) ; INT = -1

@DEF('symbol' | symbol)

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

Example:

.IF @DEFINED('ANGLE') ; is symbol ANGLE defined?
.IF @DEFINED(ANGLE) ; does label ANGLE exist?
@EXP(expression)

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

Example:

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

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

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

Example:

```assembly
VAR1 .EQU @FLD(0,1,1)   ;turn bit 0 on, VAR1=1
VAR2 .EQU @FLD(0,3,1)   ;turn bit 0 on, VAR2=1
VAR3 .EQU @FLD(0,3,2)   ;turn bits 0 and 1 on, VAR3=3
VAR4 .EQU @FLD(0,3,2,1) ;turn bits 1 and 2 on, VAR4=6
VAR5 .EQU @FLD(0,1,1,7) ;turn eighth bit on, VAR5=0x80
```

@FLR(expression)

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

Example:

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

@FRACT(expression)

Returns the 32-bit fractional representation of the floating-point expression. The expression must be in the range [-1,+1].

Example:

```assembly
.WORD  @FRACT(0.1), @FRACT(1.0)
```

@HI(expression)

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

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

@HIS(expression)

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

Example:

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

@INT(expression)

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

Example:

.IF  @INT(TERM)     ;Test if result is an integer

@L10(expression)

Returns the base 10 logarithm of expression as a floating-point value. expression must be greater than zero.

Example:

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

@LEN(string)

Returns the length of string as an integer.

Example:

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

@LNG(expression1,expression2)

Concatenates the 16-bit expression1 and expression2 into a 32-bit word value such that expression1 is the high half and expression2 is the low half.

Example:

LWORD .WORD  @LNG(HI,LO)     ;build long word

@LO(expression)

Returns the lower 16 bits of a value. @LO(expression) is equivalent to (expression & 0xffff).
Example:

```
mov.u   d2,#@LO(COUNT)        ;lower 16 bits of COUNT
addih  d2,d2,#@HI(COUNT)
```

@LOG(expression)

Returns the natural logarithm of expression as a floating-point value. expression must be greater than zero.

Example:

```
LOG    .EQU  @LOG(100.0)      ;LOG = 4.605170
```

@LOS(expression)

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

Example:

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

@LSB(expression)

Returns the least significant byte of the result of the expression. The result of the expression is calculated as 16 bit.

Example:

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

@LST()

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

Example:

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

@LUN(expression)

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

Example:
@MAC(symbol)

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

Example:

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

@MAX(expression1[,expressionN],...)

Returns the maximum value of expression1, ..., expressionN as a floating-point value.

Example:

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

@MIN(expression1[,expressionN],...)

Returns the minimum value of expression1, ..., expressionN as a floating-point value.

Example:

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

@MSB(expression)

Returns the most significant byte of the result of the expression. The result of the expression is calculated as 16 bit.

Example:

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

@MXP()

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

Example:

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

@POS(string1,string2[,start])

Returns the position of string2 in string1 as an integer. If string2 does not occur in string1, the last string position + 1 is returned.
With \textit{start} you can specify the starting position of the search. If you do not specify \textit{start}, the search is started from the beginning of \textit{string1}. Note that the first position in a string is position 0.

Example:

\begin{verbatim}
ID1 .EQU  @POS('TASKING','ASK')  ; ID1 = 1
ID2 .EQU  @POS('ABCDABCD','B',2) ; ID2 = 5
ID3 .EQU  @POS('TASKING','BUG')  ; ID3 = 7
\end{verbatim}

\textbf{@POW(expression1,expression2)}

Returns \textit{expression1} raised to the power \textit{expression2} as a floating-point value. \textit{expression1} and \textit{expression2} must be separated by a comma.

Example:

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

\textbf{@RND()}

Returns a random value in the range 0.0 to 1.0.

Example:

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

\textbf{@RVB(expression1,expression2)}

Reverse the order of bits in \textit{expression1} delimited by the number of bits in \textit{expression2}. If \textit{expression2} is omitted the field is bounded by the target word size. Both expressions must be 16-bit integer values.

Example:

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

\textbf{@SCP(string1,string2)}

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

Example:

\begin{verbatim}
.IF @SCP(STR,'MAIN')  ; does STR equal 'MAIN'?
\end{verbatim}

\textbf{@SFRACT(expression)}

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

Example:
@SGN(expression)

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

Example:

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

@SIN(expression)

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

Example:

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

@SNH(expression)

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

Example:

HSINE .EQU  @SNH(VAL)          ;hyperbolic sine

@SQT(expression)

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

Example:

SQRT1 .EQU  @SQT(3.5)         ;SQRT1 = 1.870829
SQRT2 .EQU  @SQT(16)          ;SQRT2 = 4

@SUB(string,expression1,expression2)

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

Example:

.DEFINEx  ID  "@SUB('TASKING',3,4)"  ;ID = 'KING'

@TAN(expression)

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

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

@TNH(expression)

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

Example:

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

@UNF(expression)

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

Example:

FRC .EQU @UNF(0x4000) ; FRC = 0.5

@XPN(expression)

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

Example:

EXP .EQU @XPN(1.0) ; EXP = 2.718282

3.9. Assembler Directives and Controls

An assembler directive is simply a message to the assembler. Assembler directives are not translated into machine instructions. There are three main groups of assembler directives.

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

The following directives fall under this group:

• Assembly control directives

• Symbol definition and section directives

• Data definition / Storage allocation directives

• High Level Language (HLL) directives

• Directives that are interpreted by the macro preprocessor. These directives tell the macro preprocessor how to manipulate your assembly code before it is actually being assembled. You can use these
directives to write macros and to write conditional source code. Parts of the code that do not match the condition, will not be assembled at all.

• Some directives act as assembler options and most of them indeed do have an equivalent assembler (command line) option. The advantage of using a directive is that with such a directive you can overrule the assembler option for a particular part of the code. Directives of this kind are called *controls*. A typical example is to tell the assembler with an option to generate a list file while with the controls $LIST ON and $LIST OFF you overrule this option for a part of the code that you do not want to appear in the list file. Controls always appear on a separate line and start with a `$` sign in the first column.

The following controls are available:

• Assembly listing controls
• Miscellaneous controls

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

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

### 3.9.1. Assembler Directives

#### Overview of assembly control directives

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.COMMENT</td>
<td>Start comment lines. You cannot use this directive in .IF/.ELSE/.ENDIF constructs and .MACRO/.DUP definitions.</td>
</tr>
<tr>
<td>.END</td>
<td>Indicates the end of an assembly module</td>
</tr>
<tr>
<td>.FAIL</td>
<td>Programmer generated error message</td>
</tr>
<tr>
<td>.INCLUDE</td>
<td>Include file</td>
</tr>
<tr>
<td>.MESSAGE</td>
<td>Programmer generated message</td>
</tr>
<tr>
<td>.WARNING</td>
<td>Programmer generated warning message</td>
</tr>
</tbody>
</table>

#### Overview of symbol definition and section directives

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.ALIAS</td>
<td>Create an alias for a symbol</td>
</tr>
<tr>
<td>.EQU</td>
<td>Set permanent value to a symbol</td>
</tr>
<tr>
<td>.EXTERN</td>
<td>Import global section symbol</td>
</tr>
<tr>
<td>.GLOBAL</td>
<td>Declare global section symbol</td>
</tr>
<tr>
<td>.LOCAL</td>
<td>Declare local section symbol</td>
</tr>
<tr>
<td>.NAME</td>
<td>Specify name of original C source file</td>
</tr>
<tr>
<td>Directive</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>.ORG</code></td>
<td>Initialize memory space and location counters to create a nameless section</td>
</tr>
<tr>
<td><code>.SDECL</code></td>
<td>Declare a section with name, type and attributes</td>
</tr>
<tr>
<td><code>.SECT</code></td>
<td>Activate a declared section</td>
</tr>
<tr>
<td><code>.SET</code></td>
<td>Set temporary value to a symbol</td>
</tr>
<tr>
<td><code>.SIZE</code></td>
<td>Set size of symbol in the ELF symbol table</td>
</tr>
<tr>
<td><code>.TYPE</code></td>
<td>Set symbol type in the ELF symbol table</td>
</tr>
<tr>
<td><code>.WEAK</code></td>
<td>Mark a symbol as 'weak'</td>
</tr>
</tbody>
</table>

### Overview of data definition / storage allocation directives

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>.ACCUM</code></td>
<td>Define 64-bit constant of 18 + 46 bits format</td>
</tr>
<tr>
<td><code>.ALIGN</code></td>
<td>Align location counter</td>
</tr>
<tr>
<td><code>.ASCII, .ASCIIZ</code></td>
<td>Define ASCII string without / with ending NULL byte</td>
</tr>
<tr>
<td><code>.BYTE</code></td>
<td>Define byte</td>
</tr>
<tr>
<td><code>.DOUBLE</code></td>
<td>Define a 64-bit floating-point constant</td>
</tr>
<tr>
<td><code>.FLOAT</code></td>
<td>Define a 32-bit floating-point constant</td>
</tr>
<tr>
<td><code>.FRACT</code></td>
<td>Define a 32-bit constant fraction</td>
</tr>
<tr>
<td><code>.HALF</code></td>
<td>Define half-word (16 bits)</td>
</tr>
<tr>
<td><code>.SFRACT</code></td>
<td>Define a 16-bit constant fraction</td>
</tr>
<tr>
<td><code>.SPACE</code></td>
<td>Define storage</td>
</tr>
<tr>
<td><code>.WORD</code></td>
<td>Define word (32 bits)</td>
</tr>
</tbody>
</table>

### Overview of macro and conditional assembly directives

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>.DEFINE</code></td>
<td>Define substitution string</td>
</tr>
<tr>
<td><code>.DUP, .ENDM</code></td>
<td>Duplicate sequence of source lines</td>
</tr>
<tr>
<td><code>.DUPA, .ENDM</code></td>
<td>Duplicate sequence with arguments</td>
</tr>
<tr>
<td><code>.DUPC, .ENDM</code></td>
<td>Duplicate sequence with characters</td>
</tr>
<tr>
<td><code>.DUPF, .ENDM</code></td>
<td>Duplicate sequence in loop</td>
</tr>
<tr>
<td><code>.IF, .ELIF, .ELSE</code></td>
<td>Conditional assembly directive</td>
</tr>
<tr>
<td><code>.ENDIF</code></td>
<td>End of conditional assembly directive</td>
</tr>
<tr>
<td><code>.EXITM</code></td>
<td>Exit macro</td>
</tr>
<tr>
<td><code>.MACRO, .ENDM</code></td>
<td>Define macro</td>
</tr>
<tr>
<td><code>.PMACRO</code></td>
<td>Undefine (purge) macro</td>
</tr>
<tr>
<td><code>.UNDEF</code></td>
<td>Undefine <code>.DEFINE</code> symbol</td>
</tr>
</tbody>
</table>
# Overview of HLL directives

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.CALLS</td>
<td>Pass call tree information and/or stack usage information</td>
</tr>
<tr>
<td>.MISRAC</td>
<td>Pass MISRA-C information</td>
</tr>
</tbody>
</table>
.ACCUM

Syntax

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

Description

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

An expression can be:

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

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

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

Example

ACC: .ACCUM 0.1,0.2,0.3

Related Information

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

.SPACE (Define Storage)
**.ALIAS**

**Syntax**

alias-name .ALIAS function-name

**Description**

With the .ALIAS directive you can create an alias of a symbol. The C compiler generates this directive when you use the #pragma alias.

**Example**

exit .ALIAS _Exit
.ALIGN

Syntax

.ALIGN expression

Description

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

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

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

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

A label is not allowed before this directive.

Example

.sdecl '.text.mod.csec',code
.sect '.text.mod.csec'
.ALIGN 16 ; the assembler aligns
instruction ; this instruction at 16 MAUs and
 ; fills the 'gap' with NOP instructions.

.sdecl '.text.mod.csec2',code
.sect '.text.mod.csec2'
.ALIGN 12 ; WRONG: not a power of two, the
instruction ; assembler aligns this instruction at
 ; 16 MAUs and issues a warning.
.ASCII, .ASCIIIZ

Syntax

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

[label:] .ASCIIIZ string[,string]...

Description

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

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

Example

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

Note that with the .BYTE directive you can obtain exactly the same effect:

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

Related Information

.BYTE (Define a constant byte)

.SPACE (Define Storage)
.BYTE

Syntax

[label:] .BYTE argument[,argument]...

Description

With the .BYTE directive the assembler allocates and initializes a byte of memory for each argument.

An argument can be:

• a single or multiple character string constant

• an integer expression

• NULL (indicated by two adjacent commas: ,,)

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

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

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

String constants

Single-character strings are stored in a byte whose lower seven bits represent the ASCII value of the character, for example:

```
.BYTE 'R'        ; = 0x52
```

Multiple-character strings are stored in consecutive byte addresses, as shown below. The standard C language escape characters like \n are permitted.

```
.BYTE 'AB',,'C'  ; = 0x41420043 (second argument is empty)
```

Example

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

Related Information

.ASCII, .ASCIIZ (Define ASCII string without/with ending NULL)

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

.SPACE (Define Storage)
.CALLS

Syntax

.CALLS 'caller', 'callee'

or

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

Description

The first syntax creates a call graph reference between caller and callee. The linker needs this information to build a call graph. caller and callee are names of functions.

The second syntax specifies stack information. When callee is an empty name, this means we define the stack usage of the function itself. The value specified is the stack usage in bytes at the time of the call including the return address. A function can use multiple stacks.

This information is used by the linker to compute the used stack within the application. The information is found in the generated linker map file within the Memory Usage.

This directive is generated by the C compiler. Normally you will not use it in hand-coded assembly.

A label is not allowed before this directive.

Example

.CALLS 'main', 'nfunc'

Indicates that the function main calls the function nfunc.

.CALLS 'main', '', 8

The function main uses 8 bytes on the stack.
.COMMENT

Syntax

.COMMENT delimiter
.
delimiter

description

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

A label is not allowed before this directive.

Example

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

Syntax

.DEFINE symbol string

Description

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

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

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

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

A label is not allowed before this directive.

Example

Suppose you defined the symbol LEN with the substitution string "32":

.DEFINE LEN "32"

Then you can use the symbol LEN for example as follows:

.SPACE LEN
.MESSAGE "The length is: LEN"

The assembler preprocessor replaces LEN with "32" and assembles the following lines:

.SPACE 32
.MESSAGE "The length is: 32"

Related Information

.UNDEF (Undefine a .DEFINE symbol)

.MACRO, .ENDM (Define a macro)
.DUP, .ENDM

Syntax

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

Description

With the .DUP/.ENDM directive you can duplicate a sequence of assembly source lines. With expression you specify the number of duplications. If the expression evaluates to a number less than or equal to 0, the sequence of lines will not be included in the assembler output. The expression result must be an absolute integer and cannot contain any forward references (symbols that have not already been defined). The .DUP directive may be nested to any level.

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

Example

In this example the loop is repeated three times. Effectively, the preprocessor repeats the source lines (.BYTE 10) three times, then the assembler assembles the result:

.DUP 3
 .BYTE 10  ; assembly source lines
 .ENDM

Related Information

.DUPA, .ENDM (Duplicate sequence with arguments)
.DUPC, .ENDM (Duplicate sequence with characters)
.DUPF, .ENDM (Duplicate sequence in loop)
.MACRO, .ENDM (Define a macro)
.DUPA, .ENDM

Syntax

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

Description

With the .DUPA/.ENDM directive you can repeat a block of source statements for each argument. For each repetition, every occurrence of the formal_arg parameter within the block is replaced with each succeeding argument string. If an argument includes an embedded blank or other assembler-significant character, it must be enclosed with single quotes.

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

Example

Consider the following source input statements,

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

This is expanded as follows:

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

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

Related Information

.DUP, .ENDM (Duplicate sequence of source lines)

.DUPC, .ENDM (Duplicate sequence with characters)

.DUPF, .ENDM (Duplicate sequence in loop)

.MACRO, .ENDM (Define a macro)
.DUPC, .ENDM

Syntax

[label:] .DUPC formal_arg, string
....
. ENDM

Description

With the .DUPC/.ENDM directive you can repeat a block of source statements for each character within string. For each character in the string, the formal_arg parameter within the block is replaced with that character. If the string is empty, then the block is skipped.

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

Example

Consider the following source input statements,

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

This is expanded as follows:

.BYTE 1
.BYTE 2
.BYTE 3

Related Information

.DUP, .ENDM (Duplicate sequence of source lines)
.DUPA, .ENDM (Duplicate sequence with arguments)
.DUPF, .ENDM (Duplicate sequence in loop)
.MACRO, .ENDM (Define a macro)
.DUPF, .ENDM

Syntax

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

....

.ENDM

Description

With the .DUPF/.ENDM directive you can repeat a block of source statements \((end - start) + 1 / increment\) times. \(start\) is the starting value for the loop index; \(end\) represents the final value. \(increment\) is the increment for the loop index; it defaults to 1 if omitted (as does the \(start\) value). The \(formal\_arg\) parameter holds the loop index value and may be used within the body of instructions.

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

Example

Consider the following source input statements,

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

This is expanded as follows:

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

Related Information

.\DUP, \ENDM (Duplicate sequence of source lines)

.\DUPA, \ENDM (Duplicate sequence with arguments)

.\DUPC, \ENDM (Duplicate sequence with characters)

.\MACRO, \ENDM (Define a macro)
.END

Syntax

.END

Description

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

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

The assembler does not allow a label with this directive.

Example

    ; source lines
    .END                ; End of assembly module

Related Information

-
.EQU

Syntax

symbol .EQU expression

Description

With the .EQU directive you assign the value of expression to symbol permanently. The expression can be relocatable or absolute and forward references are allowed. Once defined, you cannot redefine the symbol. With the .GLOBAL directive you can declare the symbol global.

Example

To assign the value 0x400 permanently to the symbol MYSYMBOL:

MYSYMBOL .EQU 0x4000

You cannot redefine the symbol MYSYMBOL after this.

Related Information

.SET (Set temporary value to a symbol)
.EXITM

Syntax

.EXITM

Description

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

A label is not allowed before this directive.

Example

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

Related Information

.DUP, .ENDM (Duplicate sequence of source lines)
.DUPA, .ENDM (Duplicate sequence with arguments)
.DUPC, .ENDM (Duplicate sequence with characters)
.DUPF, .ENDM (Duplicate sequence in loop)
.MACRO, .ENDM (Define a macro)
.EXTERN

Syntax

```assembly
.EXTERN symbol[,symbol]...
```

Description

With the `.EXTERN` directive you define an external symbol. It means that the specified symbol is referenced in the current module, but is not defined within the current module. This symbol must either have been defined outside of any module or declared as globally accessible within another module with the `.GLOBAL` directive.

If you do not use the `.EXTERN` directive and the symbol is not defined within the current module, the assembler issues a warning and inserts the `.EXTERN` directive.

A label is not allowed with this directive.

Example

```assembly
.EXTERN AA,CC,DD       ;defined elsewhere
.sdecl ".text.code", code
.sect ".text.code"
.
.
MOV D0, #AA    ; AA is used here
.
```

Related Information

`.GLOBAL` (Declare global section symbol)

`.LOCAL` (Declare local section symbol)
.FAIL

Syntax

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

Description

With the .FAIL directive you tell the assembler to print an error message to stderr during the assembling process.

An arbitrary number of strings and expressions, in any order but separated by commas with no intervening white space, can be specified to describe the nature of the generated error. If you use expressions, the assembler outputs the result. The assembler outputs a space between each argument.

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

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

A label is not allowed with this directive.

Example

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

This results in the error:

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

Related Information

.MESSAGE (Programmer generated message)

.WARNING (Programmer generated warning)
.FLOAT, .DOUBLE

Syntax

[label:].FLOAT expression[,expression]...

[label:].DOUBLE expression[,expression]...

Description

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

An expression can be:

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

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

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

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

Example

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

Related Information

.SPACE (Define Storage)
.FRACT, .SFRACT

Syntax

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

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

Description

With the .FRACT or .SFRACT directive the assembler allocates and initializes a 32-bit or 16-bit constant fraction in memory for each argument. Use commas to separate multiple arguments.

An expression can be:

• a fractional fixed point expression (range [-1, +1>)

• NULL (indicated by two adjacent commas: ,,)

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

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

Example

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

Related Information

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

.SPACE (Define Storage)
.GLOBAL

Syntax

.GLOBAL  symbol[,]...

Description

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

With the .GLOBAL directive you declare one of more symbols as global. It means that the specified symbols are defined within the current section or module, and that those definitions should be accessible by all modules.

To access a symbol, defined with .GLOBAL, from another module, use the .EXTERN directive.

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

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

The assembler does not allow a label with this directive.

Example

LOOPA .EQU 1          ; definition of symbol LOOPA
   .GLOBAL  LOOPA ; LOOPA will be globally
                   ; accessible by other modules

Related Information

.EXTERN  (Import global section symbol)

.LOCAL   (Declare local section symbol)
.IF, .ELIF, .ELSE, .ENDIF

Syntax

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

Description

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

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

If the optional .ELSE and/or .ELIF directives are not present, then the source statements following the .IF directive and up to the next .ENDIF directive will be included as part of the source file being assembled only if the expression had a non-zero result.

If the expression has a value of zero, the source file will be assembled as if those statements between the .IF and the .ENDIF directives were never encountered.

If the .ELSE directive is present and expression has a nonzero result, then the statements between the .IF and .ELSE directives will be assembled, and the statement between the .ELSE and .ENDIF directives will be skipped. Alternatively, if expression has a value of zero, then the statements between the .IF and .ELSE directives will be skipped, and the statements between the .ELSE and .ENDIF directives will be assembled.

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

A label is not allowed with this directive.

Example

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

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

TEST .SET 0
DEMO .SET 1

You can also define the symbols on the command line with the assembler option --define (-D):

astc --define=DEMO --define=TEST=0 test.src
.INCLUDE

Syntax

.INCLUDE "filename" | <filename>

Description

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

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

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

1. The current directory if you use the "filename" construction.
   - The current directory is not searched if you use the <filename> syntax.
2. The path that is specified with the assembler option --include-directory.
3. The path that is specified in the environment variable ASTCINC when the product was installed.
4. The default include directory in the installation directory.

The assembler does not allow a label with this directive.

Example

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

Syntax

.LOCAL  symbol[,symbol]...

Description

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

With the .LOCAL directive you declare one of more symbols as local. It means that the specified symbols are explicitly local to the module in which you define them.

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

The assembler does not allow a label with this directive.

Example

.SDECL  ".data.io",DATA
.SECT   ".data.io"
.LOCAL  LOOPA  ; LOOPA is local to this section

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

Related Information

.EXTERN (Import global section symbol)

.GLOBAL (Declare global section symbol)
.MACRO, .ENDM

Syntax

macro_name .MACRO [argument[,argument]...]
  ... macro_definition_statements
  ... .ENMD

Description

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

The definition of a macro consists of three parts:

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

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

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

You can use the following operators in macro definition statements:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\</td>
<td>Macro argument concatenation</td>
<td>Concatenates a macro argument with adjacent alphanumeric characters.</td>
</tr>
<tr>
<td>?</td>
<td>Return decimal value of symbol</td>
<td>Substitutes the ?symbol sequence with a character string that represents the decimal value of the symbol.</td>
</tr>
<tr>
<td>%</td>
<td>Return hex value of symbol</td>
<td>Substitutes the %symbol sequence with a character string that represents the hexadecimal value of the symbol.</td>
</tr>
<tr>
<td>&quot;</td>
<td>Macro string delimiter</td>
<td>Allows the use of macro arguments as literal strings.</td>
</tr>
<tr>
<td>^</td>
<td>Macro local label override</td>
<td>Prevents name mangling on labels in macros.</td>
</tr>
</tbody>
</table>

Example

The macro definition:

```
CONST.D .MACRO dx,v
  movh    dx,#@his(v) ;body
```
addi    dx,dx,#@los(v)
.label    .ENDM ;terminator

The macro call:

.SDECL   ".text",code
.SECT    ".text"
CONST.D  d4,0x12345678

The macro expands as follows:

movh   d4,#@his(0x12345678)
addi   d4,d4,#@los(0x12345678)

Related Information

Section 3.10, Macro Operations

.DUP, .ENDM (Duplicate sequence of source lines)
.DUPA, .ENDM (Duplicate sequence with arguments)
.DUPC, .ENDM (Duplicate sequence with characters)
.DUPF, .ENDM (Duplicate sequence in loop)
.PMACRO (Undefine macro)
.DEFINE (Define a substitution string)
.MESSAGE

Syntax

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

Description

With the `.MESSAGE` directive you tell the assembler to print a message to stderr during the assembling process.

An arbitrary number of strings and expressions, in any order but separated by commas with no intervening white space, can be specified to describe the nature of the generated message. If you use expressions, the assembler outputs the result. The assembler outputs a space between each argument.

The error and warning counts will not be affected. The `.MESSAGE` directive is for example useful in combination with conditional assembly to indicate which part is assembled. The assembling process proceeds normally after the message has been printed.

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

A label is not allowed with this directive.

Example

```assembly
.DEFINES LONG "SHORT"
.MESSAGE 'This is a LONG string'
.MESSAGE "This is a LONG string"
```

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

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

Related Information

`.FAIL` (Programmer generated error)

`.WARNING` (Programmer generated warning)
.MISRAC

Syntax

.MISRAC string

Description

The C compiler can generate the .MISRAC directive to pass the compiler’s MISRA-C settings to the object file. The linker performs checks on these settings and can generate a report. It is not recommended to use this directive in hand-coded assembly.

Example

.MISRAC 'MISRA-C:2004,64,e2,0b,e,e11,27,6,ef83,e1,ef,66,cb75,af1,eff,e7,e7f,8d,63,87ff7,4'

Related Information

Section 4.7, C Code Checking: MISRA-C

C compiler option --misrac
.NAME

Syntax

(NAME string

Description

With the .NAME directive you specify the name of the original C source module. This directive is generated by the C compiler. You do not need this directive in hand-written assembly.

Example

.NAME "main.c"
.ORG

Syntax

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

Description

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

This directive uses the following arguments:

| abs-loc | Initial value to assign to the run-time location counter. abs-loc must be an absolute expression. If abs_loc is not specified, then the value is zero. |
| sect_type | An optional section type: code or data |
| attribute | An optional section attribute: init, noread, noclear, max, rom, group(string), cluster(string), protect |

For more information about the section types and attributes see the assembler directive .SDECL.

A label is not allowed with this directive.

Example

; define a section at location 100 decimal
.org 100

; define a relocatable nameless section
.org

; define a relocatable data section
.org ,data

; define a data section at 0x8000
.org 0x8000,data

Related Information

.SDECL (Declare section name and attributes)

.SECT (Activate a declared section)
.PMACRO

Syntax

.PMACRO  symbol[,symbol]...

Description

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

The assembler does not allow a label with this directive.

Example

.PMACRO MAC1,MAC2

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

Related Information

.MACRO, .ENDM (Define a macro)
**.SDECL**

**Syntax**

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

**Description**

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

The *name* specifies the name of the section. The *type* operand specifies the section’s type and must be one of:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CODE</td>
<td>Code section.</td>
</tr>
<tr>
<td>DATA</td>
<td>Data section.</td>
</tr>
</tbody>
</table>

The defined *attributes* are:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Allowed on type</th>
</tr>
</thead>
<tbody>
<tr>
<td>AT <em>address</em></td>
<td>Locate the section at the given <em>address</em>.</td>
<td>CODE, DATA</td>
</tr>
<tr>
<td>CLEAR</td>
<td>Sections are zeroed at startup.</td>
<td>DATA</td>
</tr>
<tr>
<td>CLUSTER(‘*name’)</td>
<td>Cluster code sections with companion debug sections. Used by the linker during removal of unreferenced sections. The name must be unique for this module (not for the application).</td>
<td>CODE, DATA</td>
</tr>
<tr>
<td>INIT</td>
<td>Defines that the section contains initialization data, which is copied from ROM to RAM at program startup.</td>
<td>CODE, DATA</td>
</tr>
<tr>
<td>LINKONCE ‘*tag’</td>
<td>For internal use only.</td>
<td></td>
</tr>
<tr>
<td>MAX</td>
<td>When data sections with the same name occur in different object modules with the MAX attribute, the linker generates a section of which the size is the maximum of the sizes in the individual object modules.</td>
<td>DATA</td>
</tr>
<tr>
<td>NOCLEAR</td>
<td>Sections are not zeroed at startup. This is a default attribute for data sections. This attribute is only useful with BSS sections, which are cleared at startup by default.</td>
<td>DATA</td>
</tr>
<tr>
<td>NOINIT</td>
<td>Defines that the section contains no initialization data.</td>
<td>CODE, DATA</td>
</tr>
<tr>
<td>NOREAD</td>
<td>Defines that the section can be executed from but not read.</td>
<td>CODE</td>
</tr>
<tr>
<td>PROTECT</td>
<td>Tells the linker to exclude a section from unreferenced section removal and duplicate section removal.</td>
<td>CODE, DATA</td>
</tr>
<tr>
<td>ROM</td>
<td>Section contains data to be placed in ROM. This ROM area is not executable.</td>
<td>CODE, DATA</td>
</tr>
</tbody>
</table>
Section names

The *name* of a section can have a special meaning for locating sections. The name of code sections should always start with ".text". With data sections, the prefix in the name is important. The prefix determines if the section is initialized, constant or uninitialized and which addressing mode is used. See the following table.

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

Note that the compiler uses the following name convention:

*prefix.module_name.function_or_object_name*

**Example**

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

.sdecl  ".data.t.var1", DATA  ; declare data section
.sect   ".data.t.var1"        ; activate section

.sdecl  ".text.intvec.00a", CODE  ; declare interrupt
    ; vector table entry for interrupt 10
.sect   ".text.intvec.00a"    ; activate section

.sdecl  ".data.t.abssec",data at 0x100
    ; absolute section
.sect   ".data.t.abssec"     ; activate section
```
Related Information

.\texttt{SECT} (Activate a declared section)

.\texttt{ORG} (Initialize a nameless section)
.SECT

Syntax

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

Description

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

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

Example

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

Related Information

.SDECL (Declare section name and attributes)

.ORG (Initialize a nameless section)
.SET

Syntax

\texttt{symbol \ .SET \ expression}

\texttt{.SET \ symbol \ expression}

Description

With the \texttt{.SET} directive you assign the value of \texttt{expression} to symbol \textit{temporarily}. If a symbol was defined with the \texttt{.SET} directive, you can redefine that symbol in another part of the assembly source, using the \texttt{.SET} directive again. Symbols that you define with the \texttt{.SET} directive are always local: you cannot define the symbol global with the \texttt{.GLOBAL} directive.

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

Example

\texttt{COUNT \ .SET \ 0 \ ; Initialize count. Later on you can}
\texttt{\ ; assign other values to the symbol}

Related Information

\texttt{.EQU} (Set permanent value to a symbol)
**.SIZE**

**Syntax**

```
.SIZE symbol,expression
```

**Description**

With the `.SIZE` directive you set the size of the specified `symbol` to the value represented by `expression`. The `.SIZE` directive may occur anywhere in the source file unless the specified symbol is a function. In this case, the `.SIZE` directive must occur after the function has been defined.

**Example**

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

**Related Information**

`.TYPE` (Set symbol type)
.SPACE

Syntax

[label:] .SPACE expression

Description

The .SPACE directive reserves a block in memory. The reserved block of memory is not initialized to any value.

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

The expression specifies the number of MAUs (Minimal Addressable Units) to be reserved, and how much the location counter will advance. The expression must evaluate to an integer greater than zero and cannot contain any forward references (symbols that have not yet been defined). For the TriCore the MAU size is 8 (1 byte).

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

Example

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

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

Related Information

.BYTE (Define a constant byte)
**.TYPE**

**Syntax**

```
symbol .TYPE typeid
```

**Description**

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

- **FUNC** The symbol is associated with a function or other executable code.
- **OBJECT** The symbol is associated with an object such as a variable, an array, or a structure.
- **FILE** The symbol name represents the filename of the compilation unit.

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

**Example**

```
Afunc: .type func
```

**Related Information**

```
.SIZE (Set symbol size)
```
.UNDEF

Syntax

.UNDEF symbol

Description

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

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

The assembler does not allow a label with this directive.

Example

The following example undefines the LEN substitution string that was previously defined with the .DEFINE directive:

.UNDEF LEN

Related Information

.DEFINITE (Define a substitution string)
.WARNING

Syntax

.WARNING {str|exp}[,{str|exp}]...

Description

With the .WARNING directive you tell the assembler to print a warning message to stderr during the assembling process.

An arbitrary number of strings and expressions, in any order but separated by commas with no intervening white space, can be specified to describe the nature of the generated warning. If you use expressions, the assembler outputs the result. The assembler outputs a space between each argument.

The total warning count will be incremented as with any other warning. The .WARNING directive is for example useful in combination with conditional assembly to indicate which part is assembled. The assembling process proceeds normally after the message has been printed.

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

A label is not allowed with this directive.

Example

.WARNING 'Parameter out of range'

This results in the warning:

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

Related Information

.FAIL (Programmer generated error)

.MESSAGE (Programmer generated message)
.WEAK Syntax

.WEAK symbol[,symbol]...

Description

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

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

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

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

Example

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

Related Information

.EXTERN (Import global section symbol)

.GLOBAL (Declare global section symbol)
.WORD, .HALF

Syntax

[label:] .WORD argument[,argument]...
[label:] .HALF argument[,argument]...

Description

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

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

An argument can be a single- or multiple-character string constant, an expression or empty.

Multiple arguments are stored in sets of four or two bytes. One or more arguments can be null (indicated by two adjacent commas), in which case the corresponding byte location will be filled with zeros.

The value of the arguments must be in range with the size of the directive; floating-point numbers are not allowed. If the evaluated argument is too large to be represented in a word / halfword, the assembler issues a warning and truncates the value.

String constants

Single-character strings are stored in the most significant byte of a word / halfword, where the lower seven bits in that byte represent the ASCII value of the character, for example:

```assembly
.WORD 'R' ; = 0x52000000
.HALF 'R' ; = 0x5200
```

Multiple-character strings are stored in consecutive byte addresses, as shown below. The standard C language escape characters like \n are permitted.

```assembly
.WORD 'ABCD' ; = 0x44434241
```

Example

When a string is supplied as argument of a directive that initializes multiple bytes, each character in the string is stored in consecutive bytes whose lower seven bits represent the ASCII value of the character. For example:

```assembly
HTBL: .HALF 'ABC','D' ; results in 0x42410004400, the 'C' is truncated
WTBL: .WORD 'ABC' ; results in 0x43424100
```

Related Information

_BYTE (Define a constant byte)

_SPACE (Define Storage)
3.9.2. Assembler Controls

Controls start with a $ as the first character on the line. Unknown controls are ignored after a warning is issued.

Overview of assembler listing controls

<table>
<thead>
<tr>
<th>Control</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$LIST ON/OFF</td>
<td>Print / do not print source lines to list file</td>
</tr>
<tr>
<td>$LIST &quot;flags&quot;</td>
<td>Exclude / include lines in assembly list file</td>
</tr>
<tr>
<td>$PAGE</td>
<td>Generate form feed in list file</td>
</tr>
<tr>
<td>$PAGE settings</td>
<td>Define page layout for assembly list file</td>
</tr>
<tr>
<td>$PRCTL</td>
<td>Send control string to printer</td>
</tr>
<tr>
<td>$STITLE</td>
<td>Set program subtitle in header of assembly list file</td>
</tr>
<tr>
<td>$TITLE</td>
<td>Set program title in header of assembly list file</td>
</tr>
</tbody>
</table>

Overview of miscellaneous assembler controls

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

Syntax

$CASE ON
$CASE OFF

Default

$CASE ON

Description

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

Note that the instruction mnemonics, register names, directives and controls are always treated case insensitive.

Example

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

Related Information

Assembler option --case-insensitive
$CPU_TCnum, $DMU_TCnum, $PMI_TCnum, $PMU_TCnum

Syntax

$CPU_TCnum  ON
$CPU_TCnum  OFF

$DMU_TCnum  ON
$DMU_TCnum  OFF

$PMI_TCnum  ON
$PMI_TCnum  OFF

$PMU_TCnum  ON
$PMU_TCnum  OFF

Description

With these controls you can enable or disable specific CPU functional problem checks.

When you use this control, the define __defect_TCnum__ is set to 1.

Example

$CPU_TC018 ON   ; enable assembler check for CPU
               ; functional problem CPU_TC.018,
               ; __CPU_TC018__ is defined

Related Information

Assembler option --silicon-bug

Chapter 16, CPU Problem Bypasses and Checks
$DEBUG

Syntax

$DEBUG ON
$DEBUG OFF
$DEBUG "flags"

Default

$DEBUG "AhLS"

Description

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

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

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a/A</td>
<td>Assembly source line information</td>
</tr>
<tr>
<td>h/H</td>
<td>Pass high level language debug information (HLL)</td>
</tr>
<tr>
<td>l/L</td>
<td>Assembler local symbols debug information</td>
</tr>
<tr>
<td>s/S</td>
<td>Smart debug information</td>
</tr>
</tbody>
</table>

You cannot specify $DEBUG "ah". Either the assembler generates assembly source line information, or it passes HLL debug information.

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

Example

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

Related Information

Assembler option --debug-info
$FPU

Syntax

$FPU

Description

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

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

Example

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

Related Information

Assembler option --fpu-present
**$HWONLY**

**Syntax**

$HWONLY

**Description**

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

With the $HWONLY control you instruct the assembler to encode all instruction as they are. The assembler does not substitute instructions with other, faster or smaller instructions.

**Example**

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

**Related Information**

Assembler option `--optimize=+generics`
$IDENT

Syntax

$IDENT LOCAL
$IDENT GLOBAL

Default

$IDENT LOCAL

Description

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

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

Example

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

Related Information

Assembler directive .GLOBAL

Assembler directive .LOCAL

Assembler option --symbol-scope
$LIST ON/OFF

Syntax

$LIST ON
$LIST OFF

Default

$LIST ON

Description

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

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

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

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

Example

... ; source line in list file
$LIST OFF
... ; source line not in list file
$LIST ON
... ; source line also in list file

Related Information

Assembler option --list-file

Assembler control $LIST "flags"

Assembler function @LST()
$LIST "flags"

Syntax

$LIST "flags"

You can set the following flags:

- **d/D** List section directives (.SDECL, .SECT)
- **e/E** List symbol definition directives
- **g/G** List expansion of generic instructions
- **i/I** List generic instructions
- **m/M** List macro definitions
- **n/N** List empty source lines (newline)
- **p/P** List conditional assembly
- **q/Q** List equate and set directives (.EQU, .SET)
- **r/R** List relocations characters 'r'
- **v/V** List equate and set values
- **w/W** Wrap source lines
- **x/X** List macro expansions
- **y/Y** List cycle counts
- **z/Z** List define expansions

**Default**

$LIST "dEGiMnPqrVwXyZ"

**Description**

If you generate a list file with the assembler option --list-file, you can use the $LIST control to specify which type of source lines the assembler must exclude from the list file. Without the assembler option --list-file this control has no effect.

To switch a flag 'on', use a lowercase letter. To switch a flag off, use an uppercase letter.

**Example**

The following example also includes macro definitions and equate and set values in the list file:

```
; begin of source
$LIST "mv"
```

**Related Information**

Assembler option --list-file

Assembler control $LIST ON/OFF
Assembler option --list-format
$MMU

Syntax

$MMU

Description

With the $MMU control you instruct the assembler to accept and encode memory management instructions in the assembly source file.

When you use this control, the define __MMU__ is set to 1.

Example

; begin of source
$MMU ; the use of memory management instructions
      ; in this source is allowed.

Related Information

Assembler option --mmu-present
$OBJECT

Syntax

$OBJECT "file"
$OBJECT OFF

Default

$OBJECT

Description

With the $OBJECT control you can specify an alternative name for the generated object file. With the $OBJECT OFF control, the assembler does not generate an object file at all.

Example

; Begin of source
$object "x1.o" ; generate object file x1.o

Related Information

Assembler option --output
$PAGE

Syntax

$PAGE [pagewidth[,pagelength[,blanktop[,blankbtm[,blankleft]]]]]

Default

$PAGE 132,72,0,0,0

Description

If you generate a list file with the assembler option --list-file, you can use the $PAGE control to format the generated list file.

The arguments may be any positive absolute integer expression, and must be separated by commas.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pagewidth</td>
<td>Number of columns per line. The default is 132, the minimum is 40.</td>
</tr>
<tr>
<td>pagelength</td>
<td>Total number of lines per page. The default is 72, the minimum is 10.</td>
</tr>
<tr>
<td></td>
<td>As a special case, a page length of 0 turns off page breaks.</td>
</tr>
<tr>
<td>blanktop</td>
<td>Number of blank lines at the top of the page. The default is 0, the</td>
</tr>
<tr>
<td></td>
<td>minimum is 0 and the maximum must be a value so that (blanktop + blankbtm) ≤ (pagelength - 10).</td>
</tr>
<tr>
<td>blankbtm</td>
<td>Number of blank lines at the bottom of the page. The default is 0, the</td>
</tr>
<tr>
<td></td>
<td>minimum is 0 and the maximum must be a value so that (blanktop + blankbtm) ≤ (pagelength - 10).</td>
</tr>
<tr>
<td>blankleft</td>
<td>Number of blank columns at the left of the page. The default is 0, the</td>
</tr>
<tr>
<td></td>
<td>minimum is 0, and the maximum must maintain the relationship: blankleft &lt; pagewidth.</td>
</tr>
</tbody>
</table>

If you use the $PAGE control without arguments, it causes a 'formfeed': the next source line is printed on the next page in the list file. The $PAGE control itself is not printed.

Example

$PAGE ; formfeed, the next source line is printed
       ; on the next page in the list file.

$PAGE 96 ; set page width to 96. Note that you can
       ; omit the last four arguments.

$PAGE ,,3,3 ; use 3 line top/bottom margins.

Related Information

Assembler option --list-file
**$PRCTL**

**Syntax**

```
$PRCTL exp|string[,exp|string]...
```

**Description**

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

The `$PRCTL` control simply concatenates its arguments and sends them to the listing file (the control line itself is not printed unless there is an error).

You can specify the following arguments:

- `expr` A byte expression which may be used to encode non-printing control characters, such as ESC.
- `string` An assembler string, which may be of arbitrary length, up to the maximum assembler-defined limits.

The `$PRCTL` control can appear anywhere in the source file; the assembler sends out the control string at the corresponding place in the listing file.

If a `$PRCTL` control is the last line in the last input file to be processed, the assembler insures that all error summaries, symbol tables, and cross-references have been printed before sending out the control string. In this manner, you can use a `$PRCTL` control to restore a printer to a previous mode after printing is done.

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

**Example**

```
$PRCTL $1B,'E' ; Reset HP LaserJet printer
```

**Related Information**

Assembler option `--list-file`
$STITLE

Syntax

$STITLE "string"

Default

$STITLE ""

Description

If you generate a list file with the assembler option --list-file, you can use the $STITLE control to specify the program subtitle which is printed at the top of all succeeding pages in the assembler list file below the title.

The specified subtitle is valid until the assembler encounters a new $STITLE control. By default, the subtitle is empty.

The $STITLE control itself will not be printed in the source listing.

If the page width is too small for the title to fit in the header, it will be truncated.

Example

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

Related Information

Assembler option --list-file

Assembler control $TITLE
$TC2

Syntax

$TC2

Description

With the $TC2 control you instruct the assembler to accept and encode TriCore 2 instructions in the assembly source file.

When you use this control, the define __TC2__ is set to 1.

Example

; begin of source
$TC2 ; the use of TriCore 2 instructions
      ; in this source is allowed.

Related Information

Assembler option --core
$TITLE

Syntax

$TITLE "string"

Default

$TITLE ""

Description

If you generate a list file with the assembler option --list-file, you can use the $TITLE control to specify the program title which is printed at the top of each page in the assembler list file.

The specified title is valid until the assembler encounters a new $TITLE control. By default, the title is empty.

The $TITLE control itself will not be printed in the source listing.

If the page width is too small for the title to fit in the header, it will be truncated.

Example

$TITLE 'This is the title'

Related Information

Assembler option --list-file

Assembler control $STITLE
$WARNING OFF

Syntax

$WARNING OFF [number]

Default

All warnings are reported.

Description

This control allows you to disable all or individual warnings. The number argument must be a valid warning message number.

Example

$WARNING OFF ; all warning messages are suppressed

$WARNING OFF 135 ; suppress warning message 135

Related Information

Assembler option --no-warnings
3.10. Macro Operations

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

Some patterns contain variable entries which change for each repetition of the pattern. Others are subject to conditional assembly.

When a macro is called, the assembler executes the macro and replaces the call by the resulting in-line source statements. 'In-line' means that all replacements act as if they are on the same line as the macro call. The generated statements may contain substitutable arguments. The statements produced by a macro can be any processor instruction, almost any assembler directive, or any previously-defined macro. Source statements resulting from a macro call are subject to the same conditions and restrictions as any other statements.

Macros can be nested. The assembler processes nested macros when the outer macro is expanded.

3.10.1. Defining a Macro

The first step in using a macro is to define it.

The definition of a macro consists of three parts:

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

A macro definition takes the following form:

```assembly
macro_name .MACRO [argument[,argument]...]  
...  
macro_definition_statements  
...  
.ENDM
```

For more information on the definition see the description of the .MACRO directive.

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

3.10.2. Calling a Macro

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

```assembly
[label] macro_name [argument[,argument]...]  ; comment
```

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

The name of the macro. This may not start in the first column.

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

An optional comment.

The following applies to macro arguments:

- Each argument must correspond one-to-one with the formal arguments of the macro definition. If the macro call does not contain the same number of arguments as the macro definition, the assembler issues a warning.

- If an argument has an embedded comma or space, you must surround the argument by single quotes (').

- You can declare a macro call argument as null in three ways:
  - enter delimiting commas in succession with no intervening spaces
    
    `macroname ARG1,,ARG3 ; the second argument is a null argument`

  - terminate the argument list with a comma, the arguments that normally would follow, are now considered null
    
    `macroname ARG1, ; the second and all following arguments are null`

  - declare the argument as a null string

- No character is substituted in the generated statements that reference a null argument.

3.10.3. Using Operators for Macro Arguments

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\</td>
<td>Macro argument concatenation</td>
<td>Concatenates a macro argument with adjacent alphanumeric characters.</td>
</tr>
<tr>
<td>?</td>
<td>Return decimal value of symbol</td>
<td>Substitutes the <code>?symbol</code> sequence with a character string that represents the decimal value of the symbol.</td>
</tr>
<tr>
<td>%</td>
<td>Return hex value of symbol</td>
<td>Substitutes the <code>%symbol</code> sequence with a character string that represents the hexadecimal value of the symbol.</td>
</tr>
<tr>
<td>&quot;</td>
<td>Macro string delimiter</td>
<td>Allows the use of macro arguments as literal strings.</td>
</tr>
<tr>
<td>^</td>
<td>Macro local label override</td>
<td>Prevents name mangling on labels in macros.</td>
</tr>
</tbody>
</table>
Example: Argument Concatenation Operator - \

Consider the following macro definition:

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

The macro is called as follows:

```
SWAP_MEM  0,1
```

The macro expands as follows:

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

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

Without the `\` operator the macro would expand as:

```
LD.W  D0,[AREG1]
LD.W  D1,[AREG2]
ST.W  [AREG1],D1
ST.W  [AREG2],D0
```

which results in an assembler error (invalid operand).

Example: Decimal Value Operator - ?

Instead of substituting the formal arguments with the actual macro call arguments, you can also use the value of the macro call arguments.

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

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

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

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

.ENDM

The macro first expands as follows:

LD.W D0,lab\AREG
LD.W D1,lab\BREG
ST.W _lab\AREG,D1
ST.W _lab\BREG,D0

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

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

Because of the concatenation operator '"' the strings are concatenated:

LD.W D0,lab1
LD.W D1,lab2
ST.W _lab1,D1
ST.W _lab2,D0

Example: Hex Value Operator - %

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

Consider the following macro definition:

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

The macro is called after NUM has been set to 10:

NUM .SET 10
 GEN_LAB HEX,NUM,NOP

The macro expands as follows:

HEXA NOP

The %VAL argument is replaced by the character 'A' which represents the hexadecimal value 10 of the argument VAL.
Example: Argument String Operator - "

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

Consider the following macro definition:

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

The macro is called as follows:

```assembly
STR_MAC  ABCD
```

The macro expands as follows:

```assembly
 .BYTE   'ABCD'
```

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

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

If the macro is called as follows:

```assembly
STR_MAC  sentence
```

it expands as:

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

Macro Local Label Override Operator - ^

If you use labels in macros, the assembler normally generates another unique name for the labels (such as LAB__M_L000001).

The macro ^-operator prevents name mangling on macro local labels.

Consider the following macro definition:

```assembly
INIT    .MACRO  addr
LAB:     LD.W   D0,^addr
       .ENDM
```

The macro is called as follows:
LAB:
    INIT LAB

The macro expands as:

LAB__M_L000001: LD.W D0,LAB

If you would have omitted the ^ operator, the macro preprocessor would choose another name for LAB because the label already exists. The macro would expand like:

LAB__M_L000001: LD.W D0,LAB__M_L000001
Chapter 4. Using the C Compiler

This chapter describes the compilation process and explains how to call the C compiler.

The TASKING VX-toolset for TriCore under Eclipse can use the internal builder (default) or the TASKING
makefile generator (external builder) to build your entire embedded project, from C source till the final
ELF/DWARF object file which serves as input for the debugger.

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

On the command line it is possible to call the C compiler separately from the other tools. However, it is
recommended to use the control program for command line invocations of the toolset (see Section 9.1,
Control Program). With the control program it is possible to call the entire toolset with only one command
line.

The C compiler takes the following files for input and output:

This chapter first describes the compilation process which consists of a frontend and a backend part.
Next it is described how to call the C compiler and how to use its options. An extensive list of all options
and their descriptions is included in Section 11.1, C Compiler Options. Finally, a few important basic tasks
are described, such as including the C startup code and performing various optimizations.

4.1. Compilation Process

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

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

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

Frontend phases

1. The preprocessor phase:

   The preprocessor includes files and substitutes macros by C source. It uses only string manipulations
   on the C source. The syntax for the preprocessor is independent of the C syntax but is also described

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

3. The parser phase:

The tokens are fed to a parser for the C grammar. The parser performs a syntactic and semantic analysis of the program, and generates an intermediate representation of the program. This code is called MIL (Medium level Intermediate Language).

4. The frontend optimization phase:

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

**Backend phases**

1. Instruction selector phase:

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

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

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

3. Register allocator phase:

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

4. The backend optimization phase:

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

5. The code generation/formatter phase:

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

**4.2. Calling the C Compiler**

The TASKING VX-toolset for TriCore under Eclipse can use the internal builder (default) or the TASKING makefile generator (external builder) to build your entire project. After you have built your project, the output files are available in a subdirectory of your project directory, depending on the active configuration you have set in the **C/C++ Build » Settings** page of the **Project » Properties** dialog.

**Building a project under Eclipse**

You have several ways of building your project:

- Incremental Build All ( ). Only the components affected by modified files in that particular project are built.
To incrementally build all open projects, select **Project » Build All**.

- **Build Selected File(s)**. This compiles and assembles the selected file(s) without calling the linker.
  1. In the C/C++ Projects view, select the files you want to compile.
  2. Right-click in the C/C++ Projects view and select **Build Selected File(s)**.

- **Build Individual Project**.

  To build individual projects incrementally, select **Project » Build Project**.

- **Rebuild Project**. This builds every file in the project whether or not a file has been modified since the last build. A rebuild is a clean followed by a build.
  1. Select **Project » Clean...**
  2. Enable the option **Start a build immediately** and click **OK**.

- **Build Automatically**. This performs a Build All whenever any project file is saved, such as your makefile.

  This way of building is not recommended for C/C++ development, but to enable this feature select **Project » Build Automatically** and ensure there is a check mark beside the **Build Automatically** menu item.

**Select a target processor (core)**

Processor options affect the invocation of all tools in the toolset. In Eclipse you only need to set them once. Based on the target processor, the compiler includes a special function register file. This is a regular include file which enables you to use virtual registers that are located in memory.

1. From the **Project** menu, select **Properties**
   
   *The Properties dialog appears.*

2. In the left pane, expand **C/C++ Build** and select **Processor**.
   
   *In the right pane the Processor page appears.*

3. From the **Processor Selection** list, select a processor.

**To access the C/C++ compiler options**

1. From the **Project** menu, select **Properties**
   
   *The Properties dialog appears.*

2. In the left pane, expand **C/C++ Build** and select **Settings**.
   
   *In the right pane the Settings appear.*

3. On the Tool Settings tab, select **C/C++ Compiler**.
4. Select the sub-entries and set the options in the various pages.

Note that the C/C++ compiler options are used to create an object file from a C or C++ file. The options you enter in the Assembler page are not only used for hand-coded assembly files, but also for intermediate assembly files.

You can find a detailed description of all C compiler options in Section 11.1, C Compiler Options.

Invocation syntax on the command line (Windows Command Prompt):

c.tc [ option]... [file]...

4.3. The C Startup Code

You need the run-time startup code to build an executable application. The startup code consists of the following components:

- **Initialization code.** This code is executed when the program is initiated and before the function main() is called. It initializes the processor’s registers and the application C variables.

- **Exit code.** This controls the close down of the application after the program’s main function terminates.

- **Trap vector table.** This contains default trap vectors. See also Section 1.8.4, Interrupt and Trap Functions.

To add the C startup code to your project

When you create a new project with the New C/C++ Project wizard (File » New » Other... » TASKING C/C++ » TASKING VX-toolset for TriCore C/C++ Project), fill in the dialogs and enable the option Add C startup code to the project in the following dialog (this is the default setting).
This adds the files `cstart.c` and `cstart.h` to your project. These files are copies of `lib/src/cstart.c` and `include/cstart.h`. If you do not add the startup code here, you can always add it later with File » New » Other... » TASKING C/C++ » cstart.c/cstart.h Files.

**To change the C Startup Code in Eclipse**

1. Double-click on the file `cstart.c`.

   *The cstart.c file opens in the editor area with several tabs.*
2. You can edit the C startup code directly in the `cstart.c` tab or make changes to the other tabs (Configuration, Register, cstart.h).

   The file `cstart.c` is updated automatically according to the changes you make in the tabs. A * appears in front of the name of the file to indicate that the file has changes.

3. Click 🔄 or select File » Save to save the changes.

### Configuration tab

On the **Configuration** tab, you can make changes to the C startup code configuration. You can disable specific initializations and you can disable the trap vectors you do not want to be automatically defined in the startup code.

### Register tab

On the **Register** tab, you can specify the registers and their settings that must be known to the startup code. Enable the option **Initialize in startup code** to add a register setting to the startup code. If you made changes to the registers and you want to reset the registers to their original values, click on the **Set CPU defaults** button.

### 4.4. How the Compiler Searches Include Files

When you use include files (with the `#include` statement), you can specify their location in several ways. The compiler searches the specified locations in the following order:
1. If the `#include` statement contains an absolute pathname, the compiler looks for this file. If no path or a relative path is specified, the compiler looks in the same directory as the source file. This is only possible for include files that are enclosed in "".

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

2. When the compiler did not find the include file, it looks in the directories that are specified in the C/C++ Compiler » Include Paths page in the C/C++ Build » Settings » Tool Settings tab of the Project Properties dialog (equivalent to the `-I` command line option).

3. When the compiler did not find the include file (because it is not in the specified include directory or because no directory is specified), it looks in the path(s) specified in the environment variable `CTCINC`.

4. When the compiler still did not find the include file, it finally tries the default include directory relative to the installation directory (unless you specified option `--no-stdinc`).

**Example**

Suppose that the C source file `test.c` contains the following lines:

```c
#include <stdio.h>
#include "myinc.h"
```

You can call the compiler as follows:

```bash
ctc -Imyinclusive test.c
```

First the compiler looks for the file `stdio.h` in the directory `myinclusive` relative to the current directory. If it was not found, the compiler searches in the environment variable `CTCINC` and then in the default include directory.

The compiler now looks for the file `myinc.h`, in the directory where `test.c` is located. If the file is not there the compiler searches in the directory `myinclusive`. If it was still not found, the compiler searches in the environment variable `CTCINC` and then in the default include directory.

**4.5. Compiling for Debugging**

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

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

**To include symbolic debug information**

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

   *The Properties dialog appears.*

2. In the left pane, expand **C/C++ Build** and select **Settings**.
In the right pane the Settings appear.

3. On the Tool Settings tab, select C/C++ Compiler » Debugging.
4. Select Default in the Generate symbolic debug information box.

**Debug and optimizations**

Due to different compiler optimizations, it might be possible that certain debug information is optimized away. Therefore, if you encounter strange behavior during debugging it might be necessary to reduce the optimization level, so that the source code is still suitable for debugging. For more information on optimization see Section 4.6, Compiler Optimizations.

**Invocation syntax on the command line (Windows Command Prompt)**

The invocation syntax on the command line is:

ctc -g file.c

### 4.6. Compiler Optimizations

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

1. From the Project menu, select Properties
   
   *The Properties dialog appears.*

2. In the left pane, expand C/C++ Build and select Settings.
   
   *In the right pane the Settings appear.*

3. On the Tool Settings tab, select C/C++ Compiler » Optimization.
4. Select an optimization level in the Optimization level box.
   
or:

   In the Optimization level box select Custom optimization and enable the optimizations you want on the Custom optimization page.

**Optimization levels**

The TASKING C compiler offers four optimization levels and a custom level, at each level a specific set of optimizations is enabled.

- **Level 0 - No optimization**: No optimizations are performed. The compiler tries to achieve a 1-to-1 resemblance between source code and produced code. Expressions are evaluated in the order written in the source code, associative and commutative properties are not used.

- **Level 1 - Optimize**: Enables optimizations that do not affect the debug-ability of the source code. Use this level when you encounter problems during debugging your source code with optimization level 2.
• **Level 2 - Optimize more (default):** Enables more optimizations to reduce the memory footprint and/or execution time. This is the default optimization level.

• **Level 3 - Optimize most:** This is the highest optimization level. Use this level when your program/hardware has become too slow to meet your real-time requirements.

• **Custom optimization:** you can enable/disable specific optimizations on the Custom optimization page.

### Optimization pragmas

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

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

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

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

#### 4.6.1. Generic Optimizations (frontend)

### Common subexpression elimination (CSE) (option -Oc/-OC)

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

### Expression simplification (option -Oe/-OE)

Multiplication by 0 or 1 and additions or subtractions of 0 are removed. Such useless expressions may be introduced by macros or by the compiler itself (for example, array subcripting).

### Constant propagation (option -Op/-OP)

A variable with a known value is replaced by that value.
Automatic function inlining (option -Oi/-OI)

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

Code compaction (reverse inlining) (option -Or/-OR)

Compaction is the opposite of inlining functions: large chunks of code that occur more than once, are transformed into a function. This reduces code size at the cost of execution speed.

Control flow simplification (option -Of/-OF)

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

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

- **Jump chaining**: A (conditional) jump to a label which is immediately followed by an unconditional jump may be replaced by a jump to the destination label of the second jump. This optimization speeds up execution.

- **Conditional jump reversal**: A conditional jump over an unconditional jump is transformed into one conditional jump with the jump condition reversed. This reduces both the code size and the execution time.

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

Subscript strength reduction (option -Os/-OS)

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

Loop transformations (option -Ol/-OL)

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

Forward store (option -Oo/-OO)

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

4.6.2. Core Specific Optimizations (backend)

Coalescer (option -Oa/-OA)

The coalescer seeks for possibilities to reduce the number of moves (MOV instruction) by smart use of registers. This optimizes both speed as code size.
Peephole optimizations (option -Oy/-OY)

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

Align loop bodies (option -On/-ON)

Loop bodies are aligned to lower the number of fetches required to retrieve the loop body.

Instruction Scheduler (option -Ok/-OK)

When two instructions need the same machine resource - like a bus, register or functional unit - at the same time, they suffer a structural hazard, which stalls the pipeline. This optimization tries to rearrange instructions to avoid structural hazards, for example by inserting another non-related instruction, or pairing a L/S instruction with a data arithmetic instruction in order to fill both pipelines as much as possible.

IFconversion (option -Ov/-OV)

IF - ELSE constructions are transformed into predicated instructions. This avoids unnecessary jumps while the predicated instructions are optimized by the pipeline scheduler and the predicate optimization.

Software pipelining (option -Ow/-OW)

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

Use of SIMD instructions (option -Om/-OM)

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

Generic assembly optimizations (option -Og/-OG)

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

4.6.3. Optimize for Size or Speed

You can tell the compiler to focus on execution speed or code size during optimizations. You can do this by specifying a size/speed trade-off level from 0 (speed) to 4 (size). This trade-off does not turn optimization phases on or off. Instead, its level is a weight factor that is used in the different optimization phases to influence the heuristics. The higher the level, the more the compiler focusses on code size optimization. To choose a trade-off value read the description below about which optimizations are affected and the impact of the different trade-off values.

Note that the trade-off settings are directions and there is no guarantee that these are followed. The compiler may decide to generate different code if it assessed that this would improve the result.

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

1. From the Project menu, select Properties
The Properties dialog appears.

2. In the left pane, expand C/C++ Build and select Settings.

In the right pane the Settings appear.

3. On the Tool Settings tab, select C/C++ Compiler » Optimization.

4. Select a trade-off level in the Trade-off between speed and size box.

See also C compiler option --tradeoff (-t)

Instruction Selection

Trade-off levels 0, 1 and 2: the compiler selects the instructions with the smallest number of cycles.

Trade-off levels 3 and 4: the compiler selects the instructions with the smallest number of bytes.

Switch Jump Chain versus Jump Table

Instruction selection for the switch statements follows different trade-off rules. A switch statement can result in a jump chain or a jump table. The compiler makes the decision between those by measuring and weighing bytes and cycles. This weigh is controlled with the trade-off values:

<table>
<thead>
<tr>
<th>Trade-off value</th>
<th>Time</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>100%</td>
<td>0%</td>
</tr>
<tr>
<td>1</td>
<td>75%</td>
<td>25%</td>
</tr>
<tr>
<td>2</td>
<td>50%</td>
<td>50%</td>
</tr>
<tr>
<td>3</td>
<td>25%</td>
<td>75%</td>
</tr>
<tr>
<td>4</td>
<td>0%</td>
<td>100%</td>
</tr>
</tbody>
</table>

Loop Optimization

For a top-loop, the loop is entered at the top of the loop. A bottom-loop is entered at the bottom. Every loop has a test and a jump at the bottom of the loop, otherwise it is not possible to create a loop. Some top-loops also have a conditional jump before the loop. This is only necessary when the number of loop iterations is unknown. The number of iterations might be zero, in this case the conditional jump jumps over the loop.

Bottom loops always have an unconditional jump to the loop test at the bottom of the loop.

<table>
<thead>
<tr>
<th>Trade-off value</th>
<th>Try to rewrite top-loops to bottom-loops</th>
<th>Optimize loops for size/speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>no</td>
<td>speed</td>
</tr>
<tr>
<td>1</td>
<td>yes</td>
<td>speed</td>
</tr>
<tr>
<td>2</td>
<td>yes</td>
<td>speed</td>
</tr>
<tr>
<td>3</td>
<td>yes</td>
<td>size</td>
</tr>
</tbody>
</table>
Optimize loops for size/speed

Try to rewrite top-loops to bottom-loops

Optimize loops for size/speed

<table>
<thead>
<tr>
<th>Trade-off value</th>
<th>Try to rewrite top-loops to bottom-loops</th>
<th>Optimize loops for size/speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>yes</td>
<td>size</td>
</tr>
</tbody>
</table>

Example:

```c
int a;

void i( int l, int m )
{
    int i;

    for ( i = m; i < l; i++ )
    {
        a++;
    }
    return;
}
```

Coded as a bottom loop (compiled with `--tradeoff=4`) is:

```assembly
j .L2 ;; unconditional jump to loop test at bottom
.L3:
    ld.w d15,1
    add16 d15,#1
    st.w a,d15
    add16 d5,#1
.L2: ;; loop entry point
    jlt d5,d4,.L3
```

Coded as a top loop (compiled with `--tradeoff=0`) is:

```assembly
    ld.w d15,a ;; test for at least one loop iteration
    jlt d5,d4,.L2 ;; can be omitted when number of iterations is known
.L3: ;; loop entry point
    add16 d15,#1
    add16 d5,#1
    jlt d5,d4,.L3
.L2:
    st.w a,d15
```

**Automatic Function Inlining**

Trade-off levels 0, 1 and 2: the compiler inlines functions as long as the function will not grow more than the percentage specified with option `--inline-max-incr`.

Trade-off levels 3 and 4: no automatic inlining.

The following inlining is independent of the trade-off level:
Functions that are smaller or equal to the threshold specified with the option `--inline-max-size` are always inlined.

Static functions that are called only once, are always inlined.

**Code Compaction**

Trade-off levels 0 and 1: code compaction is disabled.

Trade-off level 2: only code compaction of matches outside loops.

Trade-off level 3: code compaction of matches outside loops, and matches inside loops of patterns that have an estimate execution frequency lower or equal to 10.

Trade-off level 4: code compaction of matches outside loops, and matches inside loops of patterns that have an estimate execution frequency lower or equal to 100.

For loops where the iteration count is unknown an iteration count of 10 is assumed.

For the execution frequency the compiler also accounts nested loops.

**4.7. C Code Checking: MISRA-C**

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

MISRA-C specifies a subset of the C programming language which is intended to be suitable for embedded automotive systems. It consists of a set of rules, defined in *MISRA-C:2004, Guidelines for the Use of the C Language in Critical Systems* (Motor Industry Research Association (MIRA), 2004).

The compiler also supports MISRA-C:1998, the first version of MISRA-C. You can select this version with the following C compiler option:

```
--misrac-version=1998
```

For a complete overview of all MISRA-C rules, see Chapter 17, *MISRA-C Rules*.

**Implementation issues**

The MISRA-C implementation in the compiler supports nearly all rules. Only a few rules are not supported because they address documentation, run-time behavior, or other issues that cannot be checked by static source code inspection, or because they require an application-wide overview.

During compilation of the code, violations of the enabled MISRA-C rules are indicated with error messages and the build process is halted.

MISRA-C rules are divided in required rules and advisory rules. If rules are violated, errors are generated causing the compiler to stop. With the following options warnings, instead of errors, are generated for either or both the required rules and the advisory rules:
Note that not all MISRA-C violations will be reported when other errors are detected in the input source. For instance, when there is a syntax error, all semantic checks will be skipped, including some of the MISRA-C checks. Also note that some checks cannot be performed when the optimizations are switched off.

Quality Assurance report

To ensure compliance to the MISRA-C rules throughout the entire project, the TASKING linker can generate a MISRA-C Quality Assurance report. This report lists the various modules in the project with the respective MISRA-C settings at the time of compilation. You can use this in your company’s quality assurance system to provide proof that company rules for best practice programming have been applied in the particular project.

To apply MISRA-C code checking to your application

1. From the Project menu, select Properties
   
   *The Properties dialog appears.*

2. In the left pane, expand C/C++ Build and select Settings.

   *In the right pane the Settings appear.*

3. On the Tool Settings tab, select C/C++ Compiler » MISRA-C.


5. In the MISRA-C checking box select a MISRA-C configuration. Select a predefined configuration for conformance with the required rules in the MISRA-C guidelines.

6. (Optional) In the Custom 2004 or Custom 1998 entry, specify the individual rules.

On the command line you can use the option --misrac.

c tc --misrac={all | number [-number],...}

4.8. C Compiler Error Messages

The C compiler reports the following types of error messages in the Problems view of Eclipse.

F (Fatal errors)

After a fatal error the compiler immediately aborts compilation.
E (Errors)

Errors are reported, but the compiler continues compilation. No output files are produced unless you have set the **C compiler option** `--keep-output-files` (the resulting output file may be incomplete).

W (Warnings)

Warning messages do not result into an erroneous assembly output file. They are meant to draw your attention to assumptions of the compiler for a situation which may not be correct. You can control warnings in the C/C++ Build » Settings » Tool Settings » C/C++ Compiler » Diagnostics page of the Project » Properties menu (C compiler option `--no-warnings`).

I (Information)

Information messages are always preceded by an error message. Information messages give extra information about the error.

S (System errors)

System errors occur when internal consistency checks fail and should never occur. When you still receive the system error message

```
S9##: internal consistency check failed - please report
```

please report the error number and as many details as possible about the context in which the error occurred.

Display detailed information on diagnostics

1. From the **Window** menu, select **Show View » Other » General » Problems**.

   *The Problems view is added to the current perspective.*

2. In the Problems view right-click on a message.

   *A popup menu appears.*

3. Select **Detailed Diagnostics Info**.

   *A dialog box appears with additional information.*

On the command line you can use the **C compiler option** `--diag` to see an explanation of a diagnostic message:

```
ctc --diag=[format:]{all | number,...}
```
Chapter 5. Using the C++ Compiler

This chapter describes the compilation process and explains how to call the C++ compiler. You should be familiar with the C++ language and with the ISO C language.

The C++ compiler can be seen as a preprocessor or front end which accepts C++ source files or sources using C++ language features. The output generated by the TriCore C++ compiler (cptc) is TriCore C, which can be translated with the TriCore C compiler (ctc).

The C++ compiler is part of a complete toolset, the TASKING VX-toolset for TriCore. For details about the C compiler see Chapter 4, Using the C Compiler.

Although in Eclipse you cannot run the C++ compiler separately from the other tools, this section discusses the options that you can specify for the compiler.

On the command line it is possible to call the C++ compiler separately from the other tools. However, it is recommended to use the control program for command line invocations of the toolset (see Section 9.1, Control Program). With the control program it is possible to call the entire toolset with only one command line. Eclipse also uses the control program to call the C++ compiler. Files with the extensions .cc, .cpp or .cxx are seen as C++ source files and passed to the C++ compiler.


The C++ compiler does no optimization. Its goal is to produce quickly a complete and clean parsed form of the source program, and to diagnose errors. It does complete error checking, produces clear error messages (including the position of the error within the source line), and avoids cascading of errors. It also tries to avoid seeming overly finicky to a knowledgeable C or C++ programmer.

5.1. C++ Development Structure

The next figure explains the relationship between the different parts of the TASKING VX-toolset for TriCore.
5.1.1. The Prelinker Phase

The C++ compiler provides a complete prototype implementation of an automatic instantiation mechanism. The automatic instantiation mechanism is a "linker feedback" mechanism. It works by providing additional information in the object file that is used by a "prelinker" to determine which template entities require instantiation so that the program can be linked successfully. Unlike most aspects of the C++ compiler the automatic instantiation mechanism is, by its nature, dependent on certain operating system and object file format properties. In particular, the prelinker is a separate program that accesses information about the symbols defined in object files.

Template information and instantiation request files

At the end of each compilation, the C++ compiler determines whether any template entities were referenced in the translation unit. If so, a template information file (.ti) is created (or an instantiation request file (.ii), if template information files are not being used). If no template entities were referenced in the translation unit, the file will not be created and any existing file will be removed. When template information files are being used, the instantiation request file is also removed if the template information file is removed. If an error occurs during compilation, the state of the .ii file is unchanged.

Prelinker

Once a complete set of object files has been generated, including the appropriate flags, the prelinker is invoked to determine whether any new instantiations are required or if any existing instantiations are no longer required. The command line arguments to the prelinker include a list of input files to be analyzed. The input files are the object files and libraries that constitute the application. The prelinker begins by looking for template information files for each of the object files (or instantiation request files, if template information files are not being used). If no template information (or instantiation request) files are present, the prelinker concludes that no further action is required.
If there are template information (or instantiation request) files, the prelinker reads the template information file and the current instantiation list from the instantiation request file. The instantiation list is the list of instantiations assigned to a given source file by a previous invocation of the prelinker. The prelinker produces a list of the global symbols that are referenced or defined by each of the input files. The prelinker then simulates a link operation to determine which symbols must be defined for the application to link successfully.

When the link simulation has been completed, the prelinker processes each input file to determine whether any new instantiations should be assigned to the input file or if any existing instantiations should be removed. The prelinker goes through the current instantiation list from the instantiation request file to determine whether any of the existing instantiations are no longer needed. An instantiation may be no longer needed because the template entity is no longer referenced by the program or because a user supplied specialization has been provided. If the instantiation is no longer needed, it is removed from the list (internally; the file will be updated later) and the file is flagged as requiring recompilation.

The prelinker then examines any symbols referenced by the input file. The responsibility for generating an instantiation of a given entity that has not already been defined is assigned to the first file that is capable of generating that instantiation.

If a given file contains instantiations of exported templates, the template information file contains dependency information. If a source file that is part of an exported template translation unit changes, the dependency information is used to detect such a change so that the prelinker can recompile the file and regenerate the instantiations of the exported templates.

Recompilation

Once all of the assignments have been updated, the prelinker once again goes through the list of object files. For each, if the corresponding instantiation request file must be updated, the new file is written. Only source files whose corresponding .ii file has been modified will be recompiled.

At this point the combination of the template information and instantiation request files contains the information needed to recompile the source file and a list of instantiations assigned to the source file, in the form of mangled function and static data member names.

When the prelinker invokes the C++ compiler, it provides a "definition list file", which contains a list of all the external definitions found in the object files and libraries specified on the prelinker command line. The C++ compiler reads the definition list file and determines whether each of the entries on its instantiations required list has already been defined elsewhere. The definition list file permits the C++ compiler to perform instantiations that have become necessary as a result of other instantiations that were assigned by the prelinker, without requiring that the prelinker invoke the C++ compiler an additional time to perform those instantiations. This reduces the number of iterations of the prelinker and C++ compiler that are required to generate a complete set of instantiations for a program.

If an error occurs during a recompilation, the prelinker exits without updating the remaining information files and without attempting any additional compilations.

Iteration and termination

If all recompliations complete without error, the prelink process is repeated, since an instantiation can produce the demand for another instantiation. This prelink cycle (finding uninstantiated templates, updating

Using the C++ Compiler
the appropriate .ii files, and dispatching recompilations) continues until no further recompilations are required.

When the prelinker is finished, the linker is invoked. Note that simply because the prelinker completes successfully does not assure that the linker will not detect errors. Unresolvable template references and other linker errors will not be diagnosed by the prelinker.

5.1.2. The Muncher Phase

The muncher phase implements global initialization and termination code.

The muncher phase is a special part of the linker that creates sections containing a list of pointers to the initialization and termination routines. The list of pointers is consulted at run-time by startup code invoked from _main, and the routines on the list are invoked at the appropriate times.

5.2. Calling the C++ Compiler

Under Eclipse you cannot run the C++ compiler separately. However, you can set options specific for the C++ compiler. After you have built your project, the output files are available in a subdirectory of your project directory, depending on the active configuration you have set in the C/C++ Build » Settings page of the Project » Properties dialog.

Building a project under Eclipse

You have several ways of building your project:

- Incremental Build All. Only the components affected by modified files in that particular project are built.
  
  To incrementally build all open projects, select Project » Build All.

- Build Selected File(s). This compiles and assembles the selected file(s) without calling the linker.
  
  1. In the C/C++ Projects view, select the files you want to compile.
  2. Right-click in the C/C++ Projects view and select Build Selected File(s).

- Build Individual Project.
  
  To build individual projects incrementally, select Project » Build Project.

- Rebuild Project. This builds every file in the project whether or not a file has been modified since the last build. A rebuild is a clean followed by a build.
  
  1. Select Project » Clean...
  2. Enable the option Start a build immediately and click OK.

- Build Automatically. This performs a Build All whenever any project file is saved, such as your makefile.
This way of building is not recommended for C/C++ development, but to enable this feature select **Project » Build Automatically** and ensure there is a check mark beside the **Build Automatically** menu item.

**Select a target processor (core)**

Processor options affect the invocation of all tools in the toolset. In Eclipse you only need to set them once. Based on the target processor, the compiler includes a special function register file. This is a regular include file which enables you to use virtual registers that are located in memory.

1. From the **Project** menu, select **Properties**
   
   *The Properties dialog appears.*

2. In the left pane, expand **C/C++ Build** and select **Processor**.
   
   *In the right pane the Processor page appears.*

3. From the **Processor Selection** list, select a processor.

**To access the C/C++ compiler options**

1. From the **Project** menu, select **Properties**
   
   *The Properties dialog appears.*

2. In the left pane, expand **C/C++ Build** and select **Settings**.
   
   *In the right pane the Settings appear.*

3. On the Tool Settings tab, select **C/C++ Compiler**.

4. Select the sub-entries and set the options in the various pages.

Note that C++ compiler options are only enabled if you have added a C++ file to your project, a file with the extension `.cc`, `.cpp` or `.cxx`.

Note that the options you enter in the Assembler page are also used for intermediate assembly files.

You can find a detailed description of all C++ compiler options in Section 11.2, *C++ Compiler Options*.

**Invocation syntax on the command line (Windows Command Prompt):**

```
cptc [ option]... [file]... ...
```
5.3. How the C++ Compiler Searches Include Files

When you use include files (with the `#include` statement), you can specify their location in several ways. The C++ compiler searches the specified locations in the following order:

1. If the `#include` statement contains an absolute pathname, the C++ compiler looks for this file. If no path or a relative path is specified, the C++ compiler looks in the same directory as the source file. This is only possible for include files that are enclosed in "". This first step is not done for include files enclosed in < >.

2. When the C++ compiler did not find the include file, it looks in the directories that are specified in the C/C++ Compiler » Include Paths page in the C/C++ Build » Settings » Tool Settings tab of the Project Properties dialog (equivalent to the `--include-directory (-I)` command line option).

3. When the C++ compiler did not find the include file (because it is not in the specified include directory or because no directory is specified), it looks in the path(s) specified in the environment variable CPTCINC.

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

5. If the include file is still not found, the directories specified in the `--sys-include` option are searched.

   - If the include directory is specified as -, e.g., `-I-`, the option indicates the point in the list of `-I` or `--include-directory` options at which the search for file names enclosed in `<...>` should begin. That is, the search for `<...>` names should only consider directories named in `-I` or `--include-directory` options following the `-I-`, and the directories of items 3 and 4 above. `-I-` also removes the directory containing the current input file (item 1 above) from the search path for file names enclosed in "...".

   - An include directory specified with the `--sys-include` option is considered a "system" include directory. Warnings are suppressed when processing files found in system include directories.

   - If the filename has no suffix it will be searched for by appending each of a set of include file suffixes. When searching in a given directory all of the suffixes are tried in that directory before moving on to the next search directory. The default set of suffixes is, no extension and `.stdh`. The default can be overridden using the `--incl-suffixes` command line option. A null file suffix cannot be used unless it is present in the suffix list (that is, the C++ compiler will always attempt to add a suffix from the suffix list when the filename has no suffix).

**Example**

Suppose that the C++ source file `test.cc` contains the following lines:

```cpp
#include <stdio.h>
#include "myinc.h"
```

You can call the C++ compiler as follows:

```
cptc -Imyinclude test.cc
```
First the C++ compiler looks for the file `stdio.h` in the directory `myinclude` relative to the current directory. If it was not found, the C++ compiler searches in the environment variable `CPTCINC` and then in the default `include` directory.

The C++ compiler now looks for the file `myinc.h`, in the directory where `test.cc` is located. If the file is not there the C++ compiler searches in the directory `myinclude`. If it was still not found, the C++ compiler searches in the environment variable `CPTCINC` and then in the default `include.cpp` and `include` directories.

### 5.4. C++ Compiler Error Messages

The C++ compiler reports the following types of error messages in the Problems view of Eclipse.

**F (Fatal errors)**

Catastrophic errors, also called ‘fatal errors’, indicate problems of such severity that the compilation cannot continue. For example: command-line errors, internal errors, and missing include files. If multiple source files are being compiled, any source files after the current one will not be compiled.

**E (Errors)**

Errors indicate violations of the syntax or semantic rules of the C++ language. Compilation continues, but object code is not generated.

**W (Warnings)**

Warnings indicate something valid but questionable. Compilation continues and object code is generated (if no errors are detected). You can control warnings in the C/C++ Build » Settings » Tool Settings » C/C++ Compiler » Diagnostics page of the Project » Properties menu (C++ compiler option --no-warnings).

**R (Remarks)**

Remarks indicate something that is valid and probably intended, but which a careful programmer may want to check. These diagnostics are not issued by default. Compilation continues and object code is generated (if no errors are detected). To enable remarks, enable the option Issue remarks on C++ code in the C/C++ Build » Settings » Tool Settings » C/C++ Compiler » Diagnostics page of the Project » Properties menu (C++ compiler option --remarks).

**S (Internal errors)**

Internal compiler errors are caused by failed internal consistency checks and should never occur. However, if such a ‘SYSTEM’ error appears, please report the occurrence to Altium. Please include a small C++ program causing the error.

**Message format**

By default, diagnostics are written in a form like the following:

```
cptc E0020: ["test.cc" 3] identifier "name" is undefined
```
With the command line option \texttt{--error-file}=	exttt{file} you can redirect messages to a file instead of \texttt{stderr}.

Note that the message identifies the file and line involved. Long messages are wrapped to additional lines when necessary.

With the option \textbf{C/C++ Build » Settings » Tool Settings » Global Options » Treat warnings as errors} (option \texttt{--warnings-as-errors}) you can change the severity of warning messages to errors.

For some messages, a list of entities is useful; they are listed following the initial error message:

\begin{verbatim}
cptc E0308: ["test.cc" 4] more than one instance of overloaded function "f" matches the argument list:
  function "f(int)"
  function "f(float)"
  argument types are: (double)
\end{verbatim}

In some cases, some additional context information is provided; specifically, such context information is useful when the C++ compiler issues a diagnostic while doing a template instantiation or while generating a constructor, destructor, or assignment operator function. For example:

\begin{verbatim}
cptc E0265: ["test.cc" 7] "A::A()" is inaccessible detected during implicit generation of "B::B()" at line 7
\end{verbatim}

Without the context information, it is very hard to figure out what the error refers to.

\textbf{Termination Messages}

The C++ compiler writes sign-off messages to \texttt{stderr} (the Problems view in Eclipse) if errors are detected. For example, one of the following forms of message

\begin{verbatim}
n errors detected in the compilation of "file".
1 catastrophic error detected in the compilation of "file".
n errors and 1 catastrophic error detected in the compilation of "file".
\end{verbatim}

is written to indicate the detection of errors in the compilation. No message is written if no errors were detected. The following message

\begin{verbatim}
Error limit reached.
\end{verbatim}

is written when the count of errors reaches the error limit (see the option \texttt{--error-limit}); compilation is then terminated. The message

\begin{verbatim}
Compilation terminated.
\end{verbatim}

is written at the end of a compilation that was prematurely terminated because of a catastrophic error. The message

\begin{verbatim}
Compilation aborted
\end{verbatim}
is written at the end of a compilation that was prematurely terminated because of an internal error. Such an error indicates an internal problem in the compiler. If such an internal error appears, please report the occurrence to Altium. Please include a small C++ program causing the error.
Chapter 6. Profiling

Profiling is the process of collecting statistical data about a running application. With these data you can analyze which functions are called, how often they are called and what their execution time is. This chapter describes the TASKING profiling method with code instrumentation techniques and static profiling.

6.1. What is Profiling?

Profiling is a collection of methods to gather data about your application which helps you to identify code fragments where execution consumes the greatest amount of time.

TASKING supplies a number of profiler tools each dedicated to solve a particular type of performance tuning problem. Performance problems can be solved by:

- Identifying time-consuming algorithms and rewrite the code using a more time-efficient algorithm.
- Identifying time-consuming functions and select the appropriate compiler optimizations for these functions (for example, enable loop unrolling or function inlining).
- Identifying time consuming loops and add the appropriate pragmas to enable the compiler to further optimize these loops.

A profiler helps you to find and identify the time consuming constructs and provides you this way with valuable information to optimize your application.

TASKING employs various schemes for collecting profiling data, depending on the capabilities of the target system and different information needs.

6.1.1. Four Methods of Profiling

There are several methods of profiling: recording by an instruction set simulator, profiling using the debugger, profiling with code instrumentation techniques (dynamic profiling) and profiling by the C compiler at compile time (static profiling). Each method has its advantages and disadvantages.

Profiling by an instruction set simulator

One way to gather profiling information is built into the instruction set simulator (ISS). The ISS records the time consumed by each instruction that is executed. The debugger then retrieves this information and correlates the time spent for individual instructions to C source statements.

Advantages

- it gives (cycle) accurate information with extreme fine granularity
- the executed code is identical to the non-profiled code

Disadvantages

- the method requires an ISS as execution environment
Profiling with the debugger (intrusive profiling)

The second method of profiling is built into the debugger. You specify which functions you want to profile. The debugger places breakpoints on the function entry and all its exit addresses and measures the time spent in the function and its callees.

Advantages

• the executed code is identical to the non-profiled code

Disadvantage

• each time a profiling breakpoint is hit the target is stopped and control is passed to the debugger. Although the debugger restarts the application immediately, the applications performance is significantly reduced.

Profiling using code instrumentation techniques (Dynamic Profiling)

The TASKING C compiler has an option to add code to your application which takes care of the profiling process. This is called code instrumentation. The gathered profiling data is first stored in the target's memory and will be written to a file when the application finishes execution or when the function __prof_cleanup() is called.

Advantages

• it can give a complete call graph of the application annotated with the time spent in each function and basic block

• this profiling method is execution environment independent

• the application is profiled while it executes on its aimed target taking real-life input

Disadvantage

• instrumentation code creates a significant run-time overhead, and instrumentation code and gathered data take up target memory

This method provides a valuable complement to the other two methods and is described into more detail below.

Profiling estimation by the C compiler (Static Profiling)

The TASKING C compiler has an option to generate static profile information through various heuristics and estimates. The profiling data produced this way at compile time is stored in an XML file, which can be processed and displayed using the same tools used for dynamic (run-time) profiling.

Advantages

• it can give a quick estimation of the time spent in each function and basic block

• this profiling method is execution environment independent

• the application is profiled at compile time
• it requires no extra code instrumentation, so no extra run-time overhead

Disadvantage

• it is an estimation by the compiler and therefore less accurate than dynamic profiling

This method also is described into more detail below.

6.2. Profiling using Code Instrumentation (Dynamic Profiling)

Profiling can be used to determine which parts of a program take most of the execution time.

Once the collected data are presented, it may reveal which pieces of your code execute slower than expected and which functions contribute most to the overall execution time of a program. It gives you also information about which functions are called more or less often than expected. This information not only reveal design flaws or bugs that had otherwise been unnoticed, it also reveals parts of the program which can be effectively optimized.

Important considerations

The code instrumentation method adds code to your original application which is needed to gather the profiling data. Therefore, the code size of your application increases. Furthermore, during the profiling process, the gathered data is initially stored into dynamically allocated memory of the target. The heap of your application should be large enough to store this data. Since code instrumentation is done by the compiler, assembly functions used in your program do not show up in the profile.

The profiling information is collected during the actual execution of the program. Therefore, the input of the program influences the results. If a part/function of the program is not activated while the program is profiled, no profile data is generated for that part/function.

It is not possible to profile applications that are compiled with the optimization code compaction (C compiler option --optimize=+compact). Therefore, when you turn profiling on, the compiler automatically disables parts of the code compaction optimization.

Overview of steps to perform

To obtain a profile using code instrumentation, perform the following steps:

1. Compile and link your program with profiling enabled

2. Execute the program to generate the profile data

3. Display the profile

First you need a completed project. If you are not using your own project, use the profiling example as described below.

1. From the File menu, select Import...

   The Import dialog appears.

2. Select TASKING C/C++ » TriCore Example Projects and click Next.
3. In the **Example projects** box, disable all projects except **profiling**.

4. Click **Finish**.

   The **profiling** project should now be visible in the C/C++ view.

**6.2.1. Step 1: Build your Application for Profiling**

The first step is to add the code that takes care of the profiling, to your application. This is done with C compiler options:

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

   **The Properties for profiling dialog box appears.**

2. In the left pane, expand **C/C++ Build** and select **Settings**.

   **In the right pane the Settings appear.**

3. On the Tool Settings tab, expand the **C/C++ Compiler** entry and select **Debugging**.

4. Enable one or more of the following **Generate profiling information** options (the sample profiling project already has profiling options enabled).

   • **for block counters** (not in combination with Call graph or Function timers)

   • **to build a call graph** (not in combination with Block counters)

   • **for function counters**

   • **for function timers** (not in combination with Block counters)

   **Note that the more detailed information you request, the larger the overhead in terms of execution time, code size and heap space needed. The option Generate symbolic debug information (**-debug**) does not affect profiling, execution time or code size.**

**Block counters** (not in combination with Call graph or Time)

This will instrument the code to perform basic block counting. As the program runs, it will count how many time it executed each branch of each if statement, each iteration of a for loop, and so on. Note that though you can combine Block counters with Function counters, this has no effect because Function counters is only a subset of Block counters.

**Call graph** (not in combination with Block counters)

This will instrument the code to reconstruct the run-time call graph. As the program runs it associates the caller with the gathered profiling data.

**Function counters**

This will instrument the code to perform function call counting. This is a subset of the basic Block counters.
Function timers (not in combination with Block counters)

This will instrument the code to measure the time spent in a function. This includes the time spent in all called functions (callees).

For the command line, see the C compiler option --profile (-p).

Profiling is only possible with optimization levels 0, 1 and 2. So:

5. Open the Optimization page and set the Optimization level to 2 - Optimize more.

6. Click OK to apply the new option settings and rebuild the project.

6.2.1.1. Profiling Modules and C Libraries

Profiling individual modules

It is possible to profile individual C modules. In this case only limited profiling data is gathered for the functions in the modules compiled without the profiling option. When you use the suboption Call graph, the profiling data reveals which profiled functions are called by non-profiled functions. The profiling data does not show how often and from where the non-profiled functions themselves are called. Though this does not affect the flat profile, it might reduce the usefulness of the call graph.

Profiling C library functions

Eclipse and/or the control program will link your program with the standard version of the C library. Functions from this library which are used in your application, will not be profiled. If you do want to incorporate the library functions in the profile, you must set the appropriate C compiler options in the C library makefiles and rebuild the library.

6.2.1.2. Linking Profiling Libraries

When building your application, the application must be linked against the corresponding profile library. Eclipse (or the control program) automatically select the correct library based on the profiling options you specified. However, if you compile, assemble and link your application manually, make sure you specify the correct library.

See Section 8.3, Linking with Libraries for an overview of the (profiling) libraries.

6.2.2. Step 2: Execute the Application

Once you have compiled and linked the application for profiling, it must be executed to generate the profiling data. Run the program as usual: the program should run normally taking the same input as usual and producing the same output as usual. The application will run somewhat slower than normal because of the extra time spent on collecting the profiling data.

Eclipse has already made a default simulator debug configuration for your project. Follow the steps below to run the application on the TASKING simulator, using the debugger. (In fact, you can run the application also on a target board.)

1. From the Run menu, select Debug...
The Create, manage and run configurations dialog appears.

2. Select the simulator debug configuration (TASKING Embedded C/C++ Application » profiling.simulator).

3. Click the Debug button to start the debugger and launch the profiling application. 

   Eclipse will open the TASKING Debug perspective (as specified in the configuration) and asks for confirmation.

4. Click Yes to open the TASKING Debug perspective. 

   The TASKING Debug perspective opens while the application has stopped before it enters main().

5. In the Debug view, click on the (Resume) button. 

   A file system simulation (FSS) view appears in which the application outputs the results.

   When the program has finished, the collected profiling data is saved (for details see 'After execution' below).

**Startup code**

The startup code initializes the profiling functions by calling the function __prof_init(). Eclipse will automatically make the required modifications to the startup code. Or, when you use the control program, this extracts the correct startup code from the C library.

If you use your own startup code, you must manually insert a call to the function __prof_init just before the call to main and its stack setup.

An application can have multiple entry points, such as main() and other functions that are called by interrupt. This does not affect the profiling process.

**Small heap problem**

When the program does not run as usual, this is typically caused by a shortage of heap space. In this case a message is issued (when running with file system simulation, it is displayed on the Debug console). To solve this problem, increase the size of the heap. Information about the heap is stored in the linker script file (.lsl) file which is automatically added when a project is created.

1. In the C/C++ view, expand the project tree and double-click on the file profiling.lsl to open it.

   In the editor view, the LSL file is opened, showing a number of tabs at the bottom.

2. Open the Stack/Heap tab and enter a larger value for heap.

3. Save the file.

**Presumable incorrect call graph**

The call graph is based on the compiled source code. Due to compiler optimizations the call graph may therefore seem incorrect at first sight. For example, the compiler can replace a function call immediately
followed by a return instruction by a jump to the callee, thereby merging the callee function with the caller function. In this case the time spent in the callee function is not recorded separately anymore, but added to the time spent in the caller function (which, as said before, now holds the callee function). This represents exactly the structure of your source in assembly but may differ from the structure in the initial C source.

**After execution**

When the program has finished (returning from `main()`), the exit code calls the function `__prof_cleanup(void)`. This function writes the gathered profiling data to a file on the host system using the debugger’s file system simulation features. If your program does not return from `main()`, you can force this by inserting a call to the function `__prof_cleanup()` in your application source code. Please note the double underscores when calling from C code!

The resulting profiling data file is named `amon.prf`.

If your program does not run under control of the debugger and therefore cannot use the file system simulation (FSS) functionality to write a file to the host system, you must implement a way to pass the profiling data gathered on the target to the host. Adapt the function `__prof_cleanup()` in the profiling libraries or the underlying I/O functions for this purpose.

### 6.2.3. Step 3: Displaying Profiling Results

After the function `__prof_cleanup()` has been executed, the result of the profiler can be displayed in the TASKING Profiler perspective. The profiling data in the file `amon.prf` is then converted to an XML file. This file is read and its information is displayed. To view the profiling information, open the TASKING Profiler perspective:

1. From the **Window** menu, select **Open Perspective » Other...**
   
   *The Select Perspective dialog appears.*

2. Select the **TASKING Profiler** perspective and click **OK**.
   
   *The TASKING Profiler perspective opens.*
The TASKING Profiler perspective

The TASKING Profiler perspective contains the following Views:

**Profiler view**

Shows the profiling information of all functions in all C source modules belonging to your application.

**Callers / Callees view**

The first table in this view, the *callers* table, shows the functions that called the focus function.

The second table in this view, the *callees* table, shows the functions that are called by the focus function.

- Clicking on a function (or on its table row) makes it the focus function.
- Double-clicking on a function, opens the appropriate C source module in the Editor view at the location of the function definition.
- To sort the rows in the table, click on one of the column headers.
The profiling information

Based on the profiling options you have set before compiling your application, some profiling data may be present and some may be not. The columns in the tables represent the following information:

- **Module**: The C source module in which the function resides.
- **#Line**: The line number of the function definition in the C source module.
- **Function**: The function for which profiling data is gathered and (if present) the code blocks in each function. To show or hide the block counts, in the Profiler view click the **Menu** button ( sildenafil ) and select **Show Block Counts**.
- **Total Time**: The total amount of time in seconds that was spent in this function and all of its sub-functions.
- **Self Time**: The amount of time in seconds that was spent in the function itself. This excludes the time spent in the sub-functions.
- **% in Function**: This is the relative amount of time spent in this function. These should add up to 100%.
- **Calls**: Number of times the function has been executed.
- **#Callers**: Number of functions by which the function was called.
- **#Callees**: Number of functions that was actually called from this function.
- **Contribution %**: In the caller table: shows for which part (in percent) the caller contributes to the time spent in the focus function.
  - In the callee table: shows how much time the focus function has spent relatively in each of its callees.
- **Calls %**: In the caller table: shows how often each callee was called as a percentage of all calls from the focus function.
  - In the callee table: shows how often the focus function was called from a particular caller as a percentage of all calls to the focus function.

**Toolbar icons**

<table>
<thead>
<tr>
<th>Icon</th>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[-]</td>
<td>Show/Hide Block Counts</td>
<td>Toggle. If enabled, shows profiling information for block counters.</td>
</tr>
<tr>
<td>🌐</td>
<td>Link with Editor</td>
<td>Toggle. If enabled, updates the profiling information according to the active source file.</td>
</tr>
<tr>
<td>💌</td>
<td>Select Profiling File(s)</td>
<td>Opens a dialog where you can specify profiling files for display.</td>
</tr>
<tr>
<td>🔄</td>
<td>Refresh Profiler Data</td>
<td>Updates the views with the latest profiling information.</td>
</tr>
</tbody>
</table>

**Viewing previously recorded profiling results, combining results**

Each time you run your application, new profiling information is gathered and stored in the file `amon.prf`. You can store previous results by renaming the file `amon.prf` (keep the extension `.prf`); this prevents
the existing amon.prf from being overwritten by the new amon.prf. At any time, you can reload these profiling results in the profiler. You can even load multiple .prf files into the Profiler to view the combined results.

First, open the TASKING Profiler perspective if it is not open anymore:

1. In the Profiler view, click on the (Select Profiling File(s)) button. 
   *The Select Profiling File(s) dialog appears.*
2. In the Projects box, select the project for which you want to see profiling information.
3. In the **Profiling Type** group box, select **Dynamic Profiling**.
4. In the **Profiling Files** group box, disable the option **Use default**.
5. Click the **Add...** button, select the .prf files you want to load and click **Open** to confirm your choice.
6. Make sure the correct symbol file is selected, in this example profiling.elf.
7. Click **OK** to finish.

### 6.3. Profiling at Compile Time (Static Profiling)

Just as with dynamic profiling, static profiling can be used to determine which parts of a program take most of the execution time. It can provide a good alternative if you do not want that your code is affected by extra code.

**Overview of steps to perform**

To obtain a profile using code instrumentation, perform the following steps:

1. Compile and link your program with static profiling enabled
2. Display the profile

First you need a completed project. If you are not using your own project, use the profiling example as described in Section 6.2, **Profiling using Code Instrumentation (Dynamic Profiling)**.

#### 6.3.1. Step 1: Build your Application with Static Profiling

The first step is to tell the C compiler to make an estimation of the profiling information of your application. This is done with C compiler options:

1. From the **Project** menu, select **Properties**
   *The Properties for profiling dialog box appears.*
2. In the left pane, expand **C/C++ Build** and select **Settings**.
   *In the right pane the Settings appear.*
3. On the Tool Settings tab, expand the **C/C++ Compiler** entry and select **Debugging**.

4. Enable **Static profiling**.
   
   For the command line, see the C compiler option **--profile (-p)**.
   
   Profiling is only possible with optimization levels 0, 1 and 2. So:
   
   5. Open the **Optimization** page and set the **Optimization level** to **2 - Optimize more**.
   
   6. Click **OK** to apply the new option settings and rebuild the project.

### 6.3.2. Step 2: Displaying Static Profiling Results

After your project has been built with static profiling, the result of the profiler can be displayed in the TASKING Profiler perspective. The profiling data of each individual file (**.sxml**), is combined in the XML file **profiling.xprof**. This file is read and its information is displayed. To view the profiling information, open the TASKING Profiler perspective:

1. From the **Window** menu, select **Open Perspective » Other...**

   *The Select Perspective dialog appears.*

2. Select the **TASKING Profiler** perspective and click **OK**.

   *The TASKING Profiler perspective opens. This perspective is explained in Section 6.2.3, Step 3: Displaying Profiling Results*

**To display static profiling information in the Profiler view**

1. In the Profiler view, click on the (Select Profiling File(s)) button.

   *The Select Profiling File(s) dialog appears.*

2. In the Projects box, select the project for which you want to see profiling information.

3. In the **Profiling Type** group box, select **Static Profiling**.

4. In the **Static Profiling File** group box, enable the option **Use default**.

   By default, the file **project.xprof** is used (**profiling.xprof**). If you want to specify another file, disable the option **Use default** and use the edit field and/or Browse button to specify a static profiling file (**.xprof**).

5. Click **OK** to finish.
Chapter 7. Using the Assembler

This chapter describes the assembly process and explains how to call the assembler.

The assembler converts hand-written or compiler-generated assembly language programs into machine language, resulting in object files in the ELF/DWARF object format.

The assembler takes the following files for input and output:

![Diagram of the assembly process]

The following information is described:

- The assembly process.
- How to call the assembler and how to use its options. An extensive list of all options and their descriptions is included in Section 11.3, Assembler Options.
- The various assembler optimizations.
- How to generate a list file.
- Types of assembler messages.

7.1. Assembly Process

The assembler generates relocatable output files with the extension .o. These files serve as input for the linker.

Phases of the assembly process

- Parsing of the source file: preprocessing of assembler directives and checking of the syntax of instructions
- Instruction grouping and reordering
- Optimization (instruction size and generic instructions)
- Generation of the relocatable object file and optionally a list file

The assembler integrates file inclusion and macro facilities. See Section 3.10, Macro Operations for more information.
7.2. Calling the Assembler

The TASKING VX-toolset for TriCore under Eclipse can use the internal builder (default) or the TASKING makefile generator (external builder) to build your entire project. After you have built your project, the output files are available in a subdirectory of your project directory, depending on the active configuration you have set in the C/C++ Build » Settings page of the Project » Properties dialog.

Building a project under Eclipse

You have several ways of building your project:

* Incremental Build All ( ). Only the components affected by modified files in that particular project are built.

To incrementally build all open projects, select Project » Build All.

* Build Selected File(s) ( ). This compiles and assembles the selected file(s) without calling the linker.

  1. In the C/C++ Projects view, select the files you want to compile.
  2. Right-click in the C/C++ Projects view and select Build Selected File(s).

* Build Individual Project ( ).

To build individual projects incrementally, select Project » Build Project.

* Rebuild Project ( ). This builds every file in the project whether or not a file has been modified since the last build. A rebuild is a clean followed by a build.

  1. Select Project » Clean...
  2. Enable the option Start a build immediately and click OK.

* Build Automatically. This performs a Build All whenever any project file is saved, such as your makefile.

  This way of building is not recommended for C/C++ development, but to enable this feature select Project » Build Automatically and ensure there is a check mark beside the Build Automatically menu item.

Select a target processor (core)

Processor options affect the invocation of all tools in the toolset. In Eclipse you only need to set them once. Based on the target processor, the compiler includes a special function register file. This is a regular include file which enables you to use virtual registers that are located in memory.

1. From the Project menu, select Properties

   The Properties dialog appears.

2. In the left pane, expand C/C++ Build and select Processor.
In the right pane the Processor page appears.

3. From the Processor Selection list, select a processor.

To access the assembler options

1. From the Project menu, select Properties
   
The Properties dialog appears.

2. In the left pane, expand C/C++ Build and select Settings.
   
   In the right pane the Settings appear.

3. On the Tool Settings tab, select Assembler.

4. Select the sub-entries and set the options in the various pages.

Note that the options you enter in the Assembler page are not only used for hand-coded assembly files, but also for the assembly files generated by the compiler.

You can find a detailed description of all assembler options in Section 11.3, Assembler Options.

Invocation syntax on the command line (Windows Command Prompt):

```
astc [ [option]... [file]... ]...
```

The input file must be an assembly source file (.asm or .src).

7.3. How the Assembler Searches Include Files

When you use include files (with the .INCLUDE directive), you can specify their location in several ways. The assembler searches the specified locations in the following order:

1. If the .INCLUDE directive contains an absolute path name, the assembler looks for this file. If no path or a relative path is specified, the assembler looks in the same directory as the source file.

2. When the assembler did not find the include file, it looks in the directories that are specified in the Assembler » Include Paths page in the C/C++ Build » Settings » Tool Settings tab of the Project Properties dialog (equivalent to the -I command line option).

3. When the assembler did not find the include file (because it is not in the specified include directory or because no directory is specified), it looks in the path(s) specified in the environment variable ASTCINC.

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

Example

Suppose that the assembly source file test.asm contains the following lines:
You can call the assembler as follows:

```
astc -Imyincinclude test.asm
```

First the assembler looks for the file `myinc.asm`, in the directory where `test.asm` is located. If the file is not there the assembler searches in the directory `myinclude`. If it was still not found, the assembler searches in the environment variable `ASTCINC` and then in the default `include` directory.

### 7.4. Assembler Optimizations

The assembler can perform various optimizations that you can enable or disable.

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

   *The Properties dialog appears.*

2. In the left pane, expand **C/C++ Build** and select **Settings**.

   *In the right pane the Settings appear.*

3. On the Tool Settings tab, select **Assembler » Optimization**.

4. Enable one or more optimizations.

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

**Allow generic instructions (option -Og/-OG)**

When this option is enabled, you can use generic instructions in your assembly source. The assembler tries to replace instructions by faster or smaller instructions. For example, the instruction

```
jeq d0,#0,label1
```

is replaced by

```
jz d0,label1
```

By default this option is enabled. Because shorter instructions may influence the number of cycles, you may want to disable this option when you have written timed code. In that case the assembler encodes all instructions as they are.

**Optimize instruction size (option -Os/-OS)**

When this option is enabled, the assembler tries to find the shortest possible operand encoding for instructions. By default this option is enabled.

### 7.5. Generating a List File

The list file is an additional output file that contains information about the generated code. You can customize the amount and form of information.

If the assembler generates errors or warnings, these are reported in the list file just below the source line that caused the error or warning.
To generate a list file

1. From the **Project** menu, select **Properties**
   
   The Properties dialog appears.

2. In the left pane, expand **C/C++ Build** and select **Settings**.
   
   In the right pane the Settings appear.

3. On the Tool Settings tab, select **Assembler » List File**.

4. Enable the option **Generate list file**.

5. (Optional) Enable the options to include that information in the list file.

To generate a list file for every intermediate assembly file

1. From the **Project** menu, select **Properties**
   
   The Properties dialog appears.

2. In the left pane, expand **C/C++ Build** and select **Settings**.
   
   In the right pane the Settings appear.

3. On the Tool Settings tab, select **C/C++ Compiler » Miscellaneous**.

4. Enable the option **Generate assembly list file**.

**Example on the command line (Windows Command Prompt)**

The following command generates the list file `test.lst`:

```
astc -l test.asm
```

See [Section 13.1, Assembler List File Format](#), for an explanation of the format of the list file.

7.6. Assembler Error Messages

The assembler reports the following types of error messages in the Problems view of Eclipse.

**F (Fatal errors)**

After a fatal error the assembler immediately aborts the assembly process.

**E (Errors)**

Errors are reported, but the assembler continues assembling. No output files are produced unless you have set the assembler option `--keep-output-files` (the resulting output file may be incomplete).
W (Warnings)

Warning messages do not result into an erroneous assembly output file. They are meant to draw your attention to assumptions of the assembler for a situation which may not be correct. You can control warnings in the C/C++ Build » Settings » Tool Settings » Assembler » Diagnostics page of the Project » Properties menu (assembler option --no-warnings).

Display detailed information on diagnostics

1. From the Window menu, select Show View » Other » General » Problems.
   
   The Problems view is added to the current perspective.

2. In the Problems view right-click on a message.
   
   A popup menu appears.

3. Select Detailed Diagnostics Info.
   
   A dialog box appears with additional information.

On the command line you can use the assembler option --diag to see an explanation of a diagnostic message:

\[ \text{astc --diag=[format:]\{all | number,\ldots\}} \]
Chapter 8. Using the Linker

This chapter describes the linking process, how to call the linker and how to control the linker with a script file.

The TASKING linker is a combined linker/locator. The linker phase combines relocatable object files (\texttt{.o} files, generated by the assembler), and libraries into a single relocatable linker object file (\texttt{.out}). The locator phase assigns absolute addresses to the linker object file and creates an absolute object file which you can load into a target processor. From this point the term linker is used for the combined linker/locator.

The linker can simultaneously link and locate all programs for all cores available on a target board. The target board may be of arbitrary complexity. A simple target board may contain one standard processor with some external memory that executes one task. A complex target board may contain multiple standard processors and DSPs combined with configurable IP-cores loaded in an FPGA. Each core may execute a different program, and external memory may be shared by multiple cores.

The linker takes the following files for input and output:

\begin{itemize}
\item relocatable object files \texttt{.o}
\item relocatable linker object file \texttt{.out}
\item relocatable object library \texttt{.a}
\item linker script file \texttt{.lsl}
\item linker map file \texttt{.map}
\item error messages \texttt{.elm}
\item memory definition file \texttt{.mdf}
\item Intel Hex absolute object file \texttt{.hex}
\item ELF/DWARF2 absolute object file \texttt{.elf}
\item Motorola S-record absolute object file \texttt{.sce}
\end{itemize}

This chapter first describes the linking process. Then it describes how to call the linker and how to use its options. An extensive list of all options and their descriptions is included in \textit{Section 11.4, Linker Options}.

To control the link process, you can write a script for the linker. This chapter shortly describes the purpose and basic principles of the Linker Script Language (LSL) on the basis of an example. A complete description of the LSL is included in Linker Script Language.

8.1. Linking Process

The linker combines and transforms relocatable object files (\texttt{.o}) into a single absolute object file. This process consists of two phases: the linking phase and the locating phase.

In the first phase the linker combines the supplied relocatable object files and libraries into a single relocatable object file. In the second phase, the linker assigns absolute addresses to the object file so it can actually be loaded into a target.
## Terms used in the linking process

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute object file</td>
<td>Object code in which addresses have fixed absolute values, ready to load into a target.</td>
</tr>
<tr>
<td>Address</td>
<td>A specification of a location in an address space.</td>
</tr>
<tr>
<td>Address space</td>
<td>The set of possible addresses. A core can support multiple spaces, for example in a Harvard architecture the addresses that identify the location of an instruction refer to <em>code</em> space, whereas addresses that identify the location of a data object refer to a <em>data</em> space.</td>
</tr>
<tr>
<td>Architecture</td>
<td>A description of the characteristics of a core that are of interest for the linker. This encompasses the address space(s) and the internal bus structure. Given this information the linker can convert logical addresses into physical addresses.</td>
</tr>
<tr>
<td>Copy table</td>
<td>A section created by the linker. This section contains data that specifies how the startup code initializes the data and BSS sections. For each section the copy table contains the following fields:</td>
</tr>
<tr>
<td></td>
<td>• action: defines whether a section is copied or zeroed</td>
</tr>
<tr>
<td></td>
<td>• destination: defines the section's address in RAM</td>
</tr>
<tr>
<td></td>
<td>• source: defines the sections address in ROM, zero for BSS sections</td>
</tr>
<tr>
<td></td>
<td>• length: defines the size of the section in MAUs of the destination space</td>
</tr>
<tr>
<td>Core</td>
<td>An instance of an architecture.</td>
</tr>
<tr>
<td>Derivative</td>
<td>The design of a processor. A description of one or more cores including internal memory and any number of buses.</td>
</tr>
<tr>
<td>Library</td>
<td>Collection of relocatable object files. Usually each object file in a library contains one symbol definition (for example, a function).</td>
</tr>
<tr>
<td>Logical address</td>
<td>An address as encoded in an instruction word, an address generated by a core (CPU).</td>
</tr>
<tr>
<td>LSL file</td>
<td>The set of linker script files that are passed to the linker.</td>
</tr>
<tr>
<td>MAU</td>
<td>Minimum Addressable Unit. For a given processor the number of bits between an address and the next address. This is not necessarily a byte or a word.</td>
</tr>
<tr>
<td>Object code</td>
<td>The binary machine language representation of the C source.</td>
</tr>
<tr>
<td>Physical address</td>
<td>An address generated by the memory system.</td>
</tr>
<tr>
<td>Processor</td>
<td>An instance of a derivative. Usually implemented as a (custom) chip, but can also be implemented in an FPGA, in which case the derivative can be designed by the developer.</td>
</tr>
<tr>
<td>Relocatable object file</td>
<td>Object code in which addresses are represented by symbols and thus relocatable.</td>
</tr>
<tr>
<td>Relocation</td>
<td>The process of assigning absolute addresses.</td>
</tr>
</tbody>
</table>
### 8.1.1. Phase 1: Linking

The linker takes one or more relocatable object files and/or libraries as input. A relocatable object file, as generated by the assembler, contains the following information:

- **Header information**: Overall information about the file, such as the code size, name of the source file it was assembled from, and creation date.

- **Object code**: Binary code and data, divided into various named sections. Sections are contiguous chunks of code that have to be placed in specific parts of the memory. The program addresses start at zero for each section in the object file.

- **Symbols**: Some symbols are exported - defined within the file for use in other files. Other symbols are imported - used in the file but not defined (external symbols). Generally these symbols are names of routines or names of data objects.

- **Relocation information**: A list of places with symbolic references that the linker has to replace with actual addresses. When in the code an external symbol (a symbol defined in another file or in a library) is referenced, the assembler does not know the symbol's size and address. Instead, the assembler generates a call to a preliminary relocatable address (usually 0000), while stating the symbol name.

- **Debug information**: Other information about the object code that is used by a debugger. The assembler optionally generates this information and can consist of line numbers, C source code, local symbols and descriptions of data structures.

The linker resolves the external references between the supplied relocatable object files and/or libraries and combines the files into a single relocatable linker object file.

The linker starts its task by scanning all specified relocatable object files and libraries. If the linker encounters an unresolved symbol, it remembers its name and continues scanning. The symbol may be defined elsewhere in the same file, or in one of the other files or libraries that you specified to the linker. If the symbol is defined in a library, the linker extracts the object file with the symbol definition from the library. This way the linker collects all definitions and references of all of the symbols.

Next, the linker combines sections with the same section name and attributes into single sections. The linker also substitutes (external) symbol references by (relocatable) numerical addresses where possible.
At the end of the linking phase, the linker either writes the results to a file (a single relocatable object file) or keeps the results in memory for further processing during the locating phase.

The resulting file of the linking phase is a single relocatable object file (.out). If this file contains unresolved references, you can link this file with other relocatable object files (.o) or libraries (.a) to resolve the remaining unresolved references.

With the linker command line option --link-only, you can tell the linker to only perform this linking phase and skip the locating phase. The linker complains if any unresolved references are left.

### 8.1.2. Phase 2: Locating

In the locating phase, the linker assigns absolute addresses to the object code, placing each section in a specific part of the target memory. The linker also replaces references to symbols by the actual address of those symbols. The resulting file is an absolute object file which you can actually load into a target memory. Optionally, when the resulting file should be loaded into a ROM device the linker creates a so-called copy table section which is used by the startup code to initialize the data and BSS sections.

#### Code modification

When the linker assigns absolute addresses to the object code, it needs to modify this code according to certain rules or relocation expressions to reflect the new addresses. These relocation expressions are stored in the relocatable object file. Consider the following snippet of x86 code that moves the contents of variable a to variable b via the eax register:

```
A1 3412 0000 mov a,%eax   (a defined at 0x1234, byte reversed)
A3 0000 0000 mov %eax,b   (b is imported so the instruction refers to 0x0000 since its location is unknown)
```

Now assume that the linker links this code so that the section in which a is located is relocated by 0x10000 bytes, and b turns out to be at 0x9A12. The linker modifies the code to be:

```
A1 3412 0100 mov a,%eax   (0x10000 added to the address)
A3 129A 0000 mov %eax,b   (0x9A12 patched in for b)
```

These adjustments affect instructions, but keep in mind that any pointers in the data part of a relocatable object file have to be modified as well.

#### Output formats

The linker can produce its output in different file formats. The default ELF/DWARF 2 format (.elf) contains an image of the executable code and data, and can contain additional debug information. The Intel-Hex format (.hex) and Motorola S-record format (.sre) only contain an image of the executable code and data. You can specify a format with the options --output (-o) and --chip-output (-c).

#### Controlling the linker

Via a so-called linker script file you can gain complete control over the linker. The script language is called the Linker Script Language (LSL). Using LSL you can define:

- The memory installed in the embedded target system:
To assign locations to code and data sections, the linker must know what memory devices are actually installed in the embedded target system. For each physical memory device the linker must know its start-address, its size, and whether the memory is read-write accessible (RAM) or read-only accessible (ROM).

• How and where code and data should be placed in the physical memory:

    Embedded systems can have complex memory systems. If for example on-chip and off-chip memory devices are available, the code and data located in internal memory is typically accessed faster and with dissipating less power. To improve the performance of an application, specific code and data sections should be located in on-chip memory. By writing your own LSL file, you gain full control over the locating process.

• The underlying hardware architecture of the target processor.

    To perform its task the linker must have a model of the underlying hardware architecture of the processor you are using. For example the linker must know how to translate an address used within the object file (a logical address) into an offset in a particular memory device (a physical address). In most linkers this model is hard coded in the executable and can not be modified. For the TASKING linker this hardware model is described in the linker script file. This solution is chosen to support configurable cores that are used in system-on-chip designs.

When you want to write your own linker script file, you can use the standard linker script files with architecture descriptions delivered with the product.

See also Section 8.7, Controlling the Linker with a Script.

8.2. Calling the Linker

In Eclipse you can set options specific for the linker. After you have built your project, the output files are available in a subdirectory of your project directory, depending on the active configuration you have set in the C/C++ Build » Settings page of the Project » Properties dialog.

Building a project under Eclipse

You have several ways of building your project:

• Incremental Build All ( ). Only the components affected by modified files in that particular project are built.

    To incrementally build all open projects, select Project » Build All.

• Build Individual Project ( ).

    To build individual projects incrementally, select Project » Build Project.

• Rebuild Project ( ). This builds every file in the project whether or not a file has been modified since the last build. A rebuild is a clean followed by a build.

    1. Select Project » Clean...
2. Enable the option **Start a build immediately** and click **OK**.

- **Build Automatically.** This performs a Build All whenever any project file is saved, such as your makefile.

  This way of building is not recommended, but to enable this feature select **Project » Build Automatically** and ensure there is a check mark beside the **Build Automatically** menu item.

**To access the linker options**

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

   *The Properties dialog appears.*

2. In the left pane, expand **C/C++ Build** and select **Settings**.

   *In the right pane the Settings appear.*

3. On the Tool Settings tab, select **Linker**.

4. Select the sub-entries and set the options in the various pages.

You can find a detailed description of all linker options in **Section 11.4, Linker Options**.

**Invocation syntax on the command line (Windows Command Prompt):**

```
ltc [ [option]... [file]... ]...
```

When you are linking multiple files, either relocatable object files (.o) or libraries (.a), it is important to specify the files in the right order. This is explained in **Section 8.3, Linking with Libraries**.

Example:

```
ltc -dtc1165.lsl test.o
```

This links and locates the file **test.o** and generates the file **test.elf**.

**8.3. Linking with Libraries**

There are two kinds of libraries: system libraries and user libraries.

**System library**

System libraries are stored in the directories:

```
<TriCore installation path>\lib\tc1                   (TriCore 1 libraries)
<TriCore installation path>\lib\tc1_mmu              (MMU variant)
<TriCore installation path>\lib\tc2                   (TriCore 2 libraries)
<TriCore installation path>\lib\tc2_mmu              (MMU variant)
<TriCore installation path>\lib\p                      (protected libraries)
```

The **p** directory contains subdirectories with the protected libraries for CPU functional problems.
An overview of the system libraries is given in the following table:

<table>
<thead>
<tr>
<th>Libraries</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>libc[s].a</td>
<td>C libraries</td>
</tr>
<tr>
<td>libc[s]_fpu.a</td>
<td>Optional letter: s = single precision floating-point (compiler option --no-double)</td>
</tr>
<tr>
<td></td>
<td>_fpu = with FPU instructions (compiler option --fpu-present)</td>
</tr>
<tr>
<td>libfp[t].a</td>
<td>Floating-point libraries</td>
</tr>
<tr>
<td>libfp[t]_fpu.a</td>
<td>Optional letter: t = trapping (control program option --fp-trap)</td>
</tr>
<tr>
<td></td>
<td>_fpu = with FPU instructions (compiler option --fpu-present)</td>
</tr>
<tr>
<td>librt.a</td>
<td>Run-time library</td>
</tr>
<tr>
<td>libpb.a</td>
<td>Profiling libraries, pb = block/function counter</td>
</tr>
<tr>
<td>libpc.a</td>
<td>pc = call graph</td>
</tr>
<tr>
<td>libpct.a</td>
<td>pct = call graph and timing</td>
</tr>
<tr>
<td>libpd.a</td>
<td>pd = dummy</td>
</tr>
<tr>
<td>libpt.a</td>
<td>pt = function timing</td>
</tr>
<tr>
<td>libcp[s][x].a</td>
<td>C++ libraries</td>
</tr>
<tr>
<td></td>
<td>Optional letter: s = single precision floating-point x = exception handling</td>
</tr>
<tr>
<td>libstl[s]x.a</td>
<td>STLport C++ libraries (exception handling variants only)</td>
</tr>
<tr>
<td></td>
<td>Optional letter: s = single precision floating-point</td>
</tr>
</tbody>
</table>

To link the default C (system) libraries

1. From the Project menu, select Properties

   The Properties dialog appears.

2. In the left pane, expand C/C++ Build and select Settings.

   In the right pane the Settings appear.


4. Enable the option Link default libraries.

5. Enable or disable the option Use trapped floating-point library.

When you want to link system libraries from the command line, you must specify this with the option --library (-l). For example, to specify the system library libc.a, type:

ltc --library=c test.o
User library

You can create your own libraries. Section 9.3, Archiver describes how you can use the archiver to create your own library with object modules.

To link user libraries

1. From the Project menu, select Properties

   The Properties dialog appears.

2. In the left pane, expand C/C++ Build and select Settings.

   In the right pane the Settings appear.


4. Add your libraries to the Libraries box.

When you want to link user libraries from the command line, you must specify their filenames on the command line:

   ltc start.o mylib.a

If the library resides in a sub-directory, specify that directory with the library name:

   ltc start.o mylibs\mylib.a

If you do not specify a directory, the linker searches the library in the current directory only.

Library order

The order in which libraries appear on the command line is important. By default the linker processes object files and libraries in the order in which they appear at the command line. Therefore, when you use a weak symbol construction, like printf, in an object file or your own library, you must position this object/library before the C library.

With the option --first-library-first you can tell the linker to scan the libraries from left to right, and extract symbols from the first library where the linker finds it. This can be useful when you want to use newer versions of a library routine:

   ltc --first-library-first a.a test.o b.a

If the file test.o calls a function which is both present in a.a and b.a, normally the function in b.a would be extracted. With this option the linker first tries to extract the symbol from the first library a.a.

Note that routines in b.a that call other routines that are present in both a.a and b.a are now also resolved from a.a.
8.3.1. How the Linker Searches Libraries

System libraries

You can specify the location of system libraries in several ways. The linker searches the specified locations in the following order:

1. The linker first looks in the **Library search path** that are specified in the **Linker » Libraries** page in the **C/C++ Build » Settings » Tool Settings** tab of the Project Properties dialog (equivalent to the `-L` command line option). If you specify the `-L` option without a pathname, the linker stops searching after this step.

2. When the linker did not find the library (because it is not in the specified library directory or because no directory is specified), it looks in the path(s) specified in the environment variables `LIBTC1V1_2` / `LIBTC1V1_3` / `LIBTC2`.

3. When the linker did not find the library, it tries the default lib directory relative to the installation directory (or a processor specific sub-directory).

User library

If you use your own library, the linker searches the library in the current directory only.

8.3.2. How the Linker Extracts Objects from Libraries

A library built with the TASKING archiver `artc` always contains an index part at the beginning of the library. The linker scans this index while searching for unresolved externals. However, to keep the index as small as possible, only the defined symbols of the library members are recorded in this area.

When the linker finds a symbol that matches an unresolved external, the corresponding object file is extracted from the library and is processed. After processing the object file, the remaining library index is searched. If after a complete search of the library unresolved externals are introduced, the library index will be scanned again. After all files and libraries are processed, and there are still unresolved externals and you did not specify the `linker option --no-rescan`, all libraries are rescanned again. This way you do not have to worry about the library order on the command line and the order of the object files in the libraries. However, this rescanning does not work for 'weak symbols'. If you use a weak symbol construction, like `printf`, in an object file or your own library, you must position this object/library before the C library.

The `option --verbose (-v)` shows how libraries have been searched and which objects have been extracted.

Resolving symbols

If you are linking from libraries, only the objects that contain symbols to which you refer, are extracted from the library. This implies that if you invoke the linker like:

```
ltc mylib.a
```

nothing is linked and no output file will be produced, because there are no unresolved symbols when the linker searches through `mylib.a`.

It is possible to force a symbol as external (unresolved symbol) with the `option --extern (-e)`:
In this case the linker searches for the symbol `main` in the library and (if found) extracts the object that contains `main`.

If this module contains new unresolved symbols, the linker looks again in `mylib.a`. This process repeats until no new unresolved symbols are found.

### 8.4. Incremental Linking

With the TASKING linker it is possible to link incrementally. Incremental linking means that you link some, but not all `.o` modules to a relocatable object file `.out`. In this case the linker does not perform the locating phase. With the second invocation, you specify both new `.o` files as the `.out` file you had created with the first invocation.

Incremental linking is only possible on the command line.

```
ltc --incremental test1.o -otest.out
ltc test2.o test.out
```

This links the file `test1.o` and generates the file `test.out`. This file is used again and linked together with `test2.o` to create the file `test.elf` (the default name if no output filename is given in the default ELF/DWARF 2 format).

With incremental linking it is normal to have unresolved references in the output file until all `.o` files are linked and the final `.out` or `.elf` file has been reached. The option `--incremental (-r)` for incremental linking also suppresses warnings and errors because of unresolved symbols.

### 8.5. Importing Binary Files

With the TASKING linker it is possible to add a binary file to your absolute output file. In an embedded application you usually do not have a file system where you can get your data from. With the linker option `--import-object` you can add raw data to your application. This makes it possible for example to display images on a device or play audio. The linker puts the raw data from the binary file in a section. The section is aligned on a 4-byte boundary. The section name is derived from the filename, in which dots are replaced by an underscore. So, when importing a file called `my.mp3`, a section with the name `my_mp3` is created.

In your application you can refer to the created section by using linker labels.

For example:

```
#include <stdio.h>
__far extern char   __lc_ub_my_mp3; /* linker labels */
__far extern char   __lc_ue_my_mp3;
char*   mp3 = &_lc_ub_my_mp3;

void main(void)
{
    int size = &_lc_ue_my_mp3 - &_lc_ub_my_mp3;
    int i;
```
Because the compiler does not know in which space the linker will locate the imported binary, you have to make sure the symbols refer to the same space in which the linker will place the imported binary. You do this by using the memory qualifier __far, otherwise the linker cannot bind your linker symbols.

Also note that if you want to use the export functionality of Eclipse, the binary file has to be part of your project.

8.6. Linker Optimizations

During the linking and locating phase, the linker looks for opportunities to optimize the object code. Both code size and execution speed can be optimized.

To enable or disable optimizations

1. From the Project menu, select Properties
   
   The Properties dialog appears.

2. In the left pane, expand C/C++ Build and select Settings.
   
   In the right pane the Settings appear.

3. On the Tool Settings tab, select Linker » Optimization.

4. Enable one or more optimizations.

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

Delete unreferenced sections (option -Oc/-OC)

This optimization removes unused sections from the resulting object file.

First fit decreasing (option -Ol/-OL)

When the physical memory is fragmented or when address spaces are nested it may be possible that a given application cannot be located although the size of the available physical memory is larger than the sum of the section sizes. Enable the first-fit-decreasing optimization when this occurs and re-link your application.

The linker’s default behavior is to place sections in the order that is specified in the LSL file (that is, working from low to high memory addresses or vice versa). This also applies to sections within an unrestricted group. If a memory range is partially filled and a section must be located that is larger than the remainder of this range, then the section and all subsequent sections are placed in a next memory range. As a result of this gaps occur at the end of a memory range.
When the first-fit-decreasing optimization is enabled the linker will first place the largest sections in the smallest memory ranges that can contain the section. Small sections are located last and can likely fit in the remaining gaps.

**Compress copy table (option -Ot/-OT)**

The startup code initializes the application's data areas. The information about which memory addresses should be zeroed and which memory ranges should be copied from ROM to RAM is stored in the copy table.

When this optimization is enabled the linker will try to locate sections in such a way that the copy table is as small as possible thereby reducing the application's ROM image.

**Delete duplicate code (option -Ox/-OX)**

**Delete duplicate constant data (option -Oy/-OY)**

These two optimizations remove code and constant data that is defined more than once, from the resulting object file.

### 8.7. Controlling the Linker with a Script

With the options on the command line you can control the linker's behavior to a certain degree. From Eclipse it is also possible to determine where your sections will be located, how much memory is available, which sorts of memory are available, and so on. Eclipse passes these locating directions to the linker via a script file. If you want even more control over the locating process you can supply your own script.

The language for the script is called the Linker Script Language, or shortly LSL. You can specify the script file to the linker, which reads it and locates your application exactly as defined in the script. If you do not specify your own script file, the linker always reads a standard script file which is supplied with the toolset.

#### 8.7.1. Purpose of the Linker Script Language

The Linker Script Language (LSL) serves three purposes:

1. It provides the linker with a definition of the target's core architecture. This definition is supplied with the toolset.

2. It provides the linker with a specification of the memory attached to the target processor.

3. It provides the linker with information on how your application should be located in memory. This gives you, for example, the possibility to create overlaying sections.

The linker accepts multiple LSL files. You can use the specifications of the core architectures that Altium has supplied in the include.lsl directory. Do not change these files.

If you use a different memory layout than described in the LSL file supplied for the target core, you must specify this in a separate LSL file and pass both the LSL file that describes the core architecture and your LSL file that contains the memory specification to the linker. Next you may want to specify how sections should be located and overlaid. You can do this in the same file or in another LSL file.
LSL has its own syntax. In addition, you can use the standard C preprocessor keywords, such as `#include` and `#define`, because the linker sends the script file first to the C preprocessor before it starts interpreting the script.

The complete LSL syntax is described in Linker Script Language.

### 8.7.2. Eclipse and LSL

In Eclipse you can specify the size of the stack and heap; the physical memory attached to the processor; identify that particular address ranges are reserved; and specify which sections are located where in memory. Eclipse translates your input into an LSL file that is stored in the project directory under the name `project_name.lsl` and passes this file to the linker. If you want to learn more about LSL you can inspect the generated file `project_name.lsl`.

To add a generated Linker Script File to your project

1. From the **File** menu, select **File » New » Other... » TASKING C/C++ » TASKING VX-toolset for TriCore C/C++ Project**.

   *The New C/C++ Project wizard appears.*

2. Fill in the project settings in each dialog and click **Next >** until the following dialog appears.

   ![New C/C++ Project](image)

3. Enable the option **Add Linker script file to the project** and click **Finish**.

   *Eclipse creates your project and the file "project_name.lsl" in the project directory.*

If you do not add the linker script file here, you can always add it later with **File » New » Other... » TASKING C/C++ » Linker Script File (LSL)**
To change the Linker Script File in Eclipse

1. Double-click on the file `project_name.lsl`.

   The project LSL file opens in the editor area with several tabs.

2. You can edit the LSL file directly in the `project_name.lsl` tab or make changes to the other tabs (Memory, Sections, ...).

   The LSL file is updated automatically according to the changes you make in the tabs. A * appears in front of the name of the LSL file to indicate that the file has changes.

3. Click or select File » Save to save the changes.

You can quickly navigate through the LSL file by using the Outline view (Window » Show View » Outline).

8.7.3. Structure of a Linker Script File

A script file consists of several definitions. The definitions can appear in any order.

The architecture definition (required)

In essence an architecture definition describes how the linker should convert logical addresses into physical addresses for a given type of core. If the core supports multiple address spaces, then for each space the linker must know how to perform this conversion. In this context a physical address is an offset on a given internal or external bus. Additionally the architecture definition contains information about items such as the (hardware) stack and the interrupt vector table.
This specification is normally written by Altium. Altium supplies LSL files in the `include.lsl` directory. The file `tc_arch.lsl` defines the base architecture for all cores and includes an interrupt vector table (`inttab.lsl`) and a trap vector table (`traptab.lsl`). The files `tc1v1_2.lsl`, `tc1v1_3.lsl` and `tc2.lsl` extend the base architecture for each TriCore core.

The architecture definition of the LSL file should not be changed by you unless you also modify the core's hardware architecture. If the LSL file describes a multi-core system an architecture definition must be available for each different type of core.

The linker uses the architecture name in the LSL file to identify the target. For example, the default library search path can be different for each core architecture.

The derivative definition

The derivative definition describes the configuration of the internal (on-chip) bus and memory system. Basically it tells the linker how to convert offsets on the buses specified in the architecture definition into offsets in internal memory. Microcontrollers and DSPs often have internal memory and I/O sub-systems apart from one or more cores. The design of such a chip is called a derivative.

When you want to use multiple cores of the same type, you must instantiate the cores in a derivative definition, since the linker automatically instantiates only a single core for an unused architecture.

Altium supplies LSL files for each derivative (`derivative.lsl`), along with "SFR files", which provide easy access to registers in I/O sub-systems from C and assembly programs. When you build an ASIC or use a derivative that is not (yet) supported by the TASKING tools, you may have to write a derivative definition.

The processor definition

The processor definition describes an instance of a derivative. A processor definition is only needed in a multi-processor embedded system. It allows you to define multiple processors of the same type.

If for a derivative 'A' no processor is defined in the LSL file, the linker automatically creates a processor named 'A' of derivative 'A'. This is why for single-processor applications it is enough to specify the derivative in the LSL file.

The memory and bus definitions (optional)

Memory and bus definitions are used within the context of a derivative definition to specify internal memory and on-chip buses. In the context of a board specification the memory and bus definitions are used to define external (off-chip) memory and buses. Given the above definitions the linker can convert a logical address into an offset into an on-chip or off-chip memory device.

The board specification

The processor definition and memory and bus definitions together form a board specification. LSL provides language constructs to easily describe single-core and heterogeneous or homogeneous multi-core systems. The board specification describes all characteristics of your target board's system buses, memory devices, I/O sub-systems, and cores that are of interest to the linker. Based on the information provided in the board specification the linker can for each core:

- convert a logical address to an offset within a memory device
locate sections in physical memory

• maintain an overall view of the used and free physical memory within the whole system while locating

The section layout definition (optional)

The optional section layout definition enables you to exactly control where input sections are located. Features are provided such as: the ability to place sections at a given address, to place sections in a given order, and to overlay code and/or data sections.

Example: Skeleton of a Linker Script File

A linker script file that defines a derivative "X" based on the TC1V1.3 architecture, its external memory and how sections are located in memory, may have the following skeleton:

```
architecture TC1V1.3
{
   // Specification of the TC1V1.3 core architecture.
   // Written by Altium.
}

derivative X  // derivative name is arbitrary
{
   // Specification of the derivative.
   // Written by Altium.
   core tc       // always specify the core
   {
      architecture = TC1V1.3;
   }

   bus fpi_bus   // internal bus
   {
      // maps to bus "fpi_bus" in "tc" core
   }

   // internal memory
}

processor spe  // processor name is arbitrary
{
   derivative = X;

   // You can omit this part, except if you use a
   // multi-core system.
}

memory ext_name
{
   // external memory definition
}
```
section_layout spe:tc:linear  // section layout
{
    // section placement statements
    // sections are located in address space 'linear'
    // of core 'tc' of processor 'spe'
}

Overview of LSL files delivered by Altium

Altium supplies the following LSL files in the directory include.lsl.

<table>
<thead>
<tr>
<th>LSL file</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tc_arch.lsl</td>
<td>Defines the base architecture (TC) for all cores. It includes the files</td>
</tr>
<tr>
<td></td>
<td>inttab.lsl and traptab.lsl.</td>
</tr>
<tr>
<td>inttab.lsl</td>
<td>Defines the interrupt vector table. It is included in the file tc_arch.lsl.</td>
</tr>
<tr>
<td>traptab.lsl</td>
<td>Defines the trap vector table. It is included in the file tc_arch.lsl.</td>
</tr>
<tr>
<td>tc1v1_2.lsl</td>
<td>Extends the base architecture for cores TC1V1.2, TC1V1.3 and TC2. It includes</td>
</tr>
<tr>
<td>tc1v1_3.lsl</td>
<td>the file tc_arch.lsl.</td>
</tr>
<tr>
<td>tc2.lsl</td>
<td>Defines the derivative and defines a single processor. Contains a memory</td>
</tr>
<tr>
<td></td>
<td>definition and section layout. It includes one of the files tcversion.lsl or</td>
</tr>
<tr>
<td></td>
<td>pxbversion.lsl. The selection of the derivative is based on your CPU</td>
</tr>
<tr>
<td></td>
<td>selection (control program option --cpu).</td>
</tr>
<tr>
<td>derivative.lsl</td>
<td>Defines a user defined derivative for cores TC1V1.2, TC1V1.3 or TC2 and</td>
</tr>
<tr>
<td>userdef112.lsl</td>
<td>defines a single processor.</td>
</tr>
<tr>
<td>userdef113.lsl</td>
<td></td>
</tr>
<tr>
<td>userdef2.lsl</td>
<td></td>
</tr>
<tr>
<td>template.lsl</td>
<td>This file is used by Eclipse as a template for the project LSL file. It</td>
</tr>
<tr>
<td></td>
<td>includes the file derivative.lsl based on your CPU selection and contains a</td>
</tr>
<tr>
<td></td>
<td>default specification of the external memory attached to the target processor.</td>
</tr>
<tr>
<td>default.lsl</td>
<td>Contains a default memory definition and section layout based on the tc1920b</td>
</tr>
<tr>
<td></td>
<td>derivative. This file is used on a command line invocation of the tools, when</td>
</tr>
<tr>
<td></td>
<td>no CPU is selected (no option --cpu).</td>
</tr>
<tr>
<td>extmem.lsl</td>
<td>Template file with a specification of the external memory attached to the</td>
</tr>
<tr>
<td></td>
<td>target processor.</td>
</tr>
</tbody>
</table>

When you select to add a linker script file when you create a project in Eclipse, Eclipse makes a copy of the file template.lsl and names it “project_name.lsl”. On the command line, the linker uses the file default.lsl, unless you specify another file with the linker option --lsl-file (-d).

8.7.4. The Architecture Definition

Although you will probably not need to program the architecture definition (unless you are building your own processor core) it helps to understand the Linker Script Language and how the definitions are interrelated.
Within an *architecture definition* the characteristics of a target processor core that are important for the linking process are defined. These include:

- space definitions: the logical address spaces and their properties
- bus definitions: the I/O buses of the core architecture
- mappings: the address translations between logical address spaces, the connections between logical address spaces and buses and the address translations between buses

**Address spaces**

A logical address space is a memory range for which the core has a separate way to encode an address into instructions. Most microcontrollers and DSPs support multiple address spaces. For example, the Tricore's 32-bit linear address space encloses 16 24-bit sub-spaces and 16 14-bit sub-spaces. See also the section "Memory Model" in the *Tricore Architecture Manual*. Normally, the size of an address space is $2^N$, with $N$ the number of bits used to encode the addresses.

The relation of an address space with another address space can be one of the following:

- one space is a subset of the other. These are often used for "small" absolute, and relative addressing.
- the addresses in the two address spaces represent different locations so they do not overlap. This means the core must have separate sets of address lines for the address spaces. For example, in Harvard architectures we can identify at least a code and a data memory space.

Address spaces (even nested) can have different minimal addressable units (MAU), alignment restrictions, and page sizes. All address spaces have a number that identifies the logical space (id). The following table lists the different address spaces for the architecture TC as defined in `tc_arch.lsl`.

<table>
<thead>
<tr>
<th>Space</th>
<th>Id</th>
<th>MAU</th>
<th>Description</th>
<th>ELF sections</th>
</tr>
</thead>
<tbody>
<tr>
<td>linear</td>
<td>1</td>
<td>1</td>
<td>Linear address space.</td>
<td>.text, .bss, .data, .rodata, table, istack, ustack</td>
</tr>
<tr>
<td>abs24</td>
<td>2</td>
<td>8</td>
<td>Absolute 24-bit addressable space</td>
<td></td>
</tr>
<tr>
<td>abs18</td>
<td>3</td>
<td>8</td>
<td>Absolute 18-bit addressable space</td>
<td>.zdata, .zbss</td>
</tr>
<tr>
<td>csa</td>
<td>4</td>
<td>8</td>
<td>Context Save Area</td>
<td>csa.*</td>
</tr>
<tr>
<td>pcp_code</td>
<td>8</td>
<td>16</td>
<td>PCP code</td>
<td>.pcptext</td>
</tr>
<tr>
<td>pcp_data</td>
<td>9</td>
<td>32</td>
<td>PCP data</td>
<td>.pcpdata</td>
</tr>
</tbody>
</table>

**The TriCore architecture in LSL notation**

The best way to program the architecture definition, is to start with a drawing. The following figure shows a part of the TriCore architecture:
The figure shows three address spaces called `linear`, `abs18` and `pcp_code`. The address space `abs18` is a subset of the address space `linear`. All address spaces have attributes like a number that identifies the logical space (id), a MAU and an alignment. In LSL notation the definition of these address spaces looks as follows:

```plaintext
using the linker

space linear
{
    id = 1;
    mau = 8;

    map (src_offset=0x00000000, dest_offset=0x00000000, size=4G, dest=bus:fpi_bus);
}

space abs18
{
    id = 3;
    mau = 8;

    map (src_offset=0x00000000, dest_offset=0x00000000, size=16k, dest=space:linear);
    map (src_offset=0x10000000, dest_offset=0x10000000, size=16k, dest=space:linear);
    map (src_offset=0x20000000, dest_offset=0x20000000, size=16k, dest=space:linear);
    //...
}

space pcp_code
{
    id = 8;
    mau = 16;

    map (src_offset=0x00000000, dest_offset=0, size=0x04000000, dest=bus:pcp_code_bus);
}
```
The keyword map corresponds with the arrows in the drawing. You can map:

- address space => address space
- address space => bus
- memory => bus (not shown in the drawing)
- bus => bus (not shown in the drawing)

Next the two internal buses, named fpi_bus and pcp_code_bus must be defined in LSL:

```c
bus fpi_bus
{
    mau = 8;
    width = 32;  // there are 32 data lines on the bus
}

bus pcp_code_bus
{
    mau = 8;
    width = 8;
}
```

This completes the LSL code in the architecture definition. Note that all code above goes into the architecture definition, thus between:

```c
architecture TC1V1.3
{
    // All code above goes here.
}
```

### 8.7.5. The Derivative Definition

Although you will probably not need to program the derivative definition (unless you are using multiple cores) it helps to understand the Linker Script Language and how the definitions are interrelated.

A derivative is the design of a processor, as implemented on a chip (or FPGA). It comprises one or more cores and on-chip memory. The derivative definition includes:

- core definition: an instance of a core architecture
- bus definition: the I/O buses of the core architecture
- memory definitions: internal (or on-chip) memory (in Eclipse this is called 'System memory')

### Core

Each derivative must have at least one core and each core must have a specification of its core architecture. This core architecture must be defined somewhere in the LSL file(s).
Bus

Each derivative can contain a bus definition for connecting external memory. In this example, the bus \texttt{fpi\_bus} maps to the bus \texttt{fpi\_bus} defined in the architecture definition of core tc:

```cpp
bus fpi_bus
{
    mau = 8;
    width = 32;
    map (dest=bus:tc:fpi_bus, dest_offset=0, size=4G);
}
```

Memory

External memory is usually described in a separate memory definition, but you can specify on-chip memory for a derivative. For example:

```cpp
memory pcode
{
    mau = 8;
    size = 16k;
    type = ram;
    map (dest=bus:tc:fpi_bus, dest_offset=0xF0020000, size=16k);
    map (dest=bus:tc:pcp_code_bus, size=16k);
}
```

According to the drawing, the TriCore contains internal memory called \texttt{pcode} with a size 0x04000 (16 kB). This is physical memory which is mapped to the internal bus \texttt{pcp\_code\_bus} and to the \texttt{fpi\_bus}, so both the tc unit and the PCP can access the memory:
This completes the LSL code in the derivative definition. Note that all code above goes into the derivative definition, thus between:

```lsl
derivative X    // name of derivative
{
    // All code above goes here
}
```

### 8.7.6. The Processor Definition

The processor definition is only needed when you write an LSL file for a multi-processor embedded system. The processor definition explicitly instantiates a derivative, allowing multiple processors of the same type.

```lsl
processor name
{
    derivative = derivative_name;
}
```

If no processor definition is available that instantiates a derivative, a processor is created with the same name as the derivative.

Altium defines a “single processor environment” (spe) in each derivative.lsl file. For example:

```lsl
processor spe
{
    derivative = tc1920b;
}
```

### 8.7.7. The Memory Definition

Once the core architecture is defined in LSL, you may want to extend the processor with external (or off-chip) memory. You need to specify the location and size of the physical external memory devices in the target system.

The principle is the same as defining the core’s architecture but now you need to fill the memory definition:

```lsl
memory name
{
    // memory definitions
}
```
Suppose your embedded system has 16 kB of external ROM, named `code_rom` and 2 kB of external NVRAM, named `my_nvsram`. Both memories are connected to the bus `fpi_bus`. In LSL this looks like:

```plaintext
memory code_rom
{
  mau = 8;
  size = 16k;
  type = rom;
  map( dest=bus:spe:fpi_bus, dest_offset=0xa0000000, size=16k );
}

memory my_nvsram
{
  mau = 8;
  size = 2k;
  type = ram;
  map( dest=bus:spe:fpi_bus, dest_offset=0xc0000000, size=2k );
}
```

If you use a different memory layout than described in the LSL file supplied for the target core, you can specify this in Eclipse or you can specify this in a separate LSL file and pass both the LSL file that describes the core architecture and your LSL file that contains the memory specification to the linker.

**To add memory using Eclipse**

1. Double-click on the file `project.lsl`.
   
   *The project LSL file opens in the editor area with several tabs.*

2. Open the **Memory** tab and click on the **Add** button.
   
   *A new line is added to the list of User memory.*

3. Click in each field to change the type, name (for example `my_nvsram`) and sizes.

   *The LSL file is updated automatically according to the changes you make.*
4. Click or select File » Save to save the changes.

8.7.8. The Section Layout Definition: Locating Sections

Once you have defined the internal core architecture and optional memory, you can actually define where your application must be located in the physical memory.

During compilation, the compiler divides the application into sections. Sections have a name, an indication (section type) in which address space it should be located and attributes like writable or read-only.

In the section layout definition you can exactly define how input sections are placed in address spaces, relative to each other, and what their absolute run-time and load-time addresses will be.

Example: section propagation through the toolset

To illustrate section placement, the following example of a C program (bat.c) is used. The program saves the number of times it has been executed in battery back-upped memory, and prints the number.

```c
#define BATTERY_BACKUP_TAG 0xa5f0
#include <stdio.h>

int uninitialized_data;
int initialized_data = 1;
#pragma section all "non_volatile"
#pragma noclear
int battery_backup_tag;
int battery_backup_invok;
#pragma clear
#pragma section all

void main (void)
{
    if (battery_backup_tag != BATTERY_BACKUP_TAG )
    {
        // battery back-upped memory area contains invalid data
        // initialize the memory
        battery_backup_tag = BATTERY_BACKUP_TAG;
        battery_backup_invok = 0;
    }
    printf( "This application has been invoked %d times\n",
            battery_backup_invok++);
}
```

The compiler assigns names and attributes to sections. With the `#pragma section` the compiler’s default section naming convention is overruled and a section with the name `non_volatile` is defined. In this section the battery back-upped data is stored.

By default the compiler creates a section with the name "./zbss.bat" of section type `data` to store uninitialized data objects. The section prefix ".zbss" tells the linker to locate the section in address space abs18 and that the section content should be filled with zeros at startup.
As a result of the `#pragma section all "non_volatile"`, the data objects between the pragma pair are placed in a section with the name ".zbss.non_volatile". Note that ".zbss" sections are cleared at startup. However, battery back-upped sections should not be cleared and therefore we used `#pragma noclear`.

**Section placement**

The number of invocations of the example program should be saved in non-volatile (battery back-upped) memory. This is the memory `my_nvsram` from the example in *Section 8.7.7, The Memory Definition*.

To control the locating of sections, you need to write one or more section definitions in the LSL file. At least one for each address space where you want to change the default behavior of the linker. In our example, we need to locate sections in the address space `abs18`:

```c
section_layout ::abs18
{
   // Section placement statements
}
```

To locate sections, you must create a group in which you select sections from your program. For the battery back-up example, we need to define one group, which contains the section `.zbss.non_volatile`. All other sections are located using the defaults specified in the architecture definition. Section `.zbss.non_volatile` should be placed in non-volatile ram. To achieve this, the run address refers to our non-volatile memory called `my_nvsram`.

```c
group ( ordered, run_addr = mem:my_nvsram )
{
   select ".zbss.non_volatile";
}
```

**Section placement from Eclipse**

1. Double-click on the file `project.lsl`.

   *The project LSL file opens in the editor area with several tabs.*

2. Open the **Sections** tab and click on the **Add** button.

   *A new line is added to the list of sections.*

3. In the **Name** field enter `.zbss.non_volatile`.

4. In the **Run address** field enter `mem:my_nvsram`.

5. In the **Space** field select `abs18`.

6. In the **Order** field select `ordered`.

   *The LSL file is updated automatically according to the changes you make.*

7. Click 🗑️ or select **File » Save** to save the changes.
This completes the LSL file for the sample architecture and sample program. You can now invoke the linker with this file and the sample program to obtain an application that works for this architecture.

For a complete description of the Linker Script Language, refer to Chapter 14, Linker Script Language (LSL).

8.8. Linker Labels

The linker creates labels that you can use to refer to from within the application software. Some of these labels are real labels at the beginning or the end of a section. Other labels have a second function, these labels are used to address generated data in the locating phase. The data is only generated if the label is used.

Linker labels are labels starting with _lc_. The linker assigns addresses to the following labels when they are referenced:

<table>
<thead>
<tr>
<th>Label</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_lc_ub_name</td>
<td>Begin of section name. Also used to mark the begin of the stack or heap or copy table.</td>
</tr>
<tr>
<td>_lc_b_name</td>
<td>End of section name. Also used to mark the end of the stack or heap.</td>
</tr>
<tr>
<td>_lc_ue_name</td>
<td>Start address of an overlay section in ROM.</td>
</tr>
<tr>
<td>_lc_e_name</td>
<td>End address of an overlay section in ROM.</td>
</tr>
<tr>
<td>_lc_cb_name</td>
<td>Start address of an overlay section in ROM.</td>
</tr>
<tr>
<td>_lc_ce_name</td>
<td>End address of an overlay section in ROM.</td>
</tr>
<tr>
<td>_lc_gb_name</td>
<td>Begin of group name. This label appears in the output file even if no reference to the label exists in the input file.</td>
</tr>
<tr>
<td>_lc_ge_name</td>
<td>End of group name. This label appears in the output file even if no reference to the label exists in the input file.</td>
</tr>
<tr>
<td>_lc_s_name</td>
<td>Variable name is mapped through memory in shared memory situations.</td>
</tr>
</tbody>
</table>

The linker only allocates space for the stack and/or heap when a reference to either of the section labels exists in one of the input object files.

If you want to use linker labels in your C source for sections that have a dot (.) in the name, you have to replace all dots by underscores.

Additionally, the linker script file defines the following symbols:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_lc_cp</td>
<td>Start of copy table. Same as _lc_ub_table. The copy table gives the source and destination addresses of sections to be copied. This table will be generated by the linker only if this label is used.</td>
</tr>
<tr>
<td>_lc_bh</td>
<td>Begin of heap. Same as _lc_ub_heap.</td>
</tr>
<tr>
<td>_lc_eh</td>
<td>End of heap. Same as _lc_ue_heap.</td>
</tr>
</tbody>
</table>
Example: refer to a label with section name with dots from C

Suppose the C source file foo.c contains the following:

```c
int myfunc(int a)
{
    /* some source lines */
    return 1;
}
```

This results in a section with the name .text.foo.myfunc.

In the following source file main.c all dots of the section name are replaced by underscores:

```c
#include <stdio.h>
extern char _lc_ub__text_foo_myfunc[];

void main(void)
{
    printf("The function myfunc is located at %x\n",
           _lc_ub__text_foo_myfunc);
}
```

Example: refer to a PCP variable from TriCore C source

When memory is shared between two or more cores, for instance TriCore and PCP, the addresses of variables (or functions) on that memory may be different for the cores. For the TriCore the variable will be defined and you can access it in the usual way. For the PCP, when you would use the variable directly in your TriCore source, this would use an incorrect address (PCP address). The linker can map the address of the variable from one space to another, if you prefix the variable name with _lc_s_.

When a symbol foo is defined in a PCP assembly source file, by default it gets the symbol name foo. To use this symbol from a TriCore C source file, write:

```c
extern long _lc_s_foo;

void main(int argc, char **argv)
{
    _lc_s_foo = 7;
}
```

Example: refer to the stack

Suppose in an LSL file a stack section is defined with the name "ustack" (with the keyword stack). You can refer to the begin and end of the stack from your C source as follows:

```c
#include <stdio.h>
extern char _lc_ub_ustack[];
extern char _lc_ue_ustack[];

void main()
{
```
printf( "Size of stack is %d\n", 
    _lc_ue_ustack - _lc_ub_ustack );
/* stack grows from low to high */
}

From assembly you can refer to the end of the stack with:

.extern _lc_ue_ustack ; end of user stack

8.9. Generating a Map File

The map file is an additional output file that contains information about the location of sections and symbols. You can customize the type of information that should be included in the map file.

To generate a map file

1. From the Project menu, select Properties

   The Properties dialog appears.

2. In the left pane, expand C/C++ Build and select Settings.

   In the right pane the Settings appear.

3. On the Tool Settings tab, select Linker » Map File.

4. Enable the option Generate XML map file format (.mapxml) for map file viewer.

5. (Optional) Enable the option Generate map file (.map).

6. (Optional) Enable the options to include that information in the map file.

Example on the command line (Windows Command Prompt)

The following command generates the map file test.map:

ltc --map-file test.o

With this command the map file test.map is created.

See Section 13.2, Linker Map File Format, for an explanation of the format of the map file.

8.10. Linker Error Messages

The linker reports the following types of error messages in the Problems view of Eclipse.

F (Fatal errors)

After a fatal error the linker immediately aborts the link/locate process.
E (Errors)

Errors are reported, but the linker continues linking and locating. No output files are produced unless you have set the linker option --keep-output-files.

W (Warnings)

Warning messages do not result into an erroneous output file. They are meant to draw your attention to assumptions of the linker for a situation which may not be correct. You can control warnings in the C/C++ Build » Settings » Tool Settings » Linker » Diagnostics page of the Project » Properties menu (linker option --no-warnings).

I (Information)

Verbose information messages do not indicate an error but tell something about a process or the state of the linker. To see verbose information, use the linker option --verbose.

S (System errors)

System errors occur when internal consistency checks fail and should never occur. When you still receive the system error message

S6##: message

please report the error number and as many details as possible about the context in which the error occurred.

Display detailed information on diagnostics

1. From the Window menu, select Show View » Other » General » Problems.

   The Problems view is added to the current perspective.

2. In the Problems view right-click on a message.

   A popup menu appears.

3. Select Detailed Diagnostics Info.

   A dialog box appears with additional information.

On the command line you can use the linker option --diag to see an explanation of a diagnostic message:

ltc --diag= [format:][all | number,...]
Chapter 9. Using the Utilities

The TASKING VX-toolset for TriCore comes with a number of utilities:

- **cctc**: A control program. The control program invokes all tools in the toolset and lets you quickly generate an absolute object file from C and/or assembly source input files. Eclipse uses the control program to call the compiler, assembler and linker.

- **mktc**: A utility program to maintain, update, and reconstruct groups of programs. The make utility looks whether files are out of date, rebuilds them and determines which other files as a consequence also need to be rebuilt.

- **artc**: An archiver. With this utility you create and maintain library files with relocatable object modules (.o) generated by the assembler.

- **prelktc**: The C++ prelinker. This utility is called via the control program and manages automatic instantiation of template entities.

9.1. Control Program

The control program is a tool that invokes all tools in the toolset for you. It provides a quick and easy way to generate the final absolute object file out of your C/C++ sources without the need to invoke the compiler, assembler and linker manually.

Eclipse uses the control program to call the C++ compiler, C compiler, assembler and linker, but you can call the control program from the command line. The invocation syntax is:

```
cctc [ [option]... [file]... ]...
```

Recognized input files

- Files with a `.cc`, `.cxx` or `.cpp` suffix are interpreted as C++ source programs and are passed to the C++ compiler.
- Arguments with a `.c` suffix are interpreted as C source programs and are passed to the compiler.
- Files with a `.asm` suffix are interpreted as hand-written assembly source files which have to be passed to the assembler.
- Files with a `.src` suffix are interpreted as compiled assembly source files. They are directly passed to the assembler.
- Files with a `.a` suffix are interpreted as library files and are passed to the linker.
- Files with a `.o` suffix are interpreted as object files and are passed to the linker.
- Files with a `.out` suffix are interpreted as linked object files and are passed to the locating phase of the linker. The linker accepts only one `.out` file in the invocation.
- Files with a `.lsl` suffix are interpreted as linker script files and are passed to the linker.
Options

The control program accepts several command line options. If you specify an unknown option to the control program, the control program looks if it is an option for a specific tool. If so, it passes the option directly to the tool. However, it is recommended to use the control program options --pass-* (-Wcp, -Wc, -Wa, -Wl, -Wpi) to pass arguments directly to tools.

For a complete list and description of all control program options, see Section 11.5, Control Program Options.

Example with verbose output

cctc --verbose test.c

The control program calls all tools in the toolset and generates the absolute object file test.elf. With option --verbose (-v) you can see how the control program calls the tools:

+ "path\ctc" -o cc3248a.src test.c
+ "path\astc" -o cc3248b.o cc3248a.src
+ "path\ltc" -o test.elf -dextmem.lsl -ddefault.lsl --map-file cc3248b.o "-Lpath\lib\tcl1" -lc -lfp -lrt

The control program produces unique filenames for intermediate steps in the compilation process (such as cc3248a.src and cc3248b.o in the example above) which are removed afterwards, unless you specify command line option --keep-temporary-files (-t).

Example with argument passing to a tool

cctc --pass-compiler=-Oc test.c

The option -Oc is directly passed to the compiler.

9.2. Make Utility

If you are working with large quantities of files, or if you need to build several targets, it is rather time-consuming to call the individual tools to compile, assemble, link and locate all your files.

You save already a lot of typing if you use the control program and define an options file. You can even create a batch file or script that invokes the control program for each target you want to create. But with these methods all files are completely compiled, assembled and linked to obtain the target file, even if you changed just one C source. This may demand a lot of (CPU) time on your host.

The make utility mktc is a tool to maintain, update, and reconstruct groups of programs. The make utility looks which files are out-of-date and only recreates these files to obtain the updated target.

Make process

In order to build a target, the make utility needs the following input:

- the target it should build, specified as argument on the command line
the rules to build the target, stored in a file usually called makefile

In addition, the make utility also reads the file mktc.mk which contains predefined rules and macros. See Section 9.2.2, Writing a Makefile.

The makefile contains the relationships among your files (called dependencies) and the commands that are necessary to create each of the files (called rules). Typically, the absolute object file (.elf) is updated when one of its dependencies has changed. The absolute file depends on .o files and libraries that must be linked together. The .o files on their turn depend on .src files that must be assembled and finally, .src files depend on the C source files (.c) that must be compiled. In the makefile this looks like:

```makefile
# dependency
# rule

test.src : test.c
    ctc test.c

test.o : test.src
    astc test.src

test.elf : test.o
    ltc test.o -o test.elf --map-file -lc -lfp -lrt
```

You can use any command that is valid on the command line as a rule in the makefile. So, rules are not restricted to invocation of the toolset.

**Example**

To build the target test.elf, call mktc with one of the following lines:

```bash
mktc test.elf
mktc --option-file=mymake.mak test.elf
```

By default the make utility reads the file makefile so you do not need to specify it on the command line. If you want to use another name for the makefile, use the option `--option-file (-f)`.

If you do not specify a target, mktc uses the first target defined in the makefile. In this example it would build test.src instead of test.elf.

Based on the sample invocation, the make utility now tries to build test.elf based on the makefile and performs the following steps:

1. From the makefile the make utility reads that test.elf depends on test.o.
2. If test.o does not exist or is out-of-date, the make utility first tries to build this file and reads from the makefile that test.o depends on test.src.
3. If test.src does exist, the make utility now creates test.o by executing the rule for it: astc test.src.
4. There are no other files necessary to create test.elf so the make utility now can use test.o to create test.elf by executing the rule: ltc test.o -o test.elf ...

The make utility has now built test.elf but it only used the assembler to update test.o and the linker to create test.elf.

If you compare this to the control program:

cctc test.c

This invocation has the same effect but now all files are recompiled (assembled, linked and located).

9.2.1. Calling the Make Utility

You can only call the make utility from the command line. The invocation syntax is:

```
mktc [ [option]... [target]... [macro=def]... ]
```

For example:

```
mktc test.elf
```

- target You can specify any target that is defined in the makefile. A target can also be one of the intermediate files specified in the makefile.
- macro=def Macro definition. This definition remains fixed for the mktc invocation. It overrides any regular definitions for the specified macro within the makefiles and from the environment. It is inherited by subordinate mktc's but act as an environment variable for these. That is, depending on the -e setting, it may be overridden by a makefile definition.
- option For a complete list and description of all make utility options, see Section 11.6, Make Utility Options.

Exit status

The make utility returns an exit status of 1 when it halts as a result of an error. Otherwise it returns an exit status of 0.

9.2.2. Writing a Makefile

In addition to the standard makefile makefile, the make utility always reads the makefile mktc.mk before other inputs. This system makefile contains implicit rules and predefined macros that you can use in the makefile makefile.

With the option -r (Do not read the mktc.mk file) you can prevent the make utility from reading mktc.mk.

The default name of the makefile is makefile in the current directory. If you want to use another makefile, use the option --option-file (-f).

The makefile can contain a mixture of:

- targets and dependencies
To continue a line on the next line, terminate it with a backslash (\):

```bash
# this comment line is continued\
on the next line
```

If a line must end with a backslash, add an empty macro:

```bash
# this comment line ends with a backslash \$(EMPTY)
# this is a new line
```

### 9.2.2.1. Targets and Dependencies

The basis of the makefile is a set of targets, dependencies and rules. A target entry in the makefile has the following format:

```
target ... : [dependency ...] [; rule]
  [rule]
  ...
```

Target lines must always start at the beginning of a line, leading white spaces (tabs or spaces) are not allowed. A target line consists of one or more targets, a semicolon and a set of files which are required to build the target (dependencies). The target itself can be one or more filenames or symbolic names:

```
all: demo.elf final.elf

demo.elf final.elf: test.o demo.o final.o
```

You can now can specify the target you want to build to the make utility. The following three invocations all have the same effect:

```
mktc
mktc all
mktc demo.elf final.elf
```

If you do not specify a target, the first target in the makefile (in this example all) is built. The target all depends on demo.elf and final.elf so the second and third invocation have the same effect and the files demo.elf and final.elf are built.

You can normally use colons to denote drive letters. The following works as intended:

```
c:foo.o : a:foo.c
```
If a target is defined in more than one target line, the dependencies are added to form the target's complete dependency list:

```
all: demo.elf  # These two lines are equivalent with:
all: final.elf # all: demo.elf final.elf
```

**Special targets**

There are a number of special targets. Their names begin with a period.

<table>
<thead>
<tr>
<th>Target</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.DEFAULT</td>
<td>If you call the make utility with a target that has no definition in the makefile, this target is built.</td>
</tr>
<tr>
<td>.DONE</td>
<td>When the make utility has finished building the specified targets, it continues with the rules following this target.</td>
</tr>
<tr>
<td>.IGNORE</td>
<td>Non-zero error codes returned from commands are ignored. Encountering this in a makefile is the same as specifying the option -i on the command line.</td>
</tr>
<tr>
<td>.INIT</td>
<td>The rules following this target are executed before any other targets are built.</td>
</tr>
<tr>
<td>.PRECIOUS</td>
<td>Dependency files mentioned for this target are never removed. Normally, if a command in a rule returns an error or when the target construction is interrupted, the make utility removes that target file. You can use the option -p on the command line to make all targets precious.</td>
</tr>
<tr>
<td>.SILENT</td>
<td>Commands are not echoed before executing them. Encountering this in a makefile is the same as specifying the option -s on the command line.</td>
</tr>
<tr>
<td>.SUFFIXES</td>
<td>This target specifies a list of file extensions. Instead of building a completely specified target, you now can build a target that has a certain file extension. Implicit rules to build files with a number of extensions are included in the system makefile mktc.mk. If you specify this target with dependencies, these are added to the existing .SUFFIXES target in mktc.mk. If you specify this target without dependencies, the existing list is cleared.</td>
</tr>
</tbody>
</table>

### 9.2.2.2. Makefile Rules

A line with leading white space (tabs or spaces) is considered as a rule and associated with the most recently preceding dependency line. A rule is a line with commands that are executed to build the associated target. A target-dependency line can be followed by one or more rules.

```
final.src : final.c  # target and dependency
    move test.c final.c # rule1
    ctc final.c        # rule2
```

You can precede a rule with one or more of the following characters:

- `@` does not echo the command line, except if `-n` is used.
- `-` the make utility ignores the exit code of the command (ERRORLEVEL in MS-DOS). Normally the make utility stops if a non-zero exit code is returned. This is the same as specifying the option `-i` on the command line or specifying the special `.IGNORE` target.
+ The make utility uses a shell or COMMAND.COM to execute the command. If the '+' is not followed by a shell line, but the command is a DOS command or if redirection is used (<, |, >), the shell line is passed to COMMAND.COM anyway.

You can force `mktc` to execute multiple command lines in one shell environment. This is accomplished with the token combination `;\`. For example:

```
cd c:\Tasking\bin \\
mktc -V
```

Note that the `;` must always directly be followed by the `\` token. Whitespace is not removed when it is at the end of the previous command line or when it is in front of the next command line. The use of the `;` as an operator for a command (like a semicolon `;` separated list with each item on one line) and the `\` as a layout tool is not supported, unless they are separated with whitespace.

**Inline temporary files**

The make utility can generate inline temporary files. If a line contains `<<LABEL` (no whitespaces!) then all subsequent lines are placed in a temporary file until the line `LABEL` is encountered. Next, `<<LABEL` is replaced by the name of the temporary file. For example:

```
ltc -o $@ -f <<EOF
  $(separate "\n" $(match .o $!))
  $(separate "\n" $(match .a $!))
  $(LKFLAGS)
EOF
```

The three lines between `<<EOF` and `EOF` are written to a temporary file (for example `mkce4c0a.tmp`), and the rule is rewritten as: `ltc -o $@ -f mkce4c0a.tmp`.

**Suffix targets**

Instead of specifying a specific target, you can also define a general target. A general target specifies the rules to generate a file with extension `.ex1` to a file with extension `.ex2`. For example:

```
.SUFFIXES: .c
.c.o     :
  cctc -c $<
```

Read this as: to build a file with extension `.o` out of a file with extension `.c`, call the control program with `-c $<`. `$<` is a predefined macro that is replaced with the name of the current dependency file. The special target `.SUFFIXES:` is followed by a list of file extensions of the files that are required to build the target.

**Implicit rules**

Implicit rules are stored in the system makefile `mktc.mk` and are intimately tied to the `.SUFFIXES` special target. Each dependency that follows the `.SUFFIXES` target, defines an extension to a filename which must be used to build another file. The implicit rules then define how to actually build one file from another. These files share a common basename, but have different extensions.

If the specified target on the command line is not defined in the makefile or has not rules in the makefile, the make utility looks if there is an implicit rule to build the target.
Example:

```
LIB = -lc -lfp -lrt       # macro

prog.elf:  prog.o sub.o
    ltc  prog.o sub.o $(LIB) -o prog.elf

prog.o:    prog.c inc.h
    ctc  prog.c
    astc prog.src

sub.o:     sub.c inc.h
    ctc  sub.c
    astc sub.src
```

This makefile says that `prog.elf` depends on two files `prog.o` and `sub.o`, and that they in turn depend on their corresponding source files (`prog.c` and `sub.c`) along with the common file `inc.h`.

The following makefile uses implicit rules (from `mktc.mk`) to perform the same job.

```
LDFLAGS = -lc -lfp -lrt       # macro used by implicit rules

prog.elf: prog.o sub.o       # implicit rule used
prog.o:    prog.c inc.h      # implicit rule used
sub.o:     sub.c inc.h      # implicit rule used
```

### 9.2.2.3. Macro Definitions

A macro is a symbol name that is replaced with its definition before the makefile is executed. Although the macro name can consist of lower case or upper case characters, upper case is an accepted convention. The general form of a macro definition is:

```
MACRO = text and more text
```

Spaces around the equal sign are not significant.

To use a macro, you must access its contents:

```
$(MACRO)       # you can read this as
${MACRO}       # the contents of macro MACRO
```

If the macro name is a single character, the parentheses are optional. Note that the expansion is done recursively, so the body of a macro may contain other macros. These macros are expanded when the macro is actually used, not at the point of definition:

```
FOOD = $(EAT) and $(DRINK)
EAT = meat and/or vegetables
DRINK = water
export FOOD
```

The macro `FOOD` is expanded as `meat and/or vegetables and water` at the moment it is used in the export line, and the environment variable `FOOD` is set accordingly.
Predefined macros

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAKE</td>
<td>Holds the value mktc. Any line which uses MAKE, temporarily overrides the option -n (Show commands without executing), just for the duration of the one line. This way you can test nested calls to MAKE with the option -n.</td>
</tr>
<tr>
<td>MAKEFLAGS</td>
<td>Holds the set of options provided to mktc (except for the options -f and -d). If this macro is exported to set the environment variable MAKEFLAGS, the set of options is processed before any command line options. You can pass this macro explicitly to nested mktc's, but it is also available to these invocations as an environment variable.</td>
</tr>
<tr>
<td>PRODDIR</td>
<td>Holds the name of the directory where mktc is installed. You can use this macro to refer to files belonging to the product, for example a library source file.</td>
</tr>
<tr>
<td></td>
<td>DOPRINT = $(PRODDIR)/lib/src/_doprint.c</td>
</tr>
<tr>
<td></td>
<td>When mktc is installed in the directory c:/Tasking/bin this line expands to:</td>
</tr>
<tr>
<td></td>
<td>DOPRINT = c:/Tasking/lib/src/_doprint.c</td>
</tr>
<tr>
<td>SHELLCMD</td>
<td>Holds the default list of commands which are local to the SHELL. If a rule is an invocation of one of these commands, a SHELL is automatically spawned to handle it.</td>
</tr>
<tr>
<td>$</td>
<td>This macro translates to a dollar sign. Thus you can use &quot;$$&quot; in the makefile to represent a single &quot;$&quot;.</td>
</tr>
</tbody>
</table>

Dynamically maintained macros

There are several dynamically maintained macros that are useful as abbreviations within rules. It is best not to define them explicitly.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$*</td>
<td>The basename of the current target.</td>
</tr>
<tr>
<td>$&lt;</td>
<td>The name of the current dependency file.</td>
</tr>
<tr>
<td>$@</td>
<td>The name of the current target.</td>
</tr>
<tr>
<td>$?</td>
<td>The names of dependents which are younger than the target.</td>
</tr>
<tr>
<td>$!</td>
<td>The names of all dependents.</td>
</tr>
</tbody>
</table>

The $< and $* macros are normally used for implicit rules. They may be unreliable when used within explicit target command lines. All macros may be suffixed with $F to specify the Filename components (e.g., $(*F), @{$F}). Likewise, the macros $*, $< and $@ may be suffixed by $D to specify the Directory component.

The result of the $* macro is always without double quotes ("), regardless of the original target having double quotes ("), around it or not.

The result of using the suffix $F (Filename component) or $D (Directory component) is also always without double quotes ("), regardless of the original contents having double quotes (") around it or not.
9.2.2.4. Makefile Functions

A function not only expands but also performs a certain operation. Functions syntactically look like macros but have embedded spaces in the macro name, e.g. `$(match arg1 arg2 arg3 )`. All functions are built-in and currently there are five of them: `match`, `separate`, `protect`, `exist` and `nexist`.

**match**

The `match` function yields all arguments which match a certain suffix:

```
$(match .o prog.o sub.o mylib.a)
```

yields:

```
prog.o sub.o
```

**separate**

The `separate` function concatenates its arguments using the first argument as the separator. If the first argument is enclosed in double quotes then `\n` is interpreted as a newline character, `\t` is interpreted as a tab, `\ooo` is interpreted as an octal value (where, `ooo` is one to three octal digits), and spaces are taken literally. For example:

```
$(separate "\n" prog.o sub.o)
```

results in:

```
prog.o
sub.o
```

Function arguments may be macros or functions themselves. So,

```
$(separate "\n" $(match .o $!))
```

yields all object files the current target depends on, separated by a newline string.

**protect**

The `protect` function adds one level of quoting. This function has one argument which can contain white space. If the argument contains any white space, single quotes, double quotes, or backslashes, it is enclosed in double quotes. In addition, any double quote or backslash is escaped with a backslash.

Example:

```
echo $(protect I'll show you the "protect" function)
```

yields:

```
echo "I'll show you the \"protect\" function"
```

**exist**

The `exist` function expands to its second argument if the first argument is an existing file or directory.
Example:

$(exist test.c cctc test.c)

When the file test.c exists, it yields:

cctc test.c

When the file test.c does not exist nothing is expanded.

_nexist_

The _nexist_ function is the opposite of the _exist_ function. It expands to its second argument if the first argument is not an existing file or directory.

Example:

$(nexist test.src cctc test.c)

### 9.2.2.5. Conditional Processing

Lines containing _ifdef_, _ifndef_, _else_ or _endif_ are used for conditional processing of the makefile. They are used in the following way:

```
ifdef macro-name
if-lines
else
else-lines
endif
```

The _if-lines_ and _else-lines_ may contain any number of lines or text of any kind, even other _ifdef_, _ifndef_, _else_ and _endif_ lines, or no lines at all. The _else_ line may be omitted, along with the _else-lines_ following it.

First the _macro-name_ after the _ifdef_ command is checked for definition. If the macro is defined then the _if-lines_ are interpreted and the _else-lines_ are discarded (if present). Otherwise the _if-lines_ are discarded; and if there is an _else_ line, the _else-lines_ are interpreted; but if there is no _else_ line, then no lines are interpreted.

When you use the _ifndef_ line instead of _ifdef_, the macro is tested for not being defined. These conditional lines can be nested up to 6 levels deep.

### 9.2.2.6. Comment, Include and Export Lines

**Comment lines**

Anything after a "#" is considered as a comment, and is ignored. If the "#" is inside a quoted string, it is not treated as a comment. Completely blank lines are ignored.

```
test.src : test.c # this is comment and is
    cctc test.c # ignored by the make utility
```
Include lines

An include line is used to include the text of another makefile (like including a .h file in a C source). Macros in the name of the included file are expanded before the file is included. Include files may be nested.

include makefile2

Export lines

An export line is used to export a macro definition to the environment of any command executed by the make utility.

GREETING = Hello
export GREETING

This example creates the environment variable GREETING with the value Hello. The macro is exported at the moment the export line is read so the macro definition has to precede the export line.

9.3. Archiver

The archiver artc is a program to build and maintain your own library files. A library file is a file with extension .a and contains one or more object files (.o) that may be used by the linker.

The archiver has five main functions:

• Deleting an object module from the library
• Moving an object module to another position in the library file
• Replacing an object module in the library or add a new object module
• Showing a table of contents of the library file
• Extracting an object module from the library

The archiver takes the following files for input and output:

The linker optionally includes object modules from a library if that module resolves an external symbol definition in one of the modules that are read before.
9.3.1. Calling the Archiver

You can create a library in Eclipse, which calls the archiver or you can call the archiver on the command line.

To create a library in Eclipse

Instead of creating a TriCore absolute ELF file, you can choose to create a library. You do this when you create a new project with the New C/C++ Project wizard. (File » ) select the option in the following dialog.

1. From the File menu, select New » Other... » TASKING C/C++ » TASKING VX-toolset for TriCore C/C++ Project.

   The New C/C++ Project wizard appears.

2. Enter a project name.

3. In the Project types box, select TASKING TriCore Library and click Next >.

4. Follow the rest of the wizard and click Finish.

5. Add the files to your project.

6. Build the project as usual. For example, select Project » Build Project (1).

   Eclipse builds the library. Instead of calling the linker, Eclipse now calls the archiver.

Command line invocation

You can call the archiver from the command line. The invocation syntax is:

\[ \text{artc key_option [sub_option...] library [object_file]} \]

key_option With a key option you specify the main task which the archiver should perform. You must always specify a key option.

sub_option Sub-options specify into more detail how the archiver should perform the task that is specified with the key option. It is not obligatory to specify sub-options.

library The name of the library file on which the archiver performs the specified action. You must always specify a library name, except for the options -? and -V. When the library is not in the current directory, specify the complete path (either absolute or relative) to the library.

object_file The name of an object file. You must always specify an object file name when you add, extract, replace or remove an object file from the library.

Options of the archiver utility

The following archiver options are available:

<table>
<thead>
<tr>
<th>Description</th>
<th>Option</th>
<th>Sub-option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main functions (key options)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Description

<table>
<thead>
<tr>
<th>Description</th>
<th>Option</th>
<th>Sub-option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Replace or add an object module</td>
<td>-r</td>
<td>-r -a -b -c -u -v</td>
</tr>
<tr>
<td>Extract an object module from the library</td>
<td>-x</td>
<td>-x -v</td>
</tr>
<tr>
<td>Delete object module from library</td>
<td>-d</td>
<td>-d -v</td>
</tr>
<tr>
<td>Move object module to another position</td>
<td>-m</td>
<td>-m -a -b -v</td>
</tr>
<tr>
<td>Print a table of contents of the library</td>
<td>-t</td>
<td>-t -s0 -s1</td>
</tr>
<tr>
<td>Print object module to standard output</td>
<td>-p</td>
<td></td>
</tr>
</tbody>
</table>

### Sub-options

<table>
<thead>
<tr>
<th>Sub-options</th>
<th>Option</th>
<th>Sub-option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Append or move new modules after existing module</td>
<td>-a</td>
<td>-a name</td>
</tr>
<tr>
<td>Append or move new modules before existing module</td>
<td>-b</td>
<td>-b name</td>
</tr>
<tr>
<td>Create library without notification if library</td>
<td>-c</td>
<td></td>
</tr>
<tr>
<td>does not exist</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Preserve last-modified date from the library</td>
<td>-o</td>
<td></td>
</tr>
<tr>
<td>Print symbols in library modules</td>
<td>-s0</td>
<td>-s1</td>
</tr>
<tr>
<td>Replace only newer modules</td>
<td>-u</td>
<td></td>
</tr>
<tr>
<td>Verbose</td>
<td>-v</td>
<td></td>
</tr>
</tbody>
</table>

### Miscellaneous

<table>
<thead>
<tr>
<th>Miscellaneous</th>
<th>Option</th>
<th>Sub-option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display options</td>
<td>-?</td>
<td></td>
</tr>
<tr>
<td>Display version header</td>
<td>-V</td>
<td></td>
</tr>
<tr>
<td>Read options from file</td>
<td>-f</td>
<td>-f file</td>
</tr>
<tr>
<td>Suppress warnings above level n</td>
<td>-w</td>
<td>-wn</td>
</tr>
</tbody>
</table>

For a complete list and description of all archiver options, see Section 11.7, Archiver Options.

### 9.3.2. Archiver Examples

#### Create a new library

If you add modules to a library that does not yet exist, the library is created. To create a new library with the name `mylib.a` and add the object modules `cstart.o` and `calc.o` to it:

```
artc -r mylib.a cstart.o calc.o
```

#### Add a new module to an existing library

If you add a new module to an existing library, the module is added at the end of the module. (If the module already exists in the library, it is replaced.)

```
artc -r mylib.a mod3.o
```

#### Print a list of object modules in the library

To inspect the contents of the library:

```
```
artc -t mylib.a

The library has the following contents:

cstart.o
calc.o
mod3.o

**Move an object module to another position**

To move `mod3.o` to the beginning of the library, position it just before `cstart.o`:

artc -mb cstart.o mylib.a mod3.o

**Delete an object module from the library**

To delete the object module `cstart.o` from the library `mylib.a`:

artc -d mylib.a cstart.o

**Extract all modules from the library**

Extract all modules from the library `mylib.a`:

artc -x mylib.a

### 9.4. C++ Prelinker

The C++ prelinker `prelktc` is invoked at link time by the control program to manage automatic instantiation of template entities. It is given a complete list of the object files and libraries that are to be linked together. It examines the external names defined and referenced within those files, and finds cases where template entities are referenced but not defined. It then examines information in the object files that describes instantiations that could have been done during compilation, and assigns the needed instantiations to appropriate files. The prelinker then invokes the compiler again to compile those files, which will do the necessary instantiations.

For more information on this process see Section 5.1.1, *The Prelinker Phase*.

The command line invocation syntax of the C++ prelinker is:

```
prelktc [option]... files
```

where the `files` list includes all object files and libraries.

For a complete list and description of all prelinker options, see Section 11.8, *C++ Prelinker Options.*
Chapter 10. Using the Debugger

This chapter describes the debugger and how you can run and debug a C or C++ application. This chapter only describes the TASKING specific parts.

10.1. Reading the Eclipse Documentation

Before you start with this chapter, it is recommended to read the Eclipse documentation first. It provides general information about the debugging process. This chapter guides you through a number of examples using the TASKING debugger with simulation as target.

You can find the Eclipse documentation as follows:

1. Start Eclipse.
2. From the Help menu, select Help Contents.

The help screen overlays the Eclipse Workbench.
3. In the left pane, select C/C++ Development User Guide.
4. Open the Getting Started entry and select Debugging projects.

This Eclipse tutorial provides an overview of the debugging process. Be aware that the Eclipse example does not use the TASKING tools and TASKING debugger.

10.2. Creating a Customized Debug Configuration

Before you can debug a project, you need a Debug launch configuration. Such a configuration, identified by a name, contains all information about the debug project: which debugger is used, which project is used, which binary debug file is used, which perspective is used, ... and so forth.

When you created your project, a default launch configuration for the TASKING simulator is available. If you used the Target Board Configuration wizard, also a default debug launch configuration for your target board is available. At any time you can change this configuration or create a custom debug configuration.

To debug or run a project, you need at least one opened and active project in your workbench. In this chapter, it is assumed that the myproject is opened and active in your workbench.

Customize your debug configuration

To change or create your own debug configuration follow the steps below.

1. From the Run menu, select Debug...

The Debug dialog appears.
2. In the left pane, select the configuration you want to change, for example, TASKING Embedded C/C++ Application » myproject.simulator.
Or: click the **New launch configuration** button (/button) to add a new configuration.

The next dialog appears.

The dialog shows several tabs.

**Main tab**

On the **Main** tab, you can set the properties for the debug configuration such as a name for the configuration and the project and the application binary file which are used when you choose this configuration.

- **Name** is the name of the configuration. By default, this is the name of the project, optionally appended with **simulator** or **board**. You can give your configuration any name you want to distinguish it from the project name.

- In the **Project** field, you can choose the project for which you want to make a debug configuration. Because the project myproject is the active project, this project is filled in automatically. Click the **Browse...** button to select a different project. Only the *opened* projects in your workbench are listed.

- In the **C/C++ Application** field, you can choose the binary file to debug. The file myproject.elf is automatically selected from the active project.
You can use the option **Use linker/locator memory map file (.mdf) for memory map** to find errors in your application that cause access to non-existent memory or cause an attempt to write to read-only memory. When building your project, the linker/locator creates a memory description file (.mdf) file which describes the memory regions of the target you selected in your project properties. The debugger uses this file to initialize the debugging target.

This option is only useful in combination with a simulator as debug target. The debugger may fail to start if you use this option in combination with other debugging targets than a simulator.

### Arguments tab

If your application's `main()` function takes arguments, you can pass them in this tab. Arguments are conventionally passed in the `argv[]` array. Because this array is allocated in target memory, make sure you have allocated sufficient memory space for it.

- In the C/C++ perspective double-click to open the file `cstart.c` in the **Startup Code Editor** view. At the bottom of the view select the **Configuration** tab, enable the option **Enable passing argc/argv to main()** and specify a **Buffer size for argv**.
Debugger tab

On the Debugger tab you can set the debugger options. You can choose which debugger should be used and with what options it should work. The Debugger tab itself contains several tabs.

- **Execution Environment** tab: You can select on which target the application should be debugged. An application may run on an external evaluation board, or on a simulator using your own PC. For the evaluation board these settings should be the same as you specified in the Target Board Configuration wizard.

- **Communication Setup** tab: You can select the type of communication (RS-232, TCP/IP, CAN) for execution environments. This tab is grayed out for the simulator.

- **Initialization** tab: Enable one or more of the following options:
  - **Initial download of program**
    
    If enabled, the target application is downloaded onto the target. If disabled, only the debug information in the file is loaded, which may be useful when the application has already been downloaded (or flashed) earlier. If downloading fails, the debugger will shut down.
  - **Verify download of program**
If enabled, the debugger verifies whether the code and data has been downloaded successfully. This takes some extra time but may be useful if the connection to the target is unreliable.

- **Program flash when downloading**
  
  If enabled, also flash devices are programmed (if necessary). Flash programming will not work when you use a simulator.

- **Reset target**
  
  If enabled, the target is immediately reset after downloading has completed.

- **Goto main**
  
  If enabled, only the C startup code is processed when the debugger is launched. The application stops executing when it reaches the first C instruction in the function `main()`. Usually you enable this option in combination with the option **Reset Target**.

- **Break on exit**
  
  If enabled, the target halts automatically when the `exit()` function is called.

- **Target status polling interval**
  
  If you have set a breakpoint, the debugger checks the status of the target every `number` of seconds to find out if the breakpoint is hit. In this field you can change the polling frequency.

- **On the Miscellaneous tab you can specify several file locations.**
  
  - **Debugger location**
    
    The location of the debugger itself. This should not be changed.

  - **FSS root directory**
    
    The initial directory used by file system simulation (FSS) calls. See the description of the **FSS view**.

  - **ORTI file and KSM module**
    
    If you wish to use the debugger's special facilities for OSEK kernels, specify the name of your ORTI file and that of your KSM module (shared library) in the appropriate edit boxes. See also the description of the **RTOS view**.

  - **GDI log file and Debug instrument log file**
    
    You can use the options GDI log file and Debug instrument log file (if applicable) to control the generation of internal log files. These are primarily intended for use by or at the request of Altium support personnel.

  - **Set TCP port to**
    
    The debugger may use the TCP/IP protocol for internal purposes, for which it needs to reserve a TCP port number. In the unlikely case that the default number conflicts with a program already running, you can change the TCP port.
• Cache target access

Except when using a simulator, the debugger's performance is generally strongly dependent on the throughput and latency of the connection to the target. Depending on the situation, enabling this option may result in a noticeable improvement, as the debugger will then avoid re-reading registers and memory while the target remains halted. However, be aware that this may cause the debugger to show the wrong data if tasks with a higher priority or external sources can influence the halted target's state.

Source tab

On the **Source** tab, you can add additional source code locations in which the debugger should search for debug data.

• Usually, the default source code location is correct.

Common tab

On the **Common** tab you can set additional launch configuration settings.
10.3. Troubleshooting

If the debugger does not launch properly, this is likely due to mistakes in the settings of the execution environment or to an improper connection between the host computer and the execution environment. Always read the notes for your particular execution environment.

Some common problems you may check for, are:

<table>
<thead>
<tr>
<th>Problem</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wrong device name in the launch configuration</td>
<td>Make sure the specified device name is correct.</td>
</tr>
<tr>
<td>Invalid baud rate</td>
<td>Specify baud rate that matches the baud rate the execution environment is configured to expect.</td>
</tr>
<tr>
<td>No power to the execution environment.</td>
<td>Make sure the execution environment or attached probe is powered.</td>
</tr>
<tr>
<td>Wrong type of RS–232 cable.</td>
<td>Make sure you are using the correct type of RS-232 cable.</td>
</tr>
<tr>
<td>Cable connected to the wrong port on the execution environment or host.</td>
<td>Some target machines and hosts have several ports. Make sure you connect the cable to the correct port.</td>
</tr>
<tr>
<td>Problem</td>
<td>Solution</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Conflict between communication ports.</td>
<td>A device driver or background application may use the same communications port on the host system as the debugger. Disable any service that uses the same port-number or choose a different port-number if possible.</td>
</tr>
<tr>
<td>Port already in use by another user.</td>
<td>The port may already be in use by another user on some UNIX hosts, or being allocated by a login process. Some target machines and hosts have several ports. Make sure you connect the cable to the correct port.</td>
</tr>
</tbody>
</table>

10.4. The TASKING Debug Perspective

After you have launched the debugger, you are either asked if the TASKING Debug perspective should be opened or it is opened automatically. The Debug perspective consists of several views.

To open views in the Debug perspective:

1. Make sure the Debug perspective is opened
2. From the Window menu, select Show View »
3. Select a view from the menu or choose Other... for more views.

By default, the Debug perspective is opened with the following views:
10.4.1. Debug View

The Debug view shows the target information in a tree hierarchy shown below with a sample of the possible icons:

<table>
<thead>
<tr>
<th>Icon</th>
<th>Session item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>🕵️‍♂️</td>
<td>Launch instance</td>
<td>Launch configuration name and launch type</td>
</tr>
<tr>
<td>🕵️‍♂️</td>
<td>Debugger instance</td>
<td>Debugger name and state</td>
</tr>
<tr>
<td>🕵️‍♂️</td>
<td>Thread instance</td>
<td>Thread number and state</td>
</tr>
<tr>
<td>🕵️‍♂️</td>
<td>Stack frame instance</td>
<td>Stack frame number, function, file name, and file line number</td>
</tr>
</tbody>
</table>

The number beside the thread label is a reference counter, not a thread identification number (TID).

**Stack display**

During debugging (running) the actual stack is displayed as it increases or decreases during program execution. By default, all views present information that is related to the current stack item (variables, memory, source code etc.). To obtain the information from other stack items, click on the item you want.
The Debug view displays stack frames as child elements. It displays the reason for the suspension beside the thread, (such as end of stepping range, breakpoint hit, and signal received). When a program exits, the exit code is displayed.

The Debug view contains numerous functions for controlling the individual stepping of your programs and controlling the debug session. You can perform actions such as terminating the session and stopping the program. All functions are available from the right-click menu, though commonly used functions are also available from the toolbar in the Debug view.

### Controlling debug sessions

<table>
<thead>
<tr>
<th>Icon</th>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Remove all" /></td>
<td>Remove all</td>
<td>Removes all terminated launches.</td>
</tr>
<tr>
<td><img src="image" alt="Restart" /></td>
<td>Restart</td>
<td>Restarts the application. The target system is <em>not</em> reset.</td>
</tr>
<tr>
<td><img src="image" alt="Reset target system" /></td>
<td>Reset target system</td>
<td>Resets the target system and restarts the application.</td>
</tr>
<tr>
<td><img src="image" alt="Resume" /></td>
<td>Resume</td>
<td>Resumes the application after it was suspended (manually, breakpoint, signal).</td>
</tr>
<tr>
<td><img src="image" alt="Suspend" /></td>
<td>Suspend</td>
<td>Suspends the application (pause). Use the Resume button to continue.</td>
</tr>
<tr>
<td><img src="image" alt="Relaunch" /></td>
<td>Relaunch</td>
<td>Right-click menu. Restarts the selected debug session when it was terminated. If the debug session is still running, a new debug session is launched.</td>
</tr>
<tr>
<td><img src="image" alt="Reload current application" /></td>
<td>Reload current application</td>
<td>Reloads the current application without restarting the debug session. The application does restart of course.</td>
</tr>
<tr>
<td><img src="image" alt="Terminate" /></td>
<td>Terminate</td>
<td>Ends the selected debug session and/or process. Use Relaunch to restart this debug session, or start another debug session.</td>
</tr>
<tr>
<td><img src="image" alt="Terminate all" /></td>
<td>Terminate all</td>
<td>Right-click menu. As terminate. Ends <em>all</em> debug sessions.</td>
</tr>
<tr>
<td><img src="image" alt="Terminate and remove" /></td>
<td>Terminate and remove</td>
<td>Right-click menu. Ends the debug session and removes it from the Debug view.</td>
</tr>
<tr>
<td><img src="image" alt="Terminate and Relaunch" /></td>
<td>Terminate and Relaunch</td>
<td>Right-click menu. Ends the debug session and relaunches it. This is the same as choosing Terminate end then Relaunch.</td>
</tr>
<tr>
<td><img src="image" alt="Disconnect" /></td>
<td>Disconnect</td>
<td>Detaches the debugger from the selected process (useful for debugging attached processes)</td>
</tr>
</tbody>
</table>

### Stepping through the application

<table>
<thead>
<tr>
<th>Icon</th>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Step into" /></td>
<td>Step into</td>
<td>Steps to the next source line or instruction</td>
</tr>
<tr>
<td><img src="image" alt="Step over" /></td>
<td>Step over</td>
<td>Steps over a called function. The function is executed and the application suspends at the next instruction after the call.</td>
</tr>
<tr>
<td><img src="image" alt="Step return" /></td>
<td>Step return</td>
<td>Executes the current function. The application suspends at the next instruction after the return of the function.</td>
</tr>
</tbody>
</table>
### Description

The Breakpoints view shows a list of breakpoints that are currently set. The button bar in the Breakpoints view gives access to several common functions. The right-most button opens the Breakpoints menu.

### Types of breakpoints

To access the breakpoints dialog, add a breakpoint as follows:

1. Click the Add TASKING Breakpoint button.

   *The Breakpoints dialog appears.*

Each tab lets you set a breakpoint of a special type. You can set the following types of breakpoints:

- **File breakpoint**

  The target halts when it reaches the specified line of the specified source file. Note that it is possible that a source line corresponds to multiple addresses, for example when a header file has been included into two different source files or when inlining has occurred. If so, the breakpoint will be associated with all those addresses.

- **Function**

  The target halts when it reaches the first line of the specified function. If no source file has been specified and there are multiple functions with the given name, the target halts on all of those. Note that function breakpoints generally will not work on inlined instances of a function.
The target halts when it reaches the specified instruction address.

The target halts when it reaches the specified stack level.

The target halts when the given variable is read or written to, as specified.

The target halts when the given number of instructions has been executed.

The target halts when the given number of clock cycles has elapsed.

The target halts when the given amount of time elapsed.

In addition to the type of the breakpoint, you can specify the condition that must be met to halt the program.

In the Condition field, type a condition. The condition is an expression which evaluates to ‘true’ (non-zero) or ‘false’ (zero). The program only halts on the breakpoint if the condition evaluates to ‘true’.

In the Ignore count field, you can specify the number of times the breakpoint is ignored before the program halts. For example, if you want the program to halt only in the fifth iteration of a while-loop, type ‘4’: the first four iterations are ignored.

10.4.3. File System Simulation (FSS) View

Description

The File System Simulation (FSS) view is automatically opened when the target requests FSS input or generates FSS output. The virtual terminal that the FSS view represents, follows the VT100 standard. If you right-click in the view area of the FSS view, a menu is presented which gives access to some self-explanatory functions.

VT100 characteristics

The queens example demonstrates some of the VT100 features. (You can find the queens example in the \examples directory from where you can import it into your workspace.) Per debugging session, you can have more than one FSS view, each of which is associated with a positive integer. By default, the view "FSS #1" is associated with the standard streams stdin, stdout, stderr and stdaux. Other views can be accessed by opening a file named "terminal window <number>", as shown in the example below.
FILE * f3 = fopen("terminal window 3", "rw");
fprintf(f3, "Hello, window 3.\n");
fclose(f3);

You can set the initial working directory of the target application in the Debug configuration dialog (see also Section 10.2, Creating a Customized Debug Configuration):

1. On the **Debugger** tab, select the **Miscellaneous** sub-tab.

2. In the **FSS root directory** field, specify the FSS root directory.

The FSS implementation is designed to work without user intervention. Nevertheless, there are some aspects that you need to be aware of.

First, the interaction between the C library code (in the files `dbg*.c` and `dbg*.h`; see Section 12.1.5, `dbg.h`) and the debugger takes place via a breakpoint, which incidentally is not shown in the Breakpoints view. Depending on the situation this may be a hardware breakpoint, which may be in short supply.

Secondly, proper operation requires certain code in the C library to have debug information. This debug information should normally be present but might get lost when this information is stripped later in the development process.

### 10.4.4. Disassembly View

The Disassembly view shows target memory disassembled into instructions and / or data. If possible, the associated C / C++ source code is shown as well. The **Address** field shows the address of the current selected line of code.

To view the contents of a specific memory location, type the address in the **Address** field. If the address is invalid, the field turns red.

### 10.4.5. Expressions View

The Expressions view allows you to evaluate and watch regular C expressions.

To add an expression:

Click **OK** to add the expression.

1. Right-click in the Expressions View and select **Add Watch Expression**.

   *The Add Watch Expression dialog appears.*

2. Enter an expression you want to watch during debugging, for example, the variable name "i"

If you have added one or more expressions to watch, the right-click menu provides options to **Remove** and **Edit** or **Enable** and **Disable** added expressions.

- You can access target registers directly using `#NAME`. For example "arr[#R0 << 3]" or "#TIMER3 = m++". If a register is memory-mapped, you can also take its address, for example, "&#ADCIN".

- Expressions may contain target function calls like for example "g1 + invert(&g2)". Be aware that this will not work if the compiler has optimized the code in such a way that the original function code
does not actually exist anymore. This may be the case, for example, as a result of inlining. Also, be aware that the function and its callees use the same stack(s) as your application, which may cause problems if there is too little stack space. Finally, any breakpoints present affect the invoked code in the normal way.

10.4.6. Memory View

Use the Memory view to inspect and change process memory. The Memory view supports the same addressing as the C and C++ languages. You can address memory using expressions such as:

- 0x0847d3c
- (&y)+1024
- *ptr

**Monitors**

To monitor process memory, you need to add a monitor:

1. In the Debug view, select a debug session. Selecting a thread or stack frame automatically selects the associated session.
2. Click the **Add Memory Monitor** button in the Memory Monitors pane.
   
   *The Monitor Memory dialog appears.*
3. Type the address or expression that specifies the memory section you want to monitor and click **OK**.
   
   *The monitor appears in the monitor list and the Memory Renderings pane displays the contents of memory locations beginning at the specified address.*

To remove a monitor:

1. In the Monitors pane, right-click on a monitor.
2. From the popup menu, select **Remove Memory Monitor**.

**Renderings**

You can inspect the memory in so-called renderings. A rendering specifies how the output is displayed: hexadecimal, ASCII, signed integer or unsigned integer. You can add or remove renderings per monitor. Though you cannot change a rendering, you can add or remove them:

1. Click the **Add Rendering** button in the Memory Renderings pane.
   
   *The Add Memory Rendering dialog appears.*
2. Select the rendering you want (**Hex**, **ASCII**, **Signed Integer** or **Unsigned Integer**) and click **OK**.

To remove a rendering:

1. Right-click on a memory address in the rendering.
2. From the popup menu, select **Remove Rendering**.

**Changing memory contents**

In a rendering you can change the memory contents. Simply type a new value.

---

**Warning**: Changing process memory can cause a program to crash.

The right-click popup menu gives some more options for changing the memory contents or to change the layout of the memory representation.

### 10.4.7. Compare Application View

You can use the Compare Application view to check if the downloaded application matches the application in memory. Differences may occur, for example, if you changed memory addresses in the Memory view.

- To check for differences, click the **Compare** button.

### 10.4.8. Heap View

With the Heap view you can inspect the status of the heap memory. This can be illustrated with the following example:

```c
string = (char *) malloc(100);
strcpy ( string, "abcdefgh" );
free (string);
```

If you step through these lines during debugging, the Heap view shows the situation after each line has been executed. Before any of these lines has been executed, there is no memory allocated and the Heap view is empty.

- After the first line the Heap view shows that memory is occupied, the description tells where the block starts, how large it is (100 MAUs) and what its content is (0x0, 0x0, ...).

- After the second line, "abcdefgh" has been copied to the allocated block of memory. The description field of the Heap view again shows the actual contents of the memory block (0x61, 0x62,...).

- The third line frees the memory. The Heap view is empty again because after this line no memory is allocated anymore.

### 10.4.9. Logging View

Use the Logging view to control the generation of internal log files. This view is intended mainly for use by or at the request of TASKING support personnel.

### 10.4.10. RTOS View

The debugger has special support for debugging real-time operating systems (RTOSs). This support is implemented in an RTOS-specific shared library called a **kernel support module** (KSM) or **RTOS-aware debugging module** (RADM). Specifically, the TASKING VX-toolset for TriCore ships with a KSM supporting
the OSEK standard. You have to create your own OSEK Run Time Interface (ORTI) and specify this file on the Miscellaneous sub tab while configuring a customized debug configuration (see also Section 10.2, Creating a Customized Debug Configuration):

1. From the Run menu, select Debug...

   The Debug dialog appears.

2. In the left pane, select the configuration you want to change, for example, TASKING Embedded C/C++ Application » myproject.simulator.
   Or: click the New launch configuration button ( ) to add a new configuration.

3. On the Debugger tab, select the Miscellaneous tab

4. In the ORTI file field, specify the name of your own ORTI file.

The debugger supports ORTI specifications v2.0 and v2.1.

10.4.11. TASKING Registers View

When first opened, the TASKING Registers view shows a number of register groups, which together contain all known registers. You can expand each group to see which registers they contain and examine the register's values while stepping through your application. This view has a number of features:

• While you step through the application, the registers involved in the step turn yellow.

• You can change each register's value.

• You can copy registers and/or groups to the windows clipboard: select the groups and/or individual registers, right-click on a register(group) and from the popup menu choose Copy Registers. You can paste the copied selection as text in, for example, a text editor.

• You can change the way the register value is displayed: right-click on a register(group) and from the popup menu choose the desired display mode (Natural, Hexadecimal, Decimal, Binary, Octal)

• For registers that are depicted with the icon _gateway_mode_, the menu entry Symbolic Representation is available in their right-click popup menu. This opens a new view in which shows the internal fields of the register. (Alternatively, you can double-click on the a register). For example, the SBCU_CON register from the Slow FPI Bus group may be shown as follows:
In this view you can set the individual values in the register, either by selecting a value from a drop-down box or by simply entering a value depending on the chosen field. To update the register with the new values, click the Update button.

- You can fully organize the register groups as you like: right-click on a register and from the popup menu use the menu items Add Register Group..., Edit Register Group... or Remove Register Group. This way you not only can choose which groups should be visible in the Register view, you can also create your own groups to which you add the registers of your interest.

To restore the original groups: right-click on a register and from the popup menu choose Restore Register Groups. Be aware: groups you have created will be removed, groups you have edited are restored to their original and groups you have deleted are placed back!

Viewing a register group in a separate view

For a better overview, you can open a register group in a separate view. To do so, double-click on the register group name. A new Register view is opened, showing all registers from the group. You can consider this view as a sub view of the Register view with roughly the same features.

10.4.12. Trace View

If tracing is enabled, the Trace view shows the code was most recently executed. For example, while you step through the application, the Trace view shows the executed code of each step. To enable tracing:

- From the Run menu, select Trace.

_A check mark appears when tracing is enabled._

The view has three tabs, Source, Instruction and Raw, each of which represents the trace in a different way. However, not all target environments will support all three of these. The view is updated automatically each time the target halts.
Chapter 11. Tool Options

This chapter provides a detailed description of the options for the compiler, assembler, linker, control program, make utility and the archiver.

Tool options in Eclipse (Menu entry)

For each tool option that you can set from within Eclipse, a Menu entry description is available. In Eclipse you can customize the tools and tool options in the following dialog:

1. From the Project menu, select Properties
   The Properties dialog appears.
2. In the left pane, expand C/C++ Build and select Settings.
   In the right pane the Settings appear.
3. Open the Tool Settings tab.
   You can set all tool options here.

Unless stated otherwise, all Menu entry descriptions expect that you have this Tool Settings tab open.

11.1. C Compiler Options

This section lists all C compiler options.

Options in Eclipse versus options on the command line

Most command line options have an equivalent option in Eclipse but some options are only available on the command line. Eclipse invokes the compiler via the control program. Therefore, it uses the syntax of the control program to pass options and files to the C compiler. If there is no equivalent option in Eclipse, you can specify a command line option in Eclipse as follows:

1. From the Project menu, select Properties
   The Properties dialog appears.
2. In the left pane, expand C/C++ Build and select Settings.
   In the right pane the Settings appear.
3. On the Tool Settings tab, select C/C++ Compiler » Miscellaneous.
4. In the Additional options field, enter one or more command line options.

Because Eclipse uses the control program, you have to precede the option with -Wc to pass the option via the control program directly to the C compiler.
Be aware that some command line options are not useful in Eclipse or just do not have any effect. For example, the option `-n` sends output to stdout instead of a file and has no effect in Eclipse.

**Short and long option names**

Options can have both short and long names. Short option names always begin with a single minus (-) character, long option names always begin with two minus (--) characters. You can abbreviate long option names as long as it forms a unique name. You can mix short and long option names on the command line.

Options can have flags or suboptions. To switch a flag 'on', use a lowercase letter or a `+longflag`. To switch a flag off, use an uppercase letter or a `-longflag`. Separate `longflags` with commas. The following two invocations are equivalent:

```
ctc -Oac test.c
ctc --optimize=+coalesce,+cse test.c
```

When you do not specify an option, a default value may become active.
C compiler option: --align

Menu entry

2. Enter a value in the Minimum alignment field.

Command line syntax

--align=value

Default: --align=1

Description

By default the C compiler aligns objects to the minimum alignment required by the architecture. With this option you can increase this alignment for objects of four bytes or larger. The value must be a power of two.

Example

To align all objects of four bytes or larger on a 4-byte boundary, enter:

    ctc --align=4 test.c

Instead of this option you can also specify the following pragma in your C source:

    #pragma align 4

With #pragma align restore you can return to the previous alignment setting.

Related information

Pragma align
C compiler option: --check

Menu entry

- 

Command line syntax

--check

Description

With this option you can check the source code for syntax errors, without generating code. This saves time in developing your application because the code will not actually be compiled.

The compiler reports any warnings and/or errors.

This option is available on the command line only.

Related information

Assembler option --check (Check syntax)
C compiler option: --compact-max-size

Menu entry

1. Select C/C++ Compiler » Optimization.
2. In the Maximum size for code compaction field, enter the maximum size of a match.

Command line syntax

--compact-max-size=value

Default: 200

Description

This option is related to the compiler optimization --optimize=+compact (Code compaction or reverse inlining). Code compaction is the opposite of inlining functions: large sequences of code that occur more than once, are transformed into a function. This reduces code size (possibly at the cost of execution speed).

However, in the process of finding sequences of matching instructions, compile time and compiler memory usage increase quadratically with the number of instructions considered for code compaction. With this option you tell the compiler to limit the number of matching instructions it considers for code compaction.

Example

To limit the maximum number of instructions in functions that the compiler generates during code compaction:

ctc --optimize=+compact --compact-max-size=100 test.c

Related information

C compiler option --optimize=+compact (Optimization: code compaction)
C compiler option --max-call-depth (Maximum call depth for code compaction)
C compiler option: --core

Menu entry

1. Select C/C++ Compiler » Miscellaneous.
2. Add the option --core to the Additional options field.

Command line syntax

--core=core

You can specify the following core arguments:

- tc1.2 TriCore 1.2 architecture
- tc1.3 TriCore 1.3 architecture
- tc1.3.1 TriCore 1.3.1 architecture, TriCore 1.3.1 instructions may be generated
- tc2 TriCore 2 architecture, TriCore 2 instructions may be generated

Default: derived from --cpu, if used, otherwise tc1.3

Description

With this option you specify the core architecture for a custom target processor for which you create your application. By default the TriCore toolset derives the core from the processor you selected.

With --core=tc1.3.1, the compiler can generate TriCore 1.3.1 instructions in the assembly file. The macro __TC131__ is defined in the C source file.

With --core=tc2, the compiler can generate TriCore 2 instructions in the assembly file. The macro __TC2__ is defined in the C source file.

If you select a valid target processor (command line option --cpu (-C)), the core is automatically set, based on the chosen target processor.

Example

Specify a custom core:

```bash
ctc --core=tc2 test.c
```

Related information

- C compiler option --cpu (Select processor)
- C compiler option --fpu-present (FPU present)
- C compiler option --mmu-present (MMU present)
C compiler option: --cpu (-C)

Menu entry
1. Expand C/C++ Build and select Processor.
2. From the Processor Selection list, select a processor or select User defined TriCore ....

Command line syntax

--cpu=cpu
-C cpu

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

Based on this option the compiler always includes the special function register file regcpu.sfr, unless you disable the option Automatic inclusion of '.sfr' file on the Preprocessing page (option --no-tasking-sfr).

Based on the target processor the compiler automatically detects whether a FPU-unit is present and whether the architecture is a TriCore2. This means you do not have to specify the compiler options --fpu-present and --core=tc2 explicitly when one of the supported derivatives is selected.

To avoid conflicts, make sure you specify the same target processor to the assembler (Eclipse and the control program do this automatically).

Example
To compile the file test.c for the TC1165 processor and use the SFR file regtc1165.sfr:

ctc --cpu=tcl1165 test.c

Related information

Assembler option --cpu (Select CPU)

C compiler option --no-tasking-sfr (Do not include SFR file)

Section 1.3.2, Accessing Hardware from C
C compiler option: --cse-all-addresses

Menu entry
1. Select C/C++ Compiler » Miscellaneous.
2. Add the option --cse-all-addresses to the Additional options field.

Command line syntax
--cse-all-addresses

Description
With this option you tell the C compiler to make all addresses available for common subexpression evaluation.

Normally the C compiler ignores __near and __ax addresses for common subexpressions. However, depending on the use of address registers and whether stack and/or addressed memory are internal or external, it might be wise to consider them for CSE.

Example
With the following invocation the C compiler makes all addresses available for common subexpression evaluation.

```
ctc --cse-all-addresses -Oc test.c
```

Related information
C compiler option --optimize=+cse (Optimization: common subexpression elimination)
C compiler option: --debug-info (-g)

Menu entry

1. Select C/C++ Compiler » Debugging.
2. To generate symbolic debug information, select Default, Call-frame only or Full. To disable the generation of debug information, select None.

Command line syntax

--debug-info [=suboption]

-g [suboption]

You can set the following suboptions:

<table>
<thead>
<tr>
<th>Suboption</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>all</td>
<td>Emit full symbolic debug information.</td>
</tr>
<tr>
<td>call-frame</td>
<td>Emit DWARF call-frame information only.</td>
</tr>
<tr>
<td>default</td>
<td>Emit default symbolic debug information.</td>
</tr>
</tbody>
</table>

Default: --debug-info (same as --debug-info=default)

Description

With this option you tell the compiler to add directives to the output file for including symbolic information. This facilitates high level debugging but increases the size of the resulting assembler file (and thus the size of the object file). For the final application, compile your C files without debug information.

Call-frame information only

With this suboption only call-frame information is generated. This enables you to inspect parameters of nested functions.

Default debug information

This provides all debug information you need to debug your application. It meets the debugging requirements in most cases without resulting in oversized assembler/object files.

Full debug information

With this information extra debug information is generated. In extraordinary cases you may use this debug information (for instance, if you use your own debugger which makes use of this information). With this suboption, the resulting assembler/object file increases significantly.

Related information

-
C compiler option: --default-a0-size (-Z)

Menu entry
1. Select C/C++ Compiler » Allocation
2. Enable the option Default __a0 allocations for objects below threshold, and enter a threshold value in bytes.

Command line syntax

```
--default-a0-size[=threshold]
-Z[threshold]
```

Default: --default-a0-size=0

Description

With this option you can specify a threshold value for __a0 allocation. If you do not specify a memory qualifier such as __near or __far in the declaration of an object, the compiler chooses where to place the object based on the size of the object.

First, the size of the object is checked against the near size threshold, according to the description of the --default-near-size (-N) option. If the size of the object is larger than the near size threshold, but lower or equal to the a0 size threshold, the object is allocated in __a0 memory. Larger objects, arrays and strings will be allocated __far.

By default the a0 size threshold is zero, which means that the compiler will never use __a0 memory unless you specify the --default-a0-size (-Z) option. If you omit a threshold value, all objects not allocated __near, including arrays and string constants, will be allocated in __a0 memory.

Allocation in __a0 memory means that the object is addressed indirectly, using A0 as the base pointer. The total amount of memory that can be addressed this way is 64 kB.

Instead of this option you can also use #pragma default_a0_size in the C source.

Example

To put all data objects with a size of 12 bytes or smaller, but larger than the default near threshold of 8, in __a0 sections:

```
ctc --default-a0-size=12 test.c
```

Related information

C compiler option --default-a1-size (-Y) (maximum size in bytes for data elements that are by default located in __a1 sections)

C compiler option --default-near-size (-N) (maximum size in bytes for data elements that are by default located in __near sections)
Section 1.2.1, Memory Qualifiers
C compiler option: --default-a1-size (-Y)

Menu entry
1. Select C/C++ Compiler » Allocation
2. Enable the option Default __a1 allocations for objects below threshold, and enter a threshold value in bytes.

Command line syntax

--default-a1-size[=threshold]
-YP[threshold]

Default: --default-a1-size=0

Description

With this option you can specify a threshold value for __a1 allocation. If you do not specify a memory qualifier such as __near or __far in the declaration of an object, the compiler chooses where to place the object based on the size of the object.

First, the size of the object is checked against the near size threshold, according to the description of the --default-near-size (-N) option. If the size of the object is larger than the near size threshold, but lower or equal to the a1 size threshold, the object is allocated in __a1 memory. Larger objects, arrays and strings will be allocated __far.

By default the a1 size threshold is zero, which means that the compiler will never use __a1 memory unless you specify the --default-a1-size (-Y) option. If you omit a threshold value, all objects not allocated __near, including arrays and string constants, will be allocated in __a1 memory.

Allocation in __a1 memory means that the object is addressed indirectly, using A1 as the base pointer. The total amount of memory that can be addressed this way is 64 kB.

Instead of this option you can also use #pragma default_a1_size in the C source.

Example

To put all data objects with a size of 12 bytes or smaller, but larger than the default near threshold of 8, in __a1 sections:

ctc --default-a1-size=12 test.c

Related information

C compiler option --default-a0-size (-Z) (maximum size in bytes for data elements that are by default located in __a0 sections)

C compiler option --default-near-size (-N) (maximum size in bytes for data elements that are by default located in __near sections)
Section 1.2.1, Memory Qualifiers
C compiler option: --default-near-size (-N)

Menu entry
1. Select C/C++ Compiler » Allocation
2. In the Threshold for putting data in __near field, enter a value in bytes.

Command line syntax

--default-near-size[=threshold]
-N[threshold]

Default: --default-near-size=8

Description

With this option you can specify a threshold value for __near allocation. If you do not specify __near or __far in the declaration of an object, the compiler chooses where to place the object. The compiler allocates objects smaller or equal to the threshold in __near sections. Larger objects are allocated in __a0, __a1 or __far sections.

If you omit a threshold value, all objects will be allocated __near, including arrays and string constants.

Instead of this option you can also use #pragma default_near_size in the C source.

Example

To put all data objects with a size of 12 bytes or smaller in __near sections:

ctc --default-near-size=12 test.c

Related information

C compiler option --default-a0-size (-Z) (maximum size in bytes for data elements that are by default located in __a0 sections)

C compiler option --default-a1-size (-Y) (maximum size in bytes for data elements that are by default located in __a1 sections)

Section 1.2.1, Memory Qualifiers
C compiler option: --define (-D)

Menu entry

1. Select **C/C++ Compiler » Preprocessing**.
   
   *The Defined symbols box shows the symbols that are currently defined.*

2. To define a new symbol, click on the **Add** button in the **Defined symbols** box.

3. Type the symbol definition (for example, `demo=1`)

   Use the **Edit** and **Delete** button to change a macro definition or to remove a macro from the list.

Command line syntax

```
--define=macro_name[=macro_definition]

-Dmacro_name[=macro_definition]
```

Description

With this option you can define a macro and specify it to the preprocessor. If you only specify a macro name (no macro definition), the macro expands as '1'.

You can specify as many macros as you like. Simply use the **Add** button to add new macro definitions.

On the command line, use the option **--define (-D)** multiple times. If the command line exceeds the limit of the operating system, you can define the macros in an option file which you then must specify to the compiler with the option **--option-file (-f) file**.

Defining macros with this option (instead of in the C source) is, for example, useful to compile conditional C source as shown in the example below.

Example

Consider the following C program with conditional code to compile a demo program and a real program:

```c
void main( void )
{
    #if DEMO
        demo_func();  /* compile for the demo program */
    #else
        real_func();  /* compile for the real program */
    #endif
}
```

You can now use a macro definition to set the DEMO flag:
ctc --define=DEMO test.c
ctc --define=DEMO=1 test.c

Note that both invocations have the same effect.

The next example shows how to define a macro with arguments. Note that the macro name and definition are placed between double quotes because otherwise the spaces would indicate a new option.

ctc --define="MAX(A,B)=((A) > (B) ? (A) : (B))" test.c

**Related information**

C compiler option **--undefine** (Remove preprocessor macro)

C compiler option **--option-file** (Specify an option file)
C compiler option: --dep-file

Menu entry
-

Command line syntax

--dep-file[=file]

Description

With this option you tell the compiler to generate dependency lines that can be used in a Makefile. In contrast to the option --preprocess=+make, the dependency information will be generated in addition to the normal output file.

By default, the information is written to a file with extension .d (one for every input file). When you specify a filename, all dependencies will be combined in the specified file.

Example

ctc --dep-file=test.dep test.c

The compiler compiles the file test.c, which results in the output file test.src, and generates dependency lines in the file test.dep.

Related information

C compiler option --preprocess=+make (Generate dependencies for make)
C compiler option: --diag

Menu entry

1. From the Window menu, select Show View » Other » Basic » Problems. 
   *The Problems view is added to the current perspective.*

2. In the Problems view right-click on a message.
   *A popup menu appears.*

3. Select Detailed Diagnostics Info.
   *A dialog box appears with additional information.*

Command line syntax

```
--diag=[format:]{all | nr,...}
```

You can set the following output formats:

- `html` HTML output.
- `text` ASCII text.

Default format: text

Description

With this option you can ask for an extended description of error messages in the format you choose. The output is directed to stdout (normally your screen) and in the format you specify. You can specify the following formats: `html`, `rtf` or `text` (default). To create a file with the descriptions, you must redirect the output.

With the suboption `all`, the descriptions of all error messages are given. If you want the description of one or more selected error messages, you can specify the error message numbers, separated by commas.

Example

To display an explanation of message number 282, enter:

```
ctc --diag=282
```

This results in the following message and explanation:

```
E282: unterminated comment
```

Make sure that every comment starting with /* has a matching */. Nested comments are not possible.
To write an explanation of all errors and warnings in HTML format to file cerrors.html, use redirection and enter:

```bash
ctc --diag=html:all > cerrors.html
```

**Related information**

Section 4.8, *C Compiler Error Messages*
C compiler option: --error-file

Menu entry
-

Command line syntax

--error-file [=file]

Description

With this option the compiler redirects error messages to a file. If you do not specify a filename, the error file will be named after the input file with extension .err.

Example

To write errors to errors.err instead of stderr, enter:

ctc --error-file=errors.err test.c

Related information
-


**C compiler option: --fp-trap**

**Menu entry**

1. Select **Linker » Libraries**.
2. Enable the option **Use trapped floating-point library**.

**Command line syntax**

```
--fp-trap
```

**Description**

With this option you tell the compiler to allow trapping of floating-point exceptions.

The floating-point instructions, as implemented in the FPU, need to be handled in a special way if floating-point trapping behavior is expected from the generated code. A trapped floating-point library is required.

**Related information**

Control program option **--fp-trap** (Use trapped floating-point library)

Section 8.3, *Linking with Libraries*
C compiler option: --fpu-present

Menu entry

1. Expand C/C++ Build and select Processor.
2. From the Processor Selection list, select a processor that has an FPU.

Command line syntax

--fpu-present

Description

With this option the compiler can generate single precision floating-point instructions in the assembly file. When you select this option, the macro __FPU__ is defined in the C source file.

If you select a valid target processor (command line option --cpu (-C)), this option is automatically set, based on the chosen target processor.

Example

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

```c
ctc --fpu-present test.c
```

Related information

C compiler option --core (Select TriCore architecture)
C compiler option --cpu (Select processor)
C compiler option: --help (-?)

Menu entry

Command line syntax

`--help[=item]`

`-?`

You can specify the following arguments:

- **intrinsics** `i` Show the list of intrinsic functions
- **options** `o` Show extended option descriptions
- **pragmas** `p` Show the list of supported pragmas
- **typedefs** `t` Show the list of predefined typedefs

Description

Displays an overview of all command line options. With an argument you can specify which extended information is shown.

Example

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

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

The following invocation displays a list of the available pragmas:

```
ctc --help=pragmas
```

Related information

-
C compiler option: --immediate-in-code

Menu entry

1. Select C/C++ Compiler » Miscellaneous.
2. Add the option --immediate-in-code to the Additional options field.

Command line syntax

--immediate-in-code

Description

By default the TriCore C compiler creates a data object to represent an immediate value of 32 or 64 bits, then loading this constant value directly into a register. With this option you can tell the compiler to code the immediate values directly into the instructions, thus using less data, but more code.

Actually when option --default-near-size < 4, 32-bit immediates will be coded into instructions anyhow, when it is >= 4 they will be located in neardata. When --default-near-size < 8, 64-bit immediates will be located in fardata, when it is >= 8 they will be located in neardata as well.

Instead of this option you can also specify the following pragma in your C source:

#pragma immediate_in_code

Related information

C compiler option --default-near-size (Maximum size for objects to be allocated by default in __near sections)

Pragma immediate_in_code
C compiler option: --include-directory (-I)

Menu entry

1. Select C/C++ Compiler » Include Paths.
   
   The Include paths box shows the directories that are added to the search path for include files.

2. To define a new directory for the search path, click on the Add button in the Include paths box.

3. Type or select a path.

   Use the Edit and Delete button to change a path or to remove a path from the list.

Command line syntax

--include-directory=path,...

-I path,...

Description

With this option you can specify the path where your include files are located. A relative path will be relative to the current directory.

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

1. The pathname in the C source file and the directory of the C source (only for #include files that are enclosed in "")

2. The path that is specified with this option.

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

4. The default directory $(PRODDIR)\include.

Example

Suppose that the C source file test.c contains the following lines:

#include <stdio.h>
#include "myinc.h"

You can call the compiler as follows:

ctc --include-directory=myinclude test.c

First the compiler looks for the file stdio.h in the directory myinclude relative to the current directory. If it was not found, the compiler searches in the environment variable and then in the default include directory.
The compiler now looks for the file `myinc.h` in the directory where `test.c` is located. If the file is not there the compiler searches in the directory `myinclude`. If it was still not found, the compiler searches in the environment variable and then in the default include directory.

**Related information**

C compiler option `--include-file` (Include file at the start of a compilation)
C compiler option: --include-file (-H)

Menu entry
1. Select C/C++ Compiler » Preprocessing.
   The Pre-include files box shows the files that are currently included before the compilation starts.
2. To define a new file, click on the Add button in the Pre-include files box.
3. Type the full path and file name or select a file.

Use the Edit and Delete button to change a file name or to remove a file from the list.

Command line syntax
--include-file=file,...
-H file,...

Description
With this option you include one or more extra files at the beginning of each C source file, before other includes. This is the same as specifying #include "file" at the beginning of each of your C sources.

Example
ctc --include-file=stdio.h test1.c test2.c

The file stdio.h is included at the beginning of both test1.c and test2.c.

Related information
C compiler option --include-directory (Add directory to include file search path)
C compiler option: --indirect

Menu entry
2. Enable the option Call functions indirectly.

Command line syntax
--indirect

Description
With this option you tell the compiler to generate code for indirect function calling.

Instead of this option you can also specify the following pragma in your C source:

#pragma indirect

Example
With the following command the compiler generates far calls for all functions:

ctc --indirect test.c

Related information
C compiler option --indirect-runtime (Call run-time functions indirectly)

Section 1.8.1, “Function Calling Modes: __indirect”
C compiler option: --indirect-runtime

Menu entry
2. Enable the option Call run-time functions indirectly.

Command line syntax

--indirect-runtime

Description

With this option you tell the compiler to generate code for indirect calls to run-time functions. Use this option if you locate the entire run-time library in far memory.

Instead of this option you can also specify the following pragma in your C source:

#pragma indirect_runtime

Example

With the following command the compiler generates far calls for all run-time functions:

ctc --indirect-runtime test.c

Related information

C compiler option --indirect (Call functions indirectly)

Section 1.8.1, “Function Calling Modes: __indirect”
C compiler option: --inline

Menu entry
1. Select C/C++ Compiler » Optimization.
2. Enable the option Always inline function calls.

Command line syntax
--inline

Description
With this option you instruct the compiler to inline calls to functions without the __noinline function qualifier whenever possible. This option has the same effect as a #pragma inline at the start of the source file.

This option can be useful to increase the possibilities for code compaction (C compiler option --optimize=+compact).

Example
To always inline function calls:

c tc --optimize=+compact --inline test.c

Related information
C compiler option --optimize=+compact (Optimization: code compaction)
C compiler option: --inline-max-incr / --inline-max-size

Menu entry
1. Select C/C++ Compiler » Optimization.
2. In the **Maximum increment when inlining** field, enter a value (default 35).
3. In the **Maximum size for functions to inline** field, enter a value (default 10).

Command line syntax

```
--inline-max-incr=percentage  (default: 35)
--inline-max-size=threshold    (default: 10)
```

Description

With these options you can control the automatic function inlining optimization process of the compiler. These options have only effect when you have enabled the inlining optimization (option `--optimize=+inline` or **Optimize most**).

Regardless of the optimization process, the compiler always inlines all functions that have the function qualifier **inline**.

With the option **--inline-max-size** you can specify the maximum size of functions that the compiler inlines as part of the optimization process. The compiler always inlines all functions that are smaller than the specified threshold. The threshold is measured in compiler internal units and the compiler uses this measure to decide which functions are small enough to inline. The default threshold is 10.

After the compiler has inlined all functions that have the function qualifier **inline** and all functions that are smaller than the specified threshold, the compiler looks whether it can inline more functions without increasing the code size too much. With the option **--inline-max-incr** you can specify how much the code size is allowed to increase. By default, this is 35% which means that the compiler continues inlining functions until the resulting code size is 35% larger than the original size.

**Example**

```
ctc --inline-max-incr=40  --inline-max-size=15  test.c
```

The compiler first inlines all functions with the function qualifier **inline** and all functions that are smaller than the specified threshold of 15. If the code size has still not increased with 40%, the compiler decides which other functions it can inline.

Related information

C compiler option **--optimize=+inline** (Optimization: automatic function inlining)

Section 1.8.3, **Inlining Functions: inline**
C compiler option: --integer-enumeration

Menu entry
1. Select C/C++ Compiler » Language.
2. Enable the option Treat enumerated types always as integer.

Command line syntax

--integer-enumeration

Description

Normally the compiler treats enumerated types as the smallest data type possible (char or short instead of int). This reduces code size. With this option the compiler always treats enum-types as int as defined in the ISO C99 standard.

Related information

Section 1.1, Data Types
C compiler option: --iso (-c)

Menu entry

1. Select C/C++ Compiler » Language.
2. From the Comply to C standard list, select ISO C99 or ISO C90.

Command line syntax

--iso=\{90|99\}
-c\{90|99\}

Default: --iso=99

Description

With this option you select the ISO C standard. C90 is also referred to as the "ANSI C standard". C99 refers to the newer ISO/IEC 9899:1999 (E) standard. C99 is the default.

Example

To select the ISO C90 standard on the command line:

ctc --iso=90 test.c

Related information

C compiler option --language (Language extensions)
C compiler option: --keep-output-files (-k)

Menu entry

Eclipse always removes the .src file when errors occur during compilation.

Command line syntax

--keep-output-files

-k

Description

If an error occurs during compilation, the resulting .src file may be incomplete or incorrect. With this option you keep the generated output file (.src) when an error occurs.

By default the compiler removes the generated output file (.src) when an error occurs. This is useful when you use the make utility. If the erroneous files are not removed, the make utility may process corrupt files on a subsequent invocation.

Use this option when you still want to inspect the generated assembly source. Even if it is incomplete or incorrect.

Example

ctc --keep-output-files test.c

When an error occurs during compilation, the generated output file test.src will not be removed.

Related information

C compiler option --warnings-as-errors (Treat warnings as errors)
C compiler option: --language (-A)

Menu entry

1. Select C/C++ Compiler » Language.

2. Enable or disable one or more of the following options:
   - Allow GNU C extensions
   - Allow // comments in ISO C90 mode
   - Check assignment const/non-const string pointer

Command line syntax

--language=[flags]

-A[flags]

You can set the following flags:

+/-gcc     g/G enable a number of gcc extensions
+/-comments p/P // comments in ISO C90 mode
+/-strings x/X relaxed const check for string literals

Default: -AGpx

Default (without flags): -AGPX

Description

With this option you control the language extensions the compiler can accept. By default the TriCore compiler allows all language extensions, except for gcc extensions.

The option --language (-A) without flags disables all language extensions.

GNU C extensions

The --language=+gcc (-Ag) option enables the following gcc language extensions:

- The identifier __FUNCTION__ expands to the current function name.
- Alternative syntax for variadic macros.
- Alternative syntax for designated initializers.
- Allow zero sized arrays.
- Allow empty struct/union.
• Allow empty initializer list.
• Allow initialization of static objects by compound literals.
• The middle operand of a `? :` operator may be omitted.
• Allow a compound statement inside braces as expression.
• Allow arithmetic on void pointers and function pointers.
• Allow a range of values after a single case label.
• Additional preprocessor directive `#warning`.
• Allow comma operator, conditional operator and cast as lvalue.
• An inline function without "static" or "extern" will be global.
• An "extern inline" function will not be compiled on its own.
• An `__attribute__` directly following a struct/union definition relates to that tag instead of to the objects in the declaration.

For a more complete description of these extensions, you can refer to the UNIX gcc info pages (`info gcc`).

Comments in ISO C90 mode

With `--language=+comments (-Ap)` you tell the compiler to allow C++ style comments (`//`) in ISO C90 mode (option `--iso=90`). In ISO C99 mode this style of comments is always accepted.

Check assignment const/non-const string pointer

With `--language=+strings (-Ax)` you disable warnings about discarded `const` qualifiers when a string literal is assigned to a non-constant pointer.

```c
char *p;
void main( void ) { p = "hello"; }
```

Example

```c
ctc --language=-comments,+strings --iso=90 test.c
crc -APx c90 test.c
```

The compiler compiles in ISO C90 mode, accepts assignments of a constant string to a non-constant string pointer and does not allow C++ style comments.

Related information

C compiler option `--iso` (ISO C standard)
C compiler option: --make-target

Menu entry

- 

Command line syntax

--make-target=name

Description

With this option you can overrule the default target name in the make dependencies generated by the options --preprocess=+make (-Em) and --dep-file. The default target name is the basename of the input file, with extension .o.

Example

ctc --preprocess=+make --make-target=mytarget.o test.c

The compiler generates dependency lines with the default target name mytarget.o instead of test.o.

Related information

C compiler option --preprocess=+make (Generate dependencies for make)
C compiler option --dep-file (Generate dependencies in a file)
C compiler option: --max-call-depth

Menu entry

1. Select C/C++ Compiler » Optimization.
2. In the Maximum call depth for code compaction field, enter a value.

Command line syntax

--max-call-depth=value

Default: -1

Description

This option is related to the compiler optimization --optimize=+compact (Code compaction or reverse inlining). Code compaction is the opposite of inlining functions: large sequences of code that occur more than once, are transformed into a function. This reduces code size (possibly at the cost of execution speed).

During code compaction it is possible that the compiler generates nested calls. This may cause the program to run out of its stack. To prevent stack overflow caused by too deeply nested function calls, you can use this option to limit the call depth. This option can have the following values:

-1  Poses no limit to the call depth (default)
0  The compiler will not generate any function calls. (Effectively the same as if you turned off code compaction with option --optimize=-compact)
> 0  Code sequences are only reversed if this will not lead to code at a call depth larger than specified with value. Function calls will be placed at a call depth no larger than value-1. (Note that if you specified a value of 1, the option --optimize=+compact may remain without effect when code sequences for reversing contain function calls.)

This option does not influence the call depth of user written functions.

If you use this option with various C modules, the call depth is valid for each individual module. The call depth after linking may differ, depending on the nature of the modules.

Related information

C compiler option --optimize=+compact (Optimization: code compaction)
C compiler option --compact-max-size (Maximum size of a match for code compaction)
C compiler option: --mil / --mil-split

Menu entry

Command line syntax

--mil
--mil-split[=file,...]

Description

With option --mil the C compiler skips the code generator phase and writes the optimized intermediate representation (MIL) to a file with the suffix .mil. The C compiler accepts .mil files as input files on the command line.

Option --mil-split does the same as option --mil, but in addition, the C compiler splits the MIL representation and writes it to separate files with suffix .ms. One file is written for each input file or MIL library specified on the command line. The .ms files are only updated on a change. The C compiler accepts .ms files as input files on the command line.

With option --mil-split you can perform application-wide optimizations during the frontend phase by specifying all modules at once, and still invoke the backend phase one module at a time to reduce the total compilation time.

Optionally, you can specify another filename for the .ms file the C compiler generates. Without an argument, the basename of the C source file is used to create the .ms filename. Note that if you specify a filename, you have to specify one filename for every input file.

Related information

Section 4.1, Compilation Process

Control program option --mil-link / --mil-split
C compiler option: --misrac

Menu entry

1. Select C/C++ Compiler » MISRA-C.
2. Make a selection from the MISRA-C checking list.
3. If you selected Custom, expand the Custom 2004 or Custom 1998 entry and enable one or more individual rules.

Command line syntax

```bash
--misrac={all | nr[-nr]},...
```

Description

With this option you specify to the compiler which MISRA-C rules must be checked. With the option `--misrac=all` the compiler checks for all supported MISRA-C rules.

Example

```bash
ctc --misrac=9-13 test.c
```

The compiler generates an error for each MISRA-C rule 9, 10, 11, 12 or 13 violation in file `test.c`.

Related information

- Section 4.7, C Code Checking: MISRA-C
- C compiler option `--misrac-advisory-warnings`
- C compiler option `--misrac-required-warnings`
- Linker option `--misrac-report`
C compiler option: --misrac-advisory-warnings / --misrac-required-warnings

Menu entry

1. Select C/C++ Compiler » MISRA-C.
2. Make a selection from the MISRA-C checking list.
3. Enable one or both options Warnings instead of errors for required rules and Warnings instead of errors for advisory rules.

Command line syntax

--misrac-advisory-warnings
--misrac-required-warnings

Description

Normally, if an advisory rule or required rule is violated, the compiler generates an error. As a consequence, no output file is generated. With this option, the compiler generates a warning instead of an error.

Related information

Section 4.7, C Code Checking: MISRA-C
C compiler option --misrac
Linker option --misrac-report
C compiler option: --misrac-version

Menu entry
1. Select C/C++ Compiler » MISRA-C.

Command line syntax

--misrac-version={1998|2004}

Default: 2004

Description


Related information

Section 4.7, C Code Checking: MISRA-C

C compiler option --misrac
Menu entry

1. Expand C/C++ Build and select Processor.
2. From the Processor Selection list, select a processor that has an MMU or select a user defined processor.
3. (Optional) Select C/C++ Compiler » Miscellaneous.
4. (Optional) Add the option --mmu-present to the Additional options field.
5. Select C/C++ Compiler » Code Generation.
6. Enable the option Use the MMU if present.

For predefined TriCore processors, the option --mmu-present is set automatically, so you only need to enable the option Use the MMU if present. For user defined processors you need to enable them both.

Command line syntax

--mmu-present
--mmu-on

Description

If the processor you are using has an MMU which is activated, you can tell the compiler to use the MMU. The compiler then will align data naturally. To instruct the compiler to use the MMU, you must set both the option --mmu-present and the option --mmu-on.

With the option --mmu-present you tell the compiler that an MMU is physically present. With the option --mmu-on you tell the compiler the MMU is activated. In this case the compiler needs to align data naturally. Both options are necessary.

If you select a valid target processor (command line option --cpu (-C)), the option --mmu-present is set automatically, based on the chosen target processor. If you are using a target processor with MMU that is not predefined, you need to set this option manually.

Example

To instruct the compiler to use the activated MMU:

```bash
ctc --mmu-present --mmu-on test.c
```

Related information

C compiler option --fpu-present (FPU present)
C compiler option --cpu (Select processor)
C compiler option: --no-default-section-alignment

Menu entry
1. Select C/C++ Compiler » Miscellaneous.
2. Add the option --no-default-section-alignment to the Additional options field.

Command line syntax
--no-default-section-alignment

Description
By default the TriCore C compiler gives all sections an alignment of 4 bytes. This is required for the initialization algorithm in the startup code (as provided by the compiler). When you use this option, sections are no longer forced to a 4-byte alignment. You will need to use a custom made startup code. Furthermore you will have to remove the copy_unit = 4 part of the copytable declaration within the LSL file.

Related information
Section 1.9.3, Change Section Alignment
C compiler option: --no-double (-F)

Menu entry

1. Select C/C++ Compiler » Language.
2. Enable the option Treat double as float.

Command line syntax

--no-double
--F

Description

With this option you tell the compiler to treat variables of the type double as float. Because the float type takes less space, execution speed increases and code size decreases, both at the cost of less precision.

Example

ctc --no-double test.c

The file test.c is compiled where variables of the type double are treated as float.

Related information

-
C compiler option: --no-stdinc

Menu entry
1. Select C/C++ Compiler » Miscellaneous.
2. Add the option --no-stdinc to the Additional options field.

Command line syntax
--no-stdinc

Description
With this option you tell the compiler not to look in the default include directory relative to the installation directory, when searching for include files. This way the compiler only searches in the include file search paths you specified.

Related information
Section 4.4, How the Compiler Searches Include Files
C compiler option: --no-tasking-sfr

Menu entry
1. Select C/C++ Compiler » Preprocessing.
2. Disable the option Automatic inclusion of '.sfr' file.

Command line syntax

--no-tasking-sfr

Description

Normally, the compiler includes a special function register (SFR) file before compiling. The compiler automatically selects the SFR file belonging to the target you selected on the Processor page (C compiler option --cpu).

With this option the compiler does not include the register file `regcpu.sfr` as based on the selected target processor.

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

Related information

C compiler option --cpu (Select processor)

Section 1.3.2, Accessing Hardware from C
C compiler option: --no-warnings (-w)

Menu entry
1. Select C/C++ Compiler » Diagnostics.
   The Suppress C compiler warnings box shows the warnings that are currently suppressed.
2. To suppress a warning, click on the Add button in the Suppress warnings box.
3. Enter the numbers, separated by commas, of the warnings you want to suppress (for example 537,538). Or you can use the Add button multiple times.
4. To suppress all warnings, enable the option Suppress all warnings.

Use the Edit and Delete button to change a warning number or to remove a number from the list.

Command line syntax

--no-warnings [=number,...]
-w [number,...]

Description
With this option you can suppresses all warning messages or specific warning messages.

On the command line this option works as follows:

- If you do not specify this option, all warnings are reported.
- If you specify this option but without numbers, all warnings are suppressed.
- If you specify this option with a number, only the specified warning is suppressed. You can specify the option --no-warnings=number multiple times.

Example
To suppress warnings 537 and 538, enter:

```
ctc test.c --no-warnings=537,538
```

Related information

- C compiler option --warnings-as-errors (Treat warnings as errors)
- Pragma warning
C compiler option: --object-comment

Menu entry
1. Select C/C++ Compiler » Miscellaneous.
2. Add your comment to the Comment in object file field.

Command line syntax

--object-comment=comment

Description

With this option the compiler generates a .comment section at the end of the assembly file. The section contains the comment specified with this option. After assembling, this text is included in the .o object and .elf files. Place the comment between double quotes.

Instead of this option you can also specify the following pragma in your C source:

#pragma object_comment comment

Example

ctc --object-comment="Created by Altium" test.c

The compiler creates the file test.src with a .comment section at the end of the file. After assembling this file, the text "Created by Altium" is incorporated in the generated object file.

Related information
C compiler option: --optimize (-O)

Menu entry

1. Select **C/C++ Compiler » Optimization**.
2. Select an optimization level in the **Optimization level** box.

Command line syntax

```
--optimize [=flags]
-o flags
```

You can set the following flags:

- `+/-coalesce` a/A Coalescer: remove unnecessary moves
- `+/-cse` c/C Common subexpression elimination
- `+/-expression` e/E Expression simplification
- `+/-flow` f/F Control flow simplification
- `+/-glo` g/G Generic assembly code optimizations
- `+/-inline` i/I Automatic function inlining
- `+/-schedule` k/K Instruction scheduler
- `+/-loop` l/L Loop transformations
- `+/-simd` m/M Perform SIMD optimizations
- `+/-align-loop` n/N Align loop bodies
- `+/-forward` o/O Forward store
- `+/-propagate` p/P Constant propagation
- `+/-compact` r/R Code compaction (reverse inlining)
- `+/-subscript` s/S Subscript strength reduction
- `+/-ifconvert` v/V Convert IF statements using predicates
- `+/-pipeline` w/W Software pipelining
- `+/-peephole` y/Y Peephole optimizations

Use the following options for predefined sets of flags:

```
--optimize=0      -O0  No optimization
                Alias for -OACEFGIKLMNOPRSVWY
```

No optimizations are performed. The compiler tries to achieve an optimal resemblance between source code and produced code. Expressions are evaluated in the same order as written in the source code, associative and commutative properties are not used.
### Tool Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--optimize=1</td>
<td>Enables optimizations that do not affect the debug ability of the source code. Use this level when you encounter problems during debugging your source code with optimization level 2.</td>
</tr>
<tr>
<td>-O1</td>
<td>Alias for -OacefgIklMNopRsvwy</td>
</tr>
<tr>
<td>--optimize=2</td>
<td>Enables more optimizations to reduce code size and/or execution time. This is the default optimization level.</td>
</tr>
<tr>
<td>-O2</td>
<td>Alias for -OacefgIklMNopRsvwy</td>
</tr>
<tr>
<td>--optimize=3</td>
<td>This is the highest optimization level. Use this level to decrease execution time to meet your real-time requirements.</td>
</tr>
<tr>
<td>-O3</td>
<td>Alias for -OacefgIklMNopRsvwy</td>
</tr>
</tbody>
</table>

#### Default:

- `--optimize=2`

## Description

With this option you can control the level of optimization. If you do not use this option, the default optimization level is `Optimize more` (option `--optimize=2` or `--optimize`).

When you use this option to specify a set of optimizations, you can overrule these settings in your C source file with `#pragma optimize` flag/`#pragma endoptimize`.

In addition to the option `--optimize`, you can specify the option `--tradeoff (-t)`. With this option you specify whether the used optimizations should optimize for more speed (regardless of code size) or for smaller code size (regardless of speed).

## Example

The following invocations are equivalent and result all in the default optimization set:

```bash
ctc test.c

ctc --optimize=2 test.c
ctc -O2 test.c

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

ctc -OacefgIklMNopRsvwy test.c
c tc --optimize=+coalesce,+cse,+expression,+flow,+glo,-inline,
+schedule,+loop,-simd,-align-loop,+forward,+propagate,
-compact,+subscript,+ifconvert,+pipeline,+peephole test.c
```
Related information

C compiler option --tradeoff (Trade off between speed and size)

Pragma optimize/endoptimize

Section 4.6, Compiler Optimizations
C compiler option: --option-file (-f)

Menu entry

1. Select C/C++ Compiler » Miscellaneous.
2. Add the option --option-file to the Additional options field.

   Be aware that the options in the option file are added to the C compiler options you have set in the other pages. Only in extraordinary cases you may want to use them in combination.

Command line syntax

--option-file=file,...

-ffile,...

Description

This option is primarily intended for command line use. Instead of typing all options on the command line, you can create an option file which contains all options and flags you want to specify. With this option you specify the option file to the compiler.

Use an option file when the command line would exceed the limits of the operating system, or just to store options and save typing.

You can specify the option --option-file multiple times.

Format of an option file

• Multiple arguments on one line in the option file are allowed.

• To include whitespace in an argument, surround the argument with single or double quotes.

• If you want to use single quotes as part of the argument, surround the argument by double quotes and vise versa:

  "This has a single quote ' embedded"

  'This has a double quote " embedded'

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

• When a text line reaches its length limit, use a \ to continue the line. Whitespace between quotes is preserved.

  "This is a continuation \ line"

  "This is a continuation line"
It is possible to nest command line files up to 25 levels.

Example

Suppose the file myoptions contains the following lines:

```bash
--debug-info
--define=DEMO=1
```  
test.c

Specify the option file to the compiler:

```bash
ctc --option-file=myoptions
```  
This is equivalent to the following command line:

```bash
ctc --debug-info --define=DEMO=1 test.c
```  

Related information
C compiler option: --output (-o)

Menu entry

Eclipse names the output file always after the C source file.

Command line syntax

`--output=file`

`-o file`

Description

With this option you can specify another filename for the output file of the compiler. Without this option the basename of the C source file is used with extension .src.

Example

To create the file `output.src` instead of `test.src`, enter:

`ctc --output=output.src test.c`

Related information

-
C compiler option: --preprocess (-E)

Menu entry

1. Select **C/C++ Compiler » Preprocessing**.
2. Enable the option **Store C preprocessor output in <file>.pre**.

Command line syntax

```
--preprocess [=flags]
-E[flags]
```

You can set the following flags:

```
 +/-comments   c/C keep comments
 +/-make       m/M generate dependencies for make
 +/-noline     p/P strip #line source position information
```

Default: **--ECMP**

Description

With this option you tell the compiler to preprocess the C source. Under Eclipse the compiler sends the preprocessed output to the file **name.pre** (where **name** is the name of the C source file to compile). Eclipse also compiles the C source.

On the command line, the compiler sends the preprocessed file to stdout. To capture the information in a file, specify an output file with the option **--output**.

With **--preprocess=+comments** you tell the preprocessor to keep the comments from the C source file in the preprocessed output.

With **--preprocess=+make** the compiler will generate dependency lines that can be used in a Makefile. The preprocessor output is discarded. The default target name is the basename of the input file, with the extension .o. With the option **--make-target** you can specify a target name which overrules the default target name.

With **--preprocess=+noline** you tell the preprocessor to strip the #line source position information (lines starting with #line). These lines are normally processed by the assembler and not needed in the preprocessed output. When you leave these lines out, the output is easier to read.

Example

```
ctc --preprocess=+comments,-make,-noline test.c --output=test.pre
```

The compiler preprocesses the file **test.c** and sends the output to the file **test.pre**. Comments are included but no dependencies are generated and the line source position information is not stripped from the output file.
Related information

C compiler option --dep-file (Generate dependencies in a file)

C compiler option --make-target (Specify target name for -Em output)
C compiler option: --profile (-p)

Menu entry
1. Select C/C++ Compiler » Debugging.
2. Enable or disable Static profiling.
3. Enable or disable one or more of the following **Generate profiling information** options (dynamic profiling):
   - for block counters (not in combination with Call graph or Function timers)
   - to build a call graph
   - for function counters
   - for function timers

Note that the more detailed information you request, the larger the overhead in terms of execution time, code size and heap space needed. The option **--debug** does not affect profiling, execution time or code size.

Command line syntax

```
--profile [=flag,...]
-p [flags]
```

Use the following option for a predefined set of flags:

```
--profile=g  -pg  Profiling with call graph and function timers.  
             Alias for: -pBcFSt
```

You can set the following flags:

```
+/-block          b/B          block counters
+/-callgraph      c/C          call graph
+/-function       f/F          function counters
+/-static         s/S          static profile generation
+/-time           t/T          function timers
```

Default (without flags): **-pBcFSt**

Description

Profiling is the process of collecting statistical data about a running application. With these data you can analyze which functions are called, how often they are called and what their execution time is.
Several methods of profiling exist. One method is *code instrumentation* which adds code to your application that takes care of the profiling process when the application is executed. Another method is *static profiling*.

For an extensive description of profiling refer to *Chapter 6, Profiling*.

You can obtain the following profiling data (see flags above):

**Block counters (not in combination with Call graph or Time)**

This will instrument the code to perform basic block counting. As the program runs, it counts the number of executions of each branch in an if statement, each iteration of a for loop, and so on. Note that though you can combine Block counters with Function counters, this has no effect because Function counters is only a subset of Block counters.

**Call graph (not in combination with Block counters)**

This will instrument the code to reconstruct the run-time call graph. As the program runs it associates the caller with the gathered profiling data.

**Function counters**

This will instrument the code to perform function call counting. This is a subset of the basic Block counters.

**Time (not in combination with Block counters)**

This will instrument the code to measure the time spent in a function. This includes the time spent in all sub functions (callees).

**Static profiling**

With this option you do not need to run the application to get profiling results. The compiler generates profiling information at compile time, without adding extra code to your application.

If you use one or more profiling options that use code instrumentation, you must link the corresponding libraries too! Refer to *Section 8.3, Linking with Libraries*, for an overview of the (profiling) libraries. In Eclipse the correct libraries are linked automatically.

**Example**

To generate block count information for the module `test.c` during execution, compile as follows:

```
ctc --profile=+block test.c
```

In this case you must link the library `libpb.a`.

**Related information**

*Chapter 6, Profiling*
C compiler option: --rename-sections (-R)

Menu entry
1. Select C/C++ Compiler » Miscellaneous.
2. Add the option --rename-sections to the Additional options field.

Command line syntax

--rename-sections [=name]
-R [name]

Description

The compiler defaults to a section naming convention, using a prefix indicating the section type, the module name and a symbol name:

section_type_prefix.module_name.symbol_name

For example, .text.module_name.symbol_name for code sections.

In case a module must be loaded at a fixed address, or a data section needs a special place in memory, you can use this option to generate different section names (section_type_prefix.name where name replaces the part module_name.symbol_name). You can then use this unique section name in the linker script file for locating.

If you use this option without a value, the compiler uses the default section naming.

Example

To generate the section name section_type_prefix.NEW instead of the default section name section_type_prefix.module_name.symbol_name, enter:

c tc -RNEW test.c

To generate the section name section_type_prefix instead of the default section name section_type_prefix.module_name.symbol_name, enter:

c tc -R" " test.c (note the space between the quotes)

Related information

Section 1.9, Compiler Generated Sections
C compiler option: --runtime (-r)

Menu entry

1. Select C/C++ Compiler » Debugging.
2. Enable or disable one or more of the following run-time error checking options:
   - Generate code for bounds checking
   - Generate code to detect unhandled case in a switch
   - Generate code for malloc consistency checks

Command line syntax

--runtime [=flag, ...]
-r[flags]

You can set the following flags:

+/-bounds  b/B   bounds checking
+/-case    c/C   report unhandled case in a switch
+/-malloc  m/M   malloc consistency checks

Default (without flags): -rbcm

Description

This option controls a number of run-time checks to detect errors during program execution. Some of these checks require additional code to be inserted in the generated code, and may therefore slow down the program execution. The following checks are available:

bounds

Every pointer update and dereference will be checked to detect out-of-bounds accesses, null pointers and uninitialized automatic pointer variables. This check will increase the code size and slow down the program considerably. In addition, some heap memory is allocated to store the bounds information. You may enable bounds checking for individual modules or even parts of modules only (see #pragma runtime).

case

Report an unhandled case value in a switch without a default part. This check will add one function call to every switch without a default part, but it will have little impact on the execution speed.

malloc

This option enables the use of wrappers around the functions malloc/realloc/free that will check for common dynamic memory allocation errors like:
• buffer overflow
• write to freed memory
• multiple calls to free
• passing invalid pointer to free

Enabling this check will extract some additional code from the library, but it will not enlarge your application code. The dynamic memory usage will increase by a couple of bytes per allocation.

**Related information**

Pragma runtime
**C compiler option: --section-name-with-symbol**

**Menu entry**

1. Select **C/C++ Compiler » Miscellaneous**.
2. Add the option **--section-name-with-symbol** to the **Additional options** field.

**Command line syntax**

```plaintext
--section-name-with-symbol
```

**Description**

Normally, when you use an option or pragma to influence section naming, the symbol name is not included.

With this option you tell the compiler to include the symbol name in the section name.

Instead of this option you can use the following pragma:

```plaintext
#pragma section_name_with_symbol
```

**Related information**

Section 1.9.1, *Rename Sections*
C compiler option: --section-per-data-object

Menu entry
2. Enable the option Generate a section for each data object.

Command line syntax
--section-per-data-object

Description
Normally the compiler generates one section for each data type in a module (such as .data, .rodata, .bss, .zdata, ...).

With this option you force the compiler to generate a separate section for each data object. This provides more control about allocation during the linking process.

Instead of this option you can use the following pragma:

#pragma section_per_data_object

Related information
Section 1.9.2, Influence Section Definition
C compiler option: --silicon-bug

Menu entry

1. Expand C/C++ Build and select Processor.
2. From the Processor Selection list, select a processor.

   The CPU Problem Bypasses and Checks box shows the available workarounds/checks available for the selected processor.

3. (Optional) Select Show all CPU problem bypasses and checks.
4. Click Select All or select one or more individual options.

Command line syntax

--silicon-bug=arg,…

You can give one or more of the following arguments:

- all-tc11ib All TriCore TC11IB workarounds
- all-tc1100 All TriCore TC1100 workarounds
- all-tc1115 All TriCore TC1115 workarounds
- all-tc1130 All TriCore TC1130 workarounds
- all-tc1161 All TriCore TC1161 workarounds
- all-tc1162 All TriCore TC1162 workarounds
- all-tc1163 All TriCore TC1163 workarounds
- all-tc1164 All TriCore TC1164 workarounds
- all-tc1165 All TriCore TC1165 workarounds
- all-tc1166 All TriCore TC1166 workarounds
- all-tc1762 All TriCore TC1762 workarounds
- all-tc1764 All TriCore TC1764 workarounds
- all-tc1765 All TriCore TC1765 workarounds
- all-tc1766 All TriCore TC1766 workarounds
- all-tc1775 All TriCore TC1775 workarounds
- all-tc1792 All TriCore TC1792 workarounds
- all-tc1796 All TriCore TC1796 workarounds
- all-tc1910 All TriCore TC1910 workarounds
- all-tc1912 All TriCore TC1912 workarounds
- all-tc1920 All TriCore TC1920 workarounds

- cpu-tc013 Workaround for CPU_TC.013
**Description**

With this option you specify for which hardware problems the compiler should generate workarounds. Please refer to Chapter 16, *CPU Problem Bypasses and Checks* for more information about the individual problems and workarounds.

Instead of this option you can use the following pragmas:

```c
#pragma CPU_functional_problem
#pragma DMU_functional_problem
```

**Example**

To enable workarounds for problems CPU_TC.024 and CPU_TC.030, enter:

```bash
ctc --silicon-bug=cpu-tc024,cpu-tc030 test.c
```
Related information

Chapter 16, *CPU Problem Bypasses and Checks*

Assembler option **--silicon-bug**
C compiler option: --source (-s)

Menu entry
1. Select C/C++ Compiler » Miscellaneous.
2. Enable the option Merge C source code with generated assembly.

Command line syntax
--source
-s

Description
With this option you tell the compiler to merge C source code with generated assembly code in the output file. The C source lines are included as comments.

Related information
Pragmas source/nosource
C compiler option: --static

Menu entry

Command line syntax

--static

Description

With this option, the compiler treats external definitions at file scope (except for main) as if they were declared static. As a result, unused functions will be eliminated, and the alias checking algorithm assumes that objects with static storage cannot be referenced from functions outside the current module.

This option only makes sense when you specify all modules of an application on the command line.

Example

ctc --static module1.c module2.c module3.c ...

Related information

-
C compiler option: --stdout (-n)

Menu entry

Command line syntax

--stdout

-n

Description

With this option you tell the compiler to send the output to stdout (usually your screen). No files are created. This option is for example useful to quickly inspect the output or to redirect the output to other tools.

Related information

-
C compiler option: --switch

Menu entry

2. Select an Algorithm for switch statements.

Command line syntax

--switch==arg

You can give one of the following arguments:

- **auto**: Choose most optimal code
- **jumptab**: Generate jump tables
- **linear**: Use linear jump chain code
- **lookup**: Generate lookup tables

Default: --switch=auto

Description

With this option you tell the compiler which code must be generated for a switch statement: a jump chain (linear switch), a jump table or a lookup table. By default, the compiler will automatically choose the most efficient switch implementation based on code and data size and execution speed.

Instead of this option you can use the following pragma:

```c
#pragma switch arg
```

Example

To use a table filled with target addresses for each possible switch value, enter:

```c
ctc --switch=jumptab test.c
```

Related information

Section 1.7, Switch Statement
C compiler option: --tradeoff (-t)

Menu entry

1. Select C/C++ Compiler » Optimization.
2. Select a trade-off level in the Trade-off between speed and size box.

Command line syntax

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

Default: --tradeoff=2

Description

If the compiler uses certain optimizations (option --optimize), you can use this option to specify whether the used optimizations should optimize for more speed (regardless of code size) or for smaller code size (regardless of speed).

By default the compiler balances speed and size while optimizing (--tradeoff=2).

If you have not specified the option --optimize, the compiler uses the default Optimize more optimization. In this case it is still useful to specify a trade-off level.

Example

To set the trade-off level for the used optimizations:

ctc --tradeoff=2 test.c

The compiler uses the default Optimize more optimization level and optimizes for code size rather than for speed.

Related information

C compiler option --optimize (Specify optimization level)

Section 4.6.3, Optimize for Size or Speed
C compiler option: --uchar (-u)

Menu entry
1. Select C/C++ Compiler » Language.
2. Enable the option Treat "char" variables as unsigned.

Command line syntax

--uchar

-u

Description

By default char is the same as specifying signed char. With this option char is the same as unsigned char.

Related information

Section 1.1, Data Types
C compiler option: --undefine (-U)

Menu entry

1. Select C/C++ Compiler » Preprocessing

   The Defined symbols box shows the symbols that are currently defined.

2. To remove a defined symbol, select the symbol in the Defined symbols box and click on the Delete button.

Command line syntax

```
--undefine=macro_name

-U macro_name
```

Description

With this option you can undefine an earlier defined macro as with #undef. This option is for example useful to undefine predefined macros.

The following predefined ISO C standard macros cannot be undefined:

- `__FILE__` current source filename
- `__LINE__` current source line number (int type)
- `__TIME__` hh:mm:ss
- `__DATE__` Mmm dd yyyy
- `__STDC__` level of ANSI standard

Example

To undefine the predefined macro `__TASKING__`:

```
ctc --undefine=__TASKING__ test.c
```

Related information

C compiler option --define (Define preprocessor macro)

Section 1.6, Predefined Preprocessor Macros
C compiler option: --user-mode

Menu entry
1. Select C/C++ Compiler » Miscellaneous.
2. Add the option --user-mode to the Additional options field.

Command line syntax

--user-mode=mode

You can specify the following mode arguments:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>user-0</td>
<td>User-0 unprivileged mode</td>
</tr>
<tr>
<td>user-1</td>
<td>User-1 privileged mode</td>
</tr>
<tr>
<td>kernel</td>
<td>Kernel/Supervisor mode</td>
</tr>
</tbody>
</table>

Default: --user-mode=kernel

Description

With this option you specify the mode (I/O privilege mode) the TriCore runs in: User-0, User-1 or Kernel/Supervisor. The availability of some instructions depends on this mode. Most instructions run in all modes. The instructions enable and disable run in User-1 or kernel mode only. The instructions bisr, mtcrr, cachea.i and tlb instructions run in kernel mode only.

Instead of this option you can use the following pragma:

#pragma user_mode mode

Example

To restrict the instructions in the assembly code to User-1 mode, enter:

ctc --user-mode=user-1 test.c

Related information

Pragma user_mode
Assembler option --user-mode (Select user mode)
C compiler option: --verbose (-v)

Menu entry

Command line syntax

--verbose

-v

Description

With this option the C compiler can generate additional informational diagnostics when available. For example, when code is generated to circumvent functional problems in the processor.

Related information

-
C compiler option: --version (-V)

Menu entry
-

Command line syntax

--version
-v

Description

Display version information. The compiler ignores all other options or input files.

Example

ctc --version

The compiler does not compile any files but displays the following version information:

TASKING VX-toolset for TriCore: C compiler vx.yrz Build nnn
Copyright 2002-year Altium BV Serial# 00000000

Related information
-

C compiler option: --warnings-as-errors

Menu entry
1. Select Global Options.
2. Enable the option Treat warnings as errors.

Command line syntax

--warnings-as-errors [=number, ...]

Description
If the compiler encounters an error, it stops compiling. When you use this option without arguments, you tell the compiler to treat all warnings as errors. This means that the exit status of the compiler will be non-zero after one or more compiler warnings. As a consequence, the compiler now also stops after encountering a warning.

You can also limit this option to specific warnings by specifying a comma-separated list of warning numbers.

Related information

C compiler option --no-warnings (Suppress some or all warnings)
11.2. C++ Compiler Options

This section lists all C++ compiler options.

Options in Eclipse versus options on the command line

Most command line options have an equivalent option in Eclipse but some options are only available on the command line. Eclipse invokes the C++ compiler via the control program. Therefore, it uses the syntax of the control program to pass options and files to the C++ compiler. If there is no equivalent option in Eclipse, you can specify a command line option in Eclipse as follows:

1. From the Project menu, select Properties
   
   The Properties dialog appears.

2. In the left pane, expand C/C++ Build and select Settings.
   
   In the right pane the Settings appear.

3. On the Tool Settings tab, select C/C++ Compiler » Miscellaneous.

4. In the Additional options field, enter one or more command line options.
   
   Because Eclipse uses the control program, you have to precede the option with -Wcp to pass the option via the control program directly to the C++ compiler.

Short and long option names

Options can have both short and long names. Short option names always begin with a single minus (-) character, long option names always begin with two minus (--) characters. You can abbreviate long option names as long as it forms a unique name. You can mix short and long option names on the command line.

If an option requires an argument, the argument may be separated from the keyword by white space, or the keyword may be immediately followed by =option. When the second form is used there may not be any white space on either side of the equal sign.

Options can have flags or suboptions. To switch a flag 'on', use a lowercase letter or a +longflag. To switch a flag off, use an uppercase letter or a -longflag. Separate longflags with commas. The following two invocations are equivalent:

```
  cptc -Ecp test.cc
  cptc --preprocess=+comments,+noline test.cc
```

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

The priority of the options is left-to-right: when two options conflict, the first (most left) one takes effect. The -D and -U options are not considered conflicting options, so they are processed left-to-right for each source file. You can overrule the default output file name with the --output-file option.
C++ compiler option: --alternative-tokens

Menu entry

Command line syntax

--alternative-tokens

Description

Enable recognition of alternative tokens. This controls recognition of the digraph tokens in C++, and controls recognition of the operator keywords (e.g., not, and, bitand, etc.).

Example

To enable operator keywords (e.g., "not", "and") and digraphs, enter:

```cpp
cptc --alternative-tokens test.cc
```

Related information

-
C++ compiler option: --anachronisms

Menu entry
1. Select C/C++ Compiler » Language.
2. Enable the option C++ anachronisms.

Command line syntax
--anachronisms

Description
Enable C++ anachronisms. This option also enables --nonconst-ref-anachronism. But you can turn this off individually with option --no-nonconst-ref-anachronism.

Related information
C++ compiler option --nonconst-ref-anachronism (Nonconst reference anachronism)

Section 2.2.3, Anachronisms Accepted
C++ compiler option: --base-assign-op-is-default

Menu entry

Command line syntax

--base-assign-op-is-default

Description

Enable the anachronism of accepting a copy assignment operator that has an input parameter that is a reference to a base class as a default copy assignment operator for the derived class.

Related information

-
C++ compiler option: --building-runtime

Menu entry

Command line syntax

--building-runtime

Description

Special option for building the C++ run-time library. Used to indicate that the C++ run-time library is being compiled. This causes additional macros to be predefined that are used to pass configuration information from the C++ compiler to the run-time.

Related information

-
C++ compiler option: --c++0x

Menu entry
-

Command line syntax
--c++0x

Description
Enable the C++ extensions that are defined by the latest C++ working paper.

Related information
-
C++ compiler option: --check

Menu entry
-

Command line syntax
--check

Description
With this option you can check the source code for syntax errors, without generating code. This saves time in developing your application because the code will not actually be compiled.

The C++ compiler reports any warnings and/or errors.

This option is available on the command line only.

Related information
C compiler option --check (Check syntax)
Assembler option --check (Check syntax)
C++ compiler option: --context-limit

Menu entry

Command line syntax

--context-limit=number

Default: --context-limit=10

Description

Set the context limit to number. The context limit is the maximum number of template instantiation context entries to be displayed as part of a diagnostic message. If the number of context entries exceeds the limit, the first and last $N$ context entries are displayed, where $N$ is half of the context limit. A value of zero is used to indicate that there is no limit.

Example

To set the context limit to 5, enter:

cptc --context-limit=5 test.cc

Related information

-
**C++ compiler option: --cpu (-C)**

**Menu entry**

1. Expand C/C++ Build and select Processor.
2. From the Processor Selection list, select a processor.

**Command line syntax**

```plaintext
--cpu=cpu
-Ccpu
```

**Description**

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

Based on this option the C++ compiler always includes the special function register file `regcpu.sfr`, unless you disable the option **Automatic inclusion of '.sfr' file** on the Preprocessing page (option --no-tasking-sfr).

The macro `__CPU__` is set to the name of the `cpu`.

**Example**

To compile the file `test.cc` for the TC1165 processor and use the SFR file `regtc1165.sfr`:

```plaintext
cptc --cpu=tc1165 test.cc
```

**Related information**

C++ compiler option **--no-tasking-sfr** (Do not include SFR file)
C++ compiler option: --create-pch

Menu entry

1. Select C/C++ Compiler » Precompiled C++ Headers.
2. Enter a filename in the Create precompiled header file field.

Command line syntax

--create-pch=filename

Description

If other conditions are satisfied, create a precompiled header file with the specified name. If --pch (automatic PCH mode) or --use-pch appears on the command line following this option, its effect is erased.

Example

To create a precompiled header file with the name test.pch, enter:

cptc --create-pch=test.pch test.cc

Related information

C++ compiler option --pch (Automatic PCH mode)
C++ compiler option --use-pch (Use precompiled header file)

Section 2.9, Precompiled Headers
**C++ compiler option: --define (-D)**

**Menu entry**

1. Select **C/C++ Compiler » Preprocessing**.
   
   *The Defined symbols box shows the symbols that are currently defined.*

2. To define a new symbol, click on the **Add** button in the **Defined symbols** box.

3. Type the symbol definition (for example, `demo=1`)

   Use the **Edit** and **Delete** button to change a macro definition or to remove a macro from the list.

**Command line syntax**

```
--define=macro_name[(parm-list)][=macro_definition]

-Dmacro_name(parm-list)][=macro_definition]
```

**Description**

With this option you can define a macro and specify it to the preprocessor. If you only specify a macro name (no macro definition), the macro expands as '1'.

Function-style macros can be defined by appending a macro parameter list to `macro_name`.

You can specify as many macros as you like. Simply use the **Add** button to add new macro definitions.

On the command line, you can use the option **--define (-D)** multiple times. If the command line exceeds the limit of the operating system, you can define the macros in an option file which you then must specify to the C++ compiler with the option **--option-file (-f) file**.

The definition can be tested by the preprocessor with **#if, #ifdef and #ifndef**, for conditional compilations.

**Example**

Consider the following program with conditional code to compile a demo program and a real program:

```c
void main( void )
{
    #if DEMO
       demo_func();   /* compile for the demo program */
    #else
       real_func();   /* compile for the real program */
    #endif
}
```

You can now use a macro definition to set the DEMO flag:
cptc --define=DEMO test.cc
cptc --define=DEMO=1 test.cc

Note that both invocations have the same effect.

The next example shows how to define a macro with arguments. Note that the macro name and definition are placed between double quotes because otherwise the spaces would indicate a new option.

cptc --define="MAX(A,B)=((A) > (B) ? (A) : (B))" test.cc

Related information

C++ compiler option --undefine (Remove preprocessor macro)

C++ compiler option --option-file (Specify an option file)
**C++ compiler option: --definition-list-file**

**Menu entry**

- 

**Command line syntax**

`--definition-list-file=file`

**Description**

This option specifies the name of the template definition list file passed between the C++ compiler and the prelinker. This file is a temporary file and does not remain after the compilation is complete.

This option is supplied for use by the control program that invokes the C++ compiler and is not intended to be used by end-users.

**Related information**

-
**C++ compiler option: --dollar**

**Menu entry**

```
--dollar
```

**Command line syntax**

```
--dollar
```

Default format: No dollar signs are allowed in identifiers.

**Description**

Accept dollar signs in identifiers. Names like A$VAR are allowed.

**Related information**

-
C++ compiler option: --embedded-c++

Menu entry
1. Select C/C++ Compiler » Language.
2. Enable the option Check for embedded C++ compliance.

Command line syntax
--embedded-c++

Description
Enable the diagnostics of non-compliance with the "Embedded C++" subset (from which templates, exceptions, namespaces, new-style casts, RTTI, multiple inheritance, virtual base classes, and mutable are excluded).

Related information
-
C++ compiler option: --error-file

Menu entry
-

Command line syntax

--error-file=file

Description
With this option the C++ compiler redirects error messages to a file. If you do not specify a filename, the error file will be named after the input file with extension .ecp.

Example
To write errors to errors.ecp instead of stderr, enter:

cptc --error-file=errors.ecp test.cc

Related information
-
C++ compiler option: --error-limit (-e)

Menu entry
-

Command line syntax

--error-limit=number
-e number

Default: --error-limit=100

Description

Set the error limit to *number*. The C++ compiler will abandon compilation after this number of errors (remarks and warnings are not counted). By default, the limit is 100.

Example

When you want compilation to stop when 10 errors occurred, enter:

cptc --error-limit=10 test.cc

Related information
-

C++ compiler option: --exceptions (-x)

Menu entry

1. Select C/C++ Compiler » Language.
2. Enable the option Support for exception handling.

Command line syntax

--exceptions

-x

Description

With this option you enable support for exception handling in the C++ compiler.

The macro __EXCEPTIONS is defined when exception handling support is enabled.

Related information

-
C++ compiler option: --exported-template-file

Menu entry

Command line syntax

--exported-template-file=file

Description

This option specifies the name to be used for the exported template file used for processing of exported templates.

This option is supplied for use by the control program that invokes the C++ compiler and is not intended to be used by end-users.

Related information

-
C++ compiler option: --extended-variadic-macros

Menu entry

Command line syntax

--extended-variadic-macros

Default: macros with a variable number of arguments are not allowed.

Description

Allow macros with a variable number of arguments (implies --variadic-macros) and allow the naming of the variable argument list.

Related information

C++ compiler option --variadic-macros (Allow variadic macros)
C++ compiler option: --force-vtbl

Menu entry
1. Select C/C++ Compiler » Miscellaneous.
2. Enable the option **Force definition of virtual function tables**.

Command line syntax

```
--force-vtbl
```

**Description**

Force definition of virtual function tables in cases where the heuristic used by the C++ compiler to decide on definition of virtual function tables provides no guidance.

**Related information**

C++ compiler option **--suppress-vtbl** (Suppress definition of virtual function tables)
C++ compiler option: --friend-injection

Menu entry

Command line syntax

--friend-injection

Default: friend names are not injected.

Description

Controls whether the name of a class or function that is declared only in friend declarations is visible when using the normal lookup mechanisms. When friend names are injected, they are visible to such lookups. When friend names are not injected (as required by the standard), function names are visible only when using argument-dependent lookup, and class names are never visible.

Related information

C++ compiler option --no-arg-dep-lookup (Disable argument dependent lookup)
C++ compiler option: --g++

Menu entry
1. Select C/C++ Compiler » Language.
2. Enable the option Enable GNU C++ extensions.

Command line syntax
--g++

Description
Enable GNU C++ compiler language extensions.

Related information
Section 2.3, GNU Extensions
C++ compiler option: --gnu-version

Menu entry

Command line syntax

--gnu-version=version

Default: 30300 (version 3.3.0)

Description

It depends on the GNU C++ compiler version if a particular GNU extension is supported or not. With this option you set the GNU C++ compiler version that should be emulated in GNU C++ mode. Version x.y.z of the GNU C++ compiler is represented by the value x*10000+y*100+z.

Example

To specify version 3.4.1 of the GNU C++ compiler, enter:

cptc --g++ --gnu-version=30401 test.cc

Related information

Section 2.3, GNU Extensions
C++ compiler option: --guiding-decls

Menu entry

Command line syntax

--guiding-decls

Description

Enable recognition of "guiding declarations" of template functions. A guiding declaration is a function declaration that matches an instance of a function template but has no explicit definition (since its definition derives from the function template). For example:

```cpp
template <class T> void f(T) { ... }
void f(int);
```

When regarded as a guiding declaration, \( f(int) \) is an instance of the template; otherwise, it is an independent function for which a definition must be supplied.

Related information

C++ compiler option --old-specializations (Old-style template specializations)
C++ compiler option: --help (-?)

Menu entry

Command line syntax

--help [=item]

-?

You can specify the following arguments:

   options       Show extended option descriptions

Description

Displays an overview of all command line options. When you specify an argument you can list extended information such as a list of option descriptions.

Example

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

cptc -?
cptc --help
cptc

cptc

The following invocation displays an extended list of the available options:

cptc --help=options

Related information

-
C++ compiler option: --implicit-extern-c-type-conversion

Menu entry

Command line syntax

--implicit-extern-c-type-conversion

Description

Enable the implicit type conversion between pointers to extern "C" and extern "C++" function types.

Related information

-
C++ compiler option: --implicit-include (-B)

Menu entry
1. Select C/C++ Compiler » Miscellaneous.
2. Enable the option Implicit inclusion of source files for finding templates.

Command line syntax

--implicit-include

-B

Description
Enable implicit inclusion of source files as a method of finding definitions of template entities to be instantiated.

Related information

C++ compiler option --instantiate (Instantiation mode)

Section 2.5, Template Instantiation
C++ compiler option: --incl-suffixes

Menu entry
-

Command line syntax

```
--incl-suffixes=suffixes
```

Default: no extension and .stdh.

Description

Specifies the list of suffixes to be used when searching for an include file whose name was specified without a suffix. If a null suffix is to be allowed, it must be included in the suffix list. `suffixes` is a colon-separated list of suffixes (e.g., `::stdh`).

Example

To allow only the suffixes .h and .stdh as include file extensions, enter:

```
cptc --incl-suffixes=h:stdh test.cc
```

Related information

C++ compiler option --include-file (Include file at the start of a compilation)

Section 5.3, How the C++ Compiler Searches Include Files
C++ compiler option: --include-directory (-I)

Menu entry

1. Select C/C++ Compiler » Include Paths.

   The Include paths box shows the directories that are added to the search path for include files.

2. To define a new directory for the search path, click on the Add button in the Include paths box.

3. Type or select a path.

   Use the Edit and Delete button to change a path or to remove a path from the list.

Command line syntax

--include-directory=path,...

-I path,...

Description

Add path to the list of directories searched for #include files whose names do not have an absolute pathname. You can specify multiple directories separated by commas.

Example

To add the directory /proj/include to the include file search path, enter:

cptc --include-directory=/proj/include test.cc

Related information

C++ compiler option --include-file (Include file at the start of a compilation)

C++ compiler option --sys-include (Add directory to system include file search path)

Section 5.3, How the C++ Compiler Searches Include Files
**C++ compiler option: --include-file (-H)**

**Menu entry**

1. Select **C/C++ Compiler » Preprocessing**.

   *The Pre-include files box shows the files that are currently included before the compilation starts.*

2. To define a new file, click on the **Add** button in the **Include files at start of compilation** box.

3. Type the full path and file name or select a file.

   Use the **Edit** and **Delete** button to change a file name or to remove a file from the list.

**Command line syntax**

```
--include-file=file

-H file
```

**Description**

Include the source code of the indicated file at the beginning of the compilation. This is the same as specifying `#include "file"` at the beginning of each of your C++ sources.

All files included with `--include-file` are processed after any of the files included with `--include-macros-file`.

The filename is searched for in the directories on the include search list.

**Example**

```
cptc --include-file=extra.h test1.cc test2.cc
```

The file `extra.h` is included at the beginning of both `test1.cc` and `test2.cc`.

**Related information**

* C++ compiler option **--include-directory** (Add directory to include file search path)

* Section 5.3, *How the C++ Compiler Searches Include Files*
C++ compiler option: --include-macros-file

Menu entry

Command line syntax

--include-macros-file=file

Description

Include the macros of the indicated file at the beginning of the compilation. Only the preprocessing directives from the file are evaluated. All of the actual code is discarded. The effect of this option is that any macro definitions from the specified file will be in effect when the primary source file is compiled. All of the macro-only files are processed before any of the normal includes (--include-file). Within each group, the files are processed in the order in which they were specified.

Related information

C++ compiler option --include-file (Include file at the start of a compilation)

Section 5.3, How the C++ Compiler Searches Include Files
C++ compiler option: --init-priority

Menu entry

Command line syntax

--init-priority=number

Default: 0

Description

Normally, the C++ compiler assigns no priority to the global initialization functions and the exact order is determined by the linker. This option sets the default priority for global initialization functions. Default value is "0". You can also set the default priority with the #pragma init_priority.

Values from 1 to 100 are for internal use only and should not be used. Values 101 to 65535 are available for user code. A lower number means a higher priority.

Example

cptc --init-priority=101 test.cc

Related information

-
C++ compiler option: --instantiate (-t)

Menu entry
1. Select C/C++ Compiler » Miscellaneous.
2. Select an instantiation mode in the Instantiation mode of external template entities box.

Command line syntax

--instantiate=mode
-tmode

You can specify the following modes:

- none
- used
- all
- local

Default: --instantiate=used

Description

Control instantiation of external template entities. External template entities are external (that is, non-inline and non-static) template functions and template static data members. The instantiation mode determines the template entities for which code should be generated based on the template definition. Normally, when a file is compiled, template entities are instantiated wherever they are used (the linker will discard duplicate definitions). The overall instantiation mode can, however, be changed with this option. You can specify the following modes:

- none  Do not automatically create instantiations of any template entities.
- used  Instantiate those template entities that were used in the compilation. This will include all static data members for which there are template definitions. This is the default.
- all  Instantiate all template entities declared or referenced in the compilation unit. For each fully instantiated template class, all of its member functions and static data members will be instantiated whether or not they were used. Non-member template functions will be instantiated even if the only reference was a declaration.
- local  Similar to --instantiate=used except that the functions are given internal linkage. This is intended to provide a very simple mechanism for those getting started with templates. The compiler will instantiate the functions that are used in each compilation unit as local functions, and the program will link and run correctly (barring problems due to multiple copies of local static variables). However, one may end up with many copies of the instantiated functions, so this is not suitable for production use.

You cannot use --instantiate=local in conjunction with automatic template instantiation.
Related information

C++ compiler option --no-auto-instantiation (Disable automatic C++ instantiation)

Section 2.5, Template Instantiation
C++ compiler option: --integer-enumeration

Menu entry

1. Select C/C++ Compiler » Language.
2. Enable the option Treat enumerated types always as integer.

Command line syntax

--integer-enumeration

Description

Normally the C++ compiler treats enumerated types as the smallest data type possible (char or short instead of int). This reduces code size. With this option the C++ compiler always treats enum-types as int as defined in the ISO C99 standard.

Related information

Section 1.1, Data Types
C++ compiler option: --io-streams

Menu entry

1. Select C/C++ Compiler » Language.
2. Enable the option Support for I/O streams.

Command line syntax

--io-streams

Description

As I/O streams require substantial resources they are disabled by default. Use this option to enable I/O streams support in the C++ library.

This option also enables exception handling.

Related information

-
C++ compiler option: --late-tiebreaker

Menu entry

Command line syntax

--late-tiebreaker

Default: early tiebreaker processing.

Description

Select the way that tie-breakers (e.g., cv-qualifier differences) apply in overload resolution. In "early" tie-breaker processing, the tie-breakers are considered at the same time as other measures of the goodness of the match of an argument value and the corresponding parameter type (this is the standard approach).

In "late" tie-breaker processing, tie-breakers are ignored during the initial comparison, and considered only if two functions are otherwise equally good on all arguments; the tie-breakers can then be used to choose one function over another.

Related information
C++ compiler option: --list-file (-L)

Menu entry
-

Command line syntax

--list-file=file
-LLfile

Default: -1

Description

Generate raw listing information in the file. This information is likely to be used to generate a formatted listing. The raw listing file contains raw source lines, information on transitions into and out of include files, and diagnostics generated by the C++ compiler.

Each line of the listing file begins with a key character that identifies the type of line, as follows:

- **N**: A normal line of source; the rest of the line is the text of the line.
- **X**: The expanded form of a normal line of source; the rest of the line is the text of the line. This line appears following the N line, and only if the line contains non-trivial modifications (comments are considered trivial modifications; macro expansions, line splices, and trigraphs are considered non-trivial modifications). Comments are replaced by a single space in the expanded-form line.
- **S**: A line of source skipped by an #if or the like; the rest of the line is text. Note that the #else, #elif, or #endif that ends a skip is marked with an N.
- **L**: An indication of a change in source position. The line has a format similar to the # line-identifying directive output by the C preprocessor, that is to say

\[
\text{L } \text{line_number} \text{ "file-name" [key]}
\]

where key is, 1 for entry into an include file, or 2 for exit from an include file, and omitted otherwise.

The first line in the raw listing file is always an L line identifying the primary input file. L lines are also output for #line directives (key is omitted). L lines indicate the source position of the following source line in the raw listing file.
R, W, E, or C An indication of a diagnostic (R for remark, W for warning, E for error, and C for catastrophic error). The line has the form:

\[ S \ "file-name" \ line_number \ column-number \ message-text \]

where S is R, W, E, or C, as explained above. Errors at the end of file indicate the last line of the primary source file and a column number of zero. Command line errors are catastrophes with an empty file name ("") and a line and column number of zero. Internal errors are catastrophes with position information as usual, and message-text beginning with (internal error). When a diagnostic displays a list (e.g., all the contending routines when there is ambiguity on an overloaded call), the initial diagnostic line is followed by one or more lines with the same overall format (code letter, file name, line number, column number, and message text), but in which the code letter is the lower case version of the code letter in the initial line. The source position in such lines is the same as that in the corresponding initial line.

Example

To write raw listing information to the file test.lst, enter:

cptc --list-file=test.lst test.cc

Related information
C++ compiler option: --long-lifetime-temps

Menu entry
-

Command line syntax

--long-lifetime-temps

Description

Select the lifetime for temporaries: short means to end of full expression; long means to the earliest of end of scope, end of switch clause, or the next label. Short is the default.

Related information
-

C++ compiler option: --long-long

Menu entry

Command line syntax

--long-long

Description

Permit the use of long long in strict mode in dialects in which it is non-standard.

Related information

-
C++ compiler option: --multibyte-chars

Menu entry

Command line syntax

--multibyte-chars

Default: multibyte character sequences are not allowed.

Description

Enable processing for multibyte character sequences in comments, string literals, and character constants. Multibyte encodings are used for character sets like the Japanese SJIS.

Related information
C++ compiler option: --namespaces

Menu entry
-

Command line syntax
--namespaces
--no-namespaces

Default: namespaces are supported.

Description
When you used option --embedded-c++ namespaces are disabled. With option --namespaces you can enable support for namespaces in this case.

The macro __NAMESPACES is defined when namespace support is enabled.

Related information
C++ compiler option --embedded-c++ (Embedded C++ compliancy tests)
C++ compiler option --using-std (Implicit use of the std namespace)

Section 2.4, Namespace Support
C++ compiler option: --no-arg-dep-lookup

Menu entry

Command line syntax

--no-arg-dep-lookup

Default: argument dependent lookup of unqualified function names is performed.

Description

With this option you disable argument dependent lookup of unqualified function names.

Related information

-
C++ compiler option: --no-array-new-and-delete

Menu entry

Command line syntax

--no-array-new-and-delete

Default: array new and delete are supported.

Description

Disable support for array new and delete.

The macro __ARRAY_OPERATORS is defined when array new and delete is enabled.

Related information
**C++ compiler option: --no-auto-instantiation**

**Menu entry**

- 

**Command line syntax**

```
--no-auto-instantiation
```

Default: the C++ compiler automatically instantiates templates.

**Description**

With this option automatic instantiation of templates is disabled.

**Related information**

C++ compiler option `--instantiate` (Set instantiation mode)

Section 2.5, *Template Instantiation*
C++ compiler option: --no-bool

Menu entry
-

Command line syntax

--no-bool

Default: bool is recognized as a keyword.

Description

Disable recognition of the bool keyword.

The macro _BOOL is defined when bool is recognized as a keyword.

Related information
-

C++ compiler option: --no-class-name-injection

Menu entry

- 

Command line syntax

--no-class-name-injection

Default: the name of a class is injected into the scope of the class (as required by the standard).

Description

Do not inject the name of a class into the scope of the class (as was true in earlier versions of the C++ language).

Related information

-
C++ compiler option: --no-const-string-literals

Menu entry

Command line syntax

--no-const-string-literals

Default: C++ string literals and wide string literals are const (as required by the standard).

Description

With this option C++ string literals and wide string literals are non-const (as was true in earlier versions of the C++ language).

Related information
C++ compiler option: --no-dep-name

Menu entry
-

Command line syntax
--no-dep-name

Default: dependent name processing is enabled.

Description
Disable dependent name processing; i.e., the special lookup of names used in templates as required by the C++ standard. This option implies the use of --no-parse-templates.

Related information
C++ compiler option --no-parse-templates (Disable parsing of nonclass templates)
C++ compiler option: --no-distinct-template-signatures

Menu entry

Command line syntax

--no-distinct-template-signatures

Description

Control whether the signatures for template functions can match those for non-template functions when the functions appear in different compilation units. By default a normal function cannot be used to satisfy the need for a template instance; e.g., a function "void f(int)" could not be used to satisfy the need for an instantiation of a template "void f(T)" with T set to int.

--no-distinct-template-signatures provides the older language behavior, under which a non-template function can match a template function. Also controls whether function templates may have template parameters that are not used in the function signature of the function template.

Related information

-
C++ compiler option: --no-double (-F)

Menu entry
1. Select C/C++ Compiler » Language.
2. Enable the option Treat double as float.

Command line syntax

--no-double
--F

Description

With this option you tell the C++ compiler to treat variables of the type double as float. Because the float type takes less space, execution speed increases and code size decreases, both at the cost of less precision.

Example

cptc --no-double test.cc

The file test.cc is compiled where variables of the type double are treated as float.

Related information

-
C++ compiler option: --no-enum-overloading

Menu entry

Command line syntax

--no-enum-overloading

Description

Disable support for using operator functions to overload built-in operations on enum-typed operands.

Related information

-
C++ compiler option: --no-explicit

Menu entry

- 

Command line syntax

--no-explicit

Default: the explicit specifier is allowed.

Description

Disable support for the explicit specifier on constructor declarations.

Related information

-
C++ compiler option: --no-export

Menu entry

Command line syntax

--no-export

Default: exported templates (declared with the keyword export) are allowed.

Description

Disable recognition of exported templates. This option requires that dependent name processing be done, and cannot be used with implicit inclusion of template definitions.

Related information

Section 2.5.5, Exported Templates
C++ compiler option: --no-extern-inline

Menu entry

Command line syntax

--no-extern-inline

Default: inline functions are allowed to have external linkage.

Description

Disable support for inline functions with external linkage in C++. When inline functions are allowed to have external linkage (as required by the standard), then extern and inline are compatible specifiers on a non-member function declaration; the default linkage when inline appears alone is external (that is, inline means extern inline on non-member functions); and an inline member function takes on the linkage of its class (which is usually external). However, when inline functions have only internal linkage (using --no-extern-inline), then extern and inline are incompatible; the default linkage when inline appears alone is internal (that is, inline means static inline on non-member functions); and inline member functions have internal linkage no matter what the linkage of their class.

Related information

Section 2.6, Extern Inline Functions
C++ compiler option: --no-for-init-diff-warning

Menu entry

Command line syntax

--no-for-init-diff-warning

Description

Disable a warning that is issued when programs compiled without the --old-for-init option would have had different behavior under the old rules.

Related information

C++ compiler option --old-for-init (Use old for scoping rules)
C++ compiler option: --no-implicit-typename

Menu entry

Command line syntax

--no-implicit-typename

Default: implicit typename determination is enabled.

Description

Disable implicit determination, from context, whether a template parameter dependent name is a type or nontype.

Related information

C++ compiler option --no-typename (Disable the typename keyword)
C++ compiler option: --no-inlining

Menu entry

1. Select C/C++ Compiler » Miscellaneous.
2. Disable the option Minimal inlining of function calls (C++).

Command line syntax

--no-inlining

Description

Disable minimal inlining of function calls.

Related information

-
C++ compiler option: --nonconst-ref-anachronism

Menu entry

Command line syntax

--nonconst-ref-anachronism
--no-nonconst-ref-anachronism

Default: --no-nonconst-ref-anachronism

Description

Enable or disable the anachronism of allowing a reference to nonconst to bind to a class rvalue of the right type. This anachronism is also enabled by the --anachronisms option.

Related information

C++ compiler option --anachronisms (Enable C++ anachronisms)

Section 2.2.3, Anachronisms Accepted
C++ compiler option: --nonstd-qualifier-deduction

Menu entry

Command line syntax

--nonstd-qualifier-deduction

Description

Controls whether non-standard template argument deduction should be performed in the qualifier portion of a qualified name. With this feature enabled, a template argument for the template parameter $T$ can be deduced in contexts like $A<T>::B$ or $T::B$. The standard deduction mechanism treats these as non-deduced contexts that use the values of template parameters that were either explicitly specified or deduced elsewhere.

Related information


C++ compiler option: --nonstd-using-decl

Menu entry
-

Command line syntax
--nonstd-using-decl

Default: non-standard using declarations are not allowed.

Description
Allow a non-member using declaration that specifies an unqualified name.

Related information
-
C++ compiler option: --no-parse-templates

Menu entry

Command line syntax

--no-parse-templates

Default: parsing of nonclass templates is enabled.

Description

Disable the parsing of nonclass templates in their generic form (i.e., even if they are not really instantiated). It is done by default if dependent name processing is enabled.

Related information

C++ compiler option --no-dep-name (Disable dependent name processing)
C++ compiler option: --no-pch-messages

Menu entry

Command line syntax

`--no-pch-messages`

Default: a message is displayed indicating that a precompiled header file was created or used in the current compilation. For example,

"test.cc": creating precompiled header file "test.pch"

Description

Disable the display of a message indicating that a precompiled header file was created or used in the current compilation.

Related information

C++ compiler option --pch (Automatic PCH mode)

C++ compiler option --use-pch (Use precompiled header file)

C++ compiler option --create-pch (Create precompiled header file)

Section 2.9, Precompiled Headers
C++ compiler option: --no-preprocessing-only

Menu entry

Eclipse always does a full compilation.

Command line syntax

--no-preprocessing-only

Description

You can use this option in conjunction with the options that normally cause the C++ compiler to do preprocessing only (e.g., --preprocess, etc.) to specify that a full compilation should be done (not just preprocessing). When used with the implicit inclusion option, this makes it possible to generate a preprocessed output file that includes any implicitly included files.

Example

cptc --preprocess --implicit-include --no-preprocessing-only test.cc

Related information

C++ compiler option --preprocess (Preprocessing only)

C++ compiler option --implicit-include (Implicit source file inclusion)
C++ compiler option: --no-stdinc

Menu entry

1. Select C/C++ Compiler » Miscellaneous.
2. Add the option --no-stdinc to the Additional options field.

Command line syntax

--no-stdinc

Description

With this option you tell the C++ compiler not to look in the default include directory relative to the installation directory, when searching for include files. This way the C++ compiler only searches in the include file search paths you specified.

Related information

Section 5.3, How the C++ Compiler Searches Include Files
C++ compiler option: --no-tasking-sfr

Menu entry
1. Select C/C++ Compiler » Preprocessing.
2. Disable the option Automatic inclusion of '.sfr' file.

Command line syntax

--no-tasking-sfr

Description
Normally, the C++ compiler includes a special function register (SFR) file before compiling. The C++ compiler automatically selects the SFR file belonging to the target you selected on the Processor page (option --cpu).

With this option the C++ compiler does not include the register file regcpu.sfr as based on the selected target processor.

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

Related information
C++ compiler option --cpu (Select processor)

Section 1.3.2, Accessing Hardware from C
C++ compiler option: --no-typename

Menu entry
- 

Command line syntax
--no-typename

Default: typename is recognized as a keyword.

Description
Disable recognition of the typename keyword.

Related information
C++ compiler option --no-implicit-typename (Disable implicit typename determination)
C++ compiler option: --no-use-before-set-warnings (-j)

Menu entry
1. Select C/C++ Compiler » Diagnostics.
2. Enable the option Suppress C++ compiler "used before set" warnings.

Command line syntax
--no-use-before-set-warnings
-j

Description
Suppress warnings on local automatic variables that are used before their values are set.

Related information
C++ compiler option --no-warnings (Suppress all warnings)
C++ compiler option: --no-warnings (-w)

Menu entry
1. Select C/C++ Compiler » Diagnostics.
2. Enable the option Suppress all warnings.

Command line syntax

--no-warnings
-w

Description
With this option you suppress all warning messages. Error messages are still issued.

Related information
C++ compiler option --warnings-as-errors (Treat warnings as errors)
C++ compiler option: --old-for-init

Menu entry

Command line syntax

--old-for-init

Description

Control the scope of a declaration in a for-init-statement. The old (cfront-compatible) scoping rules mean the declaration is in the scope to which the for statement itself belongs; the default (standard-conforming) rules in effect wrap the entire for statement in its own implicitly generated scope.

Related information

C++ compiler option --no-for-init-diff-warning (Disable warning for old for-scoping)
C++ compiler option: --old-line-commands

Menu entry

Command line syntax

--old-line-commands

Description

When generating source output, put out #line directives in the form # nnn instead of #line nnn.

Example

To do preprocessing only, without comments and with old style line control information, enter:

cptc --preprocess --old-line-commands test.cc

Related information

C++ compiler option --preprocess (Preprocessing only)
C++ compiler option: --old-specializations

Menu entry

Command line syntax

--old-specializations

Description

Enable acceptance of old-style template specializations (that is, specializations that do not use the template<> syntax).

Related information
**C++ compiler option: --option-file (-f)**

**Menu entry**

- 

**Command line syntax**

```
--option-file=file
-f file
```

**Description**

This option is primarily intended for command line use. Instead of typing all options on the command line, you can create an option file which contains all options and flags you want to specify. With this option you specify the option file to the C++ compiler.

Use an option file when the command line would exceed the limits of the operating system, or just to store options and save typing.

You can specify the option `--option-file` multiple times.

**Format of an option file**

- Multiple arguments on one line in the option file are allowed.

- To include whitespace in an argument, surround the argument with single or double quotes.

- If you want to use single quotes as part of the argument, surround the argument by double quotes and vise versa:

  "This has a single quote ' embedded"

  'This has a double quote " embedded'

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

- When a text line reaches its length limit, use a \\ to continue the line. Whitespace between quotes is preserved.

  "This is a continuation \\ line"

  -> "This is a continuation line"

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

**Example**

Suppose the file `myoptions` contains the following lines:
TASKING VX-toolset for TriCore User Guide

```
--embedded-c++
--define=DEMO=1
test.cc
```

Specify the option file to the C++ compiler:

```
cptc --option-file=myoptions
```

This is equivalent to the following command line:

```
cptc --embedded-c++ --define=DEMO=1 test.cc
```

Related information

-
**C++ compiler option: --output (-o)**

**Menu entry**

Eclipse names the output file always after the C++ source file.

**Command line syntax**

`--output-file=file`

`-o file`

Default: module name with `.ic` suffix.

**Description**

With this option you can specify another filename for the output file of the C++ compiler. Without this option the basename of the C++ source file is used with extension `.ic`.

You can also use this option in combination with the option `--preprocess (-E)` to redirect the preprocessing output to a file.

**Example**

To create the file `output.ic` instead of `test.ic`, enter:

```
cptc --output=output.ic test.cc
```

To use the file `my.pre` as the preprocessing output file, enter:

```
cptc --preprocess --output=my.pre test.cc
```

**Related information**

C++ compiler option `--preprocess` (Preprocessing)
C++ compiler option: --pch

Menu entry

1. Select C/C++ Compiler » Precompiled C++ Headers.
2. Enable the option Automatically use/create precompiled header file.

Command line syntax

--pch

Description

Automatically use and/or create a precompiled header file. If --use-pch or --create-pch (manual PCH mode) appears on the command line following this option, its effect is erased.

Related information

C++ compiler option --use-pch (Use precompiled header file)
C++ compiler option --create-pch (Create precompiled header file)
Section 2.9, Precompiled Headers
C++ compiler option: --pch-dir

Menu entry
1. Select C/C++ Compiler » Precompiled C++ Headers.
2. Enter a path in the Precompiled header file directory.

Command line syntax

--pch-dir=directory-name

Description

Specify the directory in which to search for and/or create a precompiled header file. This option may be used with automatic PCH mode (--pch) or manual PCH mode (--create-pch or --use-pch).

Example

To use the directory /usr/include/pch to automatically create precompiled header files, enter:

cptc --pch-dir=/usr/include/pch --pch test.cc

Related information

C++ compiler option --pch (Automatic PCH mode)
C++ compiler option --use-pch (Use precompiled header file)
C++ compiler option --create-pch (Create precompiled header file)

Section 2.9, Precompiled Headers
C++ compiler option: --pch-verbose

Menu entry
-

Command line syntax

--pch-verbose

Description

In automatic PCH mode, for each precompiled header file that cannot be used for the current compilation, a message is displayed giving the reason that the file cannot be used.

Example

cptc --pch --pch-verbose test.cc

Related information

C++ compiler option --pch (Automatic PCH mode)

Section 2.9, Precompiled Headers
C++ compiler option: --pending-instantiations

Menu entry
-

Command line syntax

--pending-instantiations=\(n\)

where \(n\) is the maximum number of instantiations of a single template.

Default: 64

Description

Specifies the maximum number of instantiations of a given template that may be in process of being instantiated at a given time. This is used to detect runaway recursive instantiations. If \(n\) is zero, there is no limit.

Example

To specify a maximum of 32 pending instantiations, enter:

cptc --pending-instantiations=32 test.cc

Related information

Section 2.5, Template Instantiation
C++ compiler option: --preprocess (-E)

Menu entry
1. Select **C/C++ Compiler » Preprocessing**.
2. Enable the option **Store C preprocessor output in <file>.pre**.

Command line syntax

```
--preprocess [=flags]
-E [flags]
```

You can set the following flags:

- `-+/-comments`  
  - `-c/C`  
  - `keep comments`
- `-+/-make`  
  - `-m/M`  
  - `generate dependencies for make`
- `-+/-noline`  
  - `-p/P`  
  - `strip #line source position information`

Default: `-ECMP`

Description

With this option you tell the C++ compiler to preprocess the C++ source. Under Eclipse the C++ compiler sends the preprocessed output to the file `name.pre` (where `name` is the name of the C++ source file to compile). Eclipse also compiles the C++ source.

On the command line, the C++ compiler sends the preprocessed file to `stdout`. To capture the information in a file, specify an output file with the option `--output`.

With `--preprocess=+comments` you tell the preprocessor to keep the comments from the C++ source file in the preprocessed output.

With `--preprocess=+make` the C++ compiler will generate dependency lines that can be used in a Makefile. The preprocessor output is discarded.

When implicit inclusion of templates is enabled, the output may indicate false (but safe) dependencies unless `--no-preprocessing-only` is also used.

With `--preprocess=+noline` you tell the preprocessor to strip the `#line` source position information (lines starting with `#line`). These lines are normally processed by the assembler and not needed in the preprocessed output. When you leave these lines out, the output is easier to read.

Example

```
cptc --preprocess=+comments,-make,-noline test.cc --output=test.pre
```
The C++ compiler preprocesses the file test.cc and sends the output to the file test.pre. Comments are included but no dependencies are generated and the line source position information is not stripped from the output file.

**Related information**

C++ compiler option --no-preprocessing-only (Force full compilation)
C++ compiler option: --remarks (-r)

Menu entry
1. Select C/C++ Compiler » Diagnostics.
2. Enable the option Issue remarks on C++ code.

Command line syntax

--remarks

-r

Description
Issue remarks, which are diagnostic messages even milder than warnings.

Related information
Section 5.4, C++ Compiler Error Messages
C++ compiler option: --remove-unneeded-entities

Menu entry

Command line syntax

--remove-unneeded-entities

Description

Enable an optimization to remove types, variables, routines, and related constructs that are not really needed. Something may be referenced but unneeded if it is referenced only by something that is itself unneeded; certain entities, such as global variables and routines defined in the translation unit, are always considered to be needed.

Related information

-
C++ compiler option: --rtti

Menu entry

1. Select C/C++ Compiler » Language.
2. Enable the option Support for RTTI (run-time type information).

Command line syntax

--rtti

Default: RTTI (run-time type information) features are disabled.

Description

Enable support for RTTI (run-time type information) features: dynamic_cast, typeid.

The macro __RTTI is defined when RTTI support is enabled.

Related information
C++ compiler option: --schar (-s)

Menu entry
1. Select C/C++ Compiler » Language.
2. Disable the option Treat "char" variables as unsigned.

Command line syntax
--schar
-s

Description
With this option char is the same as signed char.
When plain char is signed, the macro __SIGNED_CHARS__ is defined.

Related information
C++ compiler option --uchar (Plain char is unsigned)

Section 1.1, Data Types
C++ compiler option: --special-subscript-cost

Menu entry

Command line syntax

--special-subscript-cost

Description

Enable a special nonstandard weighting of the conversion to the integral operand of the [] operator in overload resolution.

This is a compatibility feature that may be useful with some existing code. With this feature enabled, the following code compiles without error:

```cpp
struct A {
    A();
    operator int *();
    int operator[](unsigned);
};
void main() {
    A a;
    a[0];  // Ambiguous, but allowed with this option
    // operator[] is chosen
}
```

Related information

-
C++ compiler option: --strict (-A)

Menu entry

1. Select C/C++ Compiler » Language.
2. Disable the option Allow non-ANSI/ISO C++ features.

Command line syntax

--strict

-A

Default: non-ANSI/ISO C++ features are enabled.

Description

Enable strict ANSI/ISO mode, which provides diagnostic messages when non-standard features are used, and disables features that conflict with ANSI/ISO C or C++. All ANSI/ISO violations are issued as errors.

Example

To enable strict ANSI mode, with error diagnostic messages, enter:

cptc --strict test.cc

Related information

C++ compiler option --strict-warnings (Strict ANSI/ISO mode with warnings)
C++ compiler option: --strict-warnings (-a)

Menu entry

Command line syntax

--strict-warnings

-a

Default: non-ANSI/ISO C++ features are enabled.

Description

This option is similar to the option --strict, but all violations are issued as warnings instead of errors.

Example

To enable strict ANSI mode, with warning diagnostic messages, enter:

cptc --strict-warnings test.cc

Related information

C++ compiler option --strict (Strict ANSI/ISO mode with errors)


C++ compiler option: --suppress-instantiation-flags

Menu entry

Command line syntax

--suppress-instantiation-flags

Description

When automatic instantiation is being used, this option suppresses the generation of the special symbols used by the C++ prelinker to do automatic instantiation processing. This option is for use by the control program and prelinker and should not be used directly by users.

Related information

Section 2.5, Template Instantiation
C++ compiler option: --suppress-vtbl

Menu entry

1. Select C/C++ Compiler » Miscellaneous.
2. Enable the option Suppress definition of virtual function tables.

Command line syntax

--suppress-vtbl

Description

Suppress definition of virtual function tables in cases where the heuristic used by the C++ compiler to decide on definition of virtual function tables provides no guidance. The virtual function table for a class is defined in a compilation if the compilation contains a definition of the first non-inline non-pure virtual function of the class. For classes that contain no such function, the default behavior is to define the virtual function table (but to define it as a local static entity). The --suppress-vtbl option suppresses the definition of the virtual function tables for such classes, and the --force-vtbl option forces the definition of the virtual function table for such classes. --force-vtbl differs from the default behavior in that it does not force the definition to be local.

Related information

C++ compiler option --force-vtbl (Force definition of virtual function tables)
C++ compiler option: --sys-include

Menu entry

-  

Command line syntax

--sys-include=directory,...

Description

Change the algorithm for searching system include files whose names do not have an absolute pathname to look in directory.

Example

To add the directory /proj/include to the system include file search path, enter:

cptc --sys-include=/proj/include test.cc

Related information

C++ compiler option --include-directory (Add directory to include file search path)

Section 5.3, How the C++ Compiler Searches Include Files
C++ compiler option: --template-directory

Menu entry

Command line syntax

--template-directory=directory,...

Description

Specifies a directory name to be placed on the exported template search path. The directories are used to find the definitions of exported templates (.et files) and are searched in the order in which they are specified on the command line. The current directory is always the first entry on the search path.

Example

To add the directory export to the exported template search path, enter:

cptc --template-directory=export test.cc

Related information

Section 2.5.5, Exported Templates
C++ compiler option: --template-info-file

Menu entry

Command line syntax

`--template-info-file=file`

Description

This option specifies the name to be used for the template instantiation information file used in automatic instantiation mode.

This option is supplied for use by the control program that invokes the C++ compiler and is not intended to be used by end-users.

Related information

-
**C++ compiler option: --timing**

**Menu entry**

-  

**Command line syntax**

`--timing`

Default: no timing information is generated.

**Description**

Generate compilation timing information. This option causes the C++ compiler to display the amount of CPU time and elapsed time used by each phase of the compilation and a total for the entire compilation.

**Example**

```
cptc --timing test.cc
```

processed 180 lines at 8102 lines/min

**Related information**

-
C++ compiler option: --trace-includes

Menu entry
-

Command line syntax
--trace-includes

Description
Output a list of the names of files #included to the error output file. The source file is compiled normally (i.e. it is not just preprocessed) unless another option that causes preprocessing only is specified.

Example

cptc --trace-includes test.cc

iostream.h
string.h

Related information
C++ compiler option --preprocess (Preprocessing only)
C++ compiler option: --type-traits-helpers

Menu entry

Command line syntax

--type-traits-helpers
--no-type-traits-helpers

Default: in C++ mode type traits helpers are enabled by default. In GNU C++ mode, type traits helpers are never enabled by default.

Description

Enable or disable type traits helpers (like __is_union and __has_virtual_destructor). Type traits helpers are meant to ease the implementation of ISO/IEC TR 19768.

The macro __TYPE_TRAITS_ENABLED is defined when type traits pseudo-functions are enabled.

Related information

-
C++ compiler option: --uchar (-u)

Menu entry
1. Select C/C++ Compiler » Language.
2. Enable the option Treat "char" variables as unsigned.

Command line syntax
--uchar

-u

Description
By default char is the same as specifying signed char. With this option char is the same as unsigned char.

Related information
C++ compiler option --schar (Plain char is signed)

Section 1.1, Data Types
C++ compiler option: --undefine (-U)

Menu entry

1. Select **C/C++ Compiler » Preprocessing**

   *The Defined symbols box shows the symbols that are currently defined.*

2. To remove a defined symbol, select the symbol in the **Defined symbols** box and click on the **Delete** button.

Command line syntax

```
--undefine=macro_name
-U macro_name
```

Description

Remove any initial definition of *macro_name* as in `#undef`. **--undefine** options are processed after all **--define** options have been processed.

You cannot undefine a predefined macro as specified in *Section 2.8, Predefined Macros*, except for:

```
__STDC__
__cplusplus
__SIGNED_CHARS__
```

Example

To undefine the predefined macro `__cplusplus`:

```
cptc --undefine=__cplusplus test.cc
```

Related information

**C++ compiler option --define** (Define preprocessor macro)

*Section 2.8, Predefined Macros*
**C++ compiler option: --use-pch**

**Menu entry**

1. Select C/C++ Compiler » Precompiled C++ Headers.
2. Enter a filename in the Use precompiled header file field.

**Command line syntax**

```
--use-pch=filename
```

**Description**

Use a precompiled header file of the specified name as part of the current compilation. If --pch (automatic PCH mode) or --create-pch appears on the command line following this option, its effect is erased.

**Example**

To use the precompiled header file with the name test.pch, enter:

```
cptc --use-pch=test.pch test.cc
```

**Related information**

- C++ compiler option --pch (Automatic PCH mode)
- C++ compiler option --create-pch (Create precompiled header file)

Section 2.9, *Precompiled Headers*
C++ compiler option: --using-std

Menu entry

Command line syntax

--using-std

Default: implicit use of the std namespace is disabled.

Description

Enable implicit use of the std namespace when standard header files are included. Note that this does not do the equivalent of putting a "using namespace std;" in the program to allow old programs to be compiled with new header files; it has a special and localized meaning related to the TASKING versions of certain header files, and is unlikely to be of much use to end-users of the TASKING C++ compiler.

Related information

C++ compiler option --namespaces (Support for namespaces)

Section 2.4, Namespace Support
C++ compiler option: --variadic-macros

Menu entry
-

Command line syntax

--variadic-macros

Default: macros with a variable number of arguments are not allowed.

Description
Allow macros with a variable number of arguments.

Related information
C++ compiler option --extended-variadic-macros (Allow extended variadic macros)
C++ compiler option: --version (-V)

Menu entry
-

Command line syntax

--version
-v

Description

Display version information. The C++ compiler ignores all other options or input files.

Example

cptc --version

The C++ compiler does not compile any files but displays the following version information:

TASKING VX-toolset for TriCore: C++ compiler   vx.yrz Build nnn
Copyright 2002-year Altium BV                  Serial# 00000000

Related information
-


**C++ compiler option: --warnings-as-errors**

**Menu entry**

1. Select **Global Options**.
2. Enable the option **Treat warnings as errors**.

**Command line syntax**

---warnings-as-errors

**Description**

If the C++ compiler encounters an error, it stops compiling. With this option you tell the C++ compiler to treat all warnings as errors. This means that the exit status of the C++ compiler will be non-zero after one or more compiler warnings. As a consequence, the C++ compiler now also stops after encountering a warning.

**Related information**

C++ compiler option **--no-warnings** (Suppress all warnings)
C++ compiler option: --wchar_t-keyword

Menu entry

1. Select C/C++ Compiler » Language.
2. Enable the option Allow the 'wchar_t' keyword.

Command line syntax

--wchar_t-keyword

Default: wchar_t is not recognized as a keyword.

Description

Enable recognition of wchar_t as a keyword.

The macro _WCHAR_T is defined when wchar_t is recognized as a keyword.

Related information

-
C++ compiler option: --xref-file (-X)

Menu entry

- Command line syntax

--xref-file=file
-Xfile

Description

Generate cross-reference information in a file. For each reference to an identifier in the source program, a line of the form

symbol_id name X file-name line-number column-number

is written, where X is

D for definition;
d for declaration (that is, a declaration that is not a definition);
M for modification;
A for address taken;
U for used;
C for changed (but actually meaning used and modified in a single operation, such as an increment);
R for any other kind of reference, or
E for an error in which the kind of reference is indeterminate.

symbol-id is a unique decimal number for the symbol. The fields of the above line are separated by tab characters.

Related information

-
11.3. Assembler Options

This section lists all assembler options.

Options in Eclipse versus options on the command line

Most command line options have an equivalent option in Eclipse but some options are only available on
the command line. Eclipse invokes the assembler via the control program. Therefore, it uses the syntax
of the control program to pass options and files to the assembler. If there is no equivalent option in Eclipse,
you can specify a command line option in Eclipse as follows:

1. From the Project menu, select Properties
   The Properties dialog appears.
2. In the left pane, expand C/C++ Build and select Settings.
   In the right pane the Settings appear.
3. On the Tool Settings tab, select Assembler » Miscellaneous.
4. In the Additional options field, enter one or more command line options.
   Because Eclipse uses the control program, Eclipse automatically precedes the option with -Wa to
   pass the option via the control program directly to the assembler.

Note that the options you enter in the Assembler page are not only used for hand-coded assembly
files, but also for the assembly files generated by the compiler.

Be aware that some command line options are not useful in Eclipse or just do not have any effect. For
example, the option -V displays version header information and has no effect in Eclipse.

Short and long option names

Options can have both short and long names. Short option names always begin with a single minus (-)
character, long option names always begin with two minus (--) characters. You can abbreviate long option
names as long as it forms a unique name. You can mix short and long option names on the command
line.

Options can have flags or suboptions. To switch a flag 'on', use a lowercase letter or a +longflag. To
switch a flag off, use an uppercase letter or a -longflag. Separate longflags with commas. The following
two invocations are equivalent:

astc -Ogs test.src
astc --optimize=+generics,+instr-size test.src

When you do not specify an option, a default value may become active.
Assembler option: --case-insensitive (-c)

Menu entry
1. Select Assembler » Symbols.
2. Enable the option Case insensitive identifiers.

Command line syntax

--case-insensitive
-c

Default: case sensitive

Description

With this option you tell the assembler not to distinguish between upper and lower case characters. By default the assembler considers upper and lower case characters as different characters.

Assembly source files that are generated by the compiler must always be assembled case sensitive. When you are writing your own assembly code, you may want to specify the case insensitive mode.

Example

When assembling case insensitive, the label LabelName is the same label as labelname.

astc --case-insensitive test.src

Related information

Assembler control $CASE
Assembler option: --check

Menu entry

Command line syntax

--check

Description

With this option you can check the source code for syntax errors, without generating code. This saves time in developing your application.

The assembler reports any warnings and/or errors.

This option is available on the command line only.

Related information

C compiler option --check (Check syntax)
Assembler option: --core

Menu entry

1. Select Assembler » Miscellaneous.
2. Add the option --core to the Additional options field.

Command line syntax

--core=core

You can specify the following core arguments:

- tc1.2 TriCore 1.2 architecture
- tc1.3 TriCore 1.3 architecture
- tc1.3.1 TriCore 1.3.1 architecture, TriCore 1.3.1 instructions are allowed
- tc2 TriCore 2 architecture, TriCore 2 instructions are allowed

Default: derived from --cpu, if used, otherwise tc1.3

Description

With this option you specify the core architecture for a custom target processor for which you create your application. By default the TriCore toolset derives the core from the processor you selected.

With --core=tc1.3.1, you can use TriCore 1.3.1 instructions in the assembly code. The define __TC131__ is set to 1.

With --core=tc2, you can use TriCore 2 instructions in the assembly code. The define __TC2__ is set to 1.

If you select a valid target processor (command line option --cpu (-C)), the core is automatically set, based on the chosen target processor.

Example

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

astc --core=tc2 test.src

Related information

Assembler control $TC2
Assembler option --cpu (Select processor)
Assembler option --fpu-present (FPU present)
Assembler option --mmu-present (MMU present)
Assembler option: --cpu (-C)

Menu entry
1. Expand C/C++ Build and select Processor.
2. From the Processor Selection list, select a processor.

Command line syntax

--cpu=cpu
-c cpu

Description

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

Based on this option the assembler always includes the special function register file regcpu.def, unless you disable the option Automatic inclusion of '.def' file on the Preprocessing page (option --no-tasking-sfr).

Based on the target processor the assembler automatically detects whether a MMU or FPU-unit is present and whether the architecture is a TriCore2. This means you do not have to specify the assembler options --mmu-present, --fpu-present and --core=tc2 explicitly when one of the supported derivatives is selected.

To avoid conflicts, make sure you specify the same target processor as you did for the compiler (Eclipse and the control program do this automatically).

Example

To assemble the file test.src for the TC1165 processor and use the register file regtc1165.def:

astc --cpu=tc1165 test.src

Related information

Assembler option --no-tasking-sfr (Do not include .def file)

C compiler option --cpu (Select processor)

Section 3.5.1, Special Function Registers
Assembler option: --debug-info (-g)

Menu entry
1. Select Assembler » Symbols.
2. Select an option from the Generate symbolic debug list.

Command line syntax

--debug-info [=flags]
-g [flags]

You can set the following flags:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+/asm</td>
<td>Assembly source line information</td>
</tr>
<tr>
<td>+/-hll</td>
<td>Pass high level language debug information (HLL)</td>
</tr>
<tr>
<td>+/-local</td>
<td>Assembler local symbols debug information</td>
</tr>
<tr>
<td>+/-smart</td>
<td>Smart debug information</td>
</tr>
</tbody>
</table>

Default: --debug-info=+hll

Default (without flags): --debug-info=+smart

Description

With this option you tell the assembler which kind of debug information to emit in the object file.

You cannot specify --debug-info=+asm,+hll. Either the assembler generates assembly source line information, or it passes HLL debug information.

When you specify --debug-info=+smart, the assembler selects which flags to use. If high level language information is available in the source file, the assembler passes this information (same as --debug-info=-asm,+hll,-local). If not, the assembler generates assembly source line information (same as --debug-info=+asm,-hll,+local).

With --debug-info=AHLS the assembler does not generate any debug information.

Related information

Assembler control $DEBUG
Assembler option: --define (-D)

Menu entry

1. Select Assembler » Preprocessing.
   *The Defined symbols box right-below shows the symbols that are currently defined.*

2. To define a new symbol, click on the Add button in the Defined symbols box.

3. Type the symbol definition (for example, demo=1)

   Use the Edit and Delete button to change a macro definition or to remove a macro from the list.

Command line syntax

```
--define=macro_name[=macro_definition]
-Dmacro_name[=macro_definition]
```

Description

With this option you can define a macro and specify it to the assembler preprocessor. If you only specify a macro name (no macro definition), the macro expands as ‘1’.

You can specify as many macros as you like. Simply use the Add button to add new macro definitions.

On the command line, use the option --define (-D) multiple times. If the command line exceeds the limit of the operating system, you can define the macros in an option file which you then must specify to the assembler with the option --option-file (-f) file.

Defining macros with this option (instead of in the assembly source) is, for example, useful in combination with conditional assembly as shown in the example below.

This option has the same effect as defining symbols via the .DEFINE, .SET, and .EQU directives. (similar to #define in the C language). With the .MACRO directive you can define more complex macros.

Example

Consider the following assembly program with conditional code to assemble a demo program and a real program:

```
.IF DEMO == 1
...
    ; instructions for demo application
.ELSE
...
    ; instructions for the real application
.ENDIF
```
You can now use a macro definition to set the DEMO flag:

```
astc --define=DEMO test.src
astc --define=DEMO=1 test.src
```

Note that both invocations have the same effect.

**Related information**

**Assembler option** `--option-file` *(Specify an option file)*
Assembler option: --diag

Menu entry

1. From the Window menu, select Show View » Other » Basic » Problems.
   The Problems view is added to the current perspective.

2. In the Problems view right-click on a message.
   A popup menu appears.

3. Select Detailed Diagnostics Info.
   A dialog box appears with additional information.

Command line syntax

--diag=[format:] {all | nr, ...}

You can set the following output formats:

- html: HTML output.
- text: ASCII text.

Default format: text

Description

With this option you can ask for an extended description of error messages in the format you choose. The output is directed to stdout (normally your screen) and in the format you specify. You can specify the following formats: html, rtf or text (default). To create a file with the descriptions, you must redirect the output.

With the suboption all, the descriptions of all error messages are given. If you want the description of one or more selected error messages, you can specify the error message numbers, separated by commas.

Example

To display an explanation of message number 244, enter:

astc --diag=244

This results in the following message and explanation:

W244: additional input files will be ignored

The assembler supports only a single input file. All other input files are ignored.
To write an explanation of all errors and warnings in HTML format to file `cerrors.html`, use redirection and enter:

```bash
astc --diag=html:all > aserrors.html
```

**Related information**

Section 7.6, *Assembler Error Messages*
Assembler option: --emit-locals

Menu entry
1. Select Assembler » Symbols.
2. Enable or disable one or both of the following options:
   - Emit local EQU symbols
   - Emit local non-EQU symbols

Command line syntax

```
--emit-locals[=flag,...]
```

You can set the following flags:

```
+/-equ$s
e/E
+/-symbols
s/S
```

emit local EQU symbols
emit local non-EQU symbols

Default: `--emit-locals=ES`

Default (without flags): `--emit-locals=+symbols`

Description

With the option `--emit-locals=+equ$s` the assembler also emits local EQU symbols to the object file. Normally, only global symbols and non-EQU local symbols are emitted. Having local symbols in the object file can be useful for debugging.

Related information

Assembler directive `.EQU`
Assembler option: --error-file

Menu entry

Command line syntax

--error-file[=file]

Description

With this option the assembler redirects error messages to a file. If you do not specify a filename, the error file will be named after the input file with extension .ers.

Example

To write errors to errors.ers instead of stderr, enter:

astc --error-file=errors.ers test.src

Related information

Section 7.6, Assembler Error Messages
Assembler option: --error-limit

Menu entry

1. Select Assembler » Diagnostics.

2. Enter a value in the Maximum number of emitted errors field.

Command line syntax

--error-limit=number

Default: 42

Description

With this option you tell the assembler to only emit the specified maximum number of errors. When 0 (null) is specified, the assembler emits all errors. Without this option the maximum number of errors is 42.

Related information

Section 7.6, Assembler Error Messages
Assembler option: --fpu-present

Menu entry

1. Expand C/C++ Build and select Processor.
2. From the Processor Selection list, select a processor that has an FPU.

Command line syntax

--fpu-present

Description

With this option you can use single precision floating-point instructions in the assembly code. When you select this option, the define __FPU__ is set to 1.

If you select a valid target processor (command line option --cpu (-C)), this option is automatically set, based on the chosen target processor.

Example

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

astc --fpu-present test.src

Related information

Assembler control $FPU

Assembler option --cpu (Select processor)

Assembler option --core (Select TriCore architecture)
Assembler option: --help (-?)

Menu entry

-  

Command line syntax

--help[=item]

-?

You can specify the following arguments:

   options       Show extended option descriptions

Description

Displays an overview of all command line options. When you specify the argument options you can list detailed option descriptions.

Example

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

astc -?
astc --help
astc

To see a detailed description of the available options, enter:

astc --help=options

Related information

-
Assembler option: --include-directory (-I)

Menu entry

1. Select Assembler » Include Paths.
   
   The Include paths box shows the directories that are added to the search path for include files.

2. To define a new directory for the search path, click on the Add button in the Include paths box.

3. Type or select a path.

   Use the Edit and Delete button to change a path or to remove a path from the list.

Command line syntax

--include-directory=path,...

-I path,...

Description

With this option you can specify the path where your include files are located. A relative path will be relative to the current directory,

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

1. The pathname in the assembly file and the directory of the assembly source.

2. The path that is specified with this option.

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

4. The default directory $(PRODDIR)\include.

Example

Suppose that the assembly source file test.src contains the following lines:

.INCLUDE 'myinc.inc'

You can call the assembler as follows:

astc --include-directory=c:\proj\include test.src

First the assembler looks for the file myinc.inc in the directory where test.src is located. If it does not find the file, it looks in the directory c:\proj\include (this option). If the file is still not found, the assembler searches in the environment variable and then in the default include directory.
Related information

Assembler option --include-file (Include file at the start of the input file)
Assembler option: --include-file (-H)

Menu entry

1. Select Assembler » Preprocessing.

   The upper box shows the files that are currently included before the assembling starts.

2. To define a new file, click on the Add button in the Include files at start of input file box.

3. Type the full path and file name or select a file.

   Use the Edit and Delete button to change a file name or to remove a file from the list.

Command line syntax

--include-file=file,...

-H file,...

Description

With this option (set at project level) you include one extra file at the beginning of the assembly source file. The specified include file is included before all other includes. This is the same as specifying .INCLUDE 'file' at the beginning of your assembly source.

Example

astc --include-file=myinc.inc test.src

The file myinc.inc is included at the beginning of test.src before it is assembled.

Related information

Assembler option --include-directory (Add directory to include file search path)
Assembler option: --keep-output-files (-k)

Menu entry

Eclipse always removes the object file when errors occur during assembling.

Command line syntax

--keep-output-files

-k

Description

If an error occurs during assembling, the resulting object file (.o) may be incomplete or incorrect. With this option you keep the generated object file when an error occurs.

By default the assembler removes the generated object file when an error occurs. This is useful when you use the make utility. If the erroneous files are not removed, the make utility may process corrupt files on a subsequent invocation.

Use this option when you still want to use the generated object. For example when you know that a particular error does not result in a corrupt object file.

Related information

Assembler option --warnings-as-errors (Treat warnings as errors)
Assembler option: --list-file (-l)

Menu entry

1. Select Assembler » List File.
2. Enable the option Generate list file.
3. Enable or disable the types of information to be included.

Command line syntax

--list-file [=file]

-1 [file]

Default: no list file is generated

Description

With this option you tell the assembler to generate a list file. A list file shows the generated object code and the relative addresses. Note that the assembler generates a relocatable object file with relative addresses.

With the optional file you can specify an alternative name for the list file. By default, the name of the list file is the basename of the source file with the extension .lst.

Related information

Assembler option --list-format (Format list file)
Assembler option: --list-format (-L)

Menu entry
1. Select Assembler » List File.
2. Enable the option Generate list file.
3. Enable or disable the types of information to be included.

Command line syntax

--list-format=flag,...

-Lflags

You can set the following flags:

+/-section  d/D  List section directives (.SDECL, .SECT)
+/-symbol    e/E  List symbol definition directives
+/-generic-expansion  g/G  List expansion of generic instructions
+/-generic        i/I  List generic instructions
+/-macro         m/M  List macro definitions
+/-empty-line    n/N  List empty source lines (newline)
+/-conditional   p/P  List conditional assembly
+/-equate        q/Q  List equate and set directives (.EQU, .SET)
+/-relocations   r/R  List relocations characters ‘r’
+/-equate-values v/V  List equate and set values
+/-wrap-lines    w/W  Wrap source lines
+/-macro-expansion x/X  List macro expansions
+/-cycle-count   y/Y  List cycle counts
+/-define-expansion z/Z  List define expansions

Use the following options for predefined sets of flags:

--list-format=0  -L0  All options disabled
Alias for --list-format=DEGIMNPQRVWXYZ
--list-format=1  -L1  All options enabled
Alias for --list-format=DEGIMNPQRVWXYZ

Default: --list-format=dEGiMnPqrVwXyZ

Description

With this option you specify which information you want to include in the list file.
On the command line you must use this option in combination with the option --list-file (-l).

**Related information**

Assembler option --list-file (Generate list file)

Assembler option --section-info=+list (Display section information in list file)
Assembler option: --mmu-present

Menu entry
1. Expand C/C++ Build and select Processor.
2. From the Processor Selection list, select a processor that has an MMU.

Command line syntax
--mmu-present

Description
With this option you can use memory management instructions in the assembly code. When you select this option, the define __MMU__ is set to 1.

If you select a valid target processor (command line option --cpu (-C)), this option is automatically set, based on the chosen target processor.

Example
To allow the use of memory management unit (MMU) instructions in the assembly code, enter:

```
astc --mmu-present test.src
```

Related information
Assembler control $MMU
Assembler option --cpu (Select processor)
Assembler option --fpu-present (FPU present)
Assembler option --core (Select TriCore architecture)
Assembler option: --no-tasking-sfr

Menu entry

1. Select Assembler » Preprocessing.
2. Disable the option Automatic inclusion of '.def' file.

Command line syntax

--no-tasking-sfr

Description

Normally, the assembler includes a special function register (SFR) file before assembling. The assembler automatically selects the SFR file belonging to the target you select on the Processor page (assembler option --cpu).

With this option the assembler does not include the register file regcpu.def as based on the selected target processor.

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

Example

astc --cpu=tc1165 --no-tasking-sfr test.src

The register file regtc1165.def is not included, but the assembler allows the use of MMU instructions due to --cpu.

Related information

Assembler option --cpu (Select processor)
Assembler option: --no-warnings (-w)

Menu entry

1. Select Assembler » Diagnostics.

   The Suppress warnings box shows the warnings that are currently suppressed.

2. To suppress a warning, click on the Add button in the Suppress warnings box.

3. Enter the numbers, separated by commas, of the warnings you want to suppress (for example 201,202). Or you can use the Add button multiple times.

4. To suppress all warnings, enable the option Suppress all warnings.

   Use the Edit and Delete button to change a warning number or to remove a number from the list.

Command line syntax

--no-warnings[number, ...]

-w[number, ...]

Description

With this option you can suppresses all warning messages or specific warning messages.

On the command line this option works as follows:

- If you do not specify this option, all warnings are reported.

- If you specify this option but without numbers, all warnings are suppressed.

- If you specify this option with a number, only the specified warning is suppressed. You can specify the option --no-warnings=number multiple times.

Example

To suppress warnings 201 and 202, enter:

astc test.src --no-warnings=201,202

Related information

Assembler option --warnings-as-errors (Treat warnings as errors)
Assembler option: --optimize (-O)

Menu entry

1. Select Assembler » Optimization.
2. Select one or more of the following options:
   - Optimize generic instructions
   - Optimize instruction size

Command line syntax

--optimize=flag,...
-Oflags

You can set the following flags:

+/-generics g/G  Allow generic instructions
+/-instr-size s/S  Optimize instruction size

Default: --optimize=gs

Description

With this option you can control the level of optimization. For details about each optimization see Section 7.4, Assembler Optimizations.

Related information

Assembler control $HW_ONLY

Section 7.4, Assembler Optimizations
Assembler option: --option-file (-f)

Menu entry

1. Select Assembler » Miscellaneous.
2. Add the option --option-file to the Additional options field.

Be aware that the options in the option file are added to the assembler options you have set in the other pages. Only in extraordinary cases you may want to use them in combination.

Command line syntax

--option-file=file,...

-f file,...

Description

This option is primarily intended for command line use. Instead of typing all options on the command line, you can create an option file which contains all options and flags you want to specify. With this option you specify the option file to the assembler.

Use an option file when the command line would exceed the limits of the operating system, or just to store options and save typing.

Option files can also be generated on the fly, for example by the make utility. You can specify the option --option-file multiple times.

Format of an option file

• Multiple arguments on one line in the option file are allowed.

• To include whitespace in an argument, surround the argument with single or double quotes.

• If you want to use single quotes as part of the argument, surround the argument by double quotes and vice versa:

"This has a single quote ' embedded"

'This has a double quote " embedded'

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

• When a text line reaches its length limit, use a \ to continue the line. Whitespace between quotes is preserved.

"This is a continuation \ line"

   -> "This is a continuation line"
• It is possible to nest command line files up to 25 levels.

Example

Suppose the file myoptions contains the following lines:

```
--debug=+asm,-local
test.src
```

Specify the option file to the assembler:

```
astc --option-file=myoptions
```

This is equivalent to the following command line:

```
astc --debug=+asm,-local test.src
```

Related information
Assembler option: --output (-o)

Menu entry

Eclipse names the output file always after the input file.

Command line syntax

--output=file

-o file

Description

With this option you can specify another filename for the output file of the assembler. Without this option, the basename of the assembly source file is used with extension .o.

Example

To create the file relobj.o instead of asm.o, enter:

astc --output=relobj.o asm.src

Related information
**Assembler option: --page-length**

**Menu entry**

1. Select **Assembler » Miscellaneous**.
2. Add the option **--page-length** to the **Additional options** field.

**Command line syntax**

```
--page-length=number
```

Default: 72

**Description**

If you generate a list file with the assembler option **--list-file**, this option sets the number of lines in a page in the list file. The default is 72, the minimum is 10. As a special case, a page length of 0 turns off page breaks.

**Related information**

Assembler option **--list-file** (Generate list file)

Assembler control **$PAGE**
Assembler option: --page-width

Menu entry

1. Select Assembler » Miscellaneous.
2. Add the option --page-width to the Additional options field.

Command line syntax

--page-width=number

Default: 132

Description

If you generate a list file with the assembler option --list-file, this option sets the number of columns per line on a page in the list file. The default is 132, the minimum is 40.

Related information

Assembler option --list-file (Generate list file)

Assembler control $PAGE
Assembler option: --preprocess (-E)

Menu entry

Command line syntax

--preprocess

-E

Description

With this option the assembler will only preprocess the assembly source file. The assembler sends the preprocessed file to stdout.

Related information

-
**Assembler option: --preprocessor-type (-m)**

**Menu entry**

```
-                      
```

**Command line syntax**

```
--preprocessor-type=type
-m type
```

You can set the following preprocessor types:

<table>
<thead>
<tr>
<th>Preprocessor Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>n</td>
<td>No preprocessor</td>
</tr>
<tr>
<td>tasking</td>
<td>t</td>
<td>TASKING preprocessor</td>
</tr>
</tbody>
</table>

Default: `--preprocessor-type=tasking`

**Description**

With this option you select the preprocessor that the assembler will use. By default, the assembler uses the TASKING preprocessor.

When the assembly source file does not contain any preprocessor symbols, you can specify to the assembler not to use a preprocessor.

**Related information**

-
Assembler option: --section-info (-t)

Menu entry
1. Select *Assembler » List File*.
2. Enable the option *Generate list file*.
3. Enable the option *List section summary*.

and/or
1. Select *Assembler » Diagnostics*.
2. Enable the option *Display section summary*.

Command line syntax

```
--section-info [=flag,...]
-t[flags]
```

You can set the following flags:

```
+/-console       c/C  Display section summary on console
+/-list          I/L  List section summary in list file
```

Default: `--section-info=CL`

Default (without flags): `--section-info=cl`

Description

With this option you tell the assembler to display section information. For each section its memory space, size, total cycle counts and name is listed on stdout and/or in the list file.

The cycle count consists of two parts: the total accumulated count for the section and the total accumulated count for all repeated instructions. In the case of nested loops it is possible that the total supersedes the section total.

Example

To writes the section information to the list file and also display the section information on stdout, enter:

```
astc --list-file --section-info asm.src
```

Related information

Assembler option `--list-file` (Generate list file)
Assembler option: --silicon-bug

Menu entry

1. Expand C/C++ Build and select Processor.
2. From the Processor Selection list, select a processor.

The CPU Problem Bypasses and Checks box shows the available workarounds/checks available for the selected processor.

3. (Optional) Select Show all CPU problem bypasses and checks.
4. Click Select All or select one or more individual options.

Command line syntax

--silicon-bug=arg,...

You can give one or more of the following arguments:

- `all-tc11ib` All TriCore TC11IB checks
- `all-tc1100` All TriCore TC1100 checks
- `all-tc1115` All TriCore TC1115 checks
- `all-tc1130` All TriCore TC1130 checks
- `all-tc1161` All TriCore TC1161 checks
- `all-tc1162` All TriCore TC1162 checks
- `all-tc1163` All TriCore TC1163 checks
- `all-tc1164` All TriCore TC1164 checks
- `all-tc1165` All TriCore TC1165 checks
- `all-tc1166` All TriCore TC1166 checks
- `all-tc1762` All TriCore TC1762 checks
- `all-tc1764` All TriCore TC1764 checks
- `all-tc1765` All TriCore TC1765 checks
- `all-tc1766` All TriCore TC1766 checks
- `all-tc1775` All TriCore TC1775 checks
- `all-tc1792` All TriCore TC1792 checks
- `all-tc1796` All TriCore TC1796 checks
- `all-tc1910` All TriCore TC1910 checks
- `all-tc1912` All TriCore TC1912 checks
- `all-tc1920` All TriCore TC1920 checks
- `cpu-tc013` Check for CPU_TC.013
**Tool Options**

- **cpu-tc018**: Check for CPU_TC.018
- **cpu-tc021**: Check for CPU_TC.021
- **cpu-tc023**: Check for CPU_TC.023
- **cpu-tc024**: Check for CPU_TC.024
- **cpu-tc030**: Check for CPU_TC.030
- **cpu-tc031**: Check for CPU_TC.031
- **cpu-tc033**: Workaround for CPU_TC.033
- **cpu-tc034**: Check for CPU_TC.034
- **cpu-tc048**: Check for CPU_TC.048
- **cpu-tc050**: Check for CPU_TC.050
- **cpu-tc051**: Workaround for CPU_TC.051
- **cpu-tc060**: Check for CPU_TC.060
- **cpu-tc065**: Check for CPU_TC.065
- **cpu-tc068**: Check for CPU_TC.068
- **cpu-tc069**: Check for CPU_TC.069
- **cpu-tc070**: Check for CPU_TC.070
- **cpu-tc071**: Check for CPU_TC.071
- **cpu-tc072**: Check for CPU_TC.072
- **cpu-tc074**: Workaround for CPU_TC.082
- **cpu-tc081**: Check for CPU_TC.082
- **cpu-tc082**: Check for CPU_TC.082
- **cpu-tc083**: Check for CPU_TC.083
- **cpu-tc094**: Check for CPU_TC.094
- **cpu-tc095**: Check for CPU_TC.095
- **cpu-tc096**: Check for CPU_TC.096
- **cpu-tc104**: Check for CPU_TC.104
- **dmu-tc001**: Check for DMU_TC.001
- **pmi-tc003**: Workaround for PMI_TC.003
- **pmu-tc004**: Workaround for PMU_TC.004

**Description**

With this option you specify for which hardware problems the assembler should check or generate workarounds. Please refer to Chapter 16, *CPU Problem Bypasses and Checks* for more information about the individual problems and workarounds.

Instead of this option you can use the following assembler controls:

- $CPU_TCnum  {ON|OFF}
- $DMU_TCnum  {ON|OFF}
$PMI_{TCnum}$ {ON|OFF}
$PMU_{TCnum}$ {ON|OFF}

When you use this option or control, the define __defect_{TCnum}__ is set to 1.

**Example**

To check for problems CPU TC.024 and CPU TC.030, enter:

```
astc --silicon-bug=cpu-tc024,cpu-tc030 test.src
```

**Related information**

Chapter 16, *CPU Problem Bypasses and Checks*

Assembler controls $defect_{TCnum}$

C compiler option --silicon-bug
Assembler option: --symbol-scope (-i)

Menu entry
1. Select Assembler » Symbols.
2. Enable or disable the option Set default symbol scope to global.

Command line syntax

--symbol-scope=scope
-i scope

You can set the following scope:

<table>
<thead>
<tr>
<th>Scope</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>global</td>
<td>g</td>
</tr>
<tr>
<td>local</td>
<td>l</td>
</tr>
</tbody>
</table>

Default: --symbol-scope=local

Description

With this option you tell the assembler how to treat symbols that you have not specified explicitly as global or local. By default the assembler treats all symbols as local symbols unless you have defined them explicitly as global.

Related information

Assembler directive .GLOBAL
Assembler directive .LOCAL
Assembler control $IDENT
Assembler option: --user-mode

Menu entry

1. Select Assembler » Miscellaneous.
2. Add the option --user-mode to the Additional options field.

Command line syntax

--user-mode=mode

You can specify the following mode arguments:

- user-0 User-0 unprivileged mode
- user-1 User-1 privileged mode
- kernel Kernel/Supervisor mode

Default: --user-mode=kernel

Description

With this option you specify the mode (I/O privilege mode) the TriCore runs in: User-0, User-1 or Kernel/Supervisor. The availability of some instructions depends on this mode. Most instructions run in all modes. The instructions enable and disable run in User-1 or kernel mode only. The instructions bisr, mtcrr, cachea.i and tlb instructions run in kernel mode only.

The define __UM_KERNEL__ or __UM_USER_1__ is set to 1, depending on the mode.

Example

To restrict the instructions in the assembly code to User-1 mode, enter:

astc --user-mode=user-1 test.src

Related information

C compiler option --user-mode (Select user mode)
Assembler option: --version (-V)

Menu entry
-

Command line syntax

--version
-v

Description
Display version information. The assembler ignores all other options or input files.

Example

astc --version

The assembler does not assemble any files but displays the following version information:

TASKING VX-toolset for TriCore: assembler vx.yrz Build nnn
Copyright 2002-year Altium BV Serial# 00000000

Related information
-
**Assembler option: --warnings-as-errors**

**Menu entry**

1. Select **Global Options**.
2. Enable the option **Treat warnings as errors**.

**Command line syntax**

```
--warnings-as-errors[=number,...]
```

**Description**

If the assembler encounters an error, it stops assembling. When you use this option without arguments, you tell the assembler to treat all warnings as errors. This means that the exit status of the assembler will be non-zero after one or more assembler warnings. As a consequence, the assembler now also stops after encountering a warning.

You can also limit this option to specific warnings by specifying a comma-separated list of warning numbers.

**Related information**

Assembler option **--no-warnings** (Suppress some or all warnings)
11.4. Linker Options

This section lists all linker options.

Options in Eclipse versus options on the command line

Most command line options have an equivalent option in Eclipse but some options are only available on the command line. Eclipse invokes the linker via the control program. Therefore, it uses the syntax of the control program to pass options and files to the linker. If there is no equivalent option in Eclipse, you can specify a command line option in Eclipse as follows:

1. From the Project menu, select Properties

   The Properties dialog appears.

2. In the left pane, expand C/C++ Build and select Settings.

   In the right pane the Settings appear.

3. On the Tool Settings tab, select Linker » Miscellaneous.

4. In the Additional options field, enter one or more command line options.

   Because Eclipse uses the control program, Eclipse automatically precedes the option with -Wl to pass the option via the control program directly to the linker.

Be aware that some command line options are not useful in Eclipse or just do not have any effect. For example, the option --keep-output-files keeps files after an error occurred. When you specify this option in Eclipse, it will have no effect because Eclipse always removes the output file after an error had occurred.

Short and long option names

Options can have both short and long names. Short option names always begin with a single minus (-) character, long option names always begin with two minus (--) characters. You can abbreviate long option names as long as it forms a unique name. You can mix short and long option names on the command line.

Options can have flags or suboptions. To switch a flag 'on', use a lowercase letter or a +longflag. To switch a flag off, use an uppercase letter or a -longflag. Separate longflags with commas. The following two invocations are equivalent:

ltc -mfkl test.o
ltc --map-file-format=+files,+link,+locate test.o

When you do not specify an option, a default value may become active.
Linker option: --case-insensitive

Menu entry

1. Select Linker » Miscellaneous.
2. Enable the option Link case insensitive.

Command line syntax

--case-insensitive

Default: case sensitive

Description

With this option you tell the linker not to distinguish between upper and lower case characters in symbols. By default the linker considers upper and lower case characters as different characters.

Assembly source files that are generated by the compiler must always be assembled and thus linked case sensitive. When you have written your own assembly code and specified to assemble it case insensitive, you must also link the .o file case insensitive.

Related information

Assembler option --case-insensitive
Linker option: --chip-output (-c)

Menu entry

1. Select Linker » Output Format.
2. Enable the option Generate Intel Hex format file and/or Generate S-records file.
3. Enable the option Create file for each memory chip.
4. Optionally, specify the Size of addresses.

Eclipse always uses the project name as the basename for the output file.

Command line syntax

```bash
--chip-output=[basename]:format[:addr_size],...
-c[basename]:format[:addr_size],...
```

You can specify the following formats:

- **IHEX**: Intel Hex
- **SREC**: Motorola S-records

The `addr_size` specifies the size of the addresses in bytes (record length). For Intel Hex you can use the values 1, 2 or 4 bytes (default). For Motorola-S you can specify: 2 (S1 records), 3 (S2 records) or 4 bytes (S3 records, default).

Description

With this option you specify the Intel Hex or Motorola S-record output format for loading into a PROM-programmer. The linker generates a file for each ROM memory defined in the LSL file, where sections are located:

```langs
memory memname
{  type=rom;  }
```

The name of the file is the name of the Eclipse project or, on the command line, the name of the memory device that was emitted with extension .hex or .sre. Optionally, you can specify a `basename` which prepends the generated file name.

The linker always outputs a debugging file in ELF/DWARF format and optionally an absolute object file in Intel Hex-format and/or Motorola S-record format.

Example

To generate Intel Hex output files for each defined memory, enter the following on the command line:

```bash
ltc --chip-output=myfile:IHEX test1.o
```
In this case, this generates the file *myfile_memname.hex*.

**Related information**

Linker option *--output* (Output file)
**Linker option: --define (-D)**

**Menu entry**

1. Select **Linker » Miscellaneous**.
2. Add the option **--define** to the **Additional options** field.

**Command line syntax**

```
--define=macro_name[=macro_definition]
-D macro_name[=macro_definition]
```

**Description**

With this option you can define a macro and specify it to the linker LSL file preprocessor. If you only specify a macro name (no macro definition), the macro expands as ‘1’.

You can specify as many macros as you like; just use the option **--define (-D)** multiple times. If the command line exceeds the limit of the operating system, you can define the macros in an option file which you then must specify to the linker with the option **--option-file (-f) file**.

The definition can be tested by the preprocessor with `#if`, `#ifdef` and `#ifndef`, for conditional locating.

**Example**

To define the RESET vector, which is used in the linker script file `tc1v1_3.lsl`, enter:

```
ltc test.o -otest.elf --lsl-file=tc1v1_3.lsl --define=RESET=0xa0000000
```

**Related information**

Linker option **--option-file** (Specify an option file)
Linker option: --diag

Menu entry

1. From the Window menu, select Show View » Other » Basic » Problems.
   
   *The Problems view is added to the current perspective.*

2. In the Problems view right-click on a message.
   
   *A popup menu appears.*

3. Select Detailed Diagnostics Info.
   
   *A dialog box appears with additional information.*

Command line syntax

`--diag=[format:] {all | nr,...}`

You can set the following output formats:

- **html**: HTML output.
- **rtf**: Rich Text Format.
- **text**: ASCII text.

Default format: text

Description

With this option you can ask for an extended description of error messages in the format you choose. The output is directed to stdout (normally your screen) and in the format you specify. You can specify the following formats: html, rtf or text (default). To create a file with the descriptions, you must redirect the output.

With the suboption all, the descriptions of all error messages are given. If you want the description of one or more selected error messages, you can specify the error message numbers, separated by commas.

With this option the linker does not link/locate any files.

Example

To display an explanation of message number 106, enter:

```bash
ltc --diag=106
```

This results in the following message and explanation:

```
E106: unresolved external: <message>
```

The linker could not resolve all external symbols.
This is an error when the incremental linking option is disabled. 
The <message> indicates the symbol that is unresolved.

To write an explanation of all errors and warnings in HTML format to file `lkerrors.html`, use redirection and enter:

```sh
ttc --diag=html:all > lkerrors.html
```

**Related information**

Section 8.10, *Linker Error Messages*
Linker option: --error-file

Menu entry
- 

Command line syntax

```
--error-file [=file]
```

Description

With this option the linker redirects error messages to a file. If you do not specify a filename, the error file is `ltc.elk`.

Example

To write errors to `errors.elk` instead of `stderr`, enter:

```
ltc --error-file=errors.elk test.o
```

Related information

Section 8.10, Linker Error Messages
**Linker option: --error-limit**

**Menu entry**

1. Select **Linker » Diagnostics**.
2. Enter a value in the **Maximum number of emitted errors** field.

**Command line syntax**

```
--error-limit=number
```

Default: 42

**Description**

With this option you tell the linker to only emit the specified maximum number of errors. When 0 (null) is specified, the linker emits all errors. Without this option the maximum number of errors is 42.

**Related information**

Section 8.10, *Linker Error Messages*
Linker option: --extern (-e)

Menu entry
-

Command line syntax

--extern=\textit{symbol},...

-e\textit{symbol},...

Description

With this option you force the linker to consider the given symbol as an undefined reference. The linker tries to resolve this symbol, either the symbol is defined in an object file or the linker extracts the corresponding symbol definition from a library.

This option is, for example, useful if the startup code is part of a library. Because your own application does not refer to the startup code, you can force the startup code to be extracted by specifying the symbol \_START as an unresolved external.

Example

Consider the following invocation:

\texttt{ltc mylib.a}

Nothing is linked and no output file will be produced, because there are no unresolved symbols when the linker searches through \texttt{mylib.a}.

\texttt{ltc --extern=\_START mylib.a}

In this case the linker searches for the symbol _START in the library and (if found) extracts the object that contains _START, the startup code. If this module contains new unresolved symbols, the linker looks again in \texttt{mylib.a}. This process repeats until no new unresolved symbols are found.

Related information

Section 8.3, Linking with Libraries
Linker option: --first-library-first

Menu entry
-

Command line syntax

--first-library-first

Description

When the linker processes a library it searches for symbols that are referenced by the objects and libraries processed so far. If the library contains a definition for an unresolved reference the linker extracts the object that contains the definition from the library.

By default the linker processes object files and libraries in the order in which they appear on the command line. If you specify the option --first-library-first the linker always tries to take the symbol definition from the library that appears first on the command line before scanning subsequent libraries.

This is for example useful when you are working with a newer version of a library that partially overlaps the older version. Because they do not contain exactly the same functions, you have to link them both. However, when a function is present in both libraries, you may want the linker to extract the most recent function.

Example

Consider the following example:

```
ltc --first-library-first a.a test.o b.a
```

If the file `test.o` calls a function which is both present in `a.a` and `b.a`, normally the function in `b.a` would be extracted. With this option the linker first tries to extract the symbol from the first library `a.a`.

Note that routines in `b.a` that call other routines that are present in both `a.a` and `b.a` are now also resolved from `a.a`.

Related information

Linker option --no-rescan (Rescan libraries to solve unresolved externals)
Linker option: --help (-?)

Menu entry

Command line syntax

--help[=item]

-?

You can specify the following arguments:

  options Show extended option descriptions

Description

Displays an overview of all command line options. When you specify the argument options you can list detailed option descriptions.

Example

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

ltc -?
ltc --help
ltc

To see a detailed description of the available options, enter:

ltc --help=options

Related information
Linker option: --hex-format

Menu entry
1. Select **Linker » Output Format**.
2. Enable the option **Generate Intel Hex format file**.
3. Select **Linker » Miscellaneous**.
4. Add the option **--hex-format** to the **Additional options** field.

Command line syntax

```
--hex-format=flag,...
```

You can set the following flag:

```
+/-start-address s/S Emit start address record
```

Default: **--hex-format=s**

Description

With this option you can specify to emit or omit the start address record from the hex file.

Related information

**Linker option --output** (Output file)
Linker option: --import-object

Menu entry

Command line syntax

```
--import-object=file,...
```

Description

With this option the linker imports a binary file containing raw data and places it in a section. The section name is derived from the filename, in which dots are replaced by an underscore. So, when importing a file called `my.jpg`, a section with the name `my_jpg` is created. In your application you can refer to the created section by using linker labels.

Related information

Section 8.5, Importing Binary Files
Linker option: --include-directory (-I)

Menu entry
-

Command line syntax

--include-directory=path,...
-I path,...

Description

With this option you can specify the path where your LSL include files are located. A relative path will be relative to the current directory.

The order in which the linker searches for LSL include files is:

1. The pathname in the LSL file and the directory where the LSL file is located (only for #include files that are enclosed in "")

2. The path that is specified with this option.

3. The default directory $(PRODDIR)\include.lsl.

Example

Suppose that your linker script file mylsl.lsl contains the following line:

#include "myinc.inc"

You can call the linker as follows:

ltc --include-directory=c:\proj\include --lsl-file=mylsl.lsl test.o

First the linker looks for the file myinc.inc in the directory where mylsl.lsl is located. If it does not find the file, it looks in the directory c:\proj\include (this option). Finally it looks in the directory $(PRODDIR)\include.lsl.

Related information

Linker option --lsl-file (Specify linker script file)
Linker option: --incremental (-r)

Menu entry
-

Command line syntax
--incremental
-r

Description
Normally the linker links and locates the specified object files. With this option you tell the linker only to link the specified files. The linker creates a linker output file .out. You then can link this file again with other object files until you have reached the final linker output file that is ready for locating.

In the last pass, you call the linker without this option with the final linker output file .out. The linker will now locate the file.

Example
In this example, the files test1.o, test2.o and test3.o are incrementally linked:

1. ltc --incremental test1.o test2.o --output=test.out
   test1.o and test2.o are linked
2. ltc --incremental test3.o test.out
   test3.o and test.out are linked, task1.out is created
3. ltc task1.out
   task1.out is located

Related information
Section 8.4, Incremental Linking
Linker option: --keep-output-files (-k)

Menu entry

Eclipse always removes the output files when errors occurred.

Command line syntax

--keep-output-files

-k

Description

If an error occurs during linking, the resulting output file may be incomplete or incorrect. With this option you keep the generated output files when an error occurs.

By default the linker removes the generated output file when an error occurs. This is useful when you use the make utility. If the erroneous files are not removed, the make utility may process corrupt files on a subsequent invocation.

Use this option when you still want to use the generated file. For example when you know that a particular error does not result in a corrupt object file, or when you want to inspect the output file, or send it to Altium support.

Related information

Linker option --warnings-as-errors (Treat warnings as errors)
Linker option: --library (-l)

Menu entry

1. Select **Linker » Libraries.**

   *The lower box shows the list of libraries that are linked with the project.*

2. To add a library, click on the **Add** button in the **Libraries** box.

3. Type or select a library (including its path).

4. Optionally, disable the option **Link default libraries.**

   Use the **Edit** and **Delete** button to change a library or to remove a library from the list.

Command line syntax

```
--library=name
-lname
```

Description

With this option you tell the linker to use system library `libname.a`, where `name` is a string. The linker first searches for system libraries in any directories specified with **--library-directory**, then in the directories specified with the environment variables `LIBTC1V1_2 / LIBTC1V1_3 / LIBTC2`, unless you used the option **--ignore-default-library-path**.

Example

To search in the system library `libc.a` (C library):

```
ltc test.o mylib.a --library=c
```

The linker links the file `test.o` and first looks in library `mylib.a` (in the current directory only), then in the system library `libc.a` to resolve unresolved symbols.

Related information

Linker option **--library-directory** (Additional search path for system libraries)

Section 8.3, *Linking with Libraries*
Linker option: --library-directory (-L) / --ignore-default-library-path

Menu entry

1. Select **Linker » Libraries**.

   *The Libraries search path box shows the directories that are added to the search path for library files.*

2. To define a new directory for the search path, click on the **Add** button in the **Library search path** box.

3. Type or select a path.

   Use the **Edit** and **Delete** button to change a path or to remove a path from the list.

Command line syntax

`--library-directory=path,...`
`-Lpath,...`

`--ignore-default-library-path`
`-L`

Description

With this option you can specify the path(s) where your system libraries, specified with the option **--library** (-l), are located. If you want to specify multiple paths, use the option **--library-directory** for each separate path.

The default path is $(PRODDIR)\lib.

If you specify only -L (without a pathname) or the long option **--ignore-default-library-path**, the linker will not search the default path and also not in the paths specified in the environment variables LIBTC1V1_2 / LIBTC1V1_3 / LIBTC2. So, the linker ignores steps 2 and 3 as listed below.

The priority order in which the linker searches for system libraries specified with the option **--library** (-l) is:

1. The path that is specified with the option **--library-directory**.
2. The path that is specified in the environment variables LIBTC1V1_2 / LIBTC1V1_3 / LIBTC2.
3. The default directory $(PRODDIR)\lib.
4. The processor specific directory, for example $(PRODDIR)\lib\tc1.

Example

Suppose you call the linker as follows:

`ltc test.o --library-directory=c:\mylibs --library=c`
First the linker looks in the directory `c:\mylibs` for library `libc.a` (this option). If it does not find the requested libraries, it looks in the directory that is set with the environment variables `LIBTC1V1_2 / LIBTC1V1_3 / LIBTC2`. Then the linker looks in the default directory `$(PRODDIR)\lib`.

**Related information**

**Linker option** `--library` (Link system library)

Section 8.3.1, *How the Linker Searches Libraries*
Linker option: --link-only

Menu entry

Command line syntax

--link-only

Description

With this option you suppress the locating phase. The linker stops after linking and informs you about unresolved references.

Related information

Control program option -cl (Stop after linking)
Linker option: --lsl-check

Menu entry

Command line syntax

--lsl-check

Description

With this option the linker just checks the syntax of the LSL file(s) and exits. No linking or locating is performed. Use the option --lsl-file to specify the name of the Linker Script File you want to test.

Related information

Linker option --lsl-file (Linker script file)

Linker option --lsl-dump (Dump LSL info)

Section 8.7, *Controlling the Linker with a Script*
Linker option: --lsl-dump

Menu entry

Command line syntax

`--lsl-dump[=file]`

Description

With this option you tell the linker to dump the LSL part of the map file in a separate file, independent of the option `--map-file` (generate map file). If you do not specify a filename, the file `ltc.ldf` is used.

Related information

Linker option `--map-file-format` (Map file formatting)
Linker option: --lsl-file (-d)

Menu entry

An LSL file can be generated when you create your project in Eclipse:

1. From the File menu, select File » New » Other... » TASKING C/C++ » TASKING VX-toolset for TriCore C/C++ Project.

   The New C/C++ Project wizard appears.

2. Fill in the project settings in each dialog and click Next > until the TriCore Project Settings appear.

3. Enable the option add Linker script file to the project and click Finish.

   Eclipse creates your project and the file project. lsl in the project directory.

Command line syntax

--lsl-file=file

-d file

Description

A linker script file contains vital information about the core for the locating phase of the linker. A linker script file is coded in LSL and contains the following types of information:

• the architecture definition describes the core’s hardware architecture.

• the memory definition describes the physical memory available in the system.

• the section layout definition describes how to locate sections in memory.

With this option you specify a linker script file to the linker. If you do not specify this option, the linker uses a default script file. You can specify the existing file target. lsl or the name of a manually written linker script file. You can use this option multiple times. The linker processes the LSL files in the order in which they appear on the command line.

Related information

Linker option --lsl-check (Check LSL file(s) and exit)

Section 8.7, Controlling the Linker with a Script
Linker option: --map-file (-M)

Menu entry
1. Select Linker » Map File.
2. Enable the option Generate map file.
3. Enable or disable the types of information to be included.

Command line syntax

```
--map-file [=file]
-M [file]
```

Default: no map file is generated

Description

With this option you tell the linker to generate a linker map file. If you do not specify a filename and you specified the option --output, the linker uses the same basename as the output file with the extension .map. If you did not specify the option --output, the linker uses the file task1.map. Eclipse names the .map file after the project.

A linker map file is a text file that shows how the linker has mapped the sections and symbols from the various object files (.o) to the linked object file. A locate part shows the absolute position of each section. External symbols are listed per space with their absolute address, both sorted on symbol and sorted on address.

Related information

Linker option --map-file-format (Format map file)

Section 13.2, Linker Map File Format
Linker option: --map-file-format (-m)

Menu entry

1. Select Linker » Map File.
2. Enable the option Generate map file.
3. Enable or disable the types of information to be included.

Command line syntax

```
--map-file-format=flag,...

-mflags
```

You can set the following flags:

- `+/-callgraph`  
  Include call graph information
- `+/-files`  
  Include processed files information
- `+/-invocation`  
  Include invocation and tool information
- `+/-link`  
  Include link result information
- `+/-locate`  
  Include locate result information
- `+/-memory`  
  Include memory usage information
- `+/-nonalloc`  
  Include non alloc information
- `+/-overlay`  
  Include overlay information
- `+/-statics`  
  Include module local symbols information
- `+/-crossref`  
  Include cross references information
- `+/-lsl`  
  Include processor and memory information
- `+/-rules`  
  Include locate rules

Use the following options for predefined sets of flags:

```
--map-file-format=0 -m0  Link information
Alias for -mcfikLMNoQrSU

--map-file-format=1 -m1  Locate information
Alias for -mCfIKMNQrSU

--map-file-format=2 -m2  Most information
Alias for -mcfiklMNQrSu
```

Default: --map-file-format=2

Description

With this option you specify which information you want to include in the map file.
On the command line you must use this option in combination with the option `--map-file (-M)`.

**Related information**

- Linker option `--map-file` (Generate map file)
- Section 13.2, *Linker Map File Format*
Linker option: --misra-c-report

Menu entry

Command line syntax

`--misra-c-report [=file]`

Description

With this option you tell the linker to create a MISRA-C Quality Assurance report. This report lists the various modules in the project with the respective MISRA-C settings at the time of compilation. If you do not specify a filename, the file `basename.mcr` is used.

Related information

C compiler option `--misrac` (MISRA-C checking)
Linker option: --munch

Menu entry
-  

Command line syntax
  --munch

Description
With this option you tell the linker to activate the muncher in the pre-locate phase.

Related information
Section 5.1.2, The Muncher Phase
Linker option: --non-romable

Menu entry

Command line syntax
--non-romable

Description
With this option you tell the linker that the application is not romable. The linker will locate all ROM sections in RAM. A copy table is generated and is located in RAM. When the application is started, that data and BSS sections are re-initialized.

Related information
Linker option: --no-rescan

Menu entry

1. Select Linker » Libraries.
2. Disable the option Rescan libraries to solve unresolved externals.

Command line syntax

--no-rescan

Description

When the linker processes a library it searches for symbol definitions that are referenced by the objects and libraries processed so far. If the library contains a definition for an unresolved reference the linker extracts the object that contains the definition from the library. The linker processes object files and libraries in the order in which they appear on the command line.

When all objects and libraries are processed the linker checks if there are unresolved symbols left. If so, the default behavior of the linker is to rescan all libraries in the order given at the command line. The linker stops rescanning the libraries when all symbols are resolved, or when the linker could not resolve any symbol(s) during the rescan of all libraries. Notice that resolving one symbol may introduce new unresolved symbols.

With this option, you tell the linker to scan the object files and libraries only once. When the linker has not resolved all symbols after the first scan, it reports which symbols are still unresolved. This option is useful if you are building your own libraries. The libraries are most efficiently organized if the linker needs only one pass to resolve all symbols.

Related information

Linker option --first-library-first (Scan libraries in given order)
Linker option: --no-rom-copy (-N)

Menu entry
-

Command line syntax

--no-rom-copy

-N

Description

With this option the linker will not generate a ROM copy for data sections. A copy table is generated and contains entries to clear BSS sections. However, no entries to copy data sections from ROM to RAM are placed in the copy table.

The data sections are initialized when the application is downloaded. The data sections are not re-initialized when the application is restarted.

Related information
-

Linker option: --no-warnings (-w)

Menu entry

1. Select Linker » Diagnostics.

   The Suppress warnings box shows the warnings that are currently suppressed.

2. To suppress a warning, click on the Add button in the Suppress warnings box.

3. Enter the numbers, separated by commas, of the warnings you want to suppress (for example 135, 136). Or you can use the Add button multiple times.

4. To suppress all warnings, enable the option Suppress all warnings.

Use the Edit and Delete button to change a warning number or to remove a number from the list.

Command line syntax

--no-warnings [=number,...]

-w [number,...]

Description

With this option you can suppresses all warning messages or specific warning messages.

On the command line this option works as follows:

• If you do not specify this option, all warnings are reported.

• If you specify this option but without numbers, all warnings are suppressed.

• If you specify this option with a number, only the specified warning is suppressed. You can specify the option --no-warnings=number multiple times.

Example

To suppress warnings 135 and 136, enter:

ltc --no-warnings=135,136 test.o

Related information

Linker option --warnings-as-errors (Treat warnings as errors)
Linker option: --optimize (-O)

Menu entry
1. Select **Linker » Optimization**.
2. Select one or more of the following options:
   - Delete unreferenced sections
   - Use a ‘first-fit decreasing’ algorithm
   - Compress copy table
   - Delete duplicate code
   - Delete duplicate data

Command line syntax

```
--optimize=flag,...
-Oflags
```

You can set the following flags:

```
+/-delete-unreferenced-sections c/C
+/-first-fit-decreasing i/L
+/-copytable-compression t/T
+/-delete-duplicate-code x/X
+/-delete-duplicate-data y/Y
```

Delete unreferenced sections from the output file
Use a ‘first-fit decreasing’ algorithm to locate unrestricted sections in memory
Emit smart restrictions to reduce copy table size
Delete duplicate code sections from the output file
Delete duplicate constant data from the output file

Use the following options for predefined sets of flags:

```
--optimize=0 -O0 No optimization
       Alias for -OCLTXY
--optimize=1 -O1 Default optimization
       Alias for -OcLtxy
--optimize=2 -O2 All optimizations
       Alias for -Ocltxy
```

Default: `--optimize=1`
Description

With this option you can control the level of optimization.

Related information

For details about each optimization see Section 8.6, Linker Optimizations.
Linker option: --option-file (-f)

Menu entry
1. Select **Linker » Miscellaneous**.
2. Add the option **--option-file** to the **Additional options** field.

   *Be aware that the options in the option file are added to the linker options you have set in the other pages. Only in extraordinary cases you may want to use them in combination.*

Command line syntax

```
--option-file=file,...
-f file,...
```

Description

This option is primarily intended for command line use. Instead of typing all options on the command line, you can create an option file which contains all options and flags you want to specify. With this option you specify the option file to the linker.

Use an option file when the command line would exceed the limits of the operating system, or just to store options and save typing.

Option files can also be generated on the fly, for example by the make utility. You can specify the option **--option-file** multiple times.

Format of an option file

- Multiple arguments on one line in the option file are allowed.
- To include whitespace in an argument, surround the argument with single or double quotes.
- If you want to use single quotes as part of the argument, surround the argument by double quotes and vice versa:
  
  "This has a single quote ' embedded"

  'This has a double quote " embedded'

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

- When a text line reaches its length limit, use a \ to continue the line. Whitespace between quotes is preserved.
  
  "This is a continuation \\ line"

  -> "This is a continuation line"
• It is possible to nest command line files up to 25 levels.

**Example**

Suppose the file `myoptions` contains the following lines:

```plaintext
--map-file=my.map               (generate a map file)
test.o                          (input file)
--library-directory=c:\mylibs   (additional search path for system libraries)
```

Specify the option file to the linker:

```plaintext
ltc --option-file=myoptions
```

This is equivalent to the following command line:

```plaintext
ltc --map-file=my.map test.o --library-directory=c:\mylibs
```

**Related information**

-
Linker option: --output (-o)

Menu entry

1. Select Linker » Output Format.
2. Enable one or more output formats.

   For some output formats you can specify a number of suboptions.

   Eclipse always uses the project name as the basename for the output file.

Command line syntax

```
--output=[filename][:format[:addr_size],[space_name]]...
-o[filename][:format[:addr_size],[space_name]]...
```

You can specify the following formats:

- **ELF**    ELF/DWARF
- **IHEX**   Intel Hex
- **SREC**   Motorola S-records

Description

By default, the linker generates an output file in ELF/DWARF format, with the name `task1.elf`.

With this option you can specify an alternative `filename`, and an alternative output `format`. The default output format is the format of the first input file.

You can use the `--output` option multiple times. This is useful to generate multiple output formats. With the first occurrence of the `--output` option you specify the basename (the filename without extension), which is used for subsequent `--output` options with no filename specified. If you do not specify a filename, or you do not specify the `--output` option at all, the linker uses the default basename `taskn`.

IHEX and SREC formats

If you specify the Intel Hex format or the Motorola S-records format, you can use the argument `addr_size` to specify the size of addresses in bytes (record length). For Intel Hex you can use the values: 1, 2, and 4 (default). For Motorola S-records you can specify: 2 (S1 records), 3 (S2 records, default) or 4 bytes (S3 records).

With the argument `space_name` you can specify the name of the address space. The name of the output file will be filename with the extension `.hex` or `.sre` and contains the code and data allocated in the specified space. If they exist, any other address spaces are also emitted whereas their output files are named `filename_space_name` with the extension `.hex` or `.sre`.

If you do not specify `space_name`, or you specify a non-existing space, the default address space is filled in.
Use option **--chip-output** (-c) to create Intel Hex or Motorola S-record output files for each chip defined in the LSL file (suitable for loading into a PROM-programmer).

**Example**

To create the output file `myfile.hex` of the address space named `linear`, enter:

```
ltc test.o --output=myfile.hex:IHEX:2,linear
```

If they exist, any other address spaces are emitted as well and are named `myfile_spacename.hex`.

**Related information**

- **Linker option --chip-output** (Generate an output file for each chip)
- **Linker option --hex-format** (Specify Hex file format settings)
Linker option: --print-mangled-symbols (-P)

Menu entry
-

Command line syntax

--print-mangled-symbols
-P

Description

C++ compilers generate unreadable symbol names. These symbols cannot easily be related to your C++ source file anymore. Therefore the linker will by default decode these symbols conform the IA64 ABI when printed to stdout. With this option you can override this default setting and print the mangled names instead.

Related information
-

550
Linker option: --strip-debug (-S)

Menu entry

1. Select Linker » Miscellaneous.
2. Enable the option Strip symbolic debug information.

Command line syntax

--strip-debug

-s

Description

With this option you specify not to include symbolic debug information in the resulting output file.

Related information

-
**Linker option: --user-provided-initialization-code (-i)**

**Menu entry**

1. Select **Linker » Miscellaneous**.
2. Enable the option **Do not use standard copy table for initialization**.

**Command line syntax**

```
--user-provided-initialization-code
-i
```

**Description**

It is possible to use your own initialization code, for example, to save ROM space. With this option you tell the linker *not* to generate a copy table for initialize/clear sections. Use linker labels in your source code to access the positions of the sections when located.

If the linker detects references to the TASKING initialization code, an error is emitted: it is either the TASKING initialization routine or your own, not both.

Note that the options **--no-rom-copy** and **--non-romable**, may vary independently. The 'copytable-compression' optimization (**--optimize=t**) is automatically disabled when you enable this option.

**Related information**

- **Linker option --no-rom-copy** (Do not generate ROM copy)
- **Linker option --non-romable** (Application is not romable)
- **Linker option --optimize** (Specify optimization)
Linker option: --verbose (-v) / --extra-verbose (-vv)

Menu entry
-

Command line syntax

--verbose / --extra-verbose
-v / -vv

Description

With this option you put the linker in verbose mode. The linker prints the link phases while it processes the files. In the extra verbose mode, the linker also prints the filenames and it shows which objects are extracted from libraries. With this option you can monitor the current status of the linker.

Related information
-
Linker option: --version (-V)

Menu entry

Command line syntax

--version

-v

Description

Display version information. The linker ignores all other options or input files.

Example

ltc --version

The linker does not link any files but displays the following version information:

TASKING VX-toolset for TriCore: object linker vx.yrz Build nnn
Copyright 2002-year Altium BV Serial# 00000000

Related information

-
Linker option: --warnings-as-errors

Menu entry
1. Select Global Options.
2. Enable the option Treat warnings as errors.

Command line syntax

--warnings-as-errors[=number, ...]

Description

When the linker detects an error or warning, it tries to continue the link process and reports other errors and warnings. When you use this option without arguments, you tell the linker to treat all warnings as errors. This means that the exit status of the linker will be non-zero after the detection of one or more linker warnings. As a consequence, the linker will not produce any output files.

You can also limit this option to specific warnings by specifying a comma-separated list of warning numbers.

Related information

Linker option --no-warnings ( Suppress some or all warnings)
11.5. Control Program Options

The control program cctc facilitates the invocation of the various components of the TriCore toolset from a single command line.

Options in Eclipse versus options on the command line

Eclipse invokes the compiler, assembler and linker via the control program. Therefore, it uses the syntax of the control program to pass options and files to the tools. The control program processes command line options either by itself, or, when the option is unknown to the control program, it looks whether it can pass the option to one of the other tools. However, for directly passing an option to the compiler, assembler or linker, it is recommended to use the control program options --pass-c, --pass-assembler, --pass-linker.

See the previous sections for details on the options of the tools.

Short and long option names

Options can have both short and long names. Short option names always begin with a single minus (-) character, long option names always begin with two minus (-- characters. You can abbreviate long option names as long as it forms a unique name. You can mix short and long option names on the command line.

Options can have flags or suboptions. To switch a flag 'on', use a lowercase letter or a +longflag. To switch a flag off, use an uppercase letter or a -longflag. Separate longflags with commas. The following two invocations are equivalent:

```
cctc -Wc=Oac test.c
```
```
cctc --pass-c=--optimize=-coalesce,+cse test.c
```

When you do not specify an option, a default value may become active.
Control program option: --address-size

Menu entry

1. Select **Linker» Output Format**.
2. Enable the option **Generate Intel Hex format file** and/or **Generate S-records file**.
3. Specify the **Size of addresses**.

   *Eclipse always uses the project name as the basename for the output file.*

Command line syntax

```
--address-size=addr_size
```

Description

If you specify IHEX or SREC with the control option **--format**, you can additionally specify the record length to be emitted in the output files.

With this option you can specify the size of the addresses in bytes (record length). For Intel Hex you can use the values 1, 2 or 4 bytes (default). For Motorola-S you can specify: 2 (S1 records), 3 (S2 records) or 4 bytes (S3 records, default).

If you do not specify *addr_size*, the default address size is generated.

Example

To create the SREC file `test.sre` with S1 records, type:

```
cctc --format=SREC --address-size=2 test.c
```

Related information

- Control program option **--format** (Set linker output format)
- Control program option **--output** (Output file)
Control program option: --case-insensitive

Menu entry

1. Select **Assembler » Symbols**.
2. Enable the option **Case insensitive identifiers**.

Command line syntax

```bash
--case-insensitive
```

Default: case sensitive

Description

With this option you tell the assembler not to distinguish between upper and lower case characters. By default the assembler considers upper and lower case characters as different characters.

Assembly source files that are generated by the compiler must always be assembled case sensitive. When you are writing your own assembly code, you may want to specify the case insensitive mode.

Example

When assembling case insensitive, the label `LabelName` is the same label as `labelname`.

```bash
cctc --case-insensitive test.src
```

Related information

**Assembler option --case-insensitive**

**Assembler control $CASE**
Control program option: --check

Menu entry
-

Command line syntax

--check

Description

With this option you can check the source code for syntax errors, without generating code. This saves time in developing your application because the code will not actually be compiled.

The compiler/assembler reports any warnings and/or errors.

This option is available on the command line only.

Related information

C compiler option --check (Check syntax)

Assembler option --check (Check syntax)
Control program option: --core

Menu entry
1. Select C/C++ Compiler » Miscellaneous.
2. Add the option --core to the Additional options field.

Command line syntax

```
--core=core
```

You can specify the following core arguments:

```
   tc1.2     TriCore 1.2 architecture
   tc1.3     TriCore 1.3 architecture
   tc1.3.1   TriCore 1.3.1 architecture, TriCore 1.3.1 instructions may be generated
   tc2       TriCore 2 architecture, TriCore 2 instructions may be generated
```

Default: derived from --cpu, if used, otherwise tc1.3

Description

With this option you specify the core architecture for a custom target processor for which you create your application. By default the TriCore toolset derives the core from the processor you selected.

With `--core=tc1.3.1`, the compiler can generate TriCore 1.3.1 instructions in the assembly file. The macro `__TC131__` is defined in the C source file.

With `--core=tc2`, the compiler can generate TriCore 2 instructions in the assembly file. The macro `__TC2__` is defined in the C source file.

If you select a valid target processor (command line option --cpu (-C)), the core is automatically set, based on the chosen target processor.

Example

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

```
cctc --core=tc2 test.c
```

Related information

Control program option --cpu (Select processor)
Control program option --fpu-present (FPU present)
Control program option --mmu-present (MMU present)
Control program option: --cpu (-C)

Menu entry

1. Expand C/C++ Build and select Processor.
2. From the Processor Selection list, select a processor.

Command line syntax

--cpu=cpu
-C cpu

Description

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

Based on this option the compiler always includes the special function register file regcpu.sfr, and the assembler includes the file regcpu.def, unless you specify option --no-tasking-sfr.

Based on the target processor it is automatically detected whether a MMU or FPU-unit is present and whether the architecture is a TriCore2. This means you do not have to specify the options --mmu-present, --fpu-present and --core=tc2 explicitly when one of the supported derivatives is selected.

Example

To generate the file test.elf for the TC1165 processor, enter:

```c
ccctc --cpu=tc1165 test.c
```

Related information

Control program option --no-tasking-sfr (Do not include SFR file)

Section 1.3.2, Accessing Hardware from C
Control program option: --create (-c)

Menu entry
-

Command line syntax

--create [=stage]
-c [stage]

You can specify the following stages:

- intermediate-c  c     Stop after C++ files are compiled to intermediate C files (.ic)
- relocatable     l     Stop after the files are linked to a linker object file (.out)
- mil             m     Stop after C++ files or C files are compiled to MIL (.mil)
- object          o     Stop after the files are assembled to objects (.o)
- preprocess       p     Stop after C++ files or C files are preprocessed
- assembly        s     Stop after C++ files or C files are compiled to assembly (.src)

Default (without flags): --create=object

Description

Normally the control program generates an absolute object file of the specified output format from the file you supplied as input. With this option you tell the control program to stop after a certain number of phases.

Example

To generate the object file test.o:

cctc --create test.c

The control program stops after the file is assembled. It does not link nor locate the generated output.

Related information

Control program option --link-only (Link only, no locating)
Control program option: --debug-info (-g)

Menu entry

1. Select C/C++ Compiler » Debugging.

2. To generate symbolic debug information, select Default, Call-frame only or Full. To disable the generation of debug information, select None.

Command line syntax

--debug-info

-g

Description

With this option you tell the control program to include debug information in the generated object file.

The control program passes the option `--debug-info (-g)` to the C compiler and calls the assembler with `--debug-info=+smart,+local (-gsl)`.

Related information

C compiler option --debug-info (Generate symbolic debug information)

Assembler option --debug-info (Generate symbolic debug information)
Control program option: --define (-D)

Menu entry

1. Select **C/C++ Compiler » Preprocessing** and/or **Assembler » Preprocessing**.  
   The Defined symbols box right-below shows the symbols that are currently defined.

2. To define a new symbol, click on the **Add** button in the **Defined symbols** box.

3. Type the symbol definition (for example, `demo=1`)

   Use the **Edit** and **Delete** button to change a macro definition or to remove a macro from the list.

Command line syntax

```
--define=macro_name[=macro_definition]
-Dmacro_name[=macro_definition]
```

Description

With this option you can define a macro and specify it to the preprocessor. If you only specify a macro name (no macro definition), the macro expands as ‘1’.

You can specify as many macros as you like. Simply use the **Add** button to add new macro definitions.

On the command line, use the option **--define (-D)** multiple times. If the command line exceeds the limit of the operating system, you can define the macros in an option file which you then must specify to the compiler with the option **--option-file (-f) file**.

Defining macros with this option (instead of in the C source) is, for example, useful to compile conditional C source as shown in the example below.

The control program passes the option **--define (-D)** to the compiler and the assembler.

Example

Consider the following C program with conditional code to compile a demo program and a real program:

```c
void main( void )
{
  #if DEMO
    demo_func();   /* compile for the demo program */
  #else
    real_func();   /* compile for the real program */
  #endif
}
```

You can now use a macro definition to set the DEMO flag:
cctc --define=DEMO test.c
cctc --define=DEMO=1 test.c

Note that both invocations have the same effect.

The next example shows how to define a macro with arguments. Note that the macro name and definition are placed between double quotes because otherwise the spaces would indicate a new option.

cctc --define="MAX(A,B)=((A) > (B) ? (A) : (B))" test.c

**Related information**

- Control program option **--undefine** (Remove preprocessor macro)
- Control program option **--option-file** (Specify an option file)
Control program option: --diag

Menu entry

1. From the Window menu, select Show View » Other » General » Problems.
   
   The Problems view is added to the current perspective.
2. In the Problems view right-click on a message.
   
   A popup menu appears.
3. Select Detailed Diagnostics Info.
   
   A dialog box appears with additional information.

Command line syntax

```
--diag=[format:]{all | nr,...}
```

You can set the following output formats:

- `html`  HTML output.
- `text`   ASCII text.

Default format: text

Description

With this option you can ask for an extended description of error messages in the format you choose. The output is directed to stdout (normally your screen) and in the format you specify. You can specify the following formats: `html`, `rtf` or `text` (default). To create a file with the descriptions, you must redirect the output.

With the suboption `all`, the descriptions of all error messages are given. If you want the description of one or more selected error messages, you can specify the error message numbers, separated by commas.

Example

To display an explanation of message number 103, enter:

```
cctc --diag=103
```

This results in message 103 with explanation.

To write an explanation of all errors and warnings in HTML format to file `ccerrors.html`, use redirection and enter:

```
cctc --diag=html:all > ccerrors.html
```
Related information

Section 4.8, *C Compiler Error Messages*
Control program option: --dry-run (-n)

Menu entry
-

Command line syntax

--dry-run
-n

Description

With this option you put the control program in verbose mode. The control program prints the invocations of the tools it would use to process the files without actually performing the steps.

Related information

Control program option --verbose (Verbose output)
Control program option: --error-file

Menu entry
-

Command line syntax
--error-file

Description
With this option the control program tells the compiler, assembler and linker to redirect error messages to a file.

Example
To write errors to error files instead of stderr, enter:

cctc --error-file test.c

Related information
Control Program option --warnings-as-errors (Treat warnings as errors)
Control program option: --exceptions

Menu entry

1. Select C/C++ Compiler » Language.
2. Enable the option Support for exception handling.

Command line syntax

--exceptions

Description

With this option you enable support for exception handling in the C++ compiler.

Related information

-
Control program option: --force-c

Menu entry

Command line syntax

--force-c

Description

With this option you tell the control program to treat all .cc files as C files instead of C++ files. This means that the control program does not call the C++ compiler and forces the linker to link C libraries.

Related information

Control program option --force-c++ (Force C++ compilation and linking)
Control program option: --force-c++

Menu entry

Eclipse always uses this option for a C++ project.

Command line syntax

--force-c++

Description

With this option you tell the control program to treat all .c files as C++ files instead of C files. This means that the control program calls the C++ compiler prior to the C compiler and forces the linker to link C++ libraries.

Related information

Control program option --force-c (Treat C++ files as C files)
Control program option: --force-munch

Menu entry

Eclipse always uses this option for a C++ project.

Command line syntax

--force-munch

Description

With this option you force the control program to activate the muncher in the pre-locate phase.

Related information

-
Control program option: --force-prelink

Menu entry

Eclipse always uses this option for a C++ project.

Command line syntax

```
--force-prelink
```

Description

With this option you force the control program to invoke the C++ pre-linker.

Related information

-
Control program option: --format

Menu entry

1. Select Linker » Output Format.
2. Enable the option Generate Intel Hex format file and/or Generate S-records file.
3. Optionally, specify the Size of addresses.

   Eclipse always uses the project name as the basename for the output file.

Command line syntax

--format=format

You can specify the following formats:

- ELF: ELF/DWARF
- IHEX: Intel Hex
- SREC: Motorola S-records

Description

With this option you specify the output format for the resulting (absolute) object file. The default output format is ELF/DWARF, which can directly be used by the debugger.

If you choose IHEX or SREC, you can additionally specify the address size of the chosen format (option --address-size).

Example

To generate a Motorola S-record output file:

cctc --format=SREC test1.c test2.c --output=test.sre

Related information

Control program option --address-size (Set address size for linker IHEX/SREC files)
Control program option --output (Output file)
Linker option --chip-output (Generate an output file for each chip)
Control program option: --fp-trap

Menu entry

1. Select **Linker » Libraries**.
2. Enable the option **Use trapped floating-point library**.

Command line syntax

```
--fp-trap
```

Description

By default the control program uses the non-trapping floating-point library (**libfp.a**). With this option you tell the control program to use the trapping floating-point library (**libfpt.a**).

If you use the trapping floating-point library, exceptional floating-point cases are intercepted and can be handled separately by an application defined exception handler. Using this library decreases the execution speed of your application.

Related information

**C compiler option** --fp-trap (Allow trapping of floating-point exceptions)

Section 8.3, *Linking with Libraries*
Control program option: --fpu-present

Menu entry
1. Expand C/C++ Build and select Processor.
2. From the Processor Selection list, select a processor that has an FPU.

Command line syntax

--fpu-present

Description

With this option the compiler can generate single precision floating-point instructions in the assembly file. When you select this option, the macro __FPU__ is defined in the C source file.

This option automatically sets the option --no-double, which treats ‘double’ as ‘float’, unless you overrule it with option --use-double-precision-fp.

If you select a valid target processor (command line option --cpu (-C)), this option is automatically set, based on the chosen target processor.

Example

To allow the use of floating-point unit (FPU) instructions in the assembly code and treat ‘double’ as ‘float’, enter:

cctc --fpu-present test.c

Related information

Control program option --use-double-precision-fp (Do not replace doubles with floats)

Control program option --cpu (Select processor)
Control program option: --help (-?)

Menu entry

Command line syntax

--help[=item]

-?

You can specify the following argument:

options Show extended option descriptions

Description

Displays an overview of all command line options. When you specify the argument options you can list detailed option descriptions.

Example

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

cctc -?
cctc --help
cctc
cctc

to see a detailed description of the available options, enter:

cctc --help=options

Related information

-
Control program option: --include-directory (-I)

Menu entry

1. Select C/C++ Compiler » Include Paths.

   The Include paths box shows the directories that are added to the search path for include files.

2. To define a new directory for the search path, click on the Add button in the Include paths box.

3. Type or select a path.

   Use the Edit and Delete button to change a path or to remove a path from the list.

Command line syntax

```
--include-directory= path,...

-I path,...
```

Description

With this option you can specify the path where your include files are located. A relative path will be relative to the current directory.

The control program passes this option to the compiler and the assembler.

Example

Suppose that the C source file test.c contains the following lines:

```
#include <stdio.h>
#include "myinc.h"
```

You can call the control program as follows:

```
cctc --include-directory=myinclude test.c
```

First the compiler looks for the file stdio.h in the directory myinclude relative to the current directory. If it was not found, the compiler searches in the environment variable and then in the default include directory.

The compiler now looks for the file myinc.h in the directory where test.c is located. If the file is not there the compiler searches in the directory myinclude. If it was still not found, the compiler searches in the environment variable and then in the default include directory.

Related information

C compiler option --include-directory (Add directory to include file search path)

C compiler option --include-file (Include file at the start of a compilation)
Control program option: --instantiate

Menu entry

1. Select C/C++ Compiler » Miscellaneous.
2. Select an instantiation mode in the Instantiation mode of external template entities box.

Command line syntax

`--instantiate=mode`

You can specify the following modes:

- none
- used
- all
- local

Default: `--instantiate=used`

Description

Control instantiation of external template entities. External template entities are external (that is, non-inline and non-static) template functions and template static data members. The instantiation mode determines the template entities for which code should be generated based on the template definition. Normally, when a file is compiled, template entities are instantiated wherever they are used (the linker will discard duplicate definitions). The overall instantiation mode can, however, be changed with this option. You can specify the following modes:

- **none**: Do not automatically create instantiations of any template entities.
- **used**: Instantiate those template entities that were used in the compilation. This will include all static data members for which there are template definitions. This is the default.
- **all**: Instantiate all template entities declared or referenced in the compilation unit. For each fully instantiated template class, all of its member functions and static data members will be instantiated whether or not they were used. Non-member template functions will be instantiated even if the only reference was a declaration. Similar to `--instantiate=used` except that the functions are given internal linkage. This is intended to provide a very simple mechanism for those getting started with templates. The compiler will instantiate the functions that are used in each compilation unit as local functions, and the program will link and run correctly (barring problems due to multiple copies of local static variables). However, one may end up with many copies of the instantiated functions, so this is not suitable for production use.

You cannot use `--instantiate=local` in conjunction with automatic template instantiation.
Related information

Control program option `--no-auto-instantiation` (Disable automatic C++ instantiation)

Section 2.5, *Template Instantiation*
Control program option: --integer-enumeration

Menu entry

1. Select C/C++ Compiler » Language.
2. Enable the option Treat enumerated types always as integer.

Command line syntax

--integer-enumeration

Description

Normally the compiler treats enumerated types as the smallest data type possible (char or short instead of int). This reduces code size. With this option the compiler always treats enum-types as int as defined in the ISO C99 standard.

Related information

Section 1.1, Data Types
Control program option: --io-streams

Menu entry

1. Select C/C++ Compiler » Language.
2. Enable the option Support for I/O streams.

Command line syntax

--io-streams

Description

As I/O streams require substantial resources they are disabled by default. Use this option to enable I/O streams support in the C++ library.

This option also enables exception handling.

Related information
Control program option: --iso

Menu entry
1. Select C/C++ Compiler » Language.
2. From the Comply to C standard list, select ISO C99 or ISO C90.

Command line syntax

--iso={90|99}

Default: --iso=99

Description
With this option you select the ISO C standard. C90 is also referred to as the "ANSI C standard". C99 refers to the newer ISO/IEC 9899:1999 (E) standard. C99 is the default.

Independent of the chosen ISO standard, the control program always links libraries with C99 support.

Example
To select the ISO C90 standard on the command line:

cctc --iso=90 test.c

Related information
C compiler option --iso (ISO C standard)
Control program option: --keep-output-files (-k)

Menu entry

Eclipse always removes generated output files when an error occurs.

Command line syntax

--keep-output-files
-k

Description

If an error occurs during the compilation, assembling or linking process, the resulting output file may be incomplete or incorrect. With this option you keep the generated output files when an error occurs.

By default the control program removes generated output files when an error occurs. This is useful when you use the make utility. If the erroneous files are not removed, the make utility may process corrupt files on a subsequent invocation.

Use this option when you still want to use the generated files. For example when you know that a particular error does not result in a corrupt file, or when you want to inspect the output file, or send it to Altium support.

The control program passes this option to the compiler, assembler and linker.

Example

cctc --keep-output-files test.c

When an error occurs during compiling, assembling or linking, the erroneous generated output files will not be removed.

Related information

C compiler option --keep-output-files
Assembler option --keep-output-files
Linker option --keep-output-files
Control program option: --keep-temporary-files (-t)

Menu entry

1. Select Global Options.
2. Enable the option Keep temporary files.

Command line syntax

--keep-temporary-files
-t

Description

By default, the control program removes intermediate files like the .src file (result of the compiler phase) and the .o file (result of the assembler phase).

With this option you tell the control program to keep temporary files it generates during the creation of the absolute object file.

Example

```
cctc --keep-temporary-files test.c
```

The control program keeps all intermediate files it generates while creating the absolute object file test.elf.

Related information
Control program option: --library (-l)

Menu entry

1. Select **Linker » Libraries**.

   *The lower box shows the list of libraries that are linked with the project.*

2. To add a library, click on the **Add** button in the **Libraries** box.

3. Type or select a library (including its path).

4. Optionally, disable the option **Link default libraries**.

   Use the **Edit** and **Delete** button to change a library or to remove a library from the list.

Command line syntax

```
--library=\textit{name}
-l\textit{name}
```

Description

With this option you tell the linker via the control program to use system library \texttt{lib}\textit{name}.a, where \textit{name} is a string. The linker first searches for system libraries in any directories specified with \texttt{--library-directory}, then in the directories specified with the environment variables \texttt{LIBTC1V1_2 / LIBTC1V1_3 / LIBTC2}, unless you used the option \texttt{--ignore-default-library-path}.

Example

To search in the system library \texttt{libc.a} (C library):

```
cctc test.o mylib.a --library=\texttt{c}
```

The linker links the file \texttt{test.o} and first looks in library \texttt{mylib.a} (in the current directory only), then in the system library \texttt{libc.a} to resolve unresolved symbols.

Related information

- Control program option **--no-default-libraries** (Do not link default libraries)
- Control program option **--library-directory** (Additional search path for system libraries)
- Section 8.3, *Linking with Libraries*
Control program option: --library-directory (-L) / --ignore-default-library-path

Menu entry

1. Select **Linker » Libraries**.
   
   *The Library search path box shows the directories that are added to the search path for library files.*

2. To define a new directory for the search path, click on the **Add** button in the **Library search path** box.

3. Type or select a path.

   Use the **Edit** and **Delete** button to change a path or to remove a path from the list.

Command line syntax

```plaintext
--library-directory=\path, ...
-L \path, ...
--ignore-default-library-path
-L
```

Description

With this option you can specify the path(s) where your system libraries, specified with the option **--library** (-l), are located. If you want to specify multiple paths, use the option **--library-directory** for each separate path.

The default path is `$(PRODDIR)\lib`.

If you specify only -L (without a pathname) or the long option **--ignore-default-library-path**, the linker will not search the default path and also not in the paths specified in the environment variables `LIBTC1V1_2 / LIBTC1V1_3 / LIBTC2`. So, the linker ignores steps 2 and 3 as listed below.

The priority order in which the linker searches for system libraries specified with the option **--library** (-l) is:

1. The path that is specified with the option **--library-directory**.
2. The path that is specified in the environment variables `LIBTC1V1_2 / LIBTC1V1_3 / LIBTC2`.
3. The default directory `$(PRODDIR)\lib\tc1` or `$(PRODDIR)\lib\tc2`.

Example

Suppose you call the control program as follows:

```plaintext
cctc test.c --library-directory=c:\mylibs --library=c
```
First the linker looks in the directory `c:\mylibs` for library `libc.a` (this option). If it does not find the requested libraries, it looks in the directory that is set with the environment variables `LIBTC1V1_2 / LIBTC1V1_3 / LIBTC2`. Then the linker looks in the default directory `${PRODDIR}\lib\tc1` or `${PRODDIR}\lib\tc2` for libraries.

**Related information**

Control program option `--library` (Link system library)

Section 8.3.1, *How the Linker Searches Libraries*
Control program option: --link-only

Menu entry
-

Command line syntax
--link-only

Description

With this option you suppress the locating phase. The linker stops after linking and informs you about unresolved references.

Related information

Control program option -cl (Stop after linking)

Linker option --link-only (Link only, no locating)
Control program option: --list-files

Menu entry
1. Select C/C++ Compiler » Miscellaneous.
2. Enable the option Generate assembly list file.

Command line syntax

--list-files [=file]

Default: no list files are generated

Description

With this option you tell the assembler via the control program to generate a list file for each specified input file. A list file shows the generated object code and the relative addresses. Note that the assembler generates a relocatable object file with relative addresses.

With the optional file you can specify a name for the list file. This is only possible if you specify only one input file to the control program. If you do not specify a file name, or you specify more than one input file, the control program names the generated list file(s) after the specified input file(s) with extension .lst.

Note that object files and library files are not counted as input files.

Related information

Assembler option --list-file (Generate list file)

Assembler option --list-format (Format list file)
Control program option: --list-object-files

Menu entry

Command line syntax

--list-object-files

Description

With this option the list of object files that are handled by the C++ prelinker, is displayed at stdout. The list is shown when it is changed by the C++ prelinker.

Related information
Control program option: --lsl-file (-d)

Menu entry

An LSL file can be generated when you create your project in Eclipse:

1. From the File menu, select File » New » Other... » TASKING C/C++ » TASKING VX-toolset for TriCore C/C++ Project.

   The New C/C++ Project wizard appears.

2. Fill in the project settings in each dialog and click Next > until the TriCore Project Settings appear.

3. Enable the option Add Linker script file to the project and click Finish.

   Eclipse creates your project and the file project.lsl in the project directory.

Command line syntax

--lsl-file=file

-d file

Description

A linker script file contains vital information about the core for the locating phase of the linker. A linker script file is coded in LSL and contains the following types of information:

- the architecture definition describes the core’s hardware architecture.
- the memory definition describes the physical memory available in the system.
- the section layout definition describes how to locate sections in memory.

With this option you specify a linker script file via the control program to the linker. If you do not specify this option, the linker uses a default script file. You can specify the existing file target.lsl or the name of a manually written linker script file. You can use this option multiple times. The linker processes the LSL files in the order in which they appear on the command line.

Related information

Section 8.7, Controlling the Linker with a Script
Control program option: --mil-link / --mil-split

Menu entry

- Command line syntax

--mil-link
--mil-split [=file,...]

Description

With option --mil-link the C compiler links the optimized intermediate representation (MIL) of all input files and MIL libraries specified on the command line in the compiler. The result is one single module that is optimized another time.

Option --mil-split does the same as option --mil-link, but in addition, the resulting MIL representation is written to a file with the suffix .mil and the C compiler also splits the MIL representation and writes it to separate files with suffix .ms. One file is written for each input file or MIL library specified on the command line. The .ms files are only updated on a change.

With option --mil-split you can perform application-wide optimizations during the frontend phase by specifying all modules at once, and still invoke the backend phase one module at a time to reduce the total compilation time.

Optionally, you can specify another filename for the .ms file the C compiler generates. Without an argument, the basename of the C source file is used to create the .ms filename. Note that if you specify a filename, you have to specify one filename for every input file.

Related information

Section 4.1, Compilation Process

C compiler option --mil / --mil-split
Control program option: --mmu-present

Menu entry

1. Expand C/C++ Build and select Processor.
2. From the Processor Selection list, select a processor that has an MMU.

Command line syntax

--mmu-present

Description

With this option you can use memory management instructions in the assembly code. When you select this option, the define __MMU__ is set to 1.

If you select a valid target processor (command line option --cpu (-C)), this option is set automatically, based on the chosen target processor.

Example

To allow the use of memory management unit (MMU) instructions in the assembly code, enter:

cctc --mmu-present test.c

Related information

Control program option --fpu-present (FPU present)
Control program option --cpu (Select processor)
Control program option: --no-auto-instantiation

Menu entry

Command line syntax

--no-auto-instantiation

Default: the C++ compiler automatically instantiates templates.

Description

With this option automatic instantiation of templates is disabled.

Related information

Control program option --instantiate (Set instantiation mode)

Section 2.5, Template Instantiation
Control program option: --no-default-libraries

Menu entry

1. Select Linker » Libraries.
   2. Disable the option Link default libraries.

Command line syntax

--no-default-libraries

Description

By default the control program specifies the standard C libraries (C99) and run-time library to the linker. With this option you tell the control program not to specify the standard C libraries and run-time library to the linker.

In this case you must specify the libraries you want to link to the linker with the option --library=library_name or pass the libraries as files on the command line. The control program recognizes the option --library (-l) as an option for the linker and passes it as such.

Example

```
cctc --no-default-libraries test.c
```

The control program does not specify any libraries to the linker. In normal cases this would result in unresolved externals.

To specify your own libraries (libc.a) and avoid unresolved externals:

```
cctc --no-default-libraries --library=c test.c
```

Related information

Control program option --library (Link system library)

Section 8.3.1, How the Linker Searches Libraries
Control program option: --no-double (--F)

Menu entry

1. Select C/C++ Compiler » Language.
2. Enable the option Treat double as float.

Command line syntax

--no-double

--F

Description

With this option you tell the compiler to treat variables of the type double as float. Because the float type takes less space, execution speed increases and code size decreases, both at the cost of less precision.

The control program also tells the linker to link the single-precision C library.

Related information

Control program option --use-double-precision-fp (Do not replace doubles with floats)
Control program option: --no-map-file

Menu entry

1. Select **Linker » Map File**.
2. Disable the option **Generate map file**.

Command line syntax

`--no-map-file`

Description

By default the control program tells the linker to generate a linker map file.

A linker map file is a text file that shows how the linker has mapped the sections and symbols from the various object files (.o) to the linked object file. A locate part shows the absolute position of each section. External symbols are listed per space with their absolute address, both sorted on symbol and sorted on address.

With this option you prevent the generation of a map file.

Related information

-
Control program option: --no-tasking-sfr

Menu entry
1. Select C/C++ Compiler » Preprocessing.
2. Disable the option Automatic inclusion of '.sfr' file.
3. Select Assembler » Preprocessing.
4. Disable the option Automatic inclusion of '.def' file.

Command line syntax
--no-tasking-sfr

Description
Normally, the C compiler and assembler includes a special function register (SFR) file before compiling/assembling. The compiler and assembler automatically select the SFR file belonging to the target you selected on the Processor page (control program option --cpu).

With this option the compiler and assembler do not include the register file regcpu.sfr as based on the selected target processor.

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

Related information
Control program option --cpu (Select processor)

Section 1.3.2, Accessing Hardware from C
Control program option: --no-warnings (-w)

Menu entry

1. Select C/C++ Compiler » Diagnostics.

The Suppress C compiler warnings box shows the warnings that are currently suppressed.

2. To suppress a warning, click on the Add button in the Suppress warnings box.

3. Enter the numbers, separated by commas, of the warnings you want to suppress (for example 537, 538). Or you can use the Add button multiple times.

4. To suppress all warnings, enable the option Suppress all warnings.

Use the Edit and Delete button to change a warning number or to remove a number from the list.

Command line syntax

--no-warnings [=number,...]
-w[number,...]

Description

With this option you can suppresses all warning messages for the various tools or specific compiler warning messages.

On the command line this option works as follows:

- If you do not specify this option, all warnings are reported.
- If you specify this option but without numbers, all warnings of all tools are suppressed.
- If you specify this option with a number, only the specified (compiler) warning is suppressed. You can specify the option --no-warnings=number multiple times.

Example

To suppress all warnings for all tools, enter:

cctc test.c --no-warnings

Related information

Control program option --warnings-as-errors (Treat warnings as errors)
Control program option: --option-file (-f)

Menu entry
-

Command line syntax

--option-file=file,...
-f file,...

Description

This option is primarily intended for command line use. Instead of typing all options on the command line, you can create an option file which contains all options and flags you want to specify. With this option you specify the option file to the control program.

Use an option file when the command line would exceed the limits of the operating system, or just to store options and save typing.

You can specify the option --option-file multiple times.

Format of an option file

- Multiple arguments on one line in the option file are allowed.
- To include whitespace in an argument, surround the argument with single or double quotes.
- If you want to use single quotes as part of the argument, surround the argument by double quotes and vice versa:

  "This has a single quote ' embedded"

  'This has a double quote " embedded'

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

- When a text line reaches its length limit, use a \ to continue the line. Whitespace between quotes is preserved.

  "This is a continuation \ line"

  -> "This is a continuation line"

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

Example

Suppose the file myoptions contains the following lines:
--debug-info
--define=DEMO=1
test.c

Specify the option file to the control program:

cctc --option-file=myoptions

This is equivalent to the following command line:

cctc --debug-info --define=DEMO=1 test.c

Related information
Control program option: --output (-o)

Menu entry

Eclipse always uses the project name as the basename for the output file.

Command line syntax

--output=file

-o file

Description

By default, the control program generates a file with the same basename as the first specified input file. With this option you specify another name for the resulting absolute object file.

The default output format is ELF/DWARF, but you can specify another output format with option --format.

Example

cctc test.c prog.c

The control program generates an ELF/DWARF object file (default) with the name test.elf.

To generate the file result.elf:

cctc --output=result.elf test.c prog.c

Related information

Control program option --format (Set linker output format)

Linker option --output (Output file)

Linker option --chip-output (Generate an output file for each chip)
Control program option: --pass (-W)

Menu entry

1. Select C/C++ Compiler » Miscellaneous or Assembler » Miscellaneous or Linker » Miscellaneous.

2. Add an option to the Additional options field.

Be aware that the options in the option file are added to the options you have set in the other pages. Only in extraordinary cases you may want to use them in combination. The assembler options are preceded by -Wa and the linker options are preceded by -Wl. For the C/C++ options you have to do this manually.

Command line syntax

- --pass-assembler=option -Waoption  Pass option directly to the assembler
- --pass-c=option -Wcoption  Pass option directly to the C compiler
- --pass-c++=option -Wcpoption  Pass option directly to the C++ compiler
- --pass-linker=option -Wloption  Pass option directly to the linker
- --pass-prelinker=option -Wploption  Pass option directly to the C++ prelinker

Description

With this option you tell the control program to call a tool with the specified option. The control program does not use or interpret the option itself, but specifies it directly to the tool which it calls.

Example

To pass the option --verbose directly to the linker, enter:

cctc --pass-linker=--verbose test.c

Related information

-
Control program option: --prelink-copy-if-nonlocal

Menu entry

Command line syntax

--prelink-copy-if-nonlocal

Description

If a file must be recompiled and it is not in the current directory, with this option the C++ prelinker copies the prelink file (.ii) to the current directory and rewrites that .ii file so it can find its associated .cc file. As a result, the .cc file is recompiled in the current directory.

With this option you prevent that previously compiled files are overwritten during recompilation.

Related information

-
Control program option: --prelink-local-only

Menu entry
-

Command line syntax
--prelink-local-only

Description
With this option the C++ prelinker ignores all files that are outside the current directory.

Related information
-
Control program option: --prelink-remove-instantiation-flags

Menu entry

Command line syntax

--prelink-remove-instantiation-flags

Description

With this option the C++ prelinker removes all instantiation flags from the generated object files.

Related information
Control program option: --preprocess (-E) / --no-preprocessing-only

Menu entry
1. Select C/C++ Compiler » Preprocessing.
2. Enable the option Store C preprocessor output in <file>.pre.

Command line syntax

--preprocess [=flags]
-E[flags]
--no-preprocessing-only

You can set the following flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+/-comments</td>
<td>keep comments</td>
</tr>
<tr>
<td>+/-make</td>
<td>generate dependencies for make</td>
</tr>
<tr>
<td>+/-noline</td>
<td>strip #line source position information</td>
</tr>
</tbody>
</table>

Default: --ECMP

Description

With this option you tell the compiler to preprocess the C source. The C compiler sends the preprocessed output to the file name.pre (where name is the name of the C source file to compile). Eclipse also compiles the C source.

On the command line, the control program stops after preprocessing. If you also want to compile the C source you can specify the option --no-preprocessing-only. In this case the control program calls the compiler twice, once with option --preprocess and once for a regular compilation.

With --preprocess=+comments you tell the preprocessor to keep the comments from the C source file in the preprocessed output.

With --preprocess=+make the compiler will generate dependency lines that can be used in a Makefile. The preprocessor output is discarded.

With --preprocess=+noline you tell the preprocessor to strip the #line source position information (lines starting with #line). These lines are normally processed by the assembler and not needed in the preprocessed output. When you leave these lines out, the output is easier to read.

Example

cctc --preprocess=+comments,-make,-noline --no-preprocessing-only test.c

The compiler preprocesses the file test.c and sends the output to the file test.pre. Comments are included but no dependencies are generated and the line source position information is not stripped from
the output file. Next, the control program calls the compiler, assembler and linker to create the final object file test.elf

**Related information**

-
Control program option: --profile (-p)

Menu entry

1. Select C/C++ Compiler » Debugging.
2. Enable or disable Static profiling.
3. Enable or disable one or more of the following Generate profiling information options (dynamic profiling):
   - for block counters (not in combination with Call graph or Function timers)
   - to build a call graph
   - for function counters
   - for function timers

Note that the more detailed information you request, the larger the overhead in terms of execution time, code size and heap space needed. The option --debug does not affect profiling, execution time or code size.

Command line syntax

--profile [=flag,...]
-p [flags]

Use the following option for a predefined set of flags:

--profile=g -pg Profiling with call graph and function timers.
Alias for: -pBcFSt

You can set the following flags:

+/-block b/B block counters
+/-callgraph c/C call graph
+/-function f/F function counters
+/-static s/S static profile generation
+/-time t/T function timers

Default (without flags): -pBcFSt

Description

Profiling is the process of collecting statistical data about a running application. With these data you can analyze which functions are called, how often they are called and what their execution time is.
Several methods of profiling exist. One method is code instrumentation which adds code to your application that takes care of the profiling process when the application is executed. Another method is static profiling.

For an extensive description of profiling refer to Chapter 6, Profiling.

You can obtain the following profiling data (see flags above):

**Block counters (not in combination with Call graph or Time)**

This will instrument the code to perform basic block counting. As the program runs, it counts the number of executions of each branch in an if statement, each iteration of a for loop, and so on. Note that though you can combine Block counters with Function counters, this has no effect because Function counters is only a subset of Block counters.

**Call graph (not in combination with Block counters)**

This will instrument the code to reconstruct the run-time call graph. As the program runs it associates the caller with the gathered profiling data.

**Function counters**

This will instrument the code to perform function call counting. This is a subset of the basic Block counters.

**Time (not in combination with Block counters)**

This will instrument the code to measure the time spent in a function. This includes the time spent in all sub functions (callees).

**Static profiling**

With this option you do not need to run the application to get profiling results. The compiler generates profiling information at compile time, without adding extra code to your application.

Note that the more detailed information you request, the larger the overhead in terms of execution time, code size and heap space needed. The option `Generate symbolic debug information` (`--debug`) does not affect profiling, execution time or code size.

The control program automatically specifies the corresponding profiling libraries to the linker.

**Example**

To generate block count information for the module `test.c` during execution, compile as follows:

```
cctc --profile=+block test.c
```

In this case the control program tells the linker to link the library `libpb.a`.

**Related information**

Chapter 6, Profiling
Control program option: --show-c++-warnings

Menu entry
-

Command line syntax
--show-c++-warnings

Description
The C++ compiler may generate a compiled C++ file (.ic) that causes warnings during compilation or assembling. With this option you tell the control program to show these warnings. By default C++ warnings are suppressed.

Related information
-
Control program option: --silicon-bug

Menu entry

1. Expand C/C++ Build and select Processor.
2. From the Processor Selection list, select a processor.

   The CPU Problem Bypasses and Checks box shows the available workarounds/checks available for the selected processor.

3. (Optional) Select Show all CPU problem bypasses and checks.
4. Click Select All or select one or more individual options.

Command line syntax

--silicon-bug=arg,...

For a list of available arguments refer to the description of option --silicon-bug of the compiler and assembler. Depending on the available arguments this option is passed to the compiler and/or assembler.

Description

With this option the control program tells the compiler/assembler/linker to use software workarounds for some CPU functional problems. Please refer to Chapter 16, CPU Problem Bypasses and Checks for more information about the individual problems and workarounds.

Example

To enable workarounds for problems CPU_TC.024 and CPU_TC.030, enter:

cctc --silicon-bug=cpu-tc024,cpu-tc030 test.c

Related information

Chapter 16, CPU Problem Bypasses and Checks
C compiler option --silicon-bug
Assembler option --silicon-bug
Control program option: --static

Menu entry

Command line syntax

--static

Description

This option is directly passed to the compiler.

With this option, the compiler treats external definitions at file scope (except for main) as if they were declared static. As a result, unused functions will be eliminated, and the alias checking algorithm assumes that objects with static storage cannot be referenced from functions outside the current module.

This option only makes sense when you specify all modules of an application on the command line.

Example

`cctc --static module1.c module2.c module3.c ...`

Related information

-
Control program option: --template-info-file

Menu entry

Command line syntax

--template-info-file=file

Description

This option specifies the name to be used for the template instantiation information file used in automatic instantiation mode.

This option is supplied for use by the control program that invokes the C++ compiler and is not intended to be used by end-users.

Related information
Control program option: --undefine (-U)

Menu entry

1. Select C/C++ Compiler » Preprocessing
   
   *The Defined symbols box shows the symbols that are currently defined.*

2. To remove a defined symbol, select the symbol in the Defined symbols box and click on the Delete button.

Command line syntax

```
--undefine=macro_name
-Umacro_name
```

Description

With this option you can undefine an earlier defined macro as with `#undef`. This option is for example useful to undefine predefined macros.

The following predefined ISO C standard macros cannot be undefined:

```plaintext
__FILE__        current source filename
__LINE__        current source line number (int type)
__TIME__        hh:mm:ss
__DATE__        Mmm dd yyyy
__STDC__        level of ANSI standard
```

The control program passes the option `--undefine (-U)` to the compiler.

Example

To undefine the predefined macro `__TASKING__`:

```
cctc --undefine=__TASKING__ test.c
```

Related information

Control program option `--define` (Define preprocessor macro)

Section 1.6, *Predefined Preprocessor Macros*
Control program option: --use-double-precision-fp

Menu entry
1. Select C/C++ Compiler » Language.
2. Enable the option Double precision floating-point.

Command line syntax
--use-double-precision-fp

Description
When an FPU is present the control program will by default compile all doubles as floats to make full use of the FPU. When you do not want this, use the option --use-double-precision-fp.

Example
To allow the use of floating-point unit (FPU) instructions in the assembly code and treat 'double' as 'double', enter:

cctc --fpu-present --use-double-precision-fp test.c

Related information
Control program option --no-double (Treat double as float)
Control program option --fpu-present (FPU present)
**Control program option: --user-mode**

**Menu entry**

1. Select C/C++ Compiler » Miscellaneous.
2. Add the option `--user-mode` to the Additional options field.

**Command line syntax**

```
--user-mode=mode
```

You can specify the following `mode` arguments:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>user-0</td>
<td>User-0 unprivileged mode</td>
</tr>
<tr>
<td>user-1</td>
<td>User-1 privileged mode</td>
</tr>
<tr>
<td>kernel</td>
<td>Kernel/Supervisor mode</td>
</tr>
</tbody>
</table>

Default: `--user-mode=kernel`

**Description**

With this option you specify the mode (I/O privilege mode) the TriCore runs in: User-0, User-1 or Kernel/Supervisor. The availability of some instructions depends on this mode. Most instructions run in all modes. The instructions `enable` and `disable` run in User-1 or kernel mode only. The instructions `bisr`, `mtcr`, `cachea.i` and `tlb` instructions run in kernel mode only.

**Example**

To restrict the instructions in the assembly code to User-1 mode, enter:

```
cctc --user-mode=user-1 test.c
```

**Related information**

- C compiler option `--user-mode` (Select user mode)
- Assembler option `--user-mode` (Select user mode)
Control program option: --verbose (-v)

Menu entry

1. Select Global Options.
2. Enable the option Verbose mode of control program.

Command line syntax

--verbose

-v

Description

With this option you put the control program in verbose mode. The control program performs its tasks while it prints the steps it performs to stdout.

Related information

Control program option --dry-run (Verbose output and suppress execution)
Control program option: --version (-V)

Menu entry

Command line syntax

  --version
  -v

Description

Display version information. The control program ignores all other options or input files.

Related information


Control program option: --warnings-as-errors

Menu entry
1. Select Global Options.
2. Enable the option Treat warnings as errors.

Command line syntax

```
--warnings-as-errors [=number,...]
```

Description

If one of the tools encounters an error, it stops processing the file(s). With this option you tell the tools to treat warnings as errors or treat specific C compiler warning messages as errors:

- If you specify this option but without numbers, all warnings are treated as errors.
- If you specify this option with a number, only the specified C compiler warning is treated as an error.
  You can specify the option `--warnings-as-errors=number` multiple times.

Related information

Control program option --no-warnings (Suppress some or all warnings)
11.6. Make Utility Options

When you build a project in Eclipse, Eclipse generates a makefile and uses the make utility `mktc` to build all your files. However, you can also use the make utility directly from the command line to build your project.

The invocation syntax is:

```
mktc [option...] [target...] [macro=def]
```

This section describes all options for the make utility. The make utility is a command line tool so there are no equivalent options in Eclipse.

For detailed information about the make utility and using makefiles see Section 9.2, *Make Utility*. 
Defining Macros

Command line syntax

macro_name[=macro_definition]

Description

With this argument you can define a macro and specify it to the make utility.

A macro definition remains in existence during the execution of the makefile, even when the makefile recursively calls the make utility again. In the recursive call, the macro acts as an environment variable. This means that it is overruled by definitions in the recursive call. Use the option `-e` to prevent this.

You can specify as many macros as you like. If the command line exceeds the limit of the operating system, you can define the macros in an option file which you then must specify to the make utility with the option `-m file`.

Defining macros on the command line is, for example, useful in combination with conditional processing as shown in the example below.

Example

Consider the following makefile with conditional rules to build a demo program and a real program:

```makefile
ifdef DEMO       # the value of DEMO is of no importance
    real.elf : demo.o main.o
       ltc demo.o main.o -lc -lfp -lrt
else
    real.elf : real.o main.o
       ltc real.o main.o -lc -lfp -lrt
endif
```

You can now use a macro definition to set the DEMO flag:

```makefile
mktc real.elf DEMO=1
```

In both cases the absolute object file `real.elf` is created but depending on the DEMO flag it is linked with `demo.o` or with `real.o`.

Related information

Make utility option `-e` (Environment variables override macro definitions)

Make utility option `-m` (Name of invocation file)
Make utility option: -?

Command line syntax

-?

Description

Displays an overview of all command line options.

Example

The following invocation displays a list of the available command line options:

mktc -?

Related information

-
Make utility option: -a

Command line syntax

-a

Description

Normally the make utility rebuilds only those files that are out of date. With this option you tell the make utility to rebuild all files, without checking whether they are out of date.

Example

mktc -a

Rebuilds all your files, regardless of whether they are out of date or not.

Related information

-
**Make utility option: -c**

**Command line syntax**

`-c`

**Description**

Eclipse uses this option when you create sub-projects. In this case the make utility calls another instance of the make utility for the sub-project. With the option `-c`, the make utility runs as a child process of the current make.

The option `-c` overrules the option `-err`.

**Example**

`mktc -c`

The make utility runs its commands as a child processes.

**Related information**

Make utility option `-err` (Redirect error message to file)
**Make utility option: -D / -DD**

**Command line syntax**

- `D`
- `DD`

**Description**

With the option `-D` the make utility prints every line of the makefile to standard output as it is read by `mktc`.

With the option `-DD` not only the lines of the makefile are printed but also the lines of the `mktc.mk` file (implicit rules).

**Example**

```
 mktc -D
```

Each line of the makefile that is read by the make utility is printed to standard output (usually your screen).

**Related information**

-
Make utility option: -d/ -dd

Command line syntax

-d
-dd

Description

With the option -d the make utility shows which files are out of date and thus need to be rebuild. The option -dd gives more detail than the option -d.

Example

mktc -d

Shows which files are out of date and rebuilds them.

Related information

-
Make utility option: -e

Command line syntax

-e

Description

If you use macro definitions, they may overrule the settings of the environment variables. With the option -e, the settings of the environment variables are used even if macros define otherwise.

Example

mktc -e

The make utility uses the settings of the environment variables regardless of macro definitions.

Related information

-
Make utility option: -err

Command line syntax

-err file

Description

With this option the make utility redirects error messages and verbose messages to a specified file. With the option -s the make utility only displays error messages.

Example

mktc -err error.txt

The make utility writes messages to the file error.txt.

Related information

Make utility option -s (Do not print commands before execution)

Make utility option -c (Run as child process)
Make utility option: -f

Command line syntax

-f my_makefile

Description

By default the make utility uses the file makefile to build your files.

With this option you tell the make utility to use the specified file instead of the file makefile. Multiple -f options act as if all the makefiles were concatenated in a left-to-right order.

If you use '-' instead of a makefile name it means that the information is read from stdin.

Example

mktc -f mymake

The make utility uses the file mymake to build your files.

Related information

-
Make utility option: -G

Command line syntax

-\texttt{G} \textit{path}

Description

Normally you must call the make utility from the directory where your makefile and other files are stored.

With the option -\texttt{G} you can call the make utility from within another directory. The \textit{path} is the path to the directory where your makefile and other files are stored and can be absolute or relative to your current directory.

Example

Suppose your makefile and other files are stored in the directory ..\myfiles. You can call the make utility, for example, as follows:

\texttt{mktc -G ..\myfiles}

Related information
Make utility option: -i

Command line syntax

-i

Description

When an error occurs during the make process, the make utility exits with a certain exit code. With the option -i, the make utility exits without an error code, even when errors occurred.

Example

mktc -i

The make utility exits without an error code, even when an error occurs.

Related information

-
Make utility option: -K

Command line syntax

-K

Description

With this option the make utility keeps temporary files it creates during the make process. The make utility stores temporary files in the directory that you have specified with the environment variable TMPDIR or in the default 'temp' directory of your system when the TMPDIR environment variable is not specified.

Example

mkte -K

The make utility preserves all temporary files.

Related information

-
Make utility option: -k

Command line syntax

-k

Description

When during the make process the make utility encounters an error, it stops rebuilding your files.

With the option -k, the make utility only stops building the target that produced the error. All other targets defined in the makefile are built.

Example

mktc -k

If the make utility encounters an error, it stops building the current target but proceeds with the other targets that are defined in the makefile.

Related information

Make utility option -S (Undo the effect of -k)
Make utility option: -m

Command line syntax

-m file

Description

Instead of typing all options on the command line, you can create an option file which contains all options and flags you want to specify. With this option you specify the option file to the make utility.

Use an option file when the command line would exceed the limits of the operating system, or just to store options and save typing.

You can specify the option -m multiple times.

If you use ‘-’ instead of a filename it means that the options are read from stdin.

Format of an option file

• Multiple arguments on one line in the option file are allowed.

• To include whitespace in an argument, surround the argument with single or double quotes.

• If you want to use single quotes as part of the argument, surround the argument by double quotes and vice versa:

  "This has a single quote ' embedded"

  'This has a double quote " embedded'

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

  Note that adjacent strings are concatenated.

• When a text line reaches its length limit, use a \ to continue the line. Whitespace between quotes is preserved.

  "This is a continuation \\n  line"

  -> "This is a continuation line"

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

Example

Suppose the file myoptions contains the following lines:

-k
-err errors.txt
test.elf
Specify the option file to the make utility:

mktc -m myoptions

This is equivalent to the following command line:

mktc -k -err errors.txt test.elf

Related information

-
Make utility option: -n

Command line syntax

-\n
Description

With this option you tell the make utility to perform a dry run. The make utility shows what it would do but does not actually perform these tasks.

This option is for example useful to quickly inspect what would happen if you call the make utility.

Example

mktc -n

The make utility does not perform any tasks but displays what it would do if called without the option -n.

Related information

Make utility option -s (Do not print commands before execution)
Make utility option: -p

Command line syntax

-p

Description

Normally, if a command in a target rule in a makefile returns an error or when the target construction is interrupted, the make utility removes that target file. With this option you tell the make utility to make all target files precious. This means that all dependency files are never removed.

Example

mktc -p

The make utility never removes target dependency files.

Related information

Special target .PRECIOUS in Section 9.2.2.1, Targets and Dependencies
Make utility option: -q

Command line syntax

-q

Description

With this option the make utility does not perform any tasks but only returns an exit code. A zero status indicates that all target files are up to date, a non-zero status indicates that some or all target files are out of date.

Example

mktc -q

The make utility only returns an error code that indicates whether all target files are up to date or not. It does not rebuild any files.

Related information
Make utility option: -r

Command line syntax

-r

Description

When you call the make utility, it first reads the implicit rules from the file mktc.mk, then it reads the makefile with the rules to build your files. (The file mktc.mk is located in the \etc directory of the toolset.)

With this option you tell the make utility not to read mktc.mk and to rely fully on the make rules in the makefile.

Example

mktc -r

The make utility does not read the implicit make rules in mktc.mk.

Related information

-
Make utility option: -S

Command line syntax

-s

Description

With this option you cancel the effect of the option -k. This is only necessary in a recursive make where the option -k might be inherited from the top-level make via MAKEFLAGS or if you set the option -k in the environment variable MAKEFLAGS.

With this option you tell the make utility not to read mktc.mk and to rely fully on the make rules in the makefile.

Example

mktc -S

The effect of the option -k is cancelled so the make utility stops with the make process after it encounters an error.

The option -k in this example may have been set with the environment variable MAKEFLAGS or in a recursive call to mktc in the makefile.

Related information

Make utility option -k (On error, abandon the work for the current target only)
Make utility option: -s

Command line syntax

-s

Description

With this option you tell the make utility to perform its tasks without printing the commands it executes. Error messages are normally printed.

Example

mktc -s

The make utility rebuilds your files but does not print the commands it executes during the make process.

Related information

Make utility option -n (Perform a dry run)
Make utility option: -t

Command line syntax

-t

Description

With this option you tell the make utility to touch the target files, bringing them up to date, rather than performing the rules to rebuild them.

Example

mktc -t

The make utility updates out-of-date files by giving them a new date and time stamp. The files are not actually rebuild.

Related information

-
Make utility option: -time

Command line syntax

-time

Description

With this option you tell the make utility to display the current date and time on standard output.

Example

mktc -time

The make utility displays the current date and time and updates out-of-date files.

Related information

-
Make utility option: -V

Command line syntax

-V

Description

Display version information. The make utility ignores all other options or input files.

Example

mktc -V

The make utility displays the version information but does not perform any tasks.

Related information

-
Make utility option: -W

Command line syntax

-W target

Description

With this option the make utility considers the specified target file always as up to date and will not rebuild it.

Example

mktc -W test.elf

The make utility rebuilds out of date targets in the makefile except the file test.elf which is considered now as up to date.

Related information

-
Make utility option: -w

Command line syntax

-w

Description

With this option the make utility sends error messages and verbose messages to standard output. Without this option, the make utility sends these messages to standard error.

This option is only useful on UNIX systems.

Example

mktc -w

The make utility sends messages to standard out instead of standard error.

Related information

-
Make utility option: -x

Command line syntax

-\x

Description

With this option the make utility shows extended error messages. Extended error messages give more detailed information about the exit status of the make utility after errors.

Example

mktc -x

If errors occur, the make utility gives extended information.

Related information

-
### 11.7. Archiver Options

The archiver and library maintainer `artc` is a tool to build library files and it offers the possibility to replace, extract and remove modules from an existing library.

The invocation syntax is:

```
artc  key_option  [sub_option...]  library  [object_file]
```

This section describes all options for the archiver. Some suboptions can only be used in combination with certain key options. They are described together. Suboptions that can always be used are described separately.

For detailed information about the archiver, see Section 9.3, *Archiver*.

**Overview of the options of the archiver utility**

The following archiver options are available:

<table>
<thead>
<tr>
<th>Description</th>
<th>Option</th>
<th>Sub-option</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Main functions (key options)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Replace or add an object module</td>
<td>-r</td>
<td>-a -b -c -u -v</td>
</tr>
<tr>
<td>Extract an object module from the library</td>
<td>-x</td>
<td>-v</td>
</tr>
<tr>
<td>Delete object module from library</td>
<td>-d</td>
<td>-v</td>
</tr>
<tr>
<td>Move object module to another position</td>
<td>-m</td>
<td>-a -b -v</td>
</tr>
<tr>
<td>Print a table of contents of the library</td>
<td>-t</td>
<td>-s0 -s1</td>
</tr>
<tr>
<td>Print object module to standard output</td>
<td>-p</td>
<td></td>
</tr>
<tr>
<td><strong>Sub-options</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Append or move new modules after existing module <em>name</em></td>
<td>-a</td>
<td><em>name</em></td>
</tr>
<tr>
<td>Append or move new modules before existing module <em>name</em></td>
<td>-b</td>
<td><em>name</em></td>
</tr>
<tr>
<td>Create library without notification if library does not exist</td>
<td>-c</td>
<td></td>
</tr>
<tr>
<td>Preserve last-modified date from the library</td>
<td>-o</td>
<td></td>
</tr>
<tr>
<td>Print symbols in library modules</td>
<td>-s{0</td>
<td>1}</td>
</tr>
<tr>
<td>Replace only newer modules</td>
<td>-u</td>
<td></td>
</tr>
<tr>
<td>Verbose</td>
<td>-v</td>
<td></td>
</tr>
<tr>
<td><strong>Miscellaneous</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Display options</td>
<td>-?</td>
<td></td>
</tr>
<tr>
<td>Display version header</td>
<td>-V</td>
<td></td>
</tr>
<tr>
<td>Read options from <em>file</em></td>
<td>-f</td>
<td><em>file</em></td>
</tr>
<tr>
<td>Suppress warnings above level <em>n</em></td>
<td>-wn</td>
<td></td>
</tr>
</tbody>
</table>
Archiver option: -?

Command line syntax

-?

Description

Displays an overview of all command line options.

Example

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

artc -?
artc

Related information

-
Archiver option: -d

Command line syntax

-\texttt{d} \ [-v]

Description

Delete the specified object modules from a library. With the suboption \texttt{-v} the archiver shows which files are removed.

-\texttt{v} \ Verbose: the archiver shows which files are removed.

Example

\texttt{artc} \ -d \ mylib.a \ \texttt{obj1.o obj2.o}

The archiver deletes \texttt{obj1.o} and \texttt{obj2.o} from the library \texttt{mylib.a}.

\texttt{artc} \ -d \ -v \ mylib.a \ \texttt{obj1.o obj2.o}

The archiver deletes \texttt{obj1.o} and \texttt{obj2.o} from the library \texttt{mylib.a} and displays which files are removed.

Related information
Archiver option: -f

Command line syntax

-f file

Description

Instead of typing all options on the command line, you can create an option file which contains all options and flags you want to specify. With this option you specify the option file to the archiver.

Use an option file when the command line would exceed the limits of the operating system, or just to store options and save typing.

You can specify the option -f multiple times.

If you use '-' instead of a filename it means that the options are read from stdin.

Format of an option file

• Multiple arguments on one line in the option file are allowed.

• To include whitespace in an argument, surround the argument with single or double quotes.

• If you want to use single quotes as part of the argument, surround the argument by double quotes and vise versa:

  "This has a single quote ' embedded"

  'This has a double quote " embedded'

  'This has a double quote " and a single quote '" embeded"

• When a text line reaches its length limit, use a \ to continue the line. Whitespace between quotes is preserved.

  "This is a continuation \ line"

  -> "This is a continuation line"

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

Example

Suppose the file myoptions contains the following lines:

-x mylib.a obj1.o
-w5

Specify the option file to the archiver:
artc -f myoptions

This is equivalent to the following command line:

artc -x mylib.a obj1.o -w5

Related information
Archiver option: -m

Command line syntax

```
-m [-a posname] [-b posname]
```

Description

Move the specified object modules to another position in the library.

The ordering of members in a library can make a difference in how programs are linked if a symbol is
defined in more than one member.

By default, the specified members are moved to the end of the archive. Use the suboptions -a or -b to
move them to a specified place instead.

```
-a posname      Move the specified object module(s) after the existing module posname.
-b posname      Move the specified object module(s) before the existing module posname.
```

Example

Suppose the library mylib.a contains the following objects (see option -t):

```
obj1.o
obj2.o
obj3.o
```

To move obj1.o to the end of mylib.a:

```
artc -m mylib.a obj1.o
```

To move obj3.o just before obj2.o:

```
artc -m -b obj3.o mylib.a obj2.o
```

The library mylib.a after these two invocations now looks like:

```
obj3.o
obj2.o
obj1.o
```

Related information

Archiver option -t (Print library contents)
Archiver option: -p

Command line syntax

-p

Description

Print the specified object module(s) in the library to standard output.

This option is only useful when you redirect or pipe the output to other files or tools that serve your own purposes. Normally you do not need this option.

Example

artc -p mylib.a obj1.o > file.o

The archiver prints the file obj1.o to standard output where it is redirected to the file file.o. The effect of this example is very similar to extracting a file from the library but in this case the 'extracted' file gets another name.

Related information

-
Archiver option: \(-r\)

Command line syntax

\[-r \{-a \ posname\} \{-b \ posname\} \{-c\} \{-u\} \{-v\}\]

Description

You can use the option \(-r\) for several purposes:

- Adding new objects to the library
- Replacing objects in the library with the same object of a newer date
- Creating a new library

The option \(-r\) normally adds a new module to the library. However, if the library already contains a module with the specified name, the existing module is replaced. If you specify a library that does not exist, the archiver creates a new library with the specified name.

If you add a module to the library without specifying the suboption \(-a\) or \(-b\), the specified module is added at the end of the archive. Use the suboptions \(-a\) or \(-b\) to insert them to a specified place instead.

\[-a \ posname\]  Move the specified object module(s) after the existing module posname.
\[-b \ posname\]  Move the specified object module(s) before the existing module posname.
\[-c\]  Create a new library without checking whether it already exists. If the library already exists, it is overwritten.
\[-u\]  Insert the specified object module only if it is newer than the module in the library.
\[-v\]  Verbose: the archiver shows which files are removed.

The suboptions \(-a\) or \(-b\) have no effect when an object is added to the library.

Example

Suppose the library mylib.a contains the following object (see option \(-t\)):

obj1.o

To add obj2.o to the end of mylib.a:

artc -r mylib.a obj2.o

To insert obj3.o just before obj2.o:

artc -r -b obj2.o mylib.a obj3.o

The library mylib.a after these two invocations now looks like:
Creating a new library

To create a new library file, add an object file and specify a library that does not yet exist:

```
artc -r obj1.o newlib.a
```

The archiver creates the library `newlib.a` and adds the object `obj1.o` to it.

To create a new library file and overwrite an existing library, add an object file and specify an existing library with the supoption `-c`:

```
artc -r -c obj1.o mylib.a
```

The archiver overwrites the library `mylib.a` and adds the object `obj1.o` to it. The new library `mylib.a` only contains `obj1.o`.

Related information

Archiver option `-t` (Print library contents)
Archiver option: -t

Command line syntax

-t [-s0|-s1]

Description

Print a table of contents of the library to standard output. With the suboption -s0 the archiver displays all symbols per object file.

- **-s0** Displays per object the name of the object itself and all symbols in the object.
- **-s1** Displays the symbols of all object files in the library in the form

  library_name:object_name:symbol_name

Example

artc -t mylib.a

The archiver prints a list of all object modules in the library mylib.a:

artc -t -s0 mylib.a

The archiver prints per object all symbols in the library. For example:

cstart.o
  symbols:
    _START
    __init_sp
    _start
    _endinit_clear
    _endinit_set

cinit.o
  symbols:
    _c_init

Related information
Archiver option: -V

Command line syntax

-V

Description

Display version information. The archiver ignores all other options or input files.

Example

artc -V

The archiver displays the version information but does not perform any tasks.

Related information
Archiver option: -w

Command line syntax

-w level

Description

With this suboption you tell the archiver to suppress all warnings above the specified level. The level is a number between 0 - 9.

The level of a message is printed between parentheses after the warning number. If you do not use the -w option, the default warning level is 8.

Example

To suppresses warnings above level 5:

artc -x -w5 mylib.a obj1.o

Related information

-
Archiver option: -x

Command line syntax

-x [-o] [-v]

Description

Extract an existing module from the library.

- **-o** Give the extracted object module the same date as the last-modified date that was recorded in the library. Without this suboption it receives the last-modified date of the moment it is extracted.

- **-v** Verbose: the archiver shows which files are extracted.

Example

To extract the file `obj1.o` from the library `mylib.a`:

```
artc -x mylib.a obj1.o
```

If you do not specify an object module, all object modules are extracted:

```
artc -x mylib.a
```

Related information

-
11.8. C++ Prelinker Options

This section lists all C++ prelinker options.

Options in Eclipse versus options on the command line

Eclipse invokes the C++ prelinker via the control program automatically. Therefore, it uses the syntax of the control program to pass options and files to the C++ prelinker. There are no equivalent prelinker options in Eclipse, but you can specify a command line option in Eclipse as follows:

1. From the Project menu, select Properties
   
   The Properties dialog appears.

2. In the left pane, expand C/C++ Build and select Settings.
   
   In the right pane the Settings appear.

3. On the Tool Settings tab, select C/C++ Compiler » Miscellaneous.

4. In the Additional options field, enter one or more prelinker command line options.
   
   Because Eclipse uses the control program, you have to precede the option with -Wpl to pass the option via the control program directly to the C++ prelinker.

Short and long option names

Options can have both short and long names. Short option names always begin with a single minus (-) character, long option names always begin with two minus (--) characters. You can abbreviate long option names as long as it forms a unique name. You can mix short and long option names on the command line.

If an option requires an argument, the argument may be separated from the keyword by white space, or the keyword may be immediately followed by =option. When the second form is used there may not be any white space on either side of the equals sign.

Options can have flags or sub-options. To switch a flag 'on', use a lowercase letter or a +longflag. To switch a flag off, use an uppercase letter or a -longflag. Separate longflags with commas. The argument separator character '=' is only used for long options, and should be omitted when the option argument is specified separately.
C++ prelinker option: --

Command line syntax
--

Description
When the C++ prelinker encounters this option, any command line arguments following this option are not re-emitted when option--move-nonlocal-objects is used.

Related information
-
C++ prelinker option: --check-specialization-errors (-s)

Command line syntax

--check-specialization-errors

-s

Description

With this option the C++ prelinker will not ignore any specialization errors. A specialization error occurs when entities are referenced as both explicit specializations and generated instantiations.

Related information

-
C++ prelinker option: --create-definition-list-file (-a)

Command line syntax

--create-definition-list-file

-a

Description

This option enables the generation of a template definition list file for the C++ compiler.

Related information

-
C++ prelinker option: --help (-?)

Command line syntax

`--help[=item]`

`-?`

You can specify the following arguments:

```
options     Show extended option descriptions
```

**Description**

Displays an overview of all command line options. When you specify an argument you can list extended information such as a list of option descriptions.

**Example**

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

```
prelktc -?
prelktc --help
prelktc
```

The following invocation displays an extended list of the available options:

```
prelktc --help=options
```

**Related information**

-
C++ prelinker option: --ignore-invalid-dumper-output (-i)

Command line syntax

--ignore-invalid-dumper-output

-i

Description

The C++ prelinker uses the object dumper to gather the necessary information. With this option the C++ prelinker will not abort when the object dumper generates invalid output.

Related information
C++ prelinker option: --library (-l)

Menu entry

1. Select Linker » Libraries.

   The lower box shows the list of libraries that are linked with the project.

2. To add a library, click on the Add button in the Libraries box.

3. Type or select a library (including its path).

4. Optionally, disable the option Link default libraries.

   Use the Edit and Delete button to change a library or to remove a library from the list.

Command line syntax

--library=name

-l name

Description

With this option you tell the prelinker to use system library libname.a, where name is a string. The prelinker first searches for system libraries in any directories specified with --library-directory, then in the directories specified with the environment variables LIBTC1V1_2 / LIBTC1V1_3 / LIBTC2, unless you used the option --ignore-default-library-path.

Example

To search in the system library libcp.a (C++ library):

prelktc -o test.ol test.o mylib.a -- --library=cp

The prelinker first looks in library mylib.a (in the current directory only), then in the system library libcp.a to resolve unresolved symbols.

Related information

Prelinker option --library-directory (Additional search path for system libraries)
C++ prelinker option: --library-directory (-L) / --ignore-default-library-path

Menu entry

1. Select **Linker » Libraries**.

   *The Libraries search path box shows the directories that are added to the search path for library files.*

2. To define a new directory for the search path, click on the **Add** button in the **Library search path** box.

3. Type or select a path.

   *Use the **Edit** and **Delete** button to change a path or to remove a path from the list.*

Command line syntax

`--library-directory=path,...`

`-L path,...`

`--ignore-default-library-path`

`-L`

Description

With this option you can specify the path(s) where your system libraries, specified with the option --library (-l), are located. If you want to specify multiple paths, use the option --library-directory for each separate path.

The default path is `$ (PRODDIR) \lib`.

If you specify only -L (without a pathname) or the long option --ignore-default-library-path, the C++ prelinker will not search the default path and also not in the paths specified in the environment variables `LIBTC1V1_2 / LIBTC1V1_3 / LIBTC2`. So, the linker ignores steps 2 and 3 as listed below.

The priority order in which the C++ prelinker searches for system libraries specified with the option --library (-l) is:

1. The path that is specified with the option --library-directory.

2. The path that is specified in the environment variables `LIBTC1V1_2 / LIBTC1V1_3 / LIBTC2`.

3. The default directory `$ (PRODDIR) \lib` (or a specific sub-directory).

Example

Suppose you call the linker as follows:

`prelktc -o test.ol test.o --library-directory=c: \mylibs -- --library=cp`
First the linker looks in the directory `c:\mylibs` for library `libcp.a` (this option). If it does not find the requested libraries, it looks in the directory that is set with the environment variables `LIBTC1V1_2 / LIBTC1V1_3 / LIBTC2`. Then the linker looks in the default directory `$\text{(PRODDIR)}\text{lib}$.

**Related information**

Prelinker option `--library` (Link system library)
C++ prelinker option: --limit-recursion (-r)

Command line syntax

--limit-recursion=depth
-r depth

Description

Limit the number of recursions in the algorithm to the given depth. If depth is 0, the recursion is unlimited.

The instantiation process is iterative: a recompilation may bring up new template entities that need to be instantiated, which requires another recompilation, etc. Some recursive templates can cause iteration that never terminates, because each iteration introduces another new entity that was not previously there. By default, this process is stopped after a certain number of iterations.

Related information

-
C++ prelinker option: --list-object-files (-b)

Command line syntax

--list-object-files
-b

Description

With this option the C++ prelinker will display a list of all object files.

Related information

-
C++ prelinker option: --mangled-names (-m)

Command line syntax

--mangled-names

-m

Description

With this option the C++ prelinker will show the mangled names in the output.

Related information

-
C++ prelinker option: --move-nonlocal-objects (-N)

Command line syntax

--move-nonlocal-objects

-N

Description

If a file from a non-local directory needs to be recompiled, do the compilation in the current directory.

An updated list of object files and library names is written to the file specified by the -o option so that the control program can tell that alternate versions of some of the object files should be used.

Related information

C++ prelinker option --object-list-file (-o) (Object list file)
C++ prelinker option: --no-nonlocal-object-assignment (-D)

Command line syntax

--no-nonlocal-object-assignment

-D

Description

Normally, the C++ prelinker assign instantiations to all object files. When this option is used, the C++ prelinker will only assign instantiations to object files in the current directory.

Related information

-


**C++ prelinker option: --no-warnings (-w)**

**Command line syntax**

--no-warnings

-w

**Description**

With this option you suppress all warning messages. Error messages are still issued.

**Related information**

C++ prelinker option **--warnings-as-errors** (Treat warnings as errors)
C++ prelinker option: --object-list-file (-o)

Command line syntax

--object-list-file=file

-o file

Default: Module name with .ol suffix.

Description

Specifies the name of the object list file (.ol). The object list file contains a list of all generated object file names. The list of files can be different from the initial list of object files.

Use this option when option -N is used.

Related information

C++ prelinker option --move-nonlocal-objects (-N)
C++ prelinker option: --option-file (-f)

Command line syntax

--option-file=file
-f file

Description

This option is primarily intended for command line use. Instead of typing all options on the command line, you can create an option file which contains all options and flags you want to specify. With this option you specify the option file to the C++ pre linker.

Use an option file when the command line would exceed the limits of the operating system, or just to store options and save typing.

You can specify the option --option-file multiple times.

Format of an option file

- Multiple arguments on one line in the option file are allowed.

- To include whitespace in an argument, surround the argument with single or double quotes.

- If you want to use single quotes as part of the argument, surround the argument by double quotes and vise versa:

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

- When a text line reaches its length limit, use a \\ to continue the line. Whitespace between quotes is preserved.

  "This is a continuation \line"

    -> "This is a continuation line"

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

Example

Suppose the file myoptions contains the following lines:

-o test.ol
-I/proj/include
test.o
Specify the option file to the C++ prelinker:

prelktc --option-file=myoptions

This is equivalent to the following command line:

prelktc -o test.ol -I/proj/include test.o

Related information
C++ prelinker option: --quiet (-q)

Command line syntax

--quiet

-q

Description

Quiet mode. Suppresses all C++ prelinker messages.

Related information

-
C++ prelinker option: --reserved-instantiation-request-lines (-R)

Command line syntax

--reserved-instantiation-request-lines=number
-R number

Description

Set the number of reserved instantiation request lines. This number of lines is skipped when an instantiation request file is processed.

Related information

-
C++ prelinker option: --skip-symbol-prefix (-u)

Command line syntax

--skip-symbol-prefix

-u

Description

With this option the C++ prelinker will skip the leading prefix character, usually the underscore '_ ' character, in each symbol name.

You can define the prefix character with the option --symbol-prefix.

Related information

C++ prelinker option --symbol-prefix (Define symbol prefix character)
C++ prelinker option: --suppress-compilation (-n)

Command line syntax

--suppress-compilation

-n

Description

With this option the instantiation list file (.ii) is updated but no recompilation of the C++ source files is done.

Related information

-
C++ prelinker option: --suppress-dependency-checks (-e)

Command line syntax

--suppress-dependency-checks

-e

Description

With this option the C++ prelinker will not check any files that are used to define exported templates.

Related information

-
C++ prelinker option: --suppress-instantiation-flags (-S)

Command line syntax

--suppress-instantiation-flags
-S

Description

With this option the C++ prelinker recompiles all of the local object files with the --suppress-instantiation-flags option.

Related information

C++ compiler option --suppress-instantiation-flags

Section 2.5, Template Instantiation
C++ prelinker option: --symbol-prefix (-c)

Command line syntax

--symbol-prefix=char
-c char

Description

The default symbol prefix character is the '_', but with this option you can change it to another character.

Related information

C++ prelinker option --skip-symbol-prefix (Skip the symbol prefix character)
C++ prelinker option: --verbose (-v)

Command line syntax

--verbose
-v

Description

Show more C++ prelinker messages about what is being done.

Related information

-
C++ prelinker option: --version (-V)

Command line syntax

--version

-v

Description

Display version information. The C++ prelinker ignores all other options or input files.

Related information

-
**C++ prelinker option: --warnings-as-errors**

**Menu entry**
1. Select **Global Options**.
2. Enable the option **Treat warnings as errors**.

**Command line syntax**

```sh
--warnings-as-errors
```

**Description**
When used, all warnings are treated as errors by the C++ prelinker.
This also affects the return value of the application when only warnings occur. A build process will now stop when warnings occur.

**Related information**

C++ prelinker option **--no-warnings** (Suppress all warnings)
Chapter 12. Libraries

This chapter contains an overview of all library functions that you can call in your C source. This includes all functions of the standard C library (ISO C99) and some functions of the floating-point library.

Section 12.1, Library Functions, gives an overview of all library functions you can use, grouped per header file. A number of functions declared in wchar.h are parallel to functions in other header files. These are discussed together.

Section 12.2, C Library Reentrancy, gives an overview of which functions are reentrant and which are not.

The following libraries are included in the TriCore toolset. Both Eclipse and the control program cctc automatically select the appropriate libraries depending on the specified options.

C library

<table>
<thead>
<tr>
<th>Libraries</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>libc[s].a</td>
<td>C libraries</td>
</tr>
<tr>
<td>libc[s]_fpu.a</td>
<td>Optional letter: s = single precision floating-point (compiler option --no-double)</td>
</tr>
<tr>
<td></td>
<td>_fpu = with FPU instructions (compiler option --fpu-present)</td>
</tr>
<tr>
<td>libfp[t].a</td>
<td>Floating-point libraries</td>
</tr>
<tr>
<td>libfp[t]_fpu.a</td>
<td>Optional letter: t = trapping (control program option --fp-trap)</td>
</tr>
<tr>
<td></td>
<td>_fpu = with FPU instructions (compiler option --fpu-present)</td>
</tr>
<tr>
<td>librt.a</td>
<td>Run-time library</td>
</tr>
<tr>
<td>libpb.a</td>
<td>Profiling libraries</td>
</tr>
<tr>
<td>libpc.a</td>
<td>pb = block/function counter</td>
</tr>
<tr>
<td>libpct.a</td>
<td>pc = call graph</td>
</tr>
<tr>
<td>libpd.a</td>
<td>pct = call graph and timing</td>
</tr>
<tr>
<td>libpt.a</td>
<td>pd = dummy</td>
</tr>
<tr>
<td></td>
<td>pt = function timing</td>
</tr>
</tbody>
</table>

C++ Library

The TASKING C++ compiler supports the STLport C++ libraries. STLport is a multiplatform ISO C++ Standard Library implementation. It is a free, open-source product, which is delivered with the TASKING C++ compiler. The library supports standard templates and I/O streams.

The include files for the STLport C++ libraries are present in directory include.stl relative to the product installation directory.

You can find more information on the STLport library on the following site: http://stlport.sourceforge.net/

Also read the license agreement on http://stlport.sourceforge.net/License.shtml. This license agreement is applicable to the STLport C++ library only. All other product components fall under the TASKING license agreement.

The following C++ libraries are delivered with the product:

<table>
<thead>
<tr>
<th>Libraries</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>libcp[s][x].a</td>
<td>C++ libraries</td>
</tr>
<tr>
<td></td>
<td>Optional letter:</td>
</tr>
<tr>
<td></td>
<td>s = single precision floating-point</td>
</tr>
<tr>
<td></td>
<td>x = exception handling</td>
</tr>
<tr>
<td>libstl[s].x.a</td>
<td>STLport C++ libraries (exception handling variants only)</td>
</tr>
<tr>
<td></td>
<td>Optional letter:</td>
</tr>
<tr>
<td></td>
<td>s = single precision floating-point</td>
</tr>
</tbody>
</table>

To build an STLport library

1. Change to the directory `installdir\lib\src.stl[p][tc1][tc2]\libstl[s]x`, depending on the library set used by your project.
2. Run the program `install-lib-sources.exe`. This will extract the library sources.
3. Change to the directory `[p][tc1][tc1_mmu][tc2][tc2_mmu]\libstl[s]x`, depending on the library set used by your project.
4. Run the `makefile` by executing `installdir\bin\mktc.exe` without arguments.
5. Copy the generated C++ library `libstl[s].x.a` to the corresponding directory `installdir\lib\[p][tc1][tc1_mmu][tc2][tc2_mmu]`.

where,

- `[tc1]` libraries for TriCore 1 architectures
- `[tc1_mmu]` libraries for TriCore 1 architectures that use an MMU
- `[tc2]` libraries for TriCore 2 architectures
- `[tc2_mmu]` libraries for TriCore 2 architectures that use an MMU
- `[p]` protected libraries for CPU functional problems
- `[s]` STLport library with exception handling and single precision floating-point

12.1. Library Functions

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

12.1.1. assert.h

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

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

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

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

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

This implementation uses the obvious implementation for complex multiplication; and a more sophisticated implementation for division and absolute value calculations which handles underflow, overflow and infinities with more care. The ISO C99 `#pragma CX_LIMITED_RANGE` therefore has no effect.

**Trigonometric functions**

- `csin` `<csinf` `<csinl` Returns the complex sine of \( z \).
- `ccos` `<ccosf` `<ccosl` Returns the complex cosine of \( z \).
- `ctan` `<ctanf` `<ctanl` Returns the complex tangent of \( z \).
- `casin` `<casinf` `<casinl` Returns the complex arc sine \( \sin^{-1}(z) \).
- `cacos` `<cacosf` `<cacosl` Returns the complex arc cosine \( \cos^{-1}(z) \).
- `catan` `<catanf` `<catanl` Returns the complex arc tangent \( \tan^{-1}(z) \).
- `csinh` `<csinhf` `<csinhl` Returns the complex hyperbolic sine of \( z \).
- `ccosh` `<ccoshf` `<ccoshl` Returns the complex hyperbolic cosine of \( z \).
- `ctanh` `<ctanhf` `<ctanhl` Returns the complex hyperbolic tangent of \( z \).
- `casinh` `<casinhf` `<casinhl` Returns the complex arc hyperbolic sinus of \( z \).
- `cacosh` `<cacoshf` `<cacoshl` Returns the complex arc hyperbolic cosine of \( z \).
- `catanh` `<catanhf` `<catanhl` Returns the complex arc hyperbolic tangent of \( z \).

**Exponential and logarithmic functions**

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

**Power and absolute-value functions**

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

### Manipulation functions

- \( \text{carg} \) \( \text{cargf} \) \( \text{cargl} \): Returns the argument of \( z \) (also known as *phase angle*).
- \( \text{cimag} \) \( \text{cimagf} \) \( \text{cimagn} \): Returns the imaginary part of \( z \) as a real (respectively as a `double`, `float`, `long double`).
- \( \text{conj} \) \( \text{congf} \) \( \text{congl} \): Returns the complex conjugate value (the sign of its imaginary part is reversed).
- \( \text{cproj} \) \( \text{cprojf} \) \( \text{cprojl} \): Returns the value of the projection of \( z \) onto the Riemann sphere.
- \( \text{creal} \) \( \text{crealf} \) \( \text{creall} \): Returns the real part of \( z \) as a real (respectively as a `double`, `float`, `long double`).

### 12.1.3. cstart.h

The header file `cstart.h` controls the system startup code’s general settings and register initializations. It contains defines only, no functions.

### 12.1.4. ctype.h and wctype.h

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

<table>
<thead>
<tr>
<th><code>ctype.h</code></th>
<th><code>wctype.h</code></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>isalnum</code></td>
<td><code>iswalnum</code></td>
<td>Returns a non-zero value when ( c ) is an alphabetic character or a number ([A-Z][a-z][0-9]).</td>
</tr>
<tr>
<td><code>isalpha</code></td>
<td><code>iswalpha</code></td>
<td>Returns a non-zero value when ( c ) is an alphabetic character ([A-Z][a-z]).</td>
</tr>
<tr>
<td><code>isblank</code></td>
<td><code>iswblank</code></td>
<td>Returns a non-zero value when ( c ) is a blank character (tab, space...).</td>
</tr>
<tr>
<td><code>iscntrl</code></td>
<td><code>iswcntrl</code></td>
<td>Returns a non-zero value when ( c ) is a control character.</td>
</tr>
<tr>
<td><code>isdigit</code></td>
<td><code>iswdigit</code></td>
<td>Returns a non-zero value when ( c ) is a numeric character ([0-9]).</td>
</tr>
<tr>
<td><code>isgraph</code></td>
<td><code>iswgraph</code></td>
<td>Returns a non-zero value when ( c ) is printable, but not a space.</td>
</tr>
<tr>
<td><code>islower</code></td>
<td><code>iswlower</code></td>
<td>Returns a non-zero value when ( c ) is a lowercase character ([a-z]).</td>
</tr>
<tr>
<td><code>isprint</code></td>
<td><code>iswprint</code></td>
<td>Returns a non-zero value when ( c ) is printable, including spaces.</td>
</tr>
<tr>
<td><code>ispunct</code></td>
<td><code>iswpunct</code></td>
<td>Returns a non-zero value when ( c ) is a punctuation character (such as &quot;.&quot;, &quot;,&quot;, &quot;:&quot;).</td>
</tr>
<tr>
<td><code>isspace</code></td>
<td><code>iswspace</code></td>
<td>Returns a non-zero value when ( c ) is a space type character (space, tab, vertical tab, formfeed, linefeed, carriage return).</td>
</tr>
<tr>
<td><code>isupper</code></td>
<td><code>iswupper</code></td>
<td>Returns a non-zero value when ( c ) is an uppercase character ([A-Z]).</td>
</tr>
<tr>
<td><code>isdigit</code></td>
<td><code>iswxdigit</code></td>
<td>Returns a non-zero value when ( c ) is a hexadecimal digit ([0-9][A-F][a-f]).</td>
</tr>
<tr>
<td>ctype.h</td>
<td>wchar.h</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>tolower</td>
<td>towlower</td>
<td>Returns c converted to a lowercase character if it is an uppercase character, otherwise c is returned.</td>
</tr>
<tr>
<td>toupper</td>
<td>towupper</td>
<td>Returns c converted to an uppercase character if it is a lowercase character, otherwise c is returned.</td>
</tr>
<tr>
<td>_tolower</td>
<td></td>
<td>Converts c to a lowercase character, does not check if c really is an uppercase character. Implemented as macro. This macro function is not defined in ISO C99.</td>
</tr>
<tr>
<td>_toupper</td>
<td></td>
<td>Converts c to an uppercase character, does not check if c really is a lowercase character. Implemented as macro. This macro function is not defined in ISO C99.</td>
</tr>
<tr>
<td>isascii</td>
<td></td>
<td>Returns a non-zero value when c is in the range of 0 and 127. This function is not defined in ISO C99.</td>
</tr>
<tr>
<td>toascii</td>
<td></td>
<td>Converts c to an ASCII value (strip highest bit). This function is not defined in ISO C99.</td>
</tr>
</tbody>
</table>

### 12.1.5. dbg.h

The header file `dbg.h` contains the debugger call interface for file system simulation. It contains low level functions. This header file is not defined in ISO C99.

- `__dbg_trap` Low level function to trap debug events
- `_argcv(const char *buf, size_t size)` Low level function for command line argument passing

### 12.1.6. errno.h

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

The following error codes are defined as macros in `errno.h`:

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

Floating-point errors
EDOM            23   Argument too large
ERANGE          24   Result too large

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

Encoding errors set by functions like fgetwc, getwc, mbtowc, etc ...
EILSEQ          29   Invalid or incomplete multibyte or wide character

Errors returned by RTOS
ECANCELED       30   Operation canceled
ENODEV          31   No such device

12.1.7. fcntl.h
The header file fcntl.h contains the function open(), which calls the low level function _open(), and definitions of flags used by the low level function _open(). This header file is not defined in ISO C99.

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

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

fegetenv         Stores the current floating-point environment. (Not implemented)
feholdexcept     Saves the current floating-point environment and installs an environment that ignores all floating-point exceptions. (Not implemented)
fesetenv         Restores a previously saved (fegetenv or feholdexcept) floating-point environment. (Not implemented)
feupdateenv      Saves the currently raised floating-point exceptions, restores a previously saved floating-point environment and finally raises the saved exceptions. (Not implemented)
Clears the current exception status flags corresponding to the flags specified in the argument. (*Not implemented*)

**feclearexcept**

Stores the current setting of the floating-point status flags. (*Not implemented*)

**fegetexceptflag**

Raises the exceptions represented in the argument. As a result, other exceptions may be raised as well. (*Not implemented*)

**feraiseexcept**

Sets the current floating-point status flags. (*Not implemented*)

**fesetexceptflag**

Returns the bitwise-OR of the exception macros corresponding to the exception flags which are currently set and are specified in the argument. (*Not implemented*)

**fetestexcept**

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

- FE_DIVBYZERO
- FE_INEXACT
- FE_INVALID
- FE_OVERFLOW
- FE_UNDERFLOW
- FE_ALL_EXCEPT

**fegetround**

Returns the current rounding direction, represented as one of the values of the rounding direction macros. (*Not implemented*)

**fesetround**

Sets the current rounding directions. (*Not implemented*)

Currently no rounding mode macros are implemented.

### 12.1.9. float.h

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

`float.h` used to contain prototypes for the functions `copysign(f)`, `isinf(f)`, `isfinite(f)`, `isnan(f)` and `scalb(f)`. These functions have accordingly to the ISO C99 standard been moved to the header file `math.h`. See also Section 12.1.16, `math.h` and `tgmath.h`.

The following functions are only available for ISO C90:

- `copysignf(float f, float s)` Copies the sign of the second argument `s` to the value of the first argument `f` and returns the result.
- `copysign(double d, double s)` Copies the sign of the second argument `s` to the value of the first argument `d` and returns the result.
- `isinff(float f)` Test the variable `f` on being an infinite (IEEE-754) value.
- `isinf(double d)` Test the variable `d` on being an infinite (IEEE-754) value.
- `isfinitef(float f)` Test the variable `f` on being a finite (IEEE-754) value.
- `isfinite(double d)` Test the variable `d` on being a finite (IEEE-754) value.
- `isnanf(float f)` Test the variable `f` on being NaN (Not a Number, IEEE-754).
Test the variable $d$ on being NaN (Not a Number, IEEE-754).

$\text{isnan(double } d)$

Returns $f \times 2^p$ for integral values without computing $2^N$.

$\text{scalbf(float } f, \text{int } p)$

Returns $d \times 2^p$ for integral values without computing $2^N$. (See also scalbn in Section 12.1.16, math.h and tgmath.h)

$\text{scalb(double } d, \text{int } p)$

12.1.10. inttypes.h and stdint.h

The header files stdint.h and inttypes.h provide additional declarations for integer types and have various characteristics. The stdint.h header file contains basic definitions of integer types of certain sizes, and corresponding sets of macros. This header file clearly refers to the corresponding sections in the ISO C99 standard.

The inttypes.h header file includes stdint.h and adds portable formatting and conversion functions. Below the conversion functions from inttypes.h are listed.

Returns the absolute value of $j$

$\text{imaxabs(intmax_t } j)$

Computes $\text{numerator/denominator}$ and $\text{numerator } \% \text{ denominator}$. The result is stored in the quot and rem components of the imaxdiv_t structure type.

$\text{imaxdiv(intmax_t } \text{numer, intmax_t } \text{denom)}$

Convert string to maximum sized integer. (Compare strtoll)

$\text{strtoimax(const char } * \text{ restrict nptr, char ** restrict endptr, int base)}$

Convert string to maximum sized unsigned integer. (Compare strtoull)

$\text{strtoimax(const char } * \text{ restrict nptr, char ** restrict endptr, int base)}$

Convert wide string to maximum sized integer. (Compare wcstoll)

$\text{wcstoimax(const wchar_t } * \text{ restrict nptr, wchar_t ** restrict endptr, int base)}$

Convert wide string to maximum sized unsigned integer. (Compare wcstoull)

$\text{wcstoimax(const wchar_t } * \text{ restrict nptr, wchar_t ** restrict endptr, int base)}$

12.1.11. io.h

The header file io.h contains prototypes for low level I/O functions. This header file is not defined in ISO C99.

Used by the functions close and fclose. (FSS implementation)

$\text{_close(fd)}$

Used by all file positioning functions: fgetpos, fseek, fsetpos, ftell, rewind. (FSS implementation)

$\text{_lseek(fd, offset, whence)}$

Used by the functions fopen and freopen. (FSS implementation)

$\text{_open(fd, flags)}$

Reads a sequence of characters from a file. (FSS implementation)

$\text{_read(fd, *buff, cnt)}$

Used by the function remove. (FSS implementation)

$\text{_unlink(*name)}$

Writes a sequence of characters to a file. (FSS implementation)

$\text{_write(fd, *buffer, cnt)}$
12.1.12. iso646.h

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

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

12.1.13. limits.h

Contains the sizes of integral types, defined as macros.

12.1.14. locale.h

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

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

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

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

<table>
<thead>
<tr>
<th>Category</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>LC_ALL</td>
<td>0</td>
</tr>
<tr>
<td>LC_COLLATE</td>
<td>1</td>
</tr>
<tr>
<td>LC_CTYPE</td>
<td>2</td>
</tr>
<tr>
<td>LC_NUMERIC</td>
<td>3</td>
</tr>
<tr>
<td>LC_TIME</td>
<td>4</td>
</tr>
<tr>
<td>LC_MONETARY</td>
<td>5</td>
</tr>
</tbody>
</table>

struct lconv *localeconv( void )

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

12.1.15. malloc.h

The header file malloc.h contains prototypes for memory allocation functions. This include file is not defined in ISO C99, it is included for backwards compatibility with ISO C90. For ISO C99, the memory allocation functions are part of stdlib.h. See Section 12.1.24, stdlib.h and wchar.h.
malloc(size)  Allocates space for an object with size \( \text{size} \). The allocated space is not initialized. Returns a pointer to the allocated space.

calloc(nobj, size)  Allocates space for \( n \) objects with size \( \text{size} \). The allocated space is initialized with zeros. Returns a pointer to the allocated space.

free(*ptr)  Deallocation of the memory space pointed to by \( \text{ptr} \) which should be a pointer earlier returned by the malloc or calloc function.

realloc(*ptr, size)  Deallocates the old object pointed to by \( \text{ptr} \) and returns a pointer to a new object with size \( \text{size} \). The new object cannot have a size larger than the previous object.

12.1.16. math.h and tgmath.h

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

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

### Trigonometric and hyperbolic functions

<table>
<thead>
<tr>
<th>\texttt{math.h}</th>
<th>\texttt{tgmath.h}</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{sin}</td>
<td>\texttt{sinf}</td>
<td>\texttt{sinl}</td>
</tr>
<tr>
<td>\texttt{cos}</td>
<td>\texttt{cosf}</td>
<td>\texttt{cosl}</td>
</tr>
<tr>
<td>\texttt{tan}</td>
<td>\texttt{tanhf}</td>
<td>\texttt{tanhl}</td>
</tr>
<tr>
<td>\texttt{asin}</td>
<td>\texttt{asinf}</td>
<td>\texttt{asinl}</td>
</tr>
<tr>
<td>\texttt{acos}</td>
<td>\texttt{acosf}</td>
<td>\texttt{acosl}</td>
</tr>
<tr>
<td>\texttt{atan}</td>
<td>\texttt{atanhf}</td>
<td>\texttt{atanhl}</td>
</tr>
<tr>
<td>\texttt{atan2}</td>
<td>\texttt{atan2f}</td>
<td>\texttt{atan2l}</td>
</tr>
<tr>
<td>\texttt{sinh}</td>
<td>\texttt{sinhf}</td>
<td>\texttt{sinhl}</td>
</tr>
<tr>
<td>\texttt{cosh}</td>
<td>\texttt{coshf}</td>
<td>\texttt{coshl}</td>
</tr>
<tr>
<td>\texttt{tanh}</td>
<td>\texttt{tanhf}</td>
<td>\texttt{tanhl}</td>
</tr>
<tr>
<td>\texttt{asinh}</td>
<td>\texttt{asinhf}</td>
<td>\texttt{asinhl}</td>
</tr>
<tr>
<td>\texttt{acosh}</td>
<td>\texttt{acoshf}</td>
<td>\texttt{acoshl}</td>
</tr>
<tr>
<td>\texttt{atanh}</td>
<td>\texttt{atanhf}</td>
<td>\texttt{atanhl}</td>
</tr>
</tbody>
</table>
Exponential and logarithmic functions

All of these functions are new in ISO C99, except for \(\exp\), \(\log\) and \(\log_{10}\).

<table>
<thead>
<tr>
<th>math.h</th>
<th>tgamma.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp</td>
<td>expf</td>
<td>(e^x), (x \in \mathbb{R})</td>
</tr>
<tr>
<td>exp2</td>
<td>exp2f</td>
<td>(2^x), (x \in \mathbb{R})</td>
</tr>
<tr>
<td>expm1</td>
<td>expm1f</td>
<td>(e^{-x}-1), (x &gt; 0)</td>
</tr>
<tr>
<td>log</td>
<td>logf</td>
<td>(\ln(x)), (x &gt; 0)</td>
</tr>
<tr>
<td>log10</td>
<td>log10f</td>
<td>(\log_{10}(x)), (x &gt; 0)</td>
</tr>
<tr>
<td>log1p</td>
<td>log1pf</td>
<td>(\log_e(1+x)), (x \neq -1)</td>
</tr>
<tr>
<td>log2</td>
<td>log2f</td>
<td>(\log_2(x)), (x &gt; 0)</td>
</tr>
<tr>
<td>ilogb</td>
<td>ilogbf</td>
<td>([-\log_2(x)] as an integer, (x &gt; 0))</td>
</tr>
<tr>
<td>logb</td>
<td>logbf</td>
<td>(\exp(x))</td>
</tr>
</tbody>
</table>

frexp, ldexp, modf, scalbn, scalbln

<table>
<thead>
<tr>
<th>math.h</th>
<th>tgamma.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>frexp</td>
<td>frexf</td>
<td>(f) and (n) such that: (0.0 \leq</td>
</tr>
<tr>
<td>ldexp</td>
<td>ldexpf</td>
<td>Inverse of frexp. Returns the result of (x \cdot 2^n). ((x) and (n) are both arguments).</td>
</tr>
<tr>
<td>modf</td>
<td>modff</td>
<td>(f) and (n) such that: (</td>
</tr>
<tr>
<td>scalbn</td>
<td>scalbnf</td>
<td>Computes the result of (x \cdot \text{FLT_RADIX}^n), efficiently, not normally by computing (\text{FLT_RADIX}^n) explicitly.</td>
</tr>
<tr>
<td>scalbln</td>
<td>scalblnf</td>
<td>Same as scalbn but with argument (n) as long int.</td>
</tr>
</tbody>
</table>

Rounding functions

<table>
<thead>
<tr>
<th>math.h</th>
<th>tgamma.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ceil</td>
<td>ceilf</td>
<td>(\text{nearest integer not less than } x)</td>
</tr>
<tr>
<td>floor</td>
<td>floorf</td>
<td>(\text{nearest integer not greater than } x)</td>
</tr>
<tr>
<td>rint</td>
<td>rintf</td>
<td>(\text{integer value according to the current rounding direction. See fenv.h.})</td>
</tr>
</tbody>
</table>

Libraries
### Description

gmath.h  math.h  Description

lrint  lrintf  lrintl  lrint  Returns the rounded integer value as a long int according to the current rounding direction. See fenv.h. *(Not implemented)*

llrint  llrintf  llrintl  llrint  Returns the rounded integer value as a long long int according to the current rounding direction. See fenv.h. *(Not implemented)*

nearbyint  nearbyintf  nearbyintl  nearbyint  Returns the rounded integer value as a floating-point according to the current rounding direction. See fenv.h. *(Not implemented)*

round  roundf  roundl  round  Returns the nearest integer value of \( x \) as int. *(Not implemented)*

lround  lroundf  lroundl  lround  Returns the nearest integer value of \( x \) as long int. *(Not implemented)*

llround  llroundf  llroundl  llround  Returns the nearest integer value of \( x \) as long long int. *(Not implemented)*

trunc  truncf  truncl  trunc  Returns the truncated integer value \( x \). *(Not implemented)*

### Remainder after division

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

### Power and absolute-value functions

<table>
<thead>
<tr>
<th>math.h</th>
<th>tgmath.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cbrt</td>
<td>cbrtf</td>
<td>cbrtl</td>
</tr>
<tr>
<td>fabs</td>
<td>fabsf</td>
<td>fabsl</td>
</tr>
<tr>
<td>fma</td>
<td>fmaf</td>
<td>fmal</td>
</tr>
<tr>
<td>hypot</td>
<td>hypotf</td>
<td>hypotl</td>
</tr>
<tr>
<td>pow</td>
<td>powf</td>
<td>powl</td>
</tr>
<tr>
<td>sqrt</td>
<td>sqrtf</td>
<td>sqrtl</td>
</tr>
</tbody>
</table>
Manipulation functions: copysign, nan, nextafter, nexttoward

<table>
<thead>
<tr>
<th>math.h</th>
<th>tgmmath.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>copysign</td>
<td>copysignf copysignl</td>
<td>Returns the value of ( x ) with the sign of ( y ).</td>
</tr>
<tr>
<td>nan</td>
<td>nanf</td>
<td>Returns a quiet NaN, if available, with content indicated through ( tagp ).</td>
</tr>
<tr>
<td>nanl</td>
<td>-</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>nextafter</td>
<td>nextafterf nextafterl</td>
<td>Returns the next representable value in the specified format after ( x ) in the direction of ( y ). Returns ( y ) is ( x=y ).</td>
</tr>
<tr>
<td>nexttoward</td>
<td>nexttowardf nexttowardl</td>
<td>Same as nextafter, except that the second argument in all three variants is of type long double. Returns ( y ) if ( x=y ).</td>
</tr>
</tbody>
</table>

Positive difference, maximum, minimum

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

Error and gamma (Not implemented)

<table>
<thead>
<tr>
<th>math.h</th>
<th>tgmmath.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>erf</td>
<td>erff</td>
<td>Computes the error function of ( x ).</td>
</tr>
<tr>
<td>erfl</td>
<td>erf</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>erfc</td>
<td>erfcf</td>
<td>Computes the complementary error function of ( x ).</td>
</tr>
<tr>
<td>erfcl</td>
<td>erc</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>lgamma</td>
<td>lgammaf</td>
<td>Computes the ( \log_e</td>
</tr>
<tr>
<td>lgammal</td>
<td>lgamma</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>tgamma</td>
<td>tgammaf</td>
<td>Computes ( \Gamma(x) ).</td>
</tr>
<tr>
<td>tgammal</td>
<td>tgamma</td>
<td>(Not implemented)</td>
</tr>
</tbody>
</table>

Comparison macros

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

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

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

### 12.1.17. setjmp.h

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

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

### 12.1.18. signal.h

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

<table>
<thead>
<tr>
<th>Signal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGINT</td>
<td>1  Receipt of an interactive attention signal</td>
</tr>
</tbody>
</table>
Detection of an invalid function message

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

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

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

The next function determines how subsequent signals will be handled:

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

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

- **SIG_DFL**  Default behavior is used
- **SIG_IGN**  The signal is ignored

The function returns the previous value of signalfunction for the specific signal, or SIG_ERR if an error occurs.

### 12.1.19. stdarg.h

The facilities in this header file gives you a portable way to access variable arguments lists, such as needed for as `fprintf` and `vfprintf`. va_copy is new in ISO C99. This header file contains the following macros:

```c
va_arg(va_list ap, type)  
```

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

```c
va_copy(va_list dest, va_list src)
```

This macro duplicates the current state of src in dest, creating a second pointer into the argument list. After this call, va_arg() may be used on src and dest independently.

```c
va_end(va_list ap)
```

This macro must be called after the arguments have been processed. It should be called before the function using the macro ‘va_start’ is terminated.

```c
va_start(va_list ap, lastarg)
```

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

### 12.1.20. stdbool.h

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

This header file defines the types for common use:

- 

- 

- 

Besides these types, the following macros are defined:

- 

- 

12.1.22. stdint.h

See Section 12.1.10, inttypes.h and stdint.h

12.1.23. stdio.h and wchar.h

Types

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

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

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

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

<table>
<thead>
<tr>
<th>stdio.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>Expands to 0 (zero).</td>
</tr>
<tr>
<td>BUFSIZ</td>
<td>Size of the buffer used by the setbuf/setvbuf function: 512</td>
</tr>
<tr>
<td>EOF</td>
<td>End of file indicator. Expands to -1.</td>
</tr>
<tr>
<td>WEOF</td>
<td>End of file indicator. Expands to UINT_MAX (defined in limits.h)</td>
</tr>
<tr>
<td></td>
<td>NOTE: WEOF need not to be a negative number as long as its value does not</td>
</tr>
<tr>
<td></td>
<td>correspond to a member of the wide character set. (Defined in wchar.h).</td>
</tr>
<tr>
<td>FOPEN_MAX</td>
<td>Number of files that can be opened simultaneously: 10</td>
</tr>
<tr>
<td>FILENAME_MAX</td>
<td>Maximum length of a filename: 100</td>
</tr>
<tr>
<td>_IOFBF</td>
<td>Expand to an integer expression, suitable for use as argument to the</td>
</tr>
<tr>
<td></td>
<td>setvbuf function.</td>
</tr>
<tr>
<td>_IOLBF</td>
<td></td>
</tr>
<tr>
<td>_IONBF</td>
<td></td>
</tr>
<tr>
<td>L_tmpnam</td>
<td>Size of the string used to hold temporary file names: 8 (tmpxxxxx)</td>
</tr>
<tr>
<td>TMP_MAX</td>
<td>Maximum number of unique temporary filenames that can be generated: 0x8000</td>
</tr>
<tr>
<td>SEEK_CUR</td>
<td>Expand to an integer expression, suitable for use as the third argument to</td>
</tr>
<tr>
<td>SEEK_END</td>
<td>the fseek function.</td>
</tr>
<tr>
<td>SEEK_SET</td>
<td></td>
</tr>
<tr>
<td>stderr</td>
<td></td>
</tr>
<tr>
<td>stdin</td>
<td></td>
</tr>
<tr>
<td>stdout</td>
<td></td>
</tr>
</tbody>
</table>

File access

<table>
<thead>
<tr>
<th>stdio.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fopen(name, mode)</td>
<td>Opens a file for a given mode. Available modes are:</td>
</tr>
<tr>
<td></td>
<td>&quot;r&quot; read; open text file for reading</td>
</tr>
<tr>
<td></td>
<td>&quot;w&quot; write; create text file for writing; if the file already exists, its</td>
</tr>
<tr>
<td></td>
<td>contents is discarded</td>
</tr>
<tr>
<td></td>
<td>&quot;a&quot; append; open existing text file or create new text file for writing at</td>
</tr>
<tr>
<td></td>
<td>end of file</td>
</tr>
<tr>
<td></td>
<td>&quot;r+&quot; open text file for update; reading and writing</td>
</tr>
<tr>
<td></td>
<td>&quot;w+&quot; create text file for update; previous contents if any is discarded</td>
</tr>
<tr>
<td></td>
<td>&quot;a+&quot; append; open or create text file for update, writes at end of file</td>
</tr>
<tr>
<td></td>
<td><em>(FSS implementation)</em></td>
</tr>
<tr>
<td>fclose(name)</td>
<td>Flushes the data stream and closes the specified file that was previously</td>
</tr>
<tr>
<td></td>
<td>opened with fopen. <em>(FSS implementation)</em></td>
</tr>
<tr>
<td>fflush(name)</td>
<td>If stream is an output stream, any buffered but unwritten data is written.</td>
</tr>
<tr>
<td></td>
<td>Else, the effect is undefined. <em>(FSS implementation)</em></td>
</tr>
</tbody>
</table>
stdio.h Description
freopen(name, mode, stream) Similar to fopen, but rather than generating a new value of type FILE *, the existing value is associated with a new stream. (FSS implementation)
setbuf(stream, buffer) If buffer is NULL, buffering is turned off for the stream. Otherwise, setbuf is equivalent to: (void) setvbuf(stream, buffer, _IOFBF, BUFSIZ).
setvbuf(stream, buffer, mode, size) Controls buffering for the stream; this function must be called before reading or writing. Mode can have the following values:
  _IOFBF causes full buffering
  _IOLBF causes line buffering of text files
  _IONBF causes no buffering.
If buffer is not NULL, it will be used as a buffer; otherwise a buffer will be allocated. size determines the buffer size.

Formatted input/output

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

• Flags (in any order):
  - specifies left adjustment of the converted argument.
  + a number is always preceded with a sign character.
    + has higher precedence than space.
  space a negative number is preceded with a sign, positive numbers with a space.
  0 specifies padding to the field width with zeros (only for numbers).
  # specifies an alternate output form. For o, the first digit will be zero. For x or X, "0x" and "0X" will be prefixed to the number. For e, E, f, g, G, the output always contains a decimal point, trailing zeros are not removed.

• A number specifying a minimum field width. The converted argument is printed in a field with at least the length specified here. If the converted argument has fewer characters than specified, it will be padded at the left side (or at the right when the flag '-' was specified) with spaces. Padding to numeric fields will be done with zeros when the flag '0' is also specified (only when padding left). Instead of a numeric value, also '*' may be specified, the value is then taken from the next argument, which is assumed to be of type int.

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

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

• A length modifier 'h', 'hh', 'i', 'll', 'l', 'j', 'z' or 't'. 'h' indicates that the argument is to be treated as a short or unsigned short. 'hh' indicates that the argument is to be treated as a char or unsigned char. 'l' should be used if the argument is a long integer, 'll' for a long long. 'L' indicates that the argument...
is a long double. ’j’ indicates a pointer to intmax_t or uintmax_t, ’z’ indicates a pointer to size_t and ’t’ indicates a pointer to ptrdiff_t.

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

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

*printf conversion characters*

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

The format string can contain:

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

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

- A ’*’, meaning that no assignment is done for this field.
- A number specifying the maximum field width.
The conversion characters \texttt{d}, \texttt{i}, \texttt{n}, \texttt{o}, \texttt{u} and \texttt{x} may be preceded by \texttt{h} if the argument is a pointer to \texttt{short} rather than \texttt{int}, or by \texttt{hh} if the argument is a pointer to \texttt{char}, or by \texttt{l'} (letter ell) if the argument is a pointer to \texttt{long} or by \texttt{ll} for a pointer to \texttt{long long}, \texttt{j'} for a pointer to \texttt{intmax_t} or \texttt{uintmax_t}, \texttt{'z'} for a pointer to \texttt{size_t} or \texttt{'t'} for a pointer to \texttt{ptrdiff_t}. The conversion characters \texttt{e}, \texttt{f}, and \texttt{g} may be preceded by \texttt{I'} if the argument is a pointer to \texttt{double} rather than \texttt{float}, and by \texttt{L'} for a pointer to a \texttt{long double}.

A conversion specifier. \texttt{*}, maximum field width and length modifier are optional, the conversion character is not. The conversion character must be one of the following, if a character following \texttt{"%"} is not in the list, the behavior is undefined.

<table>
<thead>
<tr>
<th>Character Scanned as</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{d} int, signed decimal.</td>
<td></td>
</tr>
<tr>
<td>\texttt{i} int, the integer may be given octal (i.e. a leading \texttt{0} is entered) or hexadecimal (leading &quot;\texttt{0x}&quot; or &quot;\texttt{0X}&quot;), or just decimal.</td>
<td></td>
</tr>
<tr>
<td>\texttt{o} int, unsigned octal.</td>
<td></td>
</tr>
<tr>
<td>\texttt{u} int, unsigned decimal.</td>
<td></td>
</tr>
<tr>
<td>\texttt{x} int, unsigned hexadecimal in lowercase or uppercase.</td>
<td></td>
</tr>
<tr>
<td>\texttt{c} single character (converted to unsigned char).</td>
<td></td>
</tr>
<tr>
<td>\texttt{s} char *, a string of non white space characters. The argument should point to an array of characters, large enough to hold the string and a terminating NULL character.</td>
<td></td>
</tr>
<tr>
<td>\texttt{f, F} float</td>
<td></td>
</tr>
<tr>
<td>\texttt{e, E} float</td>
<td></td>
</tr>
<tr>
<td>\texttt{g, G} float</td>
<td></td>
</tr>
<tr>
<td>\texttt{a, A} float</td>
<td></td>
</tr>
<tr>
<td>\texttt{n} int *, the number of characters written so far is written into the argument. No scanning is done.</td>
<td></td>
</tr>
<tr>
<td>\texttt{p} pointer; hexadecimal value which must be entered without 0x- prefix.</td>
<td></td>
</tr>
<tr>
<td>\texttt{r, lr __fract, __lfract}</td>
<td></td>
</tr>
<tr>
<td>\texttt{R, IR __accum, __laccum}</td>
<td></td>
</tr>
<tr>
<td>\texttt{[...]} Matches a string of input characters from the set between the brackets. A NULL character is added to terminate the string. Specifying \texttt{[...]} includes the \texttt{]} character in the set of scanning characters.</td>
<td></td>
</tr>
<tr>
<td>\texttt[^...]} Matches a string of input characters not in the set between the brackets. A NULL character is added to terminate the string. Specifying \texttt[^...]} includes the \texttt{]} character in the set.</td>
<td></td>
</tr>
<tr>
<td>% Literal '%', no assignment is done.</td>
<td></td>
</tr>
</tbody>
</table>

\textit{scanf conversion characters}
<table>
<thead>
<tr>
<th>stdin.h</th>
<th>wchar.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fscanf(stream, format, ...)</code></td>
<td><code>fwscanf(stream, format, ...)</code></td>
<td>Performs a formatted read from the given stream. Returns the number of items converted successfully. <em>(FSS implementation)</em></td>
</tr>
<tr>
<td><code>scanf(format, ...)</code></td>
<td><code>wscanf(format, ...)</code></td>
<td>Performs a formatted read from stdin. Returns the number of items converted successfully. <em>(FSS implementation)</em></td>
</tr>
<tr>
<td><code>sscanf(*s, format, ...)</code></td>
<td><code>swscanf(*s, format, ...)</code></td>
<td>Performs a formatted read from the string s. Returns the number of items converted successfully.</td>
</tr>
<tr>
<td><code>vscanf(format, ...)</code></td>
<td><code>vswscanf(*s, format, ...)</code></td>
<td>Same as <code>sscanf/wscanf</code>, but extra arguments are given as variable argument list <code>arg</code>. <em>(See Section 12.1.19, stdarg.h)</em></td>
</tr>
<tr>
<td><code>fwprintf(stream, format, ...)</code></td>
<td><code>fprintf(stream, format, ...)</code></td>
<td>Performs a formatted write to the given stream. Returns EOF/WEOF on error. <em>(FSS implementation)</em></td>
</tr>
<tr>
<td><code>printf(format, ...)</code></td>
<td><code>wprintf(format, ...)</code></td>
<td>Performs a formatted write to the stream stdout. Returns EOF/WEOF on error. <em>(FSS implementation)</em></td>
</tr>
<tr>
<td><code>sprintf(*s, format, ...)</code></td>
<td><code>swprintf(*s, n, format, ...)</code></td>
<td>Performs a formatted write to string s. Returns EOF/WEOF on error. Same as <code>sprintf</code>, but <code>n</code> specifies the maximum number of characters (including the terminating null character) to be written.</td>
</tr>
<tr>
<td><code>vfprintf(stream, format, ...)</code></td>
<td><code>vfwprintf(stream, format, ...)</code></td>
<td>Same as <code>fprintf/fwprintf</code>, but extra arguments are given as variable argument list <code>arg</code>. <em>(See Section 12.1.19, stdarg.h)</em> <em>(FSS implementation)</em></td>
</tr>
<tr>
<td><code>vprintf(format, ...)</code></td>
<td><code>vwprintf(format, ...)</code></td>
<td>Same as <code>printf/wprintf</code>, but extra arguments are given as variable argument list <code>arg</code>. <em>(See Section 12.1.19, stdarg.h)</em></td>
</tr>
<tr>
<td><code>vsscanf(*s, format, arg)</code></td>
<td><code>vsscanf(*s, format, arg)</code></td>
<td>Same as <code>scanf/wscanf</code>, but extra arguments are given as variable argument list <code>arg</code>. <em>(See Section 12.1.19, stdarg.h)</em></td>
</tr>
<tr>
<td><code>vswscanf(*s, format, arg)</code></td>
<td><code>vsscanf(*s, format, arg)</code></td>
<td>Same as <code>sscanf/wscanf</code>, but extra arguments are given as variable argument list <code>arg</code>. <em>(See Section 12.1.19, stdarg.h)</em></td>
</tr>
</tbody>
</table>
## Character input/output

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fgetc(stream)</code></td>
<td>Reads one character from <code>stream</code>. Returns the read character, or EOF/WEOF on error. <em>(FSS implementation)</em></td>
</tr>
<tr>
<td><code>getc(stream)</code></td>
<td>Same as <code>fgetc</code> except that it is implemented as a macro. <em>(FSS implementation)</em></td>
</tr>
<tr>
<td><code>getchar(stdin)</code></td>
<td>Reads one character from the <code>stdin</code> stream. Returns the character read or EOF/WEOF on error. Implemented as macro. <em>(FSS implementation)</em></td>
</tr>
<tr>
<td><code>fgets(*s, n, stream)</code></td>
<td>Reads at most the next <code>n-1</code> characters from the <code>stream</code> into array <code>s</code> until a newline is found. Returns <code>s</code> or NULL or EOF/WEOF on error. <em>(FSS implementation)</em></td>
</tr>
<tr>
<td><code>fputs(*s, stream)</code></td>
<td>Writes string <code>s</code> to the given <code>stream</code>. Returns EOF/WEOF on error. <em>(FSS implementation)</em></td>
</tr>
<tr>
<td><code>ungetc(c, stream)</code></td>
<td>Pushes character <code>c</code> back onto the input <code>stream</code>. Returns EOF/WEOF on error. <em>(FSS implementation)</em></td>
</tr>
<tr>
<td><code>fputc(c, stream)</code></td>
<td>Put character <code>c</code> onto the given <code>stream</code>. Returns EOF/WEOF on error. <em>(FSS implementation)</em></td>
</tr>
<tr>
<td><code>putc(c, stdout)</code></td>
<td>Put character <code>c</code> onto the <code>stdout</code> stream. Returns EOF/WEOF on error. Implemented as macro. <em>(FSS implementation)</em></td>
</tr>
<tr>
<td><code>fputs(*s, stream)</code></td>
<td>Writes string <code>s</code> to the given <code>stream</code>. Returns EOF/WEOF on error. <em>(FSS implementation)</em></td>
</tr>
</tbody>
</table>

### Direct input/output

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

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

### Random access

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

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

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ftell(stream)</td>
<td>Returns the current file position for stream, or -1L on error. <em>(FSS implementation)</em></td>
</tr>
</tbody>
</table>
| rewind(stream) | Sets the file position indicator for the stream to the beginning of the file. This function is equivalent to:  
  (void) fseek(stream, 0L, SEEK_SET);  
  clearerr(stream);  
  *(FSS implementation)* |
| fgetpos(stream, pos) | Stores the current value of the file position indicator for stream in the object pointed to by pos. *(FSS implementation)* |
| fsetpos(stream, pos) | Positions stream at the position recorded by fgetpos in *pos. *(FSS implementation)* |

### Operations on files

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

<table>
<thead>
<tr>
<th>stdio.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>clearerr(stream)</td>
<td>Clears the end of file and error indicators for stream.</td>
</tr>
<tr>
<td>ferror(stream)</td>
<td>Returns a non-zero value if the error indicator for stream is set.</td>
</tr>
<tr>
<td>feof(stream)</td>
<td>Returns a non-zero value if the end of file indicator for stream is set.</td>
</tr>
<tr>
<td>perror(*s)</td>
<td>Prints s and the error message belonging to the integer errno. (See Section 12.1.6, errno.h)</td>
</tr>
</tbody>
</table>

12.1.24. stdlib.h and wchar.h

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

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

Macros

EXIT_SUCCESS  Predefined exit codes that can be used in the exit function.
0
EXIT_FAILURE  1
RAND_MAX      Highest number that can be returned by the rand/srand function.
32767
MB_CUR_MAX    Maximum number of bytes in a multibyte character for the extended character set specified by the current locale (category LC_CTYPE, see Section 12.1.14, locale.h).
1

Numeric conversions

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

double atof(*s)
int atoi(*s)
long atol(*s)
long long atoll(*s)
The following functions convert the initial portion of the string \(*s\) to a float, double and long double value respectively. \(*endp\) will point to the first character not used by the conversion.

<table>
<thead>
<tr>
<th>stdlib.h</th>
<th>wchar.h</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>float</td>
</tr>
<tr>
<td>strtof(*s,**endp)</td>
<td>wcstof(*s,**endp)</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
</tr>
<tr>
<td>strtod(*s,**endp)</td>
<td>wcstod(*s,**endp)</td>
</tr>
<tr>
<td>long double</td>
<td>long double</td>
</tr>
<tr>
<td>strtold(*s,**endp)</td>
<td>wcstold(*s,**endp)</td>
</tr>
</tbody>
</table>

The following functions convert the initial portion of the string \(*s\) to a long, long long, unsigned long and unsigned long long respectively. Base specifies the radix. \(*endp\) will point to the first character not used by the conversion.

<table>
<thead>
<tr>
<th>stdlib.h</th>
<th>wchar.h</th>
</tr>
</thead>
<tbody>
<tr>
<td>long</td>
<td>long</td>
</tr>
<tr>
<td>strtol(*s,**endp,base)</td>
<td>wcstol(*s,**endp,base)</td>
</tr>
<tr>
<td>long long</td>
<td>long long</td>
</tr>
<tr>
<td>strtol(*s,**endp,base)</td>
<td>wcstol(*s,**endp,base)</td>
</tr>
<tr>
<td>unsigned</td>
<td>unsigned</td>
</tr>
<tr>
<td>strtoul(*s,**endp,base)</td>
<td>wcstoul(*s,**endp,base)</td>
</tr>
<tr>
<td>unsigned long long</td>
<td>wcstoull(*s,**endp,base)</td>
</tr>
</tbody>
</table>

**Random number generation**

- **rand** Returns a pseudo random integer in the range 0 to RAND_MAX.
- **srand(seed)** Same as rand but uses `seed` for a new sequence of pseudo random numbers.

**Memory management**

- **malloc(size)** Allocates space for an object with size `size`. The allocated space is not initialized. Returns a pointer to the allocated space.
- **calloc(nobj,size)** Allocates space for \(n\) objects with size `size`. The allocated space is initialized with zeros. Returns a pointer to the allocated space.
- **free(*ptr)** Deallocates the memory space pointed to by `ptr` which should be a pointer earlier returned by the malloc or calloc function.
- **realloc(*ptr,size)** Deallocates the old object pointed to by `ptr` and returns a pointer to a new object with size `size`. The new object cannot have a size larger than the previous object.

**Environment communication**

- **abort()** Causes abnormal program termination. If the signal SIGABRT is caught, the signal handler may take over control. (See Section 12.1.18, `signal.h`).
- **atexit(*func)** `func` points to a function that is called (without arguments) when the program normally terminates.
Causes normal program termination. Acts as if main() returns with status as the return value. Status can also be specified with the predefined macros EXIT_SUCCESS or EXIT_FAILURE.

Same as exit, but not registered by the atexit function or signal handlers registered by the signal function are called.

Searches an environment list for a string s. Returns a pointer to the contents of s.

NOTE: this function is not implemented because there is no OS.

Passes the string s to the environment for execution.

NOTE: this function is not implemented because there is no OS.

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

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

Compute the absolute value of an int, long int, and long long int j respectively.

Compute x/y and x%y in a single operation. X and y have respectively type int, long int and long long int. The result is stored in the members quot and rem of struct div_t, ldiv_t and lldiv_t which have the same types.

Determines the number of bytes in the multi-byte character pointed to by s. At most n characters will be examined. (See also mbrlen in Section 12.1.28, wchar.h).

Converts the multi-byte character in s to a wide-character code and stores it in pwc. At most n characters will be examined.

Converts the wide-character wc into a multi-byte representation and stores it in the string pointed to by s. At most MB_CUR_MAX characters are stored.

Converts a sequence of multi-byte characters in the string pointed to by s into a sequence of wide characters and stores at most n wide characters into the array pointed to by pwcs. (See also mbsrtowcs in Section 12.1.28, wchar.h).
Converts a sequence of wide characters in the array pointed to by `pwcs` into multi-byte characters and stores at most `n` multi-byte characters into the string pointed to by `s`. (See also `wcsrtowmb` in Section 12.1.28, `wchar.h`).

### 12.1.25. string.h and wchar.h

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

#### Copying and concatenation functions

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

#### Comparison functions

<table>
<thead>
<tr>
<th>string.h</th>
<th>wchar.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>memcmp(*s1,*s2,n)</code></td>
<td><code>wmemcmp(*s1,*s2,n)</code></td>
<td>Compares the first <code>n</code> characters of <code>*s1</code> to the first <code>n</code> characters of <code>*s2</code>. Returns <code>&lt; 0</code> if <code>*s1</code> &lt; <code>*s2</code>, <code>&lt; 0</code> if <code>*s1</code> = <code>*s2</code>, or &gt; 0 if <code>*s1</code> &gt; <code>*s2</code>.</td>
</tr>
<tr>
<td><code>strcmp(*s1,*s2)</code></td>
<td><code>wcscmp(*s1,*s2)</code></td>
<td>Compares string <code>*s1</code> to <code>*s2</code>. Returns <code>&lt; 0</code> if <code>*s1</code> &lt; <code>*s2</code>, <code>&lt; 0</code> if <code>*s1</code> = <code>*s2</code>, or &gt; 0 if <code>*s1</code> &gt; <code>*s2</code>.</td>
</tr>
<tr>
<td><code>strnncmp(*s1,*s2,n)</code></td>
<td><code>wcsncmp(*s1,*s2,n)</code></td>
<td>Compares the first <code>n</code> characters of <code>*s1</code> to the first <code>n</code> characters of <code>*s2</code>. Returns <code>&lt; 0</code> if <code>*s1</code> &lt; <code>*s2</code>, <code>&lt; 0</code> if <code>*s1</code> = <code>*s2</code>, or &gt; 0 if <code>*s1</code> &gt; <code>*s2</code>.</td>
</tr>
<tr>
<td><code>strnncmp(*s1,*s2,n)</code></td>
<td><code>wcsncmp(*s1,*s2,n)</code></td>
<td>Performs a local-specific comparison between string <code>*s1</code> and string <code>*s2</code> according to the LC_COLLATE category of the current locale. Returns <code>&lt; 0</code> if <code>*s1</code> &lt; <code>*s2</code>, <code>&lt; 0</code> if <code>*s1</code> = <code>*s2</code>, or &gt; 0 if <code>*s1</code> &gt; <code>*s2</code>. (See Section 12.1.14, <code>locale.h</code>)</td>
</tr>
</tbody>
</table>
Transforms (a local) string \( s2 \) so that a comparison between transformed strings with ```strcmp``` gives the same result as a comparison between non-transformed strings with ```strcoll```. Returns the transformed string \( s1 \).

### Search functions

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

### Miscellaneous functions

<table>
<thead>
<tr>
<th>string.h</th>
<th>wchar.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>memset(*s, c, n)</code></td>
<td><code>memset(*s, c, n)</code></td>
<td>Fills the first ( n ) bytes of ( s ) with character ( c ) and returns ( s ).</td>
</tr>
<tr>
<td><code>strerror(errno)</code></td>
<td>-</td>
<td>Typically, the values for errno come from int errno. This function returns a pointer to the associated error message. (See also Section 12.1.6, errno.h)</td>
</tr>
<tr>
<td><code>strlen(*s)</code></td>
<td><code>wcslen(*s)</code></td>
<td>Returns the length of string ( s ).</td>
</tr>
</tbody>
</table>

### 12.1.26. time.h and wchar.h

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

The type struct tm below is defined according to ISO C99 with one exception: this implementation does not support leap seconds. The struct tm type is defined as follows:

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

**Time manipulation**

- **clock** Returns the application’s best approximation to the processor time used by the program since it was started. This low-level routine is not implemented because it strongly depends on the hardware. To determine the time in seconds, the result of clock should be divided by the value defined by `CLOCKS_PER_SEC`.
- **difftime(t1, t0)** Returns the difference $t1 - t0$ in seconds.
- **mktime(tm *, tp)** Converts the broken-down time in the structure pointed to by tp, to a value of type `time_t`. The return value has the same encoding as the return value of the `time` function.
- **time(*timer)** Returns the current calendar time. This value is also assigned to *timer.

**Time conversion**

- **asctime(tm *, tp)** Converts the broken-down time in the structure pointed to by tp into a string in the form Mon Jan 22 15:14:14 2007
- **ctime(*timer)** Converts the calendar time pointed to by timer to local time in the form of a string. This is equivalent to: `asctime(localtime(timer))`
- **gmtime(*timer)** Converts the calendar time pointed to by timer to the broken-down time, expressed as UTC. Returns a pointer to the broken-down time.
- **localtime(*timer)** Converts the calendar time pointed to by timer to the broken-down time, expressed as local time. Returns a pointer to the broken-down time.

**Formatted time**

The next function has a parallel function defined in `wchar.h`: 
`strptime(*s, smax, *fmt, tm *tp)`

```wstrptime(*s, smax, *fmt, tm *tp)```

Formats date and time information from `struct tm *tp` into `*s` according to the specified format `*fmt`. No more than `smax` characters are placed into `*s`. The formatting of `strptime` is locale-specific using the `LC_TIME` category (see Section 12.1.14, `locale.h`).

You can use the next conversion specifiers:

- `%a` abbreviated weekday name
- `%A` full weekday name
- `%b` abbreviated month name
- `%B` full month name
- `%c` locale-specific date and time representation (same as `%a %b %e %T %Y`)
- `%C` last two digits of the year
- `%d` day of the month (01-31)
- `%D` same as `%m/%d/%y`
- `%e` day of the month (1-31), with single digits preceded by a space
- `%F` ISO 8601 date format: `%Y-%m-%d`
- `%g` last two digits of the week based year (00-99)
- `%G` week based year (0000–9999)
- `%h` same as `%b`
- `%H` hour, 24-hour clock (00-23)
- `%I` hour, 12-hour clock (01-12)
- `%j` day of the year (001-366)
- `%m` month (01-12)
- `%M` minute (00-59)
- `%n` replaced by newline character
- `%p` locale’s equivalent of AM or PM
- `%r` locale’s 12-hour clock time; same as `%I:%M:%S %p`
- `%R` same as `%H:%M`
- `%S` second (00-59)
- `%t` replaced by horizontal tab character
- `%T` ISO 8601 time format: `%H:%M:%S`
- `%u` ISO 8601 weekday number (1-7), Monday as first day of the week
- `%U` week number of the year (00-53), week 1 has the first Sunday
- `%V` ISO 8601 week number (01-53) in the week-based year
- `%w` weekday (0-6, Sunday is 0)
The file unistd.h contains standard UNIX I/O functions. These functions are all implemented using file system simulation. Except for lstat and fstat which are not implemented. This header file is not defined in ISO C99.

**access (name, mode)**  
Use file system simulation to check the permissions of a file on the host. *mode* specifies the type of access and is a bit pattern constructed by a logical OR of the following values:

- **R_OK** Checks read permission.
- **W_OK** Checks write permission.
- **X_OK** Checks execute (search) permission.
- **F_OK** Checks to see if the file exists.

(*FSS implementation*)

**chdir(path)**  
Use file system simulation to change the current directory on the host to the directory indicated by *path*. (*FSS implementation*)

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

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

**lseek(fd, offset, whence)**  
Moves read-write file offset. Calls _lseek(). (*FSS implementation*)

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

**stat(name, buff)**  
Use file system simulation to stat() a file on the host platform. (*FSS implementation*)

**lstat(name, buff)**  
This function is identical to stat(), except in the case of a symbolic link, where the link itself is 'stat'-ed, not the file that it refers to. (*Not implemented*)

**fstat(fd, buff)**  
This function is identical to stat(), except that it uses a file descriptor instead of a name. (*Not implemented*)

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

**write(fd, buff, cnt)**  
Write a sequence of characters to a file. Calls _write(). (*FSS implementation*)
12.1.28. wchar.h

Many functions in wchar.h represent the wide-character variant of other functions so these are discussed together. (See Section 12.1.23, stdio.h and wchar.h, Section 12.1.24, stdlib.h and wchar.h, Section 12.1.25, string.h and wchar.h and Section 12.1.26, time.h and wchar.h).

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

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

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

mbsinit(*ps) Determines whether the object pointed to by ps, is an initial conversion state. Returns a non-zero value if so.

mbstowcs(*pwcs,**src,n,*ps) Restartable version of mbstowcs. See Section 12.1.24, stdlib.h and wchar.h. The initial conversion state is specified by ps. The input sequence of multibyte characters is specified indirectly by src.

wcsrtombs(*s,**src,n,*ps) Restartable version of wcstombs. See Section 12.1.24, stdlib.h and wchar.h. The initial conversion state is specified by ps. The input wide string is specified indirectly by src.

mbtowc(*pwc,*s,n,*ps) Converts a multibyte character *s to a wide character *pwc according to conversion state ps. See also mbtowc in Section 12.1.24, stdlib.h and wchar.h.

wcrtomb(*s,wc,*ps) Converts a wide character wc to a multi-byte character according to conversion state ps and stores the multi-byte character in *s.

btowc(c) Returns the wide character corresponding to character c. Returns WEOF on error.

wctob(c) Returns the multi-byte character corresponding to the wide character c. The returned multi-byte character is represented as one byte. Returns EOF on error.

mbrlen(*s,n,*ps) Inspects up to n bytes from the string *s to see if those characters represent valid multibyte characters, relative to the conversion state held in *ps.
## 12.1.29. wctype.h

Most functions in `wctype.h` represent the wide-character variant of functions declared in `ctype.h` and are discussed in Section 12.1.4, `ctype.h` and `wctype.h`. In addition, this header file provides extensible, locale specific functions and wide character classification.

**wctype(property)** Constructs a value of type `wctype_t` that describes a class of wide characters identified by the string `*property`. If `property` identifies a valid class of wide characters according to the LC_TYPE category (see Section 12.1.14, `locale.h`) of the current locale, a non-zero value is returned that can be used as an argument in the `iswctype` function.

**iswctype(wc, desc)** Tests whether the wide character `wc` is a member of the class represented by `wctype_t desc`. Returns a non-zero value if tested true.

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

**wctrans(property)** Constructs a value of type `wctype_t` that describes a mapping between wide characters identified by the string `*property`. If `property` identifies a valid mapping of wide characters according to the LC_TYPE category (see Section 12.1.14, `locale.h`) of the current locale, a non-zero value is returned that can be used as an argument in the `towctrans` function.

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Equivalent to locale specific transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tolower(wc)</code></td>
<td><code>towctrans(wc,wctrans(&quot;tolower&quot;))</code></td>
</tr>
<tr>
<td><code>toupper(wc)</code></td>
<td><code>towctrans(wc,wctrans(&quot;toupper&quot;))</code></td>
</tr>
</tbody>
</table>

## 12.2. C Library Reentrancy

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

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Not reentrant because</th>
</tr>
</thead>
<tbody>
<tr>
<td>_close</td>
<td>Uses global File System Simulation buffer, _dbg_request</td>
</tr>
<tr>
<td>_doflt</td>
<td>Uses I/O functions which modify iob[ ]. See (1).</td>
</tr>
<tr>
<td>_doprint</td>
<td>Uses indirect access to static iob[ ] array. See (1).</td>
</tr>
<tr>
<td>_doscan</td>
<td>Uses indirect access to iob[ ] and calls ungetc (access to local static ungetc[ ] buffer). See (1).</td>
</tr>
<tr>
<td>_Exit</td>
<td>See exit</td>
</tr>
<tr>
<td>_filbuf</td>
<td>Uses iob[ ], which is not reentrant. See (1).</td>
</tr>
<tr>
<td>_flsbuf</td>
<td>Uses iob[ ]. See (1).</td>
</tr>
<tr>
<td>_getft</td>
<td>Uses iob[ ]. See (1).</td>
</tr>
<tr>
<td>_iob</td>
<td>Defines static iob[ ]. See (1).</td>
</tr>
<tr>
<td>_lseek</td>
<td>Uses global File System Simulation buffer, _dbg_request</td>
</tr>
<tr>
<td>_open</td>
<td>Uses global File System Simulation buffer, _dbg_request</td>
</tr>
<tr>
<td>_read</td>
<td>Uses global File System Simulation buffer, _dbg_request</td>
</tr>
<tr>
<td>_unlink</td>
<td>Uses global File System Simulation buffer, _dbg_request</td>
</tr>
<tr>
<td>_write</td>
<td>Uses global File System Simulation buffer, _dbg_request</td>
</tr>
<tr>
<td>abort</td>
<td>Calls exit</td>
</tr>
<tr>
<td>abs labs llabs</td>
<td>-</td>
</tr>
<tr>
<td>access</td>
<td>Uses global File System Simulation buffer, _dbg_request</td>
</tr>
<tr>
<td>acos acosf acosl</td>
<td>Sets errno.</td>
</tr>
<tr>
<td>acosh acoshf acoshl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>asctime</td>
<td>asctime defines static array for broken-down time string.</td>
</tr>
<tr>
<td>asin asinf asinl</td>
<td>Sets errno.</td>
</tr>
<tr>
<td>asinh asinhf asinhl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>atan atanf atanl</td>
<td>-</td>
</tr>
<tr>
<td>atan2 atan2f atan2l</td>
<td>-</td>
</tr>
<tr>
<td>atanh atanhf atanhl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>atexit</td>
<td>atexit defines static array with function pointers to execute at exit of program.</td>
</tr>
<tr>
<td>atof</td>
<td>-</td>
</tr>
<tr>
<td>atoi</td>
<td>-</td>
</tr>
<tr>
<td>atol</td>
<td>-</td>
</tr>
<tr>
<td>bsearch</td>
<td>-</td>
</tr>
<tr>
<td>Function</td>
<td>Not reentrant because</td>
</tr>
<tr>
<td>-----------------</td>
<td>--------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>btowc</td>
<td>-</td>
</tr>
<tr>
<td>cabs cabsf cabsl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>cacos cacosf cacosl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>cacosh cacoshf cacoshl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>calloc</td>
<td>calloc uses static buffer management structures. See malloc (5).</td>
</tr>
<tr>
<td>carg cargf cargl</td>
<td>-</td>
</tr>
<tr>
<td>casin casinf casinl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>casinh casinhf casinhl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>catan catanf catanl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>catanh catanhf catanhl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>cbrt cbrtf cbrtl</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>ccos ccosf ccosl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>ccosh ccoshf ccoshl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>ceil ceilf ceill</td>
<td>-</td>
</tr>
<tr>
<td>cexp cexpf cexpl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>chdir</td>
<td>Uses global File System Simulation buffer, _dbg_request</td>
</tr>
<tr>
<td>cimag cimagf cimagl</td>
<td>-</td>
</tr>
<tr>
<td>cleanup</td>
<td>Calls fclose. See (1)</td>
</tr>
<tr>
<td>clearerr</td>
<td>Modifies iob[ ]. See (1)</td>
</tr>
<tr>
<td>clock</td>
<td>Uses global File System Simulation buffer, _dbg_request</td>
</tr>
<tr>
<td>clog clogf clogl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>close</td>
<td>Calls _close</td>
</tr>
<tr>
<td>conj conjf conjl</td>
<td>-</td>
</tr>
<tr>
<td>copysign copysignf copysignl</td>
<td>-</td>
</tr>
<tr>
<td>cos cosf cosl</td>
<td>-</td>
</tr>
<tr>
<td>cosh coshf coshl</td>
<td>cosh calls exp(), which sets errno. If errno is discarded, cosh is reentrant.</td>
</tr>
<tr>
<td>cpow cpowf cpowl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>cproj cprojf cprojl</td>
<td>-</td>
</tr>
<tr>
<td>creal crealf creall</td>
<td>-</td>
</tr>
<tr>
<td>csin csinf csinl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>csinh csinhf csinhl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>csqrt csqrtf csqrtnl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>ctan ctanf ctanl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>ctanh ctanhf ctanhnl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>Function</td>
<td>Not reentrant because</td>
</tr>
<tr>
<td>--------------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td>ctime</td>
<td>Calls asctime</td>
</tr>
<tr>
<td>difftime</td>
<td></td>
</tr>
<tr>
<td>div ldiv lldiv</td>
<td></td>
</tr>
<tr>
<td>erf erfl erff</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>erfc erfcl</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>exit</td>
<td>Calls fclose indirectly which uses iob[ ] calls functions in _atexit array. See (1). To make exit reentrant kernel support is required.</td>
</tr>
<tr>
<td>exp expf expl</td>
<td></td>
</tr>
<tr>
<td>exp2 exp2f exp2l</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>expm1 expm1f expm1l</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>fabs fabsf fabsl</td>
<td></td>
</tr>
<tr>
<td>fclose</td>
<td>Uses values in iob[ ]. See (1).</td>
</tr>
<tr>
<td>fdim fdimf fdiml</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>feclearexcept</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>fegetenv</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>fegetexceptflag</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>fegetround</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>feholdexcept</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>feof</td>
<td>Uses values in iob[ ]. See (1).</td>
</tr>
<tr>
<td>feraisexcept</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>ferror</td>
<td>Uses values in iob[ ]. See (1).</td>
</tr>
<tr>
<td>fesetenv</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>fesetexceptflag</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>fesetround</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>fetestexcept</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>feupdateenv</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>fflush</td>
<td>Modifies iob[ ]. See (1).</td>
</tr>
<tr>
<td>fgetc fgetwc</td>
<td>Uses pointer to iob[ ]. See (1).</td>
</tr>
<tr>
<td>fgetpos</td>
<td>Sets the variable errno and uses pointer to iob[ ]. See (1) / (2).</td>
</tr>
<tr>
<td>fgets fgetws</td>
<td>Uses iob[ ]. See (1).</td>
</tr>
<tr>
<td>floor floorf floorl</td>
<td></td>
</tr>
<tr>
<td>fma fmaf fmal</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>fmax fmaxf fmaxl</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>fmin fminf fminl</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>fmod fmodf fmodl</td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Not reentrant because</td>
</tr>
<tr>
<td>-----------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>fopen</td>
<td>Uses iob[] and calls malloc when file open for buffered IO. See (1)</td>
</tr>
<tr>
<td>fpclassify</td>
<td>-</td>
</tr>
<tr>
<td>fprintf fopenw</td>
<td>Uses iob[]. See (1).</td>
</tr>
<tr>
<td>fputc fputwc</td>
<td>Uses iob[]. See (1).</td>
</tr>
<tr>
<td>fputs fputws</td>
<td>Uses iob[]. See (1).</td>
</tr>
<tr>
<td>fread</td>
<td>Calls fgetc. See (1).</td>
</tr>
<tr>
<td>free</td>
<td>free uses static buffer management structures. See malloc (5).</td>
</tr>
<tr>
<td>freopen</td>
<td>Modifies iob[]. See (1).</td>
</tr>
<tr>
<td>frexp frexpf frexpl</td>
<td></td>
</tr>
<tr>
<td>fscanf fwscanf</td>
<td>Uses iob[]. See (1)</td>
</tr>
<tr>
<td>fseek</td>
<td>Uses iob[] and calls _lseek. Accesses ungetc[] array. See (1).</td>
</tr>
<tr>
<td>fsetpos</td>
<td>Uses iob[] and sets errno. See (1) / (2).</td>
</tr>
<tr>
<td>fstat</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>ftell</td>
<td>Uses iob[] and sets errno. Calls _lseek. See (1) / (2).</td>
</tr>
<tr>
<td>fwrite</td>
<td>Uses iob[]. See (1).</td>
</tr>
<tr>
<td>getc getwc</td>
<td>Uses iob[]. See (1).</td>
</tr>
<tr>
<td>getwchar</td>
<td>Uses iob[]. See (1).</td>
</tr>
<tr>
<td>getcwd</td>
<td>Uses global File System Simulation buffer, _dbg_request</td>
</tr>
<tr>
<td>getenv</td>
<td>Skeleton only.</td>
</tr>
<tr>
<td>gets getws</td>
<td>Uses iob[]. See (1).</td>
</tr>
<tr>
<td>gmtime</td>
<td>gmttime defines static structure</td>
</tr>
<tr>
<td>hypot hypotf hypotl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>ilogb ilogbf ilogbl</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>imaxabs</td>
<td>-</td>
</tr>
<tr>
<td>imaxdiv</td>
<td>-</td>
</tr>
<tr>
<td>isalnum iswalnum</td>
<td>-</td>
</tr>
<tr>
<td>isalpha iswalpha</td>
<td>-</td>
</tr>
<tr>
<td>isascii iswascii</td>
<td>-</td>
</tr>
<tr>
<td>iscntrl iswcntrl</td>
<td>-</td>
</tr>
<tr>
<td>isdigit iswdigit</td>
<td>-</td>
</tr>
<tr>
<td>isfinite</td>
<td>-</td>
</tr>
<tr>
<td>isgraph iswgraph</td>
<td>-</td>
</tr>
<tr>
<td>isgreater</td>
<td>-</td>
</tr>
<tr>
<td>isgreaterequal</td>
<td>-</td>
</tr>
<tr>
<td>isinf</td>
<td>-</td>
</tr>
<tr>
<td>Function</td>
<td>Not reentrant because</td>
</tr>
<tr>
<td>-------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>isless</td>
<td></td>
</tr>
<tr>
<td>islessequal</td>
<td></td>
</tr>
<tr>
<td>islessgreater</td>
<td></td>
</tr>
<tr>
<td>islower iswlower</td>
<td></td>
</tr>
<tr>
<td>isnan</td>
<td></td>
</tr>
<tr>
<td>isnormal</td>
<td></td>
</tr>
<tr>
<td>isprint iswprint</td>
<td></td>
</tr>
<tr>
<td>ispunct iswpunct</td>
<td></td>
</tr>
<tr>
<td>isspace iswspace</td>
<td></td>
</tr>
<tr>
<td>isunordered</td>
<td></td>
</tr>
<tr>
<td>isupper iswupper</td>
<td></td>
</tr>
<tr>
<td>iswalnum</td>
<td></td>
</tr>
<tr>
<td>iswalpha</td>
<td></td>
</tr>
<tr>
<td>iswcntrl</td>
<td></td>
</tr>
<tr>
<td>iswctype</td>
<td></td>
</tr>
<tr>
<td>iswdigit</td>
<td></td>
</tr>
<tr>
<td>iswgraph</td>
<td></td>
</tr>
<tr>
<td>iswlower</td>
<td></td>
</tr>
<tr>
<td>iswprint</td>
<td></td>
</tr>
<tr>
<td>iswpunct</td>
<td></td>
</tr>
<tr>
<td>iswspace</td>
<td></td>
</tr>
<tr>
<td>iswupper</td>
<td></td>
</tr>
<tr>
<td>iswxdigit</td>
<td></td>
</tr>
<tr>
<td>isxdigit iswxdigit</td>
<td></td>
</tr>
<tr>
<td>ldexp ldexpf ldexpl</td>
<td>Sets errno. See (2).</td>
</tr>
<tr>
<td>lgamma lgammaf lgammal</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>llrint lrintf lrintl</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>llround llroundf llroundl</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>localeconv</td>
<td>N.A.; skeleton function</td>
</tr>
<tr>
<td>localtime</td>
<td></td>
</tr>
<tr>
<td>log logf log1</td>
<td>Sets errno. See (2).</td>
</tr>
<tr>
<td>log10 log10f log10l</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>log1p log1pf log1pl</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>log2 log2f log2l</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>logb logbf logbl</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>Function</td>
<td>Not reentrant because</td>
</tr>
<tr>
<td>-------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>longjmp</td>
<td></td>
</tr>
<tr>
<td>lrint  lrintf lrintl</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>lround lroundf lroundl</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>lseek</td>
<td>Calls _lseek</td>
</tr>
<tr>
<td>lstat</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>malloc</td>
<td>Needs kernel support. See (5).</td>
</tr>
<tr>
<td>mblen</td>
<td>N.A., skeleton function</td>
</tr>
<tr>
<td>mbstronc</td>
<td>Sets errno.</td>
</tr>
<tr>
<td>mbstroncs</td>
<td>N.A., skeleton function</td>
</tr>
<tr>
<td>mbstronc</td>
<td>N.A., skeleton function</td>
</tr>
<tr>
<td>memchr wmemchr</td>
<td></td>
</tr>
<tr>
<td>memcmp wmemcmp</td>
<td></td>
</tr>
<tr>
<td>memcpy wmemcpy</td>
<td></td>
</tr>
<tr>
<td>memmove wmemmove</td>
<td></td>
</tr>
<tr>
<td>memset wmemset</td>
<td></td>
</tr>
<tr>
<td>mktime</td>
<td></td>
</tr>
<tr>
<td>modf modff modfl</td>
<td></td>
</tr>
<tr>
<td>nan nanf nanl</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>nearbyint nearbyintf nearbyintl</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>nextafter nextafterf nextafterl</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>nexttoward nexttowardf nexttowardl</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>offsetof</td>
<td></td>
</tr>
<tr>
<td>open</td>
<td>Calls _open</td>
</tr>
<tr>
<td>perror</td>
<td>Uses errno. See (2)</td>
</tr>
<tr>
<td>pow powf powl</td>
<td>Sets errno. See (2)</td>
</tr>
<tr>
<td>printf wprintf</td>
<td>Uses iob[]. See (1)</td>
</tr>
<tr>
<td>putc putwc</td>
<td>Uses iob[]. See (1)</td>
</tr>
<tr>
<td>putchar putwchar</td>
<td>Uses iob[]. See (1)</td>
</tr>
<tr>
<td>puts</td>
<td>Uses iob[]. See (1)</td>
</tr>
<tr>
<td>qsort</td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Not reentrant because</td>
</tr>
<tr>
<td>------------------</td>
<td>---------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>raise</td>
<td>Updates the signal handler table</td>
</tr>
<tr>
<td>rand</td>
<td>Uses static variable to remember latest random number. Must diverge from ISO C standard to define reentrant rand. See (4).</td>
</tr>
<tr>
<td>read</td>
<td>Calls _read</td>
</tr>
<tr>
<td>realloc</td>
<td>See malloc (5).</td>
</tr>
<tr>
<td>remainder remainderf remainderl</td>
<td></td>
</tr>
<tr>
<td>remove</td>
<td>Uses global File System Simulation buffer, _dbg_request</td>
</tr>
<tr>
<td>remquo remquof remquol</td>
<td></td>
</tr>
<tr>
<td>rename</td>
<td>Uses global File System Simulation buffer, _dbg_request</td>
</tr>
<tr>
<td>rewind</td>
<td>Eventually calls _lseek</td>
</tr>
<tr>
<td>rint rintf rintl</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>round roundf roundl</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td>scalbln scalblnf scalblnl</td>
<td>-</td>
</tr>
<tr>
<td>scalbn scalbnf scalbnl</td>
<td>-</td>
</tr>
<tr>
<td>scanf wscanf</td>
<td>Uses iob[ ], calls _doscan. See (1).</td>
</tr>
<tr>
<td>setbuf</td>
<td>Sets iob[ ]. See (1).</td>
</tr>
<tr>
<td>setjmp</td>
<td></td>
</tr>
<tr>
<td>setlocale</td>
<td>N.A.; skeleton function</td>
</tr>
<tr>
<td>setvbuf</td>
<td>Sets iob and calls malloc. See (1) / (5).</td>
</tr>
<tr>
<td>signal</td>
<td>Updates the signal handler table</td>
</tr>
<tr>
<td>signbit</td>
<td></td>
</tr>
<tr>
<td>sin sinf sinl</td>
<td></td>
</tr>
<tr>
<td>sinh sinhf sinhhl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>snprintf swprintf</td>
<td>Sets errno. See (2).</td>
</tr>
<tr>
<td>sprintf</td>
<td>Sets errno. See (2).</td>
</tr>
<tr>
<td>sqrt sqrtf sqrtl</td>
<td>Sets errno. See (2).</td>
</tr>
<tr>
<td>srand</td>
<td>See rand</td>
</tr>
<tr>
<td>sscanf swscanf</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>stat</td>
<td>Uses global File System Simulation buffer, _dbg_request</td>
</tr>
<tr>
<td>strcat wcscat</td>
<td></td>
</tr>
<tr>
<td>strchr wcscchr</td>
<td></td>
</tr>
<tr>
<td>strcmp wcscmp</td>
<td></td>
</tr>
<tr>
<td>strcoll wcscoll</td>
<td></td>
</tr>
<tr>
<td>strncpy wcscpy</td>
<td></td>
</tr>
<tr>
<td>strcspn wcscspn</td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Not reentrant because</td>
</tr>
<tr>
<td>------------------</td>
<td>---------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>strlen wcslen</td>
<td></td>
</tr>
<tr>
<td>strncmp wcsncmp</td>
<td></td>
</tr>
<tr>
<td>strncpy wcsncpy</td>
<td></td>
</tr>
<tr>
<td>strpbrk wcspbrk</td>
<td></td>
</tr>
<tr>
<td>strchr wcsrchr</td>
<td></td>
</tr>
<tr>
<td>strspn wcsspn</td>
<td></td>
</tr>
<tr>
<td>strstr wcsstr</td>
<td></td>
</tr>
<tr>
<td>strtod wcstod</td>
<td></td>
</tr>
<tr>
<td>strtod wcstod</td>
<td></td>
</tr>
<tr>
<td>strtoimax</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>strtok wcstok</td>
<td>strtok saves last position in string in local static variable. This function</td>
</tr>
<tr>
<td></td>
<td>is not reentrant by design. See (4).</td>
</tr>
<tr>
<td>strtol wcstol</td>
<td>Sets errno. See (2).</td>
</tr>
<tr>
<td>strtold wcstold</td>
<td></td>
</tr>
<tr>
<td>strtoul wcstoul</td>
<td>Sets errno. See (2).</td>
</tr>
<tr>
<td>strtoull wcstoull</td>
<td>Sets errno. See (2).</td>
</tr>
<tr>
<td>strtoumax</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>strxfrm wcsxfrm</td>
<td></td>
</tr>
<tr>
<td>system</td>
<td>N.A; skeleton function</td>
</tr>
<tr>
<td>tan tanf tanl</td>
<td>Sets errno. See (2).</td>
</tr>
<tr>
<td>tanh tanhf tanhl</td>
<td>Sets errno via call to other functions.</td>
</tr>
<tr>
<td>tgamma tgammaf</td>
<td>(Not implemented)</td>
</tr>
<tr>
<td></td>
<td>tgammal</td>
</tr>
<tr>
<td>time</td>
<td>Uses static variable which defines initial start time</td>
</tr>
<tr>
<td>tmpfile</td>
<td>Uses iob[]. See (1).</td>
</tr>
<tr>
<td>tmpnam</td>
<td>Uses local buffer to build filename. Function can be adapted to use user buffer. This</td>
</tr>
<tr>
<td></td>
<td>makes the function non ISO C. See (4).</td>
</tr>
<tr>
<td>toascii</td>
<td></td>
</tr>
<tr>
<td>tolower</td>
<td></td>
</tr>
<tr>
<td>toupper</td>
<td></td>
</tr>
<tr>
<td>towctrans</td>
<td></td>
</tr>
<tr>
<td>tolower</td>
<td></td>
</tr>
<tr>
<td>towupper</td>
<td></td>
</tr>
</tbody>
</table>
### Function | Not reentrant because
--- | ---
trunc truncf truncl | *(Not implemented)*
ungetc ungetwc | Uses static buffer to hold unget characters for each file. Can be moved into iob structure. See (1).
unlink | Uses global File System Simulation buffer, _dbg_request
vfprintf vfwprintf | Uses iob[]. See (1).
vfscanf vfswscanf | Calls _doscan
vprintf vwprintf | Uses iob[]. See (1).
vscanf vwscanf | Calls _doscan
vsprintf vswprintf | Sets errno.
vsscanf vswscanf | Sets errno.
wctomb | Sets errno.
wcsrtombs | Sets errno.
wctob | N.A.; skeleton function
wcsstoimax | Sets errno via calls to other functions.
wctombs | N.A.; skeleton function
wcstoumax | Sets errno via calls to other functions.
wctob | -
wctomb | N.A.; skeleton function
wctrans | -
wctype | -
write | Calls _write

### Table: C library reentrancy

Several functions in the C library are not reentrant due to the following reasons:

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

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

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

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

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

(2) **errno declaration**

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

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

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

`errno` will never be set to ERR_NOLONG or ERR_NOPOINT since the C library supports long and pointer conversion routines for input and output.

`errno` can be set to ERANGE by the following functions: `exp()`, `strtol()`, `strtoul()` and `tan()`. These functions may produce results that are out of the valid range for the return type. If so, the result of the function will be the largest representable value for that type and `errno` is set to ERANGE.

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

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

(3) **ungetc**

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

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

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

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

(5) malloc

Malloc uses a heap space which is assigned at locate time. Thus this implementation is not reentrant. Making a reentrant malloc requires some sort of system call to obtain free memory space on a per process basis. This is not easy to solve within the current context of the library. This requires adaptation to a kernel.

This paragraph on reentrancy applies to multi-process environments only. If reentrancy is required for calling library functions from an exception handler, another approach is required. For such a situation it is of no use to allocate e.g. multiple iob[] structures. In such a situation several pieces of code in the library have to be declared 'atomic': this means that interrupts have to be disabled while executing an atomic piece of code.
Chapter 13. List File Formats

This chapter describes the format of the assembler list file and the linker map file.

13.1. Assembler List File Format

The assembler list file is an additional output file of the assembler that contains information about the generated code. For details on how to generate a list file, see Section 7.5, Generating a List File.

The list file consists of a page header and a source listing.

Page header

The page header is repeated on every page:

```
TASKING VX-toolset for TriCore: assembler vx.yrz Build nnn SN 00000000
Title
ADDR CODE      CYCLES  LINE SOURCE LINE
```

The first line contains version information. The second line can contain a title which you can specify with the assembler control `$TITLE` and always contains a page number. The third line is empty and the fourth line contains the headings of the columns for the source listing.

With the assembler controls `$LIST`, `$PAGE`, and with the assembler option `--list-format` you can format the list file.

Source listing

The following is a sample part of a listing. An explanation of the different columns follows below.

```
ADDR CODE      CYCLES  LINE SOURCE LINE
  1         ; Module start
  2         .

0002 85rFrrrr 1  2  27    ld.a    a15,world
0006 F4AF      1  3  28    st16.a  [a10],a15
0008 91r0rr4r 1  4  29    movh.a  a4,#@his(_2_ini)
000C D944rrrr 1  5  30    lea     a4,[a4]@los(_2_ini)
0010 1Drrrrrr 1  6  31    j       printf
  44 buf:    .space  4
```

RESERVED

0000

0003
### ADDR
This column contains the memory address. The address is a hexadecimal number that represents the offset from the beginning of a relocatable section or the absolute address for an absolute section. The address only appears on lines that generate object code.

### CODE
This is the object code generated by the assembler for this source line, displayed in hexadecimal format. The displayed code need not be the same as the generated code that is entered in the object module. The code can also be relocatable code. In this case the letter 'r' is printed for the relocatable code part in the listing. For lines that allocate space, the code field contains the text "RESERVED". For lines that initialize a buffer, the code field lists one value followed by the word "REPEATS".

### CYCLES
The first number in this column is the number of instruction cycles needed to execute the instruction(s) as generated in the CODE field. The second number is the accumulated cycle count of this section.

### LINE
This column contains the line number. This is a decimal number indicating each input line, starting from 1 and incrementing with each source line.

### SOURCE LINE
This column contains the source text. This is a copy of the source line from the assembly source file.

For the .SET and .EQU directives the ADDR and CODE columns do not apply. The symbol value is listed instead.

### 13.2. Linker Map File Format

The linker map file is an additional output file of the linker that shows how the linker has mapped the sections and symbols from the various object files (.o) to output sections. The locate part shows the absolute position of each section. External symbols are listed per space with their absolute address, both sorted on symbol and sorted on address. For details on how to generate a map file, see Section 8.9, Generating a Map File.

With the linker option --map-file-format you can specify which parts of the map file you want to see.

In Eclipse the linker map file (project.map) is generated in the output directory of the build configuration, usually Debug or Release. You can open the map file by double-clicking on the file name.
Each tab displays a part of the map file. When you right-click in the view, a popup menu appears (for example, select Find to find a symbol). The meaning of the different parts is:

### Tool and Invocation

This part of the map file contains information about the linker, its version header information, binary location and which options are used to call it.

### Used Resources

This part of the map file shows the memory usage at memory level and space level. The largest free block of memory (Largest gap) is also shown. This part also contains an estimation of the stack usage.

Explanation of the columns:

**Memory**
The names of the memory as defined in the linker script file (*.lsl).

**Code**
The size of all executable sections.

**Data**
The size of all non-executable sections (not including stacks, heaps, debug sections in non-alloc space).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>malloc.o</td>
<td>bss.libc (55)</td>
<td>0x0000000008</td>
<td>0x0</td>
<td>bss.libc (55)</td>
</tr>
<tr>
<td>_job.o</td>
<td>bss.libc (76)</td>
<td>0x00000000a0</td>
<td>0x0</td>
<td>bss.libc (76)</td>
</tr>
<tr>
<td>atexit.o</td>
<td>bss.libc (87)</td>
<td>0x0000000080</td>
<td>0x0</td>
<td>bss.libc (87)</td>
</tr>
<tr>
<td>dbg.o</td>
<td>bss.libc (91)</td>
<td>0x000000014</td>
<td>0x0</td>
<td>bss.libc (91)</td>
</tr>
<tr>
<td>clockpsrsec.o</td>
<td>.data.libc (50)</td>
<td>0x00000000c8</td>
<td>0x0</td>
<td>.data.libc (50)</td>
</tr>
<tr>
<td>_sbk.o</td>
<td>.data.libc (69)</td>
<td>0x0000000040</td>
<td>0x0</td>
<td>.data.libc (69)</td>
</tr>
<tr>
<td>_job.o</td>
<td>.data.libc (75)</td>
<td>0x00000000c3</td>
<td>0x0</td>
<td>.data.libc (75)</td>
</tr>
<tr>
<td>atexit.o</td>
<td>.data.libc (86)</td>
<td>0x000000001</td>
<td>0x0</td>
<td>.data.libc (86)</td>
</tr>
<tr>
<td>profcalltime.o</td>
<td>.data.procalltime (45)</td>
<td>0x0000000018</td>
<td>0x0</td>
<td>.data.procalltime (45)</td>
</tr>
<tr>
<td>myproject.obj</td>
<td>.rodata.myproject (25)</td>
<td>0x000000002a</td>
<td>0x0</td>
<td>.rodata.myproject (25)</td>
</tr>
<tr>
<td>profcalltime.o</td>
<td>.rodata.procalltime (46)</td>
<td>0x00000006e</td>
<td>0x0</td>
<td>.rodata.procalltime (46)</td>
</tr>
<tr>
<td>cstart.obj</td>
<td>.text.libc (2)</td>
<td>0x000000198</td>
<td>0x0</td>
<td>.text.libc (2)</td>
</tr>
<tr>
<td>cinit.o</td>
<td>.text.libc (47)</td>
<td>0x00000004e</td>
<td>0x0</td>
<td>.text.libc (47)</td>
</tr>
<tr>
<td>clock.o</td>
<td>.text.libc (49)</td>
<td>0x000000120</td>
<td>0x0</td>
<td>.text.libc (49)</td>
</tr>
<tr>
<td>calloc.o</td>
<td>.text.libc (53)</td>
<td>0x00000001c</td>
<td>0x0</td>
<td>.text.libc (53)</td>
</tr>
<tr>
<td>malloc.o</td>
<td>.text.libc (54)</td>
<td>0x0000002c</td>
<td>0x0</td>
<td>.text.libc (54)</td>
</tr>
<tr>
<td>_sbk.o</td>
<td>.text.libc (68)</td>
<td>0x000000046</td>
<td>0x0</td>
<td>.text.libc (68)</td>
</tr>
<tr>
<td>_doprint_int.o</td>
<td>.text.libc (71)</td>
<td>0x000000069a</td>
<td>0x0</td>
<td>.text.libc (71)</td>
</tr>
<tr>
<td>fprintf.o</td>
<td>.text.libc (72)</td>
<td>0x00000001c</td>
<td>0x0</td>
<td>.text.libc (72)</td>
</tr>
<tr>
<td>printf.o</td>
<td>.text.libc (73)</td>
<td>0x000000038</td>
<td>0x0</td>
<td>.text.libc (73)</td>
</tr>
<tr>
<td>_job.o</td>
<td>.text.libc (74)</td>
<td>0x00000001e</td>
<td>0x0</td>
<td>.text.libc (74)</td>
</tr>
<tr>
<td>fprintf.o</td>
<td>.text.libc (76)</td>
<td>0x000000024</td>
<td>0x0</td>
<td>.text.libc (76)</td>
</tr>
<tr>
<td>_close.o</td>
<td>.text.libc (81)</td>
<td>0x0000000b2</td>
<td>0x0</td>
<td>.text.libc (81)</td>
</tr>
<tr>
<td>fclose.o</td>
<td>.text.libc (82)</td>
<td>0x00000006a</td>
<td>0x0</td>
<td>.text.libc (82)</td>
</tr>
</tbody>
</table>
Reserved: The total size of reserved memories, reserved ranges, reserved special sections, stacks, heaps, alignment protections, sections located in non-alloc space (debug sections). In fact, this size is the same as the size in the Total column minus the size of all other columns.

Free: The free memory area addressable by this core. This area is accessible for unrestricted items.

Total: The total memory area addressable by this core.

Space: The names of the address spaces as defined in the linker script file (*.lsl). The names are constructed of the derivative name followed by a colon ':', the core name, another colon ':' and the space name. For example: spe:tc:linear.

Native used ...: The size of sections located in this space.

Foreign used: The size of all sections destined for/located in other spaces, but because of overlap in spaces consume memory in this space.

Stack Name: The name(s) of the stack(s) as defined in the linker script file (*.lsl).

Used: An estimation of the stack usage. The linker calculates the required stack size by using information (.CALLS directives) generated by the compiler. If for example recursion is detected, the calculated stack size is inaccurate, therefore this is an estimation only. The calculated stack size is supposed to be smaller than the actual allocated stack size. If that is not the case, then a warning is given.

Processed Files

This part of the map file shows all processed files. This also includes object files that are extracted from a library, with the symbol that led to the extraction.

Link Result

This part of the map file shows per object file how the link phase has mapped the sections from the various object files (.o) to output sections.


[in] Section: A section name and id from the input object file. The number between '(' ')' uniquely identifies the section.

[in] Size: The size of the input section.

[out] Offset: The offset relative to the start of the output section.

[out] Section: The resulting output section name and id.

[out] Size: The size of the output section.

Module Local Symbols

This part of the map file shows a table for each local scope within an object file. Each table has three columns, 1 the symbol name, 2 the address of the symbol and 3 the space where the symbol resides in. The table is sorted on symbol name within each space.
By default this part is not shown in the map file. You have to turn this part on manually with linker option 
--map-file-format=+statics (module local symbols).

Cross References

This part of the map file lists all symbols defined in the object modules and for each symbol the object modules that contain a reference to the symbol are shown. Also, symbols that remain undefined are shown.

Call Graph

This part of the map file contains a schematic overview that shows how (library) functions call each other. To obtain call graph information, the assembly file must contain .CALLS directives.

You can click the + or - sign to expand or collapse a node.

Overlay

This part is empty for the TriCore.

Locate Result: Sections

This part of the map file shows the absolute position of each section in the absolute object file. It is organized per address space, memory chip and group and sorted on space address.

+ Space  The names of the address spaces as defined in the linker script file (*.lsl). The names are constructed of the derivative name followed by a colon ':', the core name, another colon ':' and the space name. For example: spe:tc:linear.

Chip  The names of the memory chips as defined in the linker script file (*.lsl) in the memory definitions.

Group  Sections can be ordered in groups. These are the names of the groups as defined in the linker script file (*.lsl) with the keyword group in the section_layout definition. The name that is displayed is the name of the deepest nested group.

Section  The name and id of the section. The number between '( )' uniquely identifies the section. Names within square brackets ']' will be copied during initialization from ROM to the corresponding section name in RAM.

Size (MAU)  The size of the section in minimum addressable units.

Space addr  The absolute address of the section in the address space.

Chip addr  The absolute offset of the section from the start of a memory chip.

Locate Result: Symbols

This part of the map file lists all external symbols per address space name.

Address  The absolute address of the symbol in the address space.

Name  The name of the symbol.
The names of the address spaces as defined in the linker script file (*.lsl). The names are constructed of the derivative name followed by a colon ':', the core name, another colon ':' and the space name. For example: spe:tc:linear.

**Processor and Memory**

This part of the map file shows the processor and memory information of the linker script file.

By default this part is not shown in the map file. You have to turn this part on manually with linker option --map-file-format=+lsl (processor and memory info). You can print this information to a separate file with linker option --lsl-dump.

**Locate Rules**

This part of the map file shows the rules the linker uses to locate sections.

**Address space**

The names of the address spaces as defined in the linker script file (*.lsl). The names are constructed of the derivative name followed by a colon ':', the core name, another colon ':' and the space name.

**Type**

The rule type:

ordered/contiguous/clustered/unrestricted

Specifies how sections are grouped. By default, a group is 'unrestricted' which means that the linker has total freedom to place the sections of the group in the address space.

**absolute**

The section must be located at the address shown in the Properties column.

**address range**

The section must be located in the union of the address ranges shown in the Properties column; end addresses are not included in the range.

**address range size**

The sections must be located in some address range with size not larger than shown in the Properties column; the second number in that field is the alignment requirement for the address range.

**ballooned**

After locating all sections, the largest remaining gap in the space is used completely for the stack and/or heap.

**Properties**

The contents depends on the Type column.
The sections to which the rule applies;
restrictions between sections are shown in this column:

< ordered
| contiguous
+ clustered

For contiguous sections, the linker uses the section order as shown here. Clustered sections can be located in any relative order.
Chapter 14. Linker Script Language (LSL)

To make full use of the linker, you can write a script with information about the architecture of the target processor and locating information. The language for the script is called the Linker Script Language (LSL). This chapter first describes the structure of an LSL file. The next section contains a summary of the LSL syntax. In the remaining sections, the semantics of the Linker Script Language is explained.

The TASKING linker is a target independent linker/locator that can simultaneously link and locate all programs for all cores available on a target board. The target board may be of arbitrary complexity. A simple target board may contain one standard processor with some external memory that executes one task. A complex target board may contain multiple standard processors and DSPs combined with configurable IP-cores loaded in an FPGA. Each core may execute a different program, and external memory may be shared by multiple cores.

LSL serves two purposes. First it enables you to specify the characteristics (that are of interest to the linker) of your specific target board and of the cores installed on the board. Second it enables you to specify how sections should be located in memory.

14.1. Structure of a Linker Script File

A script file consists of several definitions. The definitions can appear in any order.

The architecture definition (required)

In essence an architecture definition describes how the linker should convert logical addresses into physical addresses for a given type of core. If the core supports multiple address spaces, then for each space the linker must know how to perform this conversion. In this context a physical address is an offset on a given internal or external bus. Additionally the architecture definition contains information about items such as the (hardware) stack and the interrupt vector table.

This specification is normally written by Altium. Altium supplies LSL files in the include.lsl directory. The architecture definition of the LSL file should not be changed by you unless you also modify the core’s hardware architecture. If the LSL file describes a multi-core system an architecture definition must be available for each different type of core.

See Section 14.4, Semantics of the Architecture Definition for detailed descriptions of LSL in the architecture definition.

The derivative definition

The derivative definition describes the configuration of the internal (on-chip) bus and memory system. Basically it tells the linker how to convert offsets on the buses specified in the architecture definition into offsets in internal memory. Microcontrollers and DSPs often have internal memory and I/O sub-systems apart from one or more cores. The design of such a chip is called a derivative.

Altium provides LSL descriptions of supported derivatives, along with “SFR files”, which provide easy access to registers in I/O sub-systems from C and assembly programs. When you build an ASIC or use a derivative that is not (yet) supported by the TASKING tools, you may have to write a derivative definition.
When you want to use multiple cores of the same type, you must instantiate the cores in a derivative definition, since the linker automatically instantiates only a single core for an unused architecture.


**The processor definition**

The *processor definition* describes an instance of a derivative. Typically the processor definition instantiates one derivative only (single-core processor). A processor that contains multiple cores having the same (homogeneous) or different (heterogeneous) architecture can also be described by instantiating multiple derivatives of the same or different types in separate processor definitions.


**The memory and bus definitions (optional)**

Memory and bus definitions are used within the context of a derivative definition to specify internal memory and on-chip buses. In the context of a board specification the memory and bus definitions are used to define external (off-chip) memory and buses. Given the above definitions the linker can convert a logical address into an offset into an on-chip or off-chip memory device.

See Section 14.6.3, *Defining External Memory and Buses*, for more information on how to specify the external physical memory layout. *Internal* memory for a processor should be defined in the derivative definition for that processor.

**The board specification**

The processor definition and memory and bus definitions together form a *board specification*. LSL provides language constructs to easily describe single-core and heterogeneous or homogeneous multi-core systems. The board specification describes all characteristics of your target board's system buses, memory devices, I/O sub-systems, and cores that are of interest to the linker. Based on the information provided in the board specification the linker can for each core:

- convert a logical address to an offset within a memory device
- locate sections in physical memory
- maintain an overall view of the used and free physical memory within the whole system while locating

**The section layout definition (optional)**

The optional section layout definition enables you to exactly control where input sections are located. Features are provided such as: the ability to place sections at a given load-address or run-time address, to place sections in a given order, and to overlay code and/or data sections.

Which object files (sections) constitute the task that will run on a given core is specified on the command line when you invoke the linker. The linker will link and locate all sections of all tasks simultaneously. From the section layout definition the linker can deduce where a given section may be located in memory,
form the board specification the linker can deduce which physical memory is (still) available while locating the section.

See Section 14.8, *Semantics of the Section Layout Definition*, for more information on how to locate a section at a specific place in memory.

**Skeleton of a Linker Script File**

```plaintext
architecture architecture_name
{
   // Specification core architecture
}

derivative derivative_name
{
   // Derivative definition
}

processor processor_name
{
   // Processor definition
}

memory and/or bus definitions

section_layout space_name
{
   // section placement statements
}
```

### 14.2. Syntax of the Linker Script Language

This section describes what the LSL language looks like. An LSL document is stored as a file coded in UTF-8 with extension `.lsl`. Before processing an LSL file, the linker preprocesses it using a standard C preprocessor. Following this, the linker interprets the LSL file using a scanner and parser. Finally, the linker uses the information found in the LSL file to guide the locating process.

#### 14.2.1. Preprocessing

When the linker loads an LSL file, the linker processes it with a C-style preprocessor. As such, it strips C and C++ comments. You can use the standard ISO C preprocessor directives, such as `#include`, `#define`, `#if/else/#endif`.

For example:

```c
#include "arch.lsl"
```

Preprocess and include the file `arch.lsl` at this point in the LSL file.
14.2.2. Lexical Syntax

The following lexicon is used to describe the syntax of the Linker Script Language:

A ::= B = A is defined as B
A ::= B C = A is defined as B and C; B is followed by C
A ::= B | C = A is defined as B or C
<B>^0|^1 = zero or one occurrence of B
<B>^>=0 = zero of more occurrences of B
<B>^>=1 = one of more occurrences of B
IDENTIFIER = a character sequence starting with 'a'-'z', 'A'-'Z' or '_'. Following characters may also be digits and dots '.'
STRING = sequence of characters not starting with \n, \r or \t
DQSTRING = " STRING " (double quoted string)
OCT_NUM = octal number, starting with a zero (06, 045)
DEC_NUM = decimal number, not starting with a zero (14, 1024)
HEX_NUM = hexadecimal number, starting with '0x' (0x0023, 0xFF00)

OCT_NUM, DEC_NUM and HEX_NUM can be followed by a k (kilo), M (mega), or G (giga).

Characters in **bold** are characters that occur literally. Words in *italics* are higher order terms that are defined in the same or in one of the other sections.

To write comments in LSL file, you can use the C style '/* */' or C++ style '//'.

14.2.3. Identifiers

arch_name ::= IDENTIFIER
bus_name ::= IDENTIFIER
core_name ::= IDENTIFIER
derivative_name ::= IDENTIFIER
file_name ::= DQSTRING
group_name ::= IDENTIFIER
mem_name ::= IDENTIFIER
proc_name ::= IDENTIFIER
section_name ::= DQSTRING
space_name ::= IDENTIFIER
stack_name ::= section_name
symbol_name ::= DQSTRING
14.2.4. Expressions

The expressions and operators in this section work the same as in ISO C.

```plaintext
number ::= OCT_NUM  # octal number
       | DEC_NUM  # decimal number
       | HEX_NUM  # hexadecimal number

expr ::= number
       | symbol_name
       | unary_op expr
       | expr binary_op expr
       | expr ? expr : expr
       | ( expr )
       | function_call

unary_op ::= !  // logical NOT
           | ~  // bitwise complement
           | -  // negative value

binary_op ::= ^  // exclusive OR
             | *  // multiplication
             | /  // division
             | %  // modulus
             | +  // addition
             | -  // subtraction
             | >>  // right shift
             | <<  // left shift
             | ==  // equal to
             | !=  // not equal to
             | >   // greater than
             | <   // less than
             | >=  // greater than or equal to
             | <=  // less than or equal to
             | &   // bitwise AND
             | |   // bitwise OR
             | &&  // logical AND
             | ||  // logical OR
```

14.2.5. Built-in Functions

```plaintext
function_call ::= absolute ( expr )
                | addressof ( addr_id )
                | exists ( section_name )
                | max ( expr , expr )
                | min ( expr , expr )
                | sizeof ( size_id )

addr_id ::= sect : section_name
           | group : group_name
```
Every space, bus, memory, section or group you refer to, must be defined in the LSL file.

The `addressof()` and `sizeof()` functions with the `group` or `sect` argument can only be used in the right hand side of an assignment. The `sizeof()` function with the `mem` argument can be used anywhere in section layouts.

You can use the following built-in functions in expressions. All functions return a numerical value. This value is a 64-bit signed integer.

**absolute()**

```c
int absolute( expr )
```

Converts the value of `expr` to a positive integer.

```c
absolute( "labelA"-"labelB" )
```

**addressof()**

```c
int addressof( addr_id )
```

Returns the address of `addr_id`, which is a named section or group. To get the offset of the section with the name `asect`:

```c
addressof( sect: "asect"
```

This function only works in assignments.

**exists()**

```c
int exists( section_name )
```

The function returns 1 if the section `section_name` exists in one or more object file, 0 otherwise. If the section is not present in input object files, but generated from LSL, the result of this function is undefined.

To check whether the section `mysection` exists in one of the object files that is specified to the linker:

```c
exists( "mysection"
```

**max()**

```c
int max( expr, expr )
```

Returns the value of the expression that has the largest value. To get the highest value of two symbols:

```c
max( "sym1", "sym2"
```
**min()**

```c
int min( expr, expr )
```

Returns the value of the expression that has the smallest value. To get the lowest value of two symbols:

```c
min( "sym1", "sym2")
```

**sizeof()**

```c
int sizeof( size_id )
```

Returns the size of the object (group, section or memory) the identifier refers to. To get the size of the section "asection":

```c
sizeof( sect: "asection")
```

The **group** and **sect** arguments only works in assignments. The **mem** argument can be used anywhere in section layouts.

### 14.2.6. LSL Definitions in the Linker Script File

```c
description ::= <definition>^{=-1}
definition ::= architecture_definition  
              | derivative_definition  
              | board_spec  
              | section_definition  
              | section_setup
```

- At least one **architecture_definition** must be present in the LSL file.

### 14.2.7. Memory and Bus Definitions

```c
mem_def ::= memory mem_name { <mem_descr ;>^{=0} }
```

- A **mem_def** defines a memory with the **mem_name** as a unique name.

```c
mem_descr ::= type = <reserved>^0|1 mem_type  
             | mau = expr  
             | size = expr  
             | speed = number  
             | mapping
```

- A **mem_def** contains exactly one **type** statement.
- A **mem_def** contains exactly one **mau** statement (non-zero size).
- A **mem_def** contains exactly one **size** statement.
- A **mem_def** contains zero or one **speed** statement (default value is 1).
A `mem_def` contains at least one `mapping`

```
mem_type ::= rom // attrs = rx
  | ram // attrs = rw
  | nvram // attrs = rwx
```

```
bus_def ::= bus bus_name { <bus_descr ;>^0 }
```

A `bus_def` statement defines a bus with the given `bus_name` as a unique name within a core architecture.

```
busedescr ::= mau = expr
  | width = expr // bus width, nr
  | mapping // legal destination
    // 'bus' only
```

- The `mau` and `width` statements appear exactly once in a `bus_descr`. The default value for `width` is the `mau` size.
- The bus width must be an integer times the bus MAU size.
- The MAU size must be non-zero.
- A bus can only have a `mapping` on a destination bus (through `dest = bus:`).

```
mapping ::= map ( map_descr <, map_descr>^0 )
```

```
map_descr ::= dest = destination
  | dest_dbits = range
  | dest_offset = expr
  | size = expr
  | src_dbits = range
  | src_offset = expr
```

- A `mapping` requires at least the `size` and `dest` statements.
- Each `map_descr` can occur only once.
- You can define multiple mappings from a single source.
- Overlap between source ranges or destination ranges is not allowed.
- If the `src_dbits` or `dest_dbits` statement is not present, its value defaults to the `width` value if the source/destination is a bus, and to the `mau` size otherwise.

```
destination ::= space : space_name
  | bus : <proc_name |
    core_name :>^0|1 bus_name
```

- A `space_name` refers to a defined address space.
- A `proc_name` refers to a defined processor.
• A core_name refers to a defined core.
• A bus_name refers to a defined bus.
• The following mappings are allowed (source to destination)
  • space => space
  • space => bus
  • bus => bus
  • memory => bus

\[
\text{range} \quad ::= \text{expr} \cdot \text{expr}
\]

14.2.8. Architecture Definition

architecture_definition
\[::= \text{architecture arch_name}
\quad <( \text{parameter_list})>^{0|1}
\quad <\text{extends arch_name}
\quad <( \text{argument_list})>^{0|1} >^{0|1}
\quad \{ <\text{arch_spec}>^{0} \}
\]

• An architecture_definition defines a core architecture with the given arch_name as a unique name.
• At least one space_def and at least one bus_def have to be present in an architecture_definition.
• An architecture_definition that uses the extends construct defines an architecture that inherits all elements of the architecture defined by the second arch_name. The parent architecture must be defined in the LSL file as well.

parameter_list
\[::= \text{parameter} <, \text{parameter}>^{0|0}
\]
parameter
\[::= \text{IDENTIFIER} <\text{expr}>^{0|1}
\]
argument_list
\[::= \text{expr} <, \text{expr}>^{0|0}
\]
arch_spec
\[::= \text{bus_def}
\quad | \text{space_def}
\quad | \text{endianness_def}
\]

space_def
\[::= \text{space space_name }\{ <\text{space_descr}>^{0} \}
\]

• A space_def defines an address space with the given space_name as a unique name within an architecture.

space_descr
\[::= \text{space_property ;}
\quad | \text{section_definition} //no space ref
\]
vector_table_statement
| reserved_range

space_property ::= id = number // as used in object
| mau = expr
| align = expr
| page_size = expr < [ range ] < [ range ] >= 0 | 1
| page
| direction = direction
| stack_def
| heap_def
| copy_table_def
| start_address
| mapping

- A `space_def` contains exactly one `id` and one `mau` statement.
- A `space_def` contains at most one `align` statement.
- A `space_def` contains at most one `page_size` statement.
- A `space_def` contains at most one `mapping`.

`stack_def` ::= stack stack_name ( stack_heap_descr <, stack_heap_descr >= 0 )

- A `stack_def` defines a stack with the `stack_name` as a unique name.

`heap_def` ::= heap heap_name ( stack_heap_descr <, stack_heap_descr >= 0 )

- A `heap_def` defines a heap with the `heap_name` as a unique name.

`stack_heap_descr` ::= min_size = expr
| grows = direction
| align = expr
| fixed
| id = expr

- The `min_size` statement must be present.
- You can specify at most one `align` statement and one `grows` statement.
- Each stack definition has its own unique `id`, the number specified corresponds to the index in the `.CALLS` directive as generated by the compiler.

`direction` ::= low_to_high
| high_to_low

- If you do not specify the `grows` statement, the stack and heap grow `low-to-high`.

`copy_table_def` ::= copytable <( copy_table_descr <, copy_table_descr >= 0 ) >= 0 | 1
• A space_def contains at most one copytable statement.
• Exactly one copy table must be defined in one of the spaces.

\[
\text{copy_table_descr} ::= \text{align} = \text{expr} \\
| \text{copy_unit} = \text{expr} \\
| \text{dest} <\text{space_name}>^{0|1} = \text{space_name} \\
| \text{page}
\]

• The copy_unit is defined by the size in MAUs in which the startup code moves data.
• The dest statement is only required when the startup code initializes memory used by another processor that has no access to ROM.

• A space_name refers to a defined address space.

\[
\text{start_addr} ::= \text{start_address} (\text{start_addr descr}<, \text{start_addr descr}>^{0} )
\]

\[
\text{start_addr descr} ::= \text{run_addr} = \text{expr} \\
| \text{symbol} = \text{symbol_name}
\]

• A symbol_name refers to the section that contains the startup code.

\[
\text{vector_table_statement} ::= \text{vector_table} \text{section_name} \\
(\text{vecttab_spec}<, \text{vecttab_spec}>^{0} ) \\
{<\text{vector_def}>^{0} }
\]

\[
\text{vecttab_spec} ::= \text{vector_size} = \text{expr} \\
| \text{size} = \text{expr} \\
| \text{id_symbol_prefix} = \text{symbol_name} \\
| \text{run_addr} = \text{addr Absolute} \\
| \text{template} = \text{section_name} \\
| \text{template_symbol} = \text{symbol_name} \\
| \text{vector_prefix} = \text{section_name} \\
| \text{fill} = \text{vector_value} \\
| \text{no_inline} \\
| \text{copy}
\]

\[
\text{vector_def} ::= \text{vector} (\text{vector_spec}<, \text{vector_spec}>^{0}) ;
\]

\[
\text{vector_spec} ::= \text{id} = \text{vector_id_spec} \\
| \text{fill} = \text{vector_value}
\]

\[
\text{vector_id_spec} ::= \text{number} \\
| \text{[range]}<[, \text{range}]^{0}>^{0}
\]

\[
\text{vector_value} ::= \text{symbol_name} \\
| \text{[number}<, \text{number}]^{0} >^{0} \\
| \text{loop} <[\text{expr}]^{0|1}
\]

\[
\text{reserved_range} ::= \text{reserved expr .. expr} ;
\]
14.2.9. Derivative Definition

A derivative_definition defines a derivative with the given derivative_name as a unique name.

14.2.10. Processor Definition and Board Specification

A proc_def defines a processor with the proc_name as a unique name.
If you do not explicitly define a processor for a derivative in an LSL file, the linker defines a processor with the same name as that derivative.

A derivative_name refers to a defined derivative.

A proc_def contains exactly one derivative statement.

14.2.11. Section Layout Definition and Section Setup

section_definition ::= section_layout <space_ref>^0^1
                     ( locate_direction )^0^1
                     { <section_statment> }^>=0^ }

A section definition inside a space definition does not have a space_ref.

All global section definitions have a space_ref.

space_ref ::= <proc_name>^0^1 : <core_name>^0^1
              : space_name

If more than one processor is present, the proc_name must be given for a global section layout.

If the section layout refers to a processor that has more than one core, the core_name must be given in the space_ref.

A proc_name refers to a defined processor.

A core_name refers to a defined core.

A space_name refers to a defined address space.

locate_direction ::= direction = direction

direction ::= low_to_high
              | high_to_low

A section layout contains at most one direction statement.

If you do not specify the direction statement, the locate direction of the section layout is low-to-high.

section_statement ::= simple_section_statement ;
                     | aggregate_section_statement

simple_section_statement ::= assignment
                           | select_section_statement
                           | special_section_statement

assignment ::= symbol_name assign_op expr
assign_op ::= =
| ::= 

select_section_statement ::= select <ref_tree>0|1 <section_name>0|1
| <section_selections>0|1

• Either a section_name or at least one section_selection must be defined.

section_selections ::= ( section_selection
| , section_selection>0 )

section_selection ::= attributes = < <+|-> attribute>0

• +attribute means: select all sections that have this attribute.

• -attribute means: select all sections that do not have this attribute.

special_section_statement ::= heap stack_name <size_spec>0|1
| stack stack_name <size_spec>0|1
| copytable
| reserved section_name <reserved_specs>0|1

• Special sections cannot be selected in load-time groups.

size_spec ::= ( size = expr )

reserved_specs ::= ( reserved_spec <, reserved_spec>0 )

reserved_spec ::= attributes
| fill_spec
| size = expr
| alloc_allowed = absolute

• If a reserved section has attributes r, rw, x, rx or rwx, and no fill pattern is defined, the section is filled with zeros. If no attributes are set, the section is created as a scratch section (attributes ws, no image).

fill_spec ::= fill = fill_values

fill_values ::= expr
| [ expr <, expr>0 ]

aggregate_section_statement ::= { <section_statement>0 }
| group_descr
| if_statement
| section_creation_statement
group descr ::= group <group_name>0|1 <( group_specs )>0|1
        section_statement

- No two groups for an address space can have the same group_name.

group_specs ::= group_spec , group_spec >^0

group_spec ::= group_alignment
        attributes
        copy
        nocopy
        group_load_address
        fill <= fill_values>0|1
        group_page
        group_run_address
        group_type
        allow_cross_references
        priority = number

- The allow-cross-references property is only allowed for overlay groups.

- Sub groups inherit all properties from a parent group.

group_alignment ::= align = expr

attributes ::= attributes = <attribute>^1

attribute ::= r // readable sections
        w // writable sections
        x // executable code sections
        i // initialized sections
        s // scratch sections
        b // blanked (cleared) sections

group_load_address ::= load_addr <= load_or_run_addr>0|1

group_page ::= page <= expr>0|1
        page_size = expr <[ range ] <[ range ]>^0>0|1

group_run_address ::= run_addr <= load_or_run_addr>0|1

group_type ::= clustered
        contiguous
        ordered
        overlay

- For non-contiguous groups, you can only specify group_alignment and attributes.

- The overlay keyword also sets the contiguous property.

- The clustered property cannot be set together with contiguous or ordered on a single group.
load_or_run_addr ::= addr_absolute
                   | addr_range < | addr_range >= 0

addr_absolute ::= expr
                | memory_reference [ expr ]

- An absolute address can only be set on ordered groups.

addr_range ::= [ expr .. expr ]
              | memory_reference
              | memory_reference [ expr .. expr ]

- The parent of a group with an addr_range or page restriction cannot be ordered, contiguous or clustered.

memory_reference ::= mem : <proc_name :>0|1 <core_name :>0|1 mem_name

- A proc_name refers to a defined processor.
- A core_name refers to a defined core.
- A mem_name refers to a defined memory.

if_statement ::= if ( expr ) section_statement
               <else section_statement>0|1

section_creation_statement
 ::= section section_name ( section_specs )
   { <section_statement2> >= 0 }

section_specs ::= section_spec <, section_spec > >= 0

section_spec ::= attributes
                | fill_spec
                | size = expr
                | blocksize = expr
                | overflow = section_name

section_statement2 ::= select_section_statement ;
                     | group_descr2
                     | { <section_statement2> >= 0 }

group_descr2 ::= group <group_name>0|1
               ( group_specs2 )
               section_statement2

group_specs2 ::= group_spec2 <, group_spec2 > >= 0

group_spec2 ::= group_alignment
              | attributes
              | load_addr
14.3. Expression Evaluation

Only *constant* expressions are allowed, including sizes, but not addresses, of sections in object files.

All expressions are evaluated with 64-bit precision integer arithmetic. The result of an expression can be absolute or relocatable. A symbol you assign is created as an absolute symbol.

14.4. Semantics of the Architecture Definition

**Keywords in the architecture definition**

```plaintext
architecture                        extends
endianness           big  little
bus                 mau
                     width
                     map
space                 id
                     mau
                     align
                     page_size
                     page
                     direction low_to_high  high_to_low
stack                 min_size
                     grows low_to_high  high_to_low
                     align
                     fixed
                     id
heap                  min_size
                     grows low_to_high  high_to_low
                     align
                     fixed
                     id
copytable            align
                     copy_unit
                     dest
```
14.4.1. Defining an Architecture

With the keyword `architecture` you define an architecture and assign a unique name to it. The name is used to refer to it at other places in the LSL file:

architecture name
{
  definitions
}

If you are defining multiple core architectures that show great resemblance, you can define the common features in a parent core architecture and extend this with a child core architecture that contains specific features. The child inherits all features of the parent. With the keyword `extends` you create a child core architecture:

architecture name_child_arch extends name_parent_arch
{
  definitions
}

A core architecture can have any number of parameters. These are identifiers which get values assigned on instantiation or extension of the architecture. You can use them in any expression within the core
architecture. Parameters can have default values, which are used when the core architecture is instantiated with less arguments than there are parameters defined for it. When you extend a core architecture you can pass arguments to the parent architecture. Arguments are expressions that set the value of the parameters of the sub-architecture.

```plaintext
architecture name_child_arch (parm1, parm2=1)
    extends name_parent_arch (arguments)
{
    definitions
}
```

### 14.4.2. Defining Internal Buses

With the `bus` keyword you define a bus (the combination of data and corresponding address bus). The bus name is used to identify a bus and does not conflict with other identifiers. Bus descriptions in an architecture definition or derivative definition define internal buses. Some internal buses are used to communicate with the components outside the core or processor. Such buses on a processor have physical pins reserved for the number of bits specified with the `width` statements.

- The `mau` field specifies the MAU size (Minimum Addressable Unit) of the data bus. This field is required.
- The `width` field specifies the width (number of address lines) of the data bus. The default value is the MAU size.
- The `map` keyword specifies how this bus maps onto another bus (if so). Mappings are described in Section 14.4.4, Mappings.

```plaintext
bus bus_name
{
    mau = 8;
    width = 8;
    map ( map_description );
}
```

### 14.4.3. Defining Address Spaces

With the `space` keyword you define a logical address space. The space name is used to identify the address space and does not conflict with other identifiers.

- The `id` field defines how the addressing space is identified in object files. In general, each address space has a unique ID. The linker locates sections with a certain ID in the address space with the same ID. This field is required.
- The `mau` field specifies the MAU size (Minimum Addressable Unit) of the space. This field is required.
- The `align` value must be a power of two. The linker uses this value to compute the start addresses when sections are concatenated. An align value of $n$ means that objects in the address space have to be aligned on $n$ MAUs.
- The `page_size` field sets the page alignment and page size in MAUs for the address space. It must be a power of 2. The default value is 1. If one or more page ranges are supplied the supplied value
only sets the page alignment. The ranges specify the available space in each page, as offsets to the page start, which is aligned at the page alignment.

See also the `page` keyword in subsection Locating a group in Section 14.8.2, Creating and Locating Groups of Sections.

• With the optional `direction` field you can specify how all sections in this space should be located. This can be either from `low_to_high` addresses (this is the default) or from `high_to_low` addresses.

• The `map` keyword specifies how this address space maps onto an internal bus or onto another address space. Mappings are described in Section 14.4.4, Mappings.

**Stacks and heaps**

• The `stack` keyword defines a stack in the address space and assigns a name to it. The architecture definition must contain at least one stack definition. Each stack of a core architecture must have a unique name. See also the `stack` keyword in Section 14.8.3, Creating or Modifying Special Sections.

The stack is described in terms of a minimum size (`min_size`) and the direction in which the stack grows (`grows`). This can be either from `low_to_high` addresses (stack grows upwards, this is the default) or from `high_to_low` addresses (stack grows downwards). The `min_size` is required.

By default, the linker tries to maximize the size of the stacks and heaps. After locating all sections, the largest remaining gap in the space is used completely for the stacks and heaps. If you specify the keyword `fixed`, you can disable this so-called ‘balloon behavior’. The size is also fixed if you used a stack or heap in the software layout definition in a restricted way. For example when you override a stack with another size or select a stack in an ordered group with other sections.

The `id` keyword matches stack information generated by the compiler with a stack name specified in LSL. This value assigned to this keyword is strongly related to the compiler’s output, so users are not supposed to change this configuration.

Optionally you can specify an alignment for the stack with the argument `align`. This alignment must be equal or larger than the alignment that you specify for the address space itself.

• The `heap` keyword defines a heap in the address space and assigns a name to it. The definition of a heap is similar to the definition of a stack. See also the `heap` keyword in Section 14.8.3, Creating or Modifying Special Sections.

Stacks and heaps are only generated by the linker if the corresponding linker labels are referenced in the object files.

See Section 14.8, Semantics of the Section Layout Definition, for information on creating and placing stack sections.

**Copy tables**

• The `copytable` keyword defines a copy table in the address space. The content of the copy table is created by the linker and contains the start address and size of all sections that should be initialized by the startup code. You must define exactly one copy table in one of the address spaces (for a core).
Optionally you can specify an alignment for the copy table with the argument `align`. This alignment must be equal or larger than the alignment that you specify for the address space itself. If smaller, the alignment for the address space is used.

The `copy_unit` argument specifies the size in MAUs of information chunks that are copied. If you do not specify the copy unit, the MAU size of the address space itself is used.

The `dest` argument specifies the destination address space that the code uses for the copy table. The linker uses this information to generate the correct addresses in the copy table. The memory into where the sections must be copied at run-time, must be accessible from this destination space.

Sections generated for the copy table may get a page restriction with the address space’s page size, by adding the `page` argument.

**Vector table**

- The `vector_table` keyword defines a vector table with `n` vectors of size `m` (This is an internal LSL object similar to an LSL group.) The `run_addr` argument specifies the location of the first vector (id=0). This can be a simple address or an offset in memory (see the description of the run-time address in subsection Locating a group in Section 14.8.2, Creating and Locating Groups of Sections). A vector table defines symbols `_lc_ub_foo` and `_lc_ue_foo` pointing to start and end of the table.

```ml
vector_table "vtable" (vector_size=m, size=n, run_addr=x, ...)
```

See the following example of a vector table definition:

```ml
vector_table "vtable" (vector_size = 4, size = 256, run_addr=0,
    template=".text.vector_template",
    template_symbol="_lc_vector_target",
    vector_prefix=".text.vector.",
    id_symbol_prefix="foo",
    no_inline,
    /* default: empty, or */
    fill="foo", /* or */
    fill=[1,2,3,4], /* or */
    fill=loop)
{
    vector (id=23, fill="main");
    vector (id=12, fill=[0xab, 0x21, 0x32, 0x43]);
    vector (id=[1..11], fill=[0]);
    vector (id=[18..23], fill=loop);
}
```

The `template` argument defines the name of the section that holds the code to jump to a handler function from the vector table. This template section does not get located and is removed when the locate phase is completed. This argument is required.

The `template_symbol` argument is the symbol reference in the template section that must be replaced by the address of the handler function. This symbol name should start with the linker prefix for the symbol to be ignored in the link phase. This argument is required.
The **vector_prefix** argument defines the names of vector sections: the section for a vector with id **vector_id** is $(vector_prefix)$(vector_id). Vectors defined in C or assembly source files that should be included in the vector table must have the correct symbol name. The compiler uses the prefix that is defined in the default LSL file(s); if this attribute is changed, the vectors declared in C source files are not included in the vector table. When a vector supplied in an object file has exactly one relocation, the linker will assume it is a branch to a handler function, and can be removed when the handler is inlined in the vector table. Otherwise, no inlining is done. This argument is required.

With the optional **no_inline** argument the vectors handlers are not inlined in the vector table.

With the optional **copy** argument a ROM copy of the vector table is made and the vector table is copied to RAM at startup.

With the optional **id_symbol_prefix** argument you can set an internal string representing a symbol name prefix that may be found on symbols in vector handler code. When the linker detects such a symbol in a handler, the symbol is assigned the vector number. If the symbol was already assigned a vector number, a warning is issued.

The **fill** argument sets the default contents of vectors. If nothing is specified for a vector, this setting is used. See below. When no default is provided, empty vectors may be used to locate large vector handlers and other sections. Only one **fill** argument is allowed.

The **vector** field defines the content of vector with the number specified by **id**. If a range is specified for **id** ([p..q,s..t]) all vectors in the ranges (inclusive) are defined the same way.

With **fill**=symbol_name, the vector must jump to this symbol. If the section in which the symbol is defined fits in the vector table (size may be >m), locate the section at the location of the vector. Otherwise, insert code to jump to the symbol's value. A template interrupt handler section name + symbol name for the target code must be supplied in the LSL file.

**fill**=[value(s)], fills the vector with the specified MAU values.

With **fill**=loop the vector jumps to itself. With the optional **offset** you can specify an offset from the vector table entry.

**Reserved address ranges**

- The **reserved** keyword specifies to reserve a part of an address space even if not all of the range is covered by memory. See also the **reserved** keyword in Section 14.8.3, Creating or Modifying Special Sections.

**Start address**

- The **start_address** keyword specifies the start address for the position where the C startup code is located. When a processor is reset, it initializes its program counter to a certain start address, sometimes called the reset vector. In the architecture definition, you must specify this start address in the correct address space in combination with the name of the label in the application code which must be located here.

The **run_addr** argument specifies the start address (reset vector). If the core starts executing using an entry from a vector table, and directly jumps to the start label, you should omit this argument.
The **symbol** argument specifies the name of the label in the application code that should be located at the specified start address. The **symbol** argument is required. The linker will resolve the start symbol and use its value after locating for the start address field in IEEE-695 files and Intel Hex files. If you also specified the **run_addr** argument, the start symbol (label) must point to a section. The linker locates this section such that the start symbol ends up on the start address.

```
space space_name{
    id = 1;
    mau = 8;
    align = 8;
    page_size = 1;
    stack name (min_size = 1k, grows = low_to_high);
    reserved start_address .. end_address;
    start_address (run_addr = 0x0000,
                   symbol = "start_label")
    map (map_description);
}
```

### 14.4.4. Mappings

You can use a mapping when you define a space, bus or memory. With the **map** field you specify how addresses from the source (space, bus or memory) are translated to addresses of a destination (space, bus). The following mappings are possible:

- space => space
- space => bus
- bus => bus
- memory => bus

With a mapping you specify a range of source addresses you want to map (specified by a source offset and a size), the destination to which you want to map them (a bus or another address space), and the offset address in the destination.

- The **dest** argument specifies the destination. This can be a **bus** or another address **space** (only for a space to space mapping). This argument is required.
- The **src_offset** argument specifies the offset of the source addresses. In combination with size, this specifies the range of address that are mapped. By default the source offset is 0x0000.
- The **size** argument specifies the number of addresses that are mapped. This argument is required.
- The **dest_offset** argument specifies the position in the destination to which the specified range of addresses is mapped. By default the destination offset is 0x0000.

If you are mapping a bus to another bus, the number of data lines of each bus may differ. In this case you have to specify a range of source data lines you want to map (**src_dbits = begin..end**) and the range of destination data lines you want to map them to (**dest_dbits = first..last**).
• The src_dbits argument specifies a range of data lines of the source bus. By default all data lines are mapped.

• The dest_dbits argument specifies a range of data lines of the destination bus. By default, all data lines from the source bus are mapped on the data lines of the destination bus (starting with line 0).

**From space to space**

If you map an address space to another address space (nesting), you can do this by mapping the subspace to the containing larger space. In this example a small space of 64 kB is mapped on a large space of 16 MB.

```
space small
{  
    id = 2;  
    mau = 4;  
    map (src_offset = 0, dest_offset = 0,  
         dest = space : large, size = 64k);  
}
```

**From space to bus**

All spaces that are not mapped to another space must map to a bus in the architecture:

```
space large
{  
    id = 1;  
    mau = 4;  
    map (src_offset = 0, dest_offset = 0,  
         dest = bus:bus_name, size = 16M );  
}
```

**From bus to bus**

The next example maps an external bus called e_bus to an internal bus called i_bus. This internal bus resides on a core called mycore. The source bus has 16 data lines whereas the destination bus has only 8 data lines. Therefore, the keywords src_dbits and dest_dbits specify which source data lines are mapped on which destination data lines.

```
arichitecture mycore  
{  
    bus i_bus  
    {  
        mau = 4;  
    }  
    space i_space  
    {  
        map (dest=bus:i_bus, size=256);  
    }  
}
```
bus e_bus
{
    mau = 16;
    width = 16;
    map (dest = bus:mycore:i_bus, src_dbits = 0..7, dest_dbits = 0..7 )
}

It is not possible to map an internal bus to an external bus.

14.5. Semantics of the Derivative Definition

Keywords in the derivative definition

derivative             extends          core
                      architecture     bus
                      mau              width
                      map
memory
                      type             reserved rom  ram  nvram
                      mau
                      size
                      speed
                      map
section_layout
section_setup

map

                      dest          bus space
                      dest_dbits
                      dest_offset
                      size
                      src_dbits
                      src_offset

14.5.1. Defining a Derivative

With the keyword derivative you define a derivative and assign a unique name to it. The name is used to refer to it at other places in the LSL file:

derivative name
{
    definitions
}
If you are defining multiple derivatives that show great resemblance, you can define the common features in a parent derivative and extend this with a child derivative that contains specific features. The child inherits all features of the parent (cores and memories). With the keyword `extends` you create a child derivative:

```plaintext
derivative name_child_deriv extends name_parent_deriv
{
   definitions
}
```

As with a core architecture, a derivative can have any number of parameters. These are identifiers which get values assigned on instantiation or extension of the derivative. You can use them in any expression within the derivative definition.

```plaintext
derivative name_child_deriv (parm1, parm2=1)
   extends name_parent_deriv (arguments)
{
   definitions
}
```

### 14.5.2. Instantiating Core Architectures

With the keyword `core` you instantiate a core architecture in a derivative.

- With the keyword `architecture` you tell the linker that the given core has a certain architecture. The architecture name refers to an existing architecture definition in the same LSL file.

  For example, if you have two cores (called `mycore_1` and `mycore_2`) that have the same architecture (called `mycorearch`), you must instantiate both cores as follows:

  ```plaintext
core mycore_1
  {
    architecture = mycorearch;
  }

core mycore_2
  {
    architecture = mycorearch;
  }
```

If the architecture definition has parameters you must specify the arguments that correspond with the parameters. For example `mycorearch1` expects two parameters which are used in the architecture definition:

  ```plaintext
core mycore
  {
    architecture = mycorearch1 (1,2);
  }
```
14.5.3. Defining Internal Memory and Buses

With the `memory` keyword you define physical memory that is present on the target board. The memory name is used to identify the memory and does not conflict with other identifiers. It is common to define internal memory (on-chip) in the derivative definition. External memory (off-chip memory) is usually defined in the board specification (See Section 14.6.3, Defining External Memory and Buses).

- The `type` field specifies a memory type:
  - `rom`: read-only memory - it can only be written at load-time
  - `ram`: random access volatile writable memory - writing at run-time is possible while writing at load-time has no use since the data is not retained after a power-down
  - `nvram`: non volatile ram - writing is possible both at load-time and run-time

The optional `reserved` qualifier before the memory type, tells the linker not to locate any section in the memory by default. You can locate sections in such memories using an absolute address or range restriction (see subsection Locating a group in Section 14.8.2, Creating and Locating Groups of Sections).

- The `mau` field specifies the MAU size (Minimum Addressable Unit) of the memory. This field is required.

- The `size` field specifies the size in MAU of the memory. This field is required.

- The `speed` field specifies a symbolic speed for the memory (0..4): 0 is the fastest, 4 the slowest. The linker uses the relative speed of the memories in such a way, that optimal speed is achieved. The default speed is 1.

- The `map` field specifies how this memory maps onto an (internal) bus. Mappings are described in Section 14.4.4, Mappings.

```
memory  mem_name
{
    type  =  rom;
    mau   =  8;
    size  =  64k;
    speed =  2;
    map   ( map_description );
}
```

With the `bus` keyword you define a bus in a derivative definition. Buses are described in Section 14.4.2, Defining Internal Buses.


Keywords in the board specification

```
processor
derivative
bus
    mau
```
## 14.6.1. Defining a Processor

If you have a target board with multiple processors that have the same derivative, you need to instantiate each individual processor in a processor definition. This information tells the linker which processor has which derivative and enables the linker to distinguish between the present processors.

If you use processors that all have a unique derivative, you may omit the processor definitions. In this case the linker assumes that for each derivative definition in the LSL file there is one processor. The linker uses the derivative name also for the processor.

With the keyword `processor` you define a processor. You can freely choose the processor name. The name is used to refer to it at other places in the LSL file:

```plaintext
processor proc_name
{
    processor definition
}
```

### 14.6.2. Instantiating Derivatives

With the keyword `derivative` you tell the linker that the given processor has a certain derivative. The derivative name refers to an existing derivative definition in the same LSL file.

For example, if you have two processors on your target board (called `myproc_1` and `myproc_2`) that have the same derivative (called `myderiv`), you must instantiate both processors as follows:

```plaintext
processor myproc_1
{
    derivative = myderiv;
}

processor myproc_2
{
```
derivative = myderiv;
}

If the derivative definition has parameters you must specify the arguments that correspond with the parameters. For example myderiv1 expects two parameters which are used in the derivative definition:

processor myproc
{
    derivative = myderiv1 (2,4);
}

14.6.3. Defining External Memory and Buses

It is common to define external memory (off-chip) and external buses at the global scope (outside any enclosing definition). Internal memory (on-chip memory) is usually defined in the scope of a derivative definition.

With the keyword memory you define physical memory that is present on the target board. The memory name is used to identify the memory and does not conflict with other identifiers. If you define memory parts in the LSL file, only the memory defined in these parts is used for placing sections.

If no external memory is defined in the LSL file and if the linker option to allocate memory on demand is set then the linker will assume that all virtual addresses are mapped on physical memory. You can override this behavior by specifying one or more memory definitions.

memory mem_name
{
    type = rom;
    mau = 8;
    size = 64k;
    speed = 2;
    map ( map_description );
}

For a description of the keywords, see Section 14.5.3, Defining Internal Memory and Buses.

With the keyword bus you define a bus (the combination of data and corresponding address bus). The bus name is used to identify a bus and does not conflict with other identifiers. Bus descriptions at the global scope (outside any definition) define external buses. These are buses that are present on the target board.

bus bus_name
{
    mau = 8;
    width = 8;
    map ( map_description );
}

For a description of the keywords, see Section 14.4.2, Defining Internal Buses.

You can connect off-chip memory to any derivative: you need to map the off-chip memory to a bus and map that bus on the internal bus of the derivative you want to connect it to.
14.7. Semantics of the Section Setup Definition

Keywords in the section setup definition

```c
section_setup
  stack
    min_size
    grows low_to_high high_to_low
    align
    fixed
    id
  heap
    min_size
    grows low_to_high high_to_low
    align
    fixed
    id
  vector_table
    vector_size
    size
    id_symbol_prefix
    run_addr
    template
    template_symbol
    vector_prefix
    fill
    no_inline
    copy
    vector
      id
      fill loop
  reserved
```

14.7.1. Setting up a Section

With the keyword `section_setup` you can define stacks, heaps, vector tables, and/or reserved address ranges outside their address space definition.

```c
section_setup ::my_space
{
  vector_table statements
  reserved address range
  stack definition
  heap definition
}
```

See the subsections Stacks and heaps, Vector table and Reserved address ranges in Section 14.4.3, Defining Address Spaces for details on the keywords stack, heap, vector_table and reserved.
14.8. Semantics of the Section Layout Definition

Keywords in the section layout definition

section_layout
  direction     low_to_high  high_to_low
group
  align
  attributes    + -  r w x b i s
  copy
  nocopy
  fill
  ordered
  contiguous
  clustered
  overlay
  allow_cross_references
  load_addr
    mem
  run_addr
    mem
  page
  page_size
  priority
select
stack
  size
heap
  size
reserved
  size
  attributes    r w x
  fill
  alloc_allowed absolute
copytable
section
  size
  blocksize
  attributes    r w x
  fill
  overflow

if
else

14.8.1. Defining a Section Layout

With the keyword section_layout you define a section layout for exactly one address space. In the section layout you can specify how input sections are placed in the address space, relative to each other, and what the absolute run and load addresses of each section will be.
You can define one or more section definitions. Each section definition arranges the sections in one address space. You can precede the address space name with a processor name and/or core name, separated by colons. You can omit the processor name and/or the core name if only one processor is defined and/or only one core is present in the processor. A reference to a space in the only core of the only processor in the system would look like ":my_space". A reference to a space of the only core on a specific processor in the system could be "my_chip::my_space". The next example shows a section definition for sections in the my_space address space of the processor called my_chip:

```plaintext
section_layout my_chip::my_space ( locate_direction )
{
    section statements
}
```

With the optional keyword `direction` you specify whether the linker starts locating sections from `low_to_high` (default) or from `high_to_low`. In the second case the linker starts locating sections at the highest addresses in the address space but preserves the order of sections when necessary (one processor and core in this example).

```plaintext
section_layout ::my_space ( direction = high_to_low )
{
    section statements
}
```

If you do not explicitly tell the linker how to locate a section, the linker decides on the basis of the section attributes in the object file and the information in the architecture definition and memory parts where to locate the section.

### 14.8.2. Creating and Locating Groups of Sections

Sections are located per group. A group can contain one or more (sets of) input sections as well as other groups. Per group you can assign a mutual order to the sets of sections and locate them into a specific memory part.

```plaintext
group ( group_specifications )
{
    section_statements
}
```

With the `section_statements` you generally select sets of sections to form the group. This is described in subsection Selecting sections for a group.

Instead of selecting sections, you can also modify special sections like stack and heap or create a reserved section. This is described in Section 14.8.3, Creating or Modifying Special Sections.

With the `group_specifications` you actually locate the sections in the group. This is described in subsection Locating a group.
Selecting sections for a group

With the keyword `select` you can select one or more sections for the group. You can select a section by name or by attributes. If you select a section by name, you can use a wildcard pattern:

* matches with all section names
? matches with a single character in the section name
\ takes the next character literally
[abc] matches with a single 'a', 'b' or 'c' character
[a-z] matches with any single character in the range 'a' to 'z'

group ( ... )
{
    select "mysection";
    select "*";
}

The first `select` statement selects the section with the name "mysection". The second `select` statement selects all sections that were not selected yet.

A section is selected by the first `select` statement that matches, in the union of all section layouts for the address space. Global section layouts are processed in the order in which they appear in the LSL file. Internal core architecture section layouts always take precedence over global section layouts.

- The `attributes` field selects all sections that carry (or do not carry) the given attribute. With `+attribute` you select sections that have the specified attribute set. With `-attribute` you select sections that do not have the specified attribute set. You can specify one or more of the following attributes:
  - r readable sections
  - w writable sections
  - x executable sections
  - i initialized sections
  - b sections that should be cleared at program startup
  - s scratch sections (not cleared and not initialized)

To select all read-only sections:

group ( ... )
{
    select (attributes = +r-w);
}

Keep in mind that all section selections are restricted to the address space of the section layout in which this group definition occurs.
• With the `ref_tree` field you can select a group of related sections. The relation between sections is often expressed by means of references. By selecting just the 'root' of tree, the complete tree is selected. This is for example useful to locate a group of related sections in special memory (e.g. fast memory). The (referenced) sections must meet the following conditions in order to be selected:

1. The sections are within the section layout's address space
2. The sections match the specified attributes
3. The sections have no absolute restriction (as is the case for all wildcard selections)

For example, to select the code sections referenced from `foo1`:

```vxml
group refgrp (ordered, contiguous, run_addr=mem:ext_c)
{
    select ref_tree "foo1" (attributes=+x);
}
```

If section `foo1` references `foo2` and `foo2` references `foo3`, then all these sections are selected by the selection shown above.

### Locating a group

```vxml
group group_name ( group_specifications )
{
    section_statements
}
```

With the `group_specifications` you actually define how the linker must locate the group. You can roughly define three things: 1) assign properties to the group like alignment and read/write attributes, 2) define the mutual order in the address space for sections in the group and 3) restrict the possible addresses for the sections in a group.

The linker creates labels that allow you to refer to the begin and end address of a group from within the application software. Labels `lc_gb_group_name` and `lc_ge_group_name` mark the begin and end of the group respectively, where the begin is the lowest address used within this group and the end is the highest address used. Notice that a group not necessarily occupies all memory between begin and end address. The given label refers to where the section is located at run-time (versus load-time).

1. Assign properties to the group like alignment and read/write attributes.

    These properties are assigned to all sections in the group (and subgroups) and override the attributes of the input sections.

    • The `align` field tells the linker to align all sections in the group and the group as a whole according to the align value. By default the linker uses the largest alignment constraint of either the input sections or the alignment of the address space.

    • The `attributes` field tells the linker to assign one or more attributes to all sections in the group. This overrules the default attributes. By default the linker uses the attributes of the input sections. You can set the `r`, `w`, or `rw` attributes and you can switch between the `b` and `s` attributes.
• The **copy** field tells the linker to locate a read-only section in RAM and generate a ROM copy and a copy action in the copy table. This property makes the sections in the group writable which causes the linker to generate ROM copies for the sections.

• The effect of the **nocopy** field is the opposite of the **copy** field. It prevents the linker from generating ROM copies of the selected sections.

2. Define the mutual order of the sections in the group.

   By default, a group is *unrestricted* which means that the linker has total freedom to place the sections of the group in the address space.

   • The **ordered** keyword tells the linker to locate the sections in the same order in the address space as they appear in the group (but not necessarily adjacent).

   Suppose you have an ordered group that contains the sections 'A', 'B' and 'C'. By default the linker places the sections in the address space like 'A' - 'B' - 'C', where section 'A' gets the lowest possible address. With `direction=high_to_low` in the section_layout space properties, the linker places the sections in the address space like 'C' - 'B' - 'A', where section 'A' gets the highest possible address.

   • The **contiguous** keyword tells the linker to locate the sections in the group in a single address range. Within a contiguous group the input sections are located in arbitrary order, however the group occupies one contiguous range of memory. Due to alignment of sections there can be 'alignment gaps' between the sections.

   When you define a group that is both **ordered** and **contiguous**, this is called a *sequential* group. In a sequential group the linker places sections in the same order in the address space as they appear in the group and it occupies a contiguous range of memory.

   • The **clustered** keyword tells the linker to locate the sections in the group in a number of *contiguous* blocks. It tries to keep the number of these blocks to a minimum. If enough memory is available, the group will be located as if it was specified as **contiguous**. Otherwise, it gets split into two or more blocks.

   If a contiguous or clustered group contains *alignment gaps*, the linker can locate sections that are not part of the group in these gaps. To prevent this, you can use the **fill** keyword. If the group is located in RAM, the gaps are treated as reserved (scratch) space. If the group is located in ROM, the alignment gaps are filled with zeros by default. You can however change the fill pattern by specifying a bit pattern. The result of the expression, or list of expressions, is used as values to write to memory, each in MAU.

• The **overlay** keyword tells the linker to overlay the sections in the group. The linker places all sections in the address space using a contiguous range of addresses. (Thus an overlay group is automatically also a contiguous group.) To overlay the sections, all sections in the overlay group share the same run-time address.

   For each input section within the overlay the linker automatically defines two symbols. The symbol `_lc_cb_section_name` is defined as the load-time start address of the section. The symbol `_lc_ce_section_name` is defined as the load-time end address of the section. C (or assembly) code may be used to copy the overlaid sections.
If sections in the overlay group contain references between groups, the linker reports an error. The keyword `allow_cross_references` tells the linker to accept cross-references. Normally, it does not make sense to have references between sections that are overlaid.

```c
group ovl (overlay)
{
  group a
  {
    select "my_ovl_p1";
    select "my_ovl_p2";
  }
  group b
  {
    select "my_ovl_q1";
  }
}
```

It may be possible that one of the sections in the overlay group already has been defined in another group where it received a load-time address. In this case the linker does not overrule this load-time address and excludes the section from the overlay group.

3. Restrict the possible addresses for the sections in a group.

The load-time address specifies where the group's elements are loaded in memory at download time. The run-time address specifies where sections are located at run-time, that is when the program is executing. If you do not explicitly restrict the address in the LSL file, the linker assigns addresses to the sections based on the restrictions relative to other sections in the LSL file and section alignments. The program is responsible for copying overlay sections at appropriate moment from its load-time location to its run-time location (this is typically done by the startup code).

- The `run_addr` keyword defines the run-time address. If the run-time location of a group is set explicitly, the given order between groups specify whether the run-time address propagates to the parent group or not. The location of the sections a group can be restricted either to a single absolute address, or to a number of address ranges. With an expression you can specify that the group should be located at the absolute address specified by the expression:

```c
group (run_addr = 0xa00f0000)
```

You can use the `[offset]` variant to locate the group at the given absolute offset in memory:

```c
group (run_addr = mem:A[0x1000])
```

A range can be an absolute space address range, written as `[expr .. expr]`, a complete memory device, written as `mem:mem_name`, or a memory address range, `mem:mem_name[expr .. expr]`

```c
group (run_addr = mem:my_dram)
```

You can use the `|` to specify an address range of more than one physical memory device:

```c
group (run_addr = mem:A | mem:B)
```
The `load_addr` keyword changes the meaning of the section selection in the group: the linker selects the load-time ROM copy of the named section(s) instead of the regular sections. Just like `run_addr` you can specify an absolute address or an address range.

```plaintext

group (contiguous, load_addr)
{
    select "mydata";  // select ROM copy of mydata:
    // "[mydata]"
}
```

The load-time and run-time addresses of a group cannot be set at the same time. If the load-time property is set for a group, the group (only) restricts the positioning at load-time of the group's sections. It is not possible to set the address of a group that has a not-unrestricted parent group.

The properties of the load-time and run-time start address are:

- At run-time, before using an element in an overlay group, the application copies the sections from their load location to their run-time location, but only if these two addresses are different. For non-overlay sections this happens at program start-up.

- The start addresses cannot be set to absolute values for unrestricted groups.

- For non-overlay groups that do not have an overlay parent, the load-time start address equals the run-time start address.

- For any group, if the run-time start address is not set, the linker selects an appropriate address.

For overlays, the linker reserves memory at the run-time start address as large as the largest element in the overlay group.

- The `page` keyword tells the linker to place the group in one page. Instead of specifying a run-time address, you can specify a page and optional a page number. Page numbers start from zero. If you omit the page number, the linker chooses a page.

```plaintext

group (priority=2)
{
    select "importantcode1";
}
```

The `page` keyword refers to pages in the address space as defined in the architecture definition.

- With the `page_size` keyword you can override the page alignment and size set on the address space. When you set the page size to zero, the linker removes simple (auto generated) page restrictions from the selected sections. See also the `page_size` keyword in Section 14.4.3, Defining Address Spaces.

- With the `priority` keyword you can change the order in which sections are located. This is useful when some sections are considered important for good performance of the application and a small amount of fast memory is available. The value is a number for which the default is 1, so higher priorities start at 2. Sections with a higher priority are located before sections with a lower priority, unless their relative locate priority is already determined by other restrictions like `run_addr` and `page`.
14.8.3. Creating or Modifying Special Sections

Instead of selecting sections, you can also create a reserved section or an output section or modify special sections like a stack or a heap. Because you cannot define these sections in the input files, you must use the linker to create them.

**Stack**

- The keyword `stack` tells the linker to reserve memory for the stack. The name for the stack section refers to the stack as defined in the architecture definition. If no name was specified in the architecture definition, the default name is `stack`.

With the keyword `size` you can specify the size for the stack. If the size is not specified, the linker uses the size given by the `min_size` argument as defined for the stack in the architecture definition. Normally the linker automatically tries to maximize the size, unless you specified the keyword `fixed`.

```cpp
group ( ... )
{
    stack "mystack" ( size = 2k );
}
```

The linker creates two labels to mark the begin and end of the stack, `_lc_ub_stack_name` for the begin of the stack and `_lc_ue_stack_name` for the end of the stack. The linker allocates space for the stack when there is a reference to either of the labels.

See also the `stack` keyword in Section 14.4.3, Defining Address Spaces.

**Heap**

- The keyword `heap` tells the linker to reserve a dynamic memory range for the `malloc()` function. Each heap section has a name. With the keyword `size` you can change the size for the heap. If the `size` is not specified, the linker uses the size given by the `min_size` argument as defined for the heap in the architecture definition. Normally the linker automatically tries to maximize the size, unless you specified the keyword `fixed`.

```cpp
group ( ... )
{
    heap "myheap" ( size = 2k );
}
```

The linker creates two labels to mark the begin and end of the heap, `_lc_ub_heap_name` for the begin of the heap and `_lc_ue_heap_name` for the end of the heap. The linker allocates space for the heap when a reference to either of the section labels exists in one of the input object files.
Reserved section

- The keyword **reserved** tells the linker to create an area or section of a given size. The linker will not locate any other sections in the memory occupied by a reserved section, with some exceptions. Each reserved section has a name. With the keyword **size** you can specify a size for a given reserved area or section.

```plaintext
group ( ... )
{
    reserved "myreserved" ( size = 2k );
}
```

The optional **fill** field contains a bit pattern that the linker writes to all memory addresses that remain unoccupied during the locate process. The result of the expression, or list of expressions, is used as values to write to memory, each in MAU. The first MAU of the fill pattern is always the first MAU in the section.

By default, no sections can overlap with a reserved section. With **alloc_allowed=absolute** sections that are located at an absolute address due to an absolute group restriction can overlap a reserved section.

With the **attributes** field you can set the access type of the reserved section. The linker locates the reserved section in its space with the restrictions that follow from the used attributes, *r*, *w* or *x* or a valid combination of them. The allowed attributes are shown in the following table. A value between < and > in the table means this value is set automatically by the linker.

<table>
<thead>
<tr>
<th>Properties set in LSL</th>
<th>Resulting section properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributes</td>
<td>filled</td>
</tr>
<tr>
<td>x</td>
<td>yes</td>
</tr>
<tr>
<td>r</td>
<td>yes</td>
</tr>
<tr>
<td>r</td>
<td>no</td>
</tr>
<tr>
<td>rx</td>
<td>yes</td>
</tr>
<tr>
<td>rw</td>
<td>yes</td>
</tr>
<tr>
<td>rw</td>
<td>no</td>
</tr>
<tr>
<td>rwx</td>
<td>yes</td>
</tr>
</tbody>
</table>

```plaintext
group ( ... )
{
    reserved "myreserved" ( size = 2k,
    attributes = rw, fill = 0xaa );
}
```

If you do not specify any attributes, the linker will reserve the given number of maus, no matter what type of memory lies beneath. If you do not specify a fill pattern, no section is generated.

The linker creates two labels to mark the begin and end of the section, **_lc_ub_name** for the begin of the section and **_lc_ue_name** for the end of the reserved section.
Output sections

- The keyword section tells the linker to accumulate sections obtained from object files ("input sections") into an output section of a fixed size in the locate phase. You can select the input sections with select statements. You can use groups inside output sections, but you can only set the align, attributes and load_addr attributes.

  The fill field contains a bit pattern that the linker writes to all unused space in the output section. When all input sections have an image (code/data) you must specify a fill pattern. If you do not specify a fill pattern, all input sections must be scratch sections. The fill pattern is aligned at the start of the output section.

  As with a reserved section you can use the attributes field to set the access type of the output section.

```plaintext
group { ... }
{
  section "myoutput" ( size = 4k, attributes = rw,
    fill = 0xaa )
  {
    select "myinput1";
    select "myinput2";
  }
}
```

The available room for input sections is determined by the size, blocksize and overflow fields. With the keyword size you specify the fixed size of the output section. Input sections are placed from output section start towards higher addresses (offsets). When the end of the output section is reached and one or more input sections are not yet placed, an error is emitted. If however, the overflow field is set to another output section, remaining sections are located as if they were selected for the overflow output section.

```plaintext
group { ... }
{
  section "tsk1_data" (size=4k, attributes=rw, fill=0,
    overflow = "overflow_data")
  {
    select ".data.tsk1.*"
  }
  section "tsk2_data" (size=4k, attributes=rw, fill=0,
    overflow = "overflow_data")
  {
    select ".data.tsk2.*"
  }
  section "overflow_data" (size=4k, attributes=rx,
    fill=0)
  {
  }
}
```
With the keyword `blocksize`, the size of the output section will adapt to the size of its content. For example:

```plaintext
group flash_area (run_addr = 0x10000)
{
    section "flash_code" (blocksize=4k, attributes=rx,
        fill=0)
    {
        select "*.flash";
    }
}
```

If the content of the section is 1 mau, the size will be 4 kB, if the content is 11 kB, the section will be 12 kB, etc. If you use `size` in combination with `blocksize`, the `size` value is used as default (minimal) size for this section. If it is omitted, the default size will be of `blocksize`. It is not allowed to omit both `size` and `blocksize` from the section definition.

The linker creates two labels to mark the begin and end of the section, `_lc_ub_name` for the begin of the section and `_lc_ue_name` for the end of the output section.

**Copy table**

- The keyword `copytable` tells the linker to select a section that is used as `copy table`. The content of the copy table is created by the linker. It contains the start address and length of all sections that should be initialized by the startup code.

The linker creates two labels to mark the begin and end of the section, `_lc_ub_table` for the begin of the section and `_lc_ue_table` for the end of the copy table. The linker generates a copy table when a reference to either of the section labels exists in one of the input object files.

**14.8.4. Creating Symbols**

You can tell the linker to create symbols before locating by putting assignments in the section layout definition. Symbol names are represented by double-quoted strings. Any string is allowed, but object files may not support all characters for symbol names. You can use two different assignment operators. With the simple assignment operator `='`, the symbol is created unconditionally. With the `:=` operator, the symbol is only created if it already exists as an undefined reference in an object file.

The expression that represents the value to assign to the symbol may contain references to other symbols. If such a referred symbol is a special section symbol, creation of the symbol in the left hand side of the assignment will cause creation of the special section.

```plaintext
section_layout
{
    "_lc_cp" := "_lc_ub_table";
    // when the symbol _lc_cp occurs as an undefined reference
    // in an object file, the linker generates a copy table
}
```
14.8.5. Conditional Group Statements

Within a group, you can conditionally select sections or create special sections.

- With the if keyword you can specify a condition. The succeeding section statement is executed if the condition evaluates to TRUE (1).

- The optional else keyword is followed by a section statement which is executed in case the if-condition evaluates to FALSE (0).

```plaintext
group ( ... )
{
    if ( exists( "mysection" ) )
        select "mysection";
    else
        reserved "myreserved" ( size=2k );
}
```
Chapter 15. Debug Target Configuration Files

DTC files (Debug Target Configuration files) define all possible configurations for a debug target. A debug target can be target hardware such as an evaluation board or a simulator. The DTC files are used by Eclipse to configure the project and the debugger. The information is used by the Target Board Configuration wizard and the debug configuration. DTC files are located in the etc directory of the installed product and use .dtc as filename suffix.

Based on the DTC files, the Target Board Configuration wizard adjust the project's LSL file and creates a debug launch configuration.

15.1. Custom Board Support

When you need support for a custom board and the board requires a different configuration than those that are in the product, it is necessary to create a dedicated DTC file.

To add a custom board

1. From the etc directory of the product, make a copy of a .dtc file and put it in your project directory (in the current workspace).

   In Eclipse, the DTC file should now be visible as part of your project.

2. Edit the file and give it a name that reflects the custom board.

The Target Board Configuration wizard in Eclipse adds DTC files that are present in your current project to the list of available target boards.

Syntax of a DTC file

DTC files are XML files. Use a delivered .dtc file as a starting point for creating a custom board specification.

Basically a DTC file consists of the definition of the debug target (debugTarget element) which embodies one or more communication methods (communicationMethod element). Within each communication method, you can define multiple configurations (configuration element). The Target Board Configuration wizard in Eclipse reflects the structure of the DTC file. The elements that determine the settings that are applied by the wizard, can be found at any level in the DTC file. The wizard will apply all elements that are within the path to the selected configuration. This is best explained by an example of a DTC file with the following basic layout:

```
debugTarget: Infineon TriBoard TC1165
  lsl
  communicationMethod: DAS over MiniWigglerII
    lsl
    configuration: Single Chip
  lsl
  communicationMethod: DAS over USB-Wiggler
    lsl
    configuration: Single Chip
```
In this example there is an LSL element at every level. If, in the Target Board Configuration wizard in Eclipse, you set the debug target configuration to "DAS over MiniWigglerII"->"Single Chip", the wizard puts the following LSL parts into the project's LSL file in this order:

- the lsl part under the debugTarget element
- the lsl part under the communicationMethod "DAS over MiniWigglerII" element
- the lsl part under the configuration "Single Chip" in the communicationMethod "DAS over MiniWigglerII" element
- the lsl part in the debugTarget element at the end of the DTC file

The same applies to all other elements that determine the underlying settings.

DTC macros in LSL

To protect the Target Board Configuration wizard from changing the LSL file, you can protect the LSL file by adding the macro __DTC_IGNORE. This can be useful for projects that need the same LSL file, but still need to run on different target boards.

#define __DTC_IGNORE

The following DTC macros can be present in the LSL file:

<table>
<thead>
<tr>
<th>LSL Define</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__DTC_IGNORE</td>
<td>If defined, protects the LSL file against changes by the Target Board Configuration wizard.</td>
</tr>
<tr>
<td>__DTC_START</td>
<td>The LSL part that is between these macros can be replaced by LSL text from the DTC file. If the macros are not present in the LSL file, the Target Board Configuration wizard will add them.</td>
</tr>
<tr>
<td>__DTC_END</td>
<td></td>
</tr>
</tbody>
</table>

15.2. Description of DTC Elements and Attributes

The following table contains a description of the DTC elements and attributes. For each element a list of allowed elements is listed and the available attributes are described.

<table>
<thead>
<tr>
<th>Element / Attribute</th>
<th>Description</th>
<th>Allowed Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>debugTarget</td>
<td>The debug target.</td>
<td>flashChips, lsl, communicationMethod, def, processor, resource, initialize</td>
</tr>
<tr>
<td>name</td>
<td>The name of the configuration.</td>
<td></td>
</tr>
<tr>
<td>manufacturer</td>
<td>The manufacturer of the debug target.</td>
<td></td>
</tr>
<tr>
<td>processor</td>
<td>Defines a processor that can be present on the debug target. Multiple processor definitions are allowed. The user should select the actual processor on the debug target.</td>
<td>-</td>
</tr>
<tr>
<td>Element / Attribute</td>
<td>Description</td>
<td>Allowed Elements</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------</td>
<td>------------------</td>
</tr>
<tr>
<td>name</td>
<td>A descriptive name of the processor derivative.</td>
<td>ref, resource, initialize, configuration, lsl, processor</td>
</tr>
<tr>
<td>cpu</td>
<td>Defines the CPU name, as for example supplied with the option --cpu of the C compiler.</td>
<td></td>
</tr>
<tr>
<td>communicationMethod</td>
<td>Defines a communication method. A communication method is the channel that is used to communicate with the target.</td>
<td>ref, resource, initialize, configuration, lsl, processor</td>
</tr>
<tr>
<td>name</td>
<td>A descriptive name of the communication method.</td>
<td></td>
</tr>
<tr>
<td>debugInstrument</td>
<td>The debug instrument DLL/Shared library file to be used for this communication method. Do not supply a path or a filename suffix.</td>
<td></td>
</tr>
<tr>
<td>gdiMethod</td>
<td>This is the method used for communication. Allowed values: rs232, tcpip, can, none</td>
<td></td>
</tr>
<tr>
<td>def</td>
<td>Define a set of elements as a macro. The macro can be expanded using the ref element.</td>
<td>Isl, resource, initialize, ref, configuration, flashMonitor</td>
</tr>
<tr>
<td>id</td>
<td>The macro name.</td>
<td></td>
</tr>
<tr>
<td>resource</td>
<td>Defines a resource definition that can be used by Eclipse, the debugger or by the debug instrument.</td>
<td></td>
</tr>
<tr>
<td>id</td>
<td>The identifier name used by the debugger or debug instrument to retrieve the value.</td>
<td></td>
</tr>
<tr>
<td>value</td>
<td>The value assigned to the resource.</td>
<td></td>
</tr>
<tr>
<td>ref</td>
<td>Reference to a macro defined with a def element. The elements contained in the def element with the same name will be expanded at the location of the ref. Multiple refs to the same def are allowed.</td>
<td></td>
</tr>
<tr>
<td>id</td>
<td>The name of the referenced macro.</td>
<td></td>
</tr>
<tr>
<td>configuration</td>
<td>Defines a configuration.</td>
<td>ref, initialize, resource, Isl, flashMonitor, processor</td>
</tr>
<tr>
<td>name</td>
<td>The descriptive name of the configuration.</td>
<td></td>
</tr>
<tr>
<td>initialize</td>
<td>This element defines an initialization expression. Each initialize element contains a resourceId attribute. If the DI requests this resource the debugger will compose a string from all initialize elements with the same resourceId. This DI can use this string to initialize registers by passing it to the debugger as an expression to be evaluated.</td>
<td></td>
</tr>
<tr>
<td>resourceId</td>
<td>The name of the resource to be used.</td>
<td></td>
</tr>
<tr>
<td>Element / Attribute</td>
<td>Description</td>
<td>Allowed Elements</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>name</td>
<td>The name of the register to be initialized.</td>
<td></td>
</tr>
<tr>
<td>value</td>
<td>When the cstart attribute is false, this is the value to be used, otherwise, it is the default value when using this configuration. It will be used by the startup code editor to set the default register values.</td>
<td></td>
</tr>
<tr>
<td>cstart</td>
<td>A boolean value. If true the debugger should ask the C startup code editor for the value, otherwise the contents of the value attribute is used. The default value is true.</td>
<td></td>
</tr>
<tr>
<td>flashMonitor</td>
<td>This element specifies the flash programming monitor to be used for this configuration.</td>
<td>-</td>
</tr>
<tr>
<td>monitor</td>
<td>Filename of the monitor, usually an Intel Hex or S-Record file.</td>
<td></td>
</tr>
<tr>
<td>workspaceAddress</td>
<td>The address of the workspace of the flash programming monitor.</td>
<td></td>
</tr>
<tr>
<td>flashSectorBufferSize</td>
<td>Specifies the buffer size for buffering a flash sector.</td>
<td></td>
</tr>
<tr>
<td>chip</td>
<td>This element defines a flash chip. It must be used by the flash properties page to add it on request to the list of flash chips.</td>
<td>debugTarget</td>
</tr>
<tr>
<td>vendor</td>
<td>The vendor of this flash chip.</td>
<td></td>
</tr>
<tr>
<td>chip</td>
<td>The name of the chip.</td>
<td></td>
</tr>
<tr>
<td>width</td>
<td>The width of the chip in bits.</td>
<td></td>
</tr>
<tr>
<td>chips</td>
<td>The number of chips present on the board.</td>
<td></td>
</tr>
<tr>
<td>baseAddress</td>
<td>The base address of the chip.</td>
<td></td>
</tr>
<tr>
<td>chipSize</td>
<td>The size of the chip in bytes.</td>
<td></td>
</tr>
<tr>
<td>flashChips</td>
<td>Specify a list of flash chips that can be available on this debug target.</td>
<td>chip</td>
</tr>
<tr>
<td>lsl</td>
<td>Defines LSL pieces belonging to the configuration part. The LSL text must be defined between the start and end tag of this element. All LSL texts of the active selection will be placed in the project’s LSL file.</td>
<td>-</td>
</tr>
</tbody>
</table>

### 15.3. Special Resource Identifiers

The following resource IDs are available in the TASKING VX-toolset for TriCore:
DAS debug instrument (DI): gdi2das

<table>
<thead>
<tr>
<th>Resource Name</th>
<th>Description</th>
<th>Possible Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>AccessPort</td>
<td>The port used to connect to the wiggler.</td>
<td>JTAG1, USB0</td>
</tr>
<tr>
<td>DASserver</td>
<td>The DAS Server used for communication.</td>
<td>JTAG JDRV LPT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>JTAG over USB Box</td>
</tr>
<tr>
<td></td>
<td></td>
<td>JTAG over USB Chip</td>
</tr>
<tr>
<td>DasTimeOut</td>
<td>The timeout value for communication with the DAS server in milliseconds. The default is 0x4000.</td>
<td></td>
</tr>
<tr>
<td>RegisterFile</td>
<td>The core register file that is used by the debug instrument. This is usually &quot;regbase_f7e1.dat&quot; or &quot;regbase_fff.dat&quot;, depending on the register base address.</td>
<td></td>
</tr>
<tr>
<td>TerminateServer</td>
<td>Terminate the DAS server when the session is closed.</td>
<td>0, 1</td>
</tr>
</tbody>
</table>

15.4. Initialize Elements

The initialize elements are used to initialize SFRs at startup. This is also done using a resource of the debug instrument. The following resource Ids exist for the DAS debug instrument (gdi2das):

<table>
<thead>
<tr>
<th>Resource Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>einit</td>
<td>Initialize an SFR that is protected with the ENDINIT flag.</td>
</tr>
<tr>
<td>init</td>
<td>Initialize an SFR that is not protected with the ENDINIT flag.</td>
</tr>
</tbody>
</table>
Chapter 16. CPU Problem Bypasses and Checks

Infineon Technologies regularly publishes microcontroller errata sheets for reporting both functional problems and deviations from the electrical and timing specifications.

For some of these functional problems in the microcontroller itself, the TASKING VX-toolset for TriCore provides workarounds. In fact these are software workarounds for hardware problems.

Support to deal with CPU functional problem is provided in three areas:

• Whenever possible and relevant, compiler bypasses will modify the code in order to avoid the identified erroneous code sequences;

• The assembler gives warnings for suspicious or erroneous code sequences;

• Ready-built, 'protected' standard C libraries with bypasses for all identified TriCore CPU functional problems are included in the toolset.

This chapter lists a summary of functional problems which can be bypassed by the TASKING VX-toolset for TriCore. Please refer to the Infineon errata sheets for the CPU step you are using, to verify if you need to use one of these bypasses.

To set a CPU bypass or check

1. From the Project menu, select Properties

   The Properties dialog appears.

2. In the left pane, expand C/C++ Build and select Processor.

   In the right pane the Processor page appears.

3. From the Processor Selection list, select a processor.

   The CPU Problem Bypasses and Checks box shows the available workarounds/checks available for the selected processor.

4. (Optional) Select Show all CPU problem bypasses and checks.

5. Click Select All or select one or more individual options.

Overview of the CPU problem bypasses and checks

The following table contains an overview of the silicon bugs you can provide to the C compiler option --silicon-bug and the assembler option --silicon-bug. WA means a workaround by the compiler, assembler and/or linker, CK means a check by the assembler.
<table>
<thead>
<tr>
<th>CPU Problem</th>
<th>Description</th>
<th>Compiler</th>
<th>Assembler</th>
<th>Linker</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU TC.013</td>
<td>Unreliable context load/store operation following an address register load instruction</td>
<td>WA</td>
<td>CK</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TC1100, TC1115, TC1130, TC111B, TC1765, TC1766, TC1775, TC1792, TC1796, TC1910, TC1912, TC1920</td>
</tr>
<tr>
<td>CPU TC.018</td>
<td>LOOP over arithmetical instruction causes a jump to an undefined address</td>
<td>WA</td>
<td>CK</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TC1765, TC1775</td>
</tr>
<tr>
<td>CPU TC.021</td>
<td>Incorrect forwarding from branch and link address register update</td>
<td>WA</td>
<td>CK</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TC1765, TC1775</td>
</tr>
<tr>
<td>CPU TC.023</td>
<td>CALLI with target address in register A11 not functional</td>
<td>WA</td>
<td>CK</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TC1765, TC1775</td>
</tr>
<tr>
<td>CPU TC.024</td>
<td>Incorrect return address in A11 when performing nested calls</td>
<td>WA</td>
<td>CK</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TC1765, TC1775</td>
</tr>
<tr>
<td>CPU TC.030</td>
<td>Loop bug following the DVSTEP type instruction</td>
<td>WA</td>
<td>CK</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TC1765, TC1775</td>
</tr>
<tr>
<td>CPU TC.031</td>
<td>Wrong return address after any divide instruction</td>
<td>WA</td>
<td>CK</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TC1765, TC1775</td>
</tr>
<tr>
<td>CPU TC.033</td>
<td>Circular addressing mode limitations</td>
<td>WA</td>
<td>WA</td>
<td>WA</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TC1765, TC1775</td>
</tr>
<tr>
<td>CPU TC.034</td>
<td>DSYNC causes corruption of up to two following instructions</td>
<td>WA</td>
<td>CK</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TC1765, TC1775</td>
</tr>
<tr>
<td>CPU TC.043</td>
<td>A load/store instruction to the last 16 bytes of a segment can lead to undefined behavior</td>
<td>WA</td>
<td></td>
<td>WA</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TC1765, TC1775</td>
</tr>
<tr>
<td>CPU TC.048</td>
<td>CPU fetches program from unexpected address</td>
<td>WA</td>
<td>CK</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TC1100, TC1115, TC1130, TC111B, TC1765, TC1766, TC1775, TC1792, TC1796, TC1910, TC1912, TC1920</td>
</tr>
<tr>
<td>CPU TC.050</td>
<td>A load instruction following a multicycle integer instruction can get lost</td>
<td>WA</td>
<td>CK</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TC1765, TC1775</td>
</tr>
<tr>
<td>CPU TC.051</td>
<td>Reduced context save area</td>
<td>WA</td>
<td>WA</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TC1910, TC1912, TC1920</td>
</tr>
<tr>
<td>CPU TC.052</td>
<td>Alignment Restrictions for Accesses using PTE-Based Translation</td>
<td>WA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TC1100, TC1115, TC1130, TC111B</td>
</tr>
<tr>
<td>CPU TC.060</td>
<td>LD.[A,DA] followed by a dependent LD.[DA,D,W] can produce unreliable results</td>
<td>WA</td>
<td>CK</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>TC1100, TC1115, TC1130, TC111B, TC1765, TC1766, TC1775, TC1792, TC1796, TC1910, TC1912, TC1920</td>
</tr>
<tr>
<td>CPU Problem</td>
<td>Description</td>
<td>Compiler</td>
<td>Assembler</td>
<td>Linker</td>
<td>CPU</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
<td>----------</td>
<td>-----------</td>
<td>--------</td>
<td>-----</td>
</tr>
<tr>
<td>CPU TC.065</td>
<td>Error when unconditional loop targets unconditional jump</td>
<td>WA</td>
<td>CK</td>
<td></td>
<td>TC1100, TC1115, TC1130, TC11IB, TC1765, TC1766, TC1775, TC1792, TC1796</td>
</tr>
<tr>
<td>CPU TC.068</td>
<td>Potential PSW corruption by cancelled DVINIT instructions</td>
<td>WA</td>
<td>CK</td>
<td></td>
<td>TC1100, TC1115, TC1130, TC11IB, TC1765, TC1766, TC1775, TC1792, TC1796</td>
</tr>
<tr>
<td>CPU TC.069</td>
<td>Potential incorrect operation of RSLCX instruction</td>
<td>WA</td>
<td>CK</td>
<td></td>
<td>TC1100, TC1115, TC1130, TC11IB, TC1765, TC1766, TC1775, TC1792, TC1796</td>
</tr>
<tr>
<td>CPU TC.070</td>
<td>Error when conditional jump precedes loop instruction</td>
<td>WA</td>
<td>CK</td>
<td></td>
<td>TC1100, TC1115, TC1130, TC11IB, TC1765, TC1766, TC1775, TC1792, TC1796</td>
</tr>
<tr>
<td>CPU TC.071</td>
<td>Error when Conditional Loop targets Unconditional Loop</td>
<td>WA</td>
<td>CK</td>
<td></td>
<td>TC1100, TC1115, TC1130, TC11IB, TC1765, TC1766, TC1775, TC1792, TC1796</td>
</tr>
<tr>
<td>CPU TC.072</td>
<td>Error when Loop Counter modified prior to Loop instruction</td>
<td>WA</td>
<td>CK</td>
<td></td>
<td>TC1100, TC1115, TC1130, TC11IB, TC1765, TC1766, TC1775, TC1792, TC1796</td>
</tr>
<tr>
<td>CPU TC.074</td>
<td>Interleaved LOOP/LOOPU instructions may cause GRWP Trap</td>
<td>WA</td>
<td></td>
<td></td>
<td>TC1100, TC1115, TC1130, TC11IB, TC1765, TC1766, TC1775, TC1792, TC1796</td>
</tr>
<tr>
<td>CPU TC.081</td>
<td>Error during Load A[10], Call / Exception Sequence</td>
<td>CK</td>
<td></td>
<td></td>
<td>TC1100, TC1115, TC1130, TC11IB, TC1765, TC1766, TC1775, TC1792, TC1796</td>
</tr>
<tr>
<td>CPU TC.082</td>
<td>Data corruption possible when Memory Load follows Context Store</td>
<td>WA</td>
<td>CK</td>
<td></td>
<td>TC1100, TC1115, TC1130, TC11IB, TC1765, TC1766, TC1775, TC1792, TC1796</td>
</tr>
<tr>
<td>CPU TC.083</td>
<td>Interrupt may be taken following DISABLE instruction</td>
<td>WA</td>
<td>CK</td>
<td></td>
<td>TC1100, TC1115, TC1130, TC11IB, TC1765, TC1766, TC1775, TC1792, TC1796</td>
</tr>
<tr>
<td>CPU TC.094</td>
<td>Potential Performance Loss when CSA Instruction follows IP Jump</td>
<td>WA</td>
<td>CK</td>
<td></td>
<td>TC1100, TC1115, TC1130, TC11IB, TC1765, TC1766, TC1775, TC1792, TC1796</td>
</tr>
<tr>
<td>CPU TC.095</td>
<td>Incorrect Forwarding in SAT, Mixed Register Instruction Sequence</td>
<td>WA</td>
<td>CK</td>
<td></td>
<td>TC1100, TC1115, TC1130, TC11IB, TC1765, TC1766, TC1775, TC1792, TC1796</td>
</tr>
<tr>
<td>CPU TC.096</td>
<td>Error when Conditional Loop targets Single Issue Group Loop</td>
<td>WA</td>
<td>CK</td>
<td></td>
<td>TC1100, TC1115, TC1130, TC11IB, TC1765, TC1766, TC1775, TC1792, TC1796</td>
</tr>
<tr>
<td>CPU TC.103</td>
<td>Spurious parity errors can be generated</td>
<td>WA</td>
<td>WA</td>
<td></td>
<td>TC1100, TC1115, TC1130, TC1766</td>
</tr>
</tbody>
</table>
### CPU Problem Description

<table>
<thead>
<tr>
<th>CPU Problem</th>
<th>Description</th>
<th>Compiler</th>
<th>Assembler</th>
<th>Linker</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU TC.104</td>
<td>Double-word Load instructions using Circular Addressing mode can produce unreliable results</td>
<td>WA</td>
<td>CK</td>
<td></td>
<td>TC1100, TC1115, TC1130, TC1766, TC1792, TC1796, TC1792, TC1796</td>
</tr>
<tr>
<td>DMU TC.001</td>
<td>RMW accesses to DMU memory are not locked</td>
<td>WA</td>
<td>CK</td>
<td></td>
<td>TC111B, TC1765, TC1775</td>
</tr>
<tr>
<td>PMI TC.003</td>
<td>MMU-PMU Address Translation</td>
<td>CK</td>
<td></td>
<td></td>
<td>TC111B, TC1920</td>
</tr>
<tr>
<td>PMU TC.004</td>
<td>PMU not addressable in split mode via the LFI, bug will trigger an LMB_ABORT</td>
<td>CK</td>
<td></td>
<td></td>
<td>TC111B, TC1920</td>
</tr>
</tbody>
</table>

TC1161, TC1162, TC1163, TC1164, TC1165, TC1166, TC1762, TC1764 have the same silicon bugs as the TC1766.
**CPU_TC.013**

**Command line option**

```
--silicon-bug=cpu-tc013
```

**Description**

To bypass this CPU functional problem, the C compiler generates a `NOP16` instruction if a 16-bit load/store address register instruction (instructions: `LD16.A` and `ST16.A`) is followed by a lower context load/store instruction (instructions: `LDLCX` and `STLCX`).

The assembler issues a warning if a 16-bit load/store address register instruction (instructions: `LD16.A` and `ST16.A`) is followed by a lower context load/store instruction (instructions: `LDLCX` and `STLCX`).
CPU_TC.018

Command line option

--silicon-bug=cpu-tc018

Description

To bypass this CPU functional problem, the C compiler generates an ISYNC instruction before each LOOP, LOOP16 and LOOPU instruction.

The assembler issues a warning when the preceding instruction of a LOOP, LOOP16 or LOOPU instruction is not an ISYNC instruction.
CPU_TC.021

Command line option

--silicon-bug=cpu-tc021

Description

To bypass this CPU functional problem, the C compiler generates a **NOP** instruction between a (target) label and the instruction following it. This is done when the instruction directly uses an An register for either an effective address calculation or as the target of an indirect branch. Optionally an integer instruction may directly follow the label.

For example, a **NOP** will be inserted after the following labels:

A_label:
   ji a4

B_label:
   add d0, d1 ; integer instruction
   ji a4

The assembler issues a warning for an instruction using an An register for either an effective address calculation or as the target of an indirect branch that is located directly after a (target) label, optionally with an integer instruction in between.
CPU_TC.023

Command line option

--silicon-bug=cpu-tc023

Description

There is no C compiler workaround required for this CPU functional problem, because the compiler does not generate CALLI instructions with a target address in register A11.

The assembler generates an error for instruction CALLI A11.
CPU_TC.024

Command line option

--silicon-bug=cpu-tc024

Description

To bypass this CPU functional problem, the C compiler generates a \texttt{NOP} instruction at the very top of any subroutine that starts with a \texttt{CALL} instruction or that starts with an integer instruction or \texttt{MAC} instruction directly followed by a \texttt{CALL} instruction.

The assembler issues a warning when the first instruction of a subroutine is a \texttt{CALL} instruction or an integer instruction or \texttt{MAC} instruction directly followed by a \texttt{CALL} instruction.
CPU_TC.030

Command line option

--silicon-bug=cpu-tc030

Description

To bypass this CPU functional problem, the C compiler generates an ISYNC instruction prior to the LOOP instruction if the last instruction in the loop is a DVSTEP or a DVSTEP.U.

The assembler issues a warning for loops where the last instruction is a DVSTEP or a DVSTEP.U.
CPU_TC.031

Command line option

--silicon-bug=cpu-tc031

Description

To bypass this CPU functional problem, the C compiler generates an ISYNC instruction prior to the LOOP instruction.

The assembler issues a warning if the LOOP instruction is not preceded by an ISYNC instruction.
CPU_TC.033

Command line option

--silicon-bug=cpu-tc033

Description

To bypass this CPU functional problem, the C compiler aligns circular qualified buffers to a quad-word boundary, and the compiler sizes all stack frames to an integral number of quad-words.

See Section 1.3.1, Circular Buffers: __circ for a description on how to declare a circular buffer.

To bypass this CPU functional problem, the assembler adds a macro to the C startup code to enable initialization of the stack pointers to a quad-word boundary.

The preprocessor define __CPU_TC033__ is used in the tc*.lsl linker script files to set the alignment of the user stack and the interrupt stack to a quad-word alignment.
CPU_TC.034

Command line option

--silicon-bug=cpu-tc034

Description

To bypass this CPU functional problem, the C compiler generates an ISYNC instruction after each DSYNC instruction.

The assembler issues a warning if a DSYNC instruction is not followed by an ISYNC instruction.
CPU_TC.043

Command line option

--silicon-bug=cpu-tc043

Description

To bypass this CPU functional problem, the preprocessor define __CPU_TC043__ is used in the tc*.lsl linker script files. The linker will not use the last 16 bytes of a segment.
CPU TC.048

Command line option

--silicon-bug=cpu-tc048

Description

To bypass this CPU functional problem, the C compiler generates a NOP instruction before a JI or CALLI instruction when this instruction is not directly preceded by either a NOP instruction or an integer instruction or a MAC instruction. The compiler also generates a NOP instruction before a RET and RET16 instruction if there is no or just one instruction before RET, starting from the function entry point.

The assembler issues a warning when a JI or CALLI instruction is not directly preceded by a NOP instruction. The assembler also issues a warning when there is no or just one instruction (not a NOP instruction) between label and RET or RET16.
CPU_TC.050

Command line option

`--silicon-bug=cpu-tc050`

Description

To bypass this CPU functional problem, the C compiler generates a `NOP` instruction between a multi-cycle integer instruction and a load instruction.

The assembler issues a warning if a multi-cycle integer instruction is directly followed by a load instruction.
**CPU_TC.051**

**Command line option**

`--silicon-bug=cpu-tc051`

**Description**

The C compiler has no workaround for this problem.

To bypass this CPU functional problem, the assembler adds a macro to the C startup code.

To bypass this CPU functional problem, the preprocessor define `__CPU_TC051__` is used in the `tc*.lsl` linker script files. The linker will use more than one section for context stores if the required CSA area exceeds the 4 kB. Each section will have a maximum size of 4 kB and will start on an 8 kB boundary.
CPU_TC.052

Command line option

--silicon-bug=cpu-tc052

Description

To bypass this CPU functional problem, the C compiler prevents load (ld) and store (st) instructions to be combined.

For example, silicon bug workaround CPU_TC052 prevents that two LD.W instructions are combined into one LD.DW instruction.

There is no assembler check for this silicon bug.
CPU TC.060

Command line option

--silicon-bug=cpu-tc060

Description

To bypass this CPU functional problem, the C compiler generates a `NOP` instruction between an `LD.A` / `LD.DA` instruction and a following `LD.W` / `LD.D` instruction, even if an integer instruction occurs in between.

The assembler issues a warning when an `LD.A` / `LD.DA` instruction is directly followed by an `LD.W` / `LD.D` instruction, or when only an integer instruction is in between.
CPU_TC.065

Command line option

--silicon-bug=cpu-tc065

Description

To bypass this CPU functional problem, the C compiler inserts a \texttt{NOP} instruction before a jump, when a label is directly followed by an unconditional jump.

The assembler issues a warning when a label is directly followed by an unconditional jump, only when debug information is turned off.
CPU_Tc.068

Command line option

--silicon-bug=cpu-tc068

Description

To bypass this CPU functional problem, the C compiler inserts a **DISABLE** and two **NOP** instructions before each **DVINIT** instruction (and if necessary an **ENABLE** after **TGE DVINIT**).

The assembler issues a warning when a **DVINIT** instruction is not preceded by a **DISABLE** and two **NOP** instructions.
CPU_TC.069

Command line option

--silicon-bug=cpu-tc069

Description

To bypass this CPU functional problem, the C compiler inserts a NOP instruction after each RSLCX instruction.

The assembler issues a warning when an RSLCX instruction is not followed by a NOP instruction.
**CPU_TC.070**

**Command line option**

`--silicon-bug=cpu-tc070`

**Description**

To bypass this CPU functional problem, the C compiler inserts a `NOP` instruction before a loop instruction, when a conditional jump, based on the value in an address register, is directly followed by a loop instruction.

The compiler inserts two `NOP` instructions before a loop instruction, when a conditional jump, based on the value in a data register, is directly followed by a loop instruction.

The assembler issues a warning when a conditional jump, based on the value in an address register, is directly followed by a loop instruction.

The assembler issues a warning when a conditional jump, based on the value in a data register, is directly followed by a loop instruction or when only a single `NOP` instruction is in between.
CPU_TC.071

Command line option

--silicon-bug=cpu-tc071

Description

To bypass this CPU functional problem, the C compiler inserts a NOP instruction before a loop instruction, when a label is directly followed by an unconditional loop instruction.

The assembler issues a warning when a label is directly followed by an unconditional loop instruction, only when debug information is turned off.
CPU_TC.072

Command line option

--silicon-bug=cpu-tc072

Description

To bypass this CPU functional problem, the C compiler inserts a `NOP` instruction before a loop instruction, when an instruction that updates an address register is followed by a conditional loop instruction which uses this address register.

The assembler issues a warning when an instruction that updates an address register is followed by a conditional loop instruction which uses this address register.
CPU_TC.074

Command line option

--silicon-bug=cpu-tc074

Description

The C compiler has no workaround for this problem.

To bypass this CPU functional problem, the assembler encodes the LOOPU instruction in such a way that bits 12-15 get the value 1.
CPU_TC.081

Command line option

--silicon-bug=cpu-tc081

Description

The C compiler has no workaround for this problem.

The assembler issues a warning when an address register load instruction, `LD.A` or `LD.DA`, targeting the A[10] register, is immediately followed by an operation causing a context switch.
CPU_TC.082

Command line option

--silicon-bug=cpu-tc082

Description

To bypass this CPU functional problem, the C compiler inserts a `NOP` instruction between a context store operation, `STUCX` or `STLCX`, and a memory load operation which reads from the last double-word address written by the context store.

The assembler issues a warning 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.
CPU_TC.083

Command line option

```bash
--silicon-bug=cpu-tc083
```

Description

To bypass this CPU functional problem, the C compiler inserts a **NOP** instruction after each **DISABLE** instruction.

The assembler issues a warning when the **DISABLE** instruction is not followed by a **NOP** instruction.
**CPU_TC.094**

**Command line option**

```bash
--silicon-bug=cpu-tc094
```

**Description**

To bypass this CPU functional problem, the C compiler inserts a *NOP* instruction between an IP jump and CSA list instruction.

The assembler issues a warning when an IP jump is followed by a CSA list instruction.
CPU_TC.095

Command line option

--silicon-bug=cpu-tc095

Description

To bypass this CPU functional problem, the C compiler inserts a \texttt{NOP} instruction between any \texttt{SAT.B/SAT.H} instruction and a following load-store instruction with a DGPR source operand (\texttt{addsc.a}, \texttt{addsc.at}, \texttt{mov.a}, \texttt{mtcr}).

The assembler issues a warning when a \texttt{SAT.B/SAT.H} instruction is immediately followed by a load-store instruction with a DGPR source operand (\texttt{addsc.a}, \texttt{addsc.at}, \texttt{mov.a}, \texttt{mtcr}).
**CPU_TC.096**

**Command line option**

```
--silicon-bug=cpu-tc096
```

**Description**

To bypass this CPU functional problem, the C compiler inserts two **NOP** instructions for a single group loop, between an IP instruction and a loop instruction targeting the IP instruction. One **NOP** is inserted between a LS and a loop instruction, when the single group loop exists of an optional IP instruction, a single LS instruction and a loop instruction targeting the first instruction.

The assembler issues a warning in the following situations:

```
label:
    loop Ax,label

label:
    <any instruction>
    loop Ax,label

label:
    <any IP-instruction>
    <any LS-instruction or NOP>
    loop Ax,label
```
**CPU_TC.103**

**Command line option**

`--silicon-bug=cpu-tc103`

**Description**

To bypass this CPU functional problem, the C compiler directs certain program flow instructions, such as RET, RFE, CALL and JI, running in spram (scratch pad ram) via a stub located in safe memory. In order to be able to tell the C compiler that certain code is predetermined for spram, the pragma `spram` and option `--spram` are introduced.

To bypass this CPU functional problem, the preprocessor define `__CPU_TC103__` is used in the `tc*.lsl` linker script files. The linker will collect the stubs as generated by the C compiler and locate them in safe non-spram memory. Furthermore it is tested if (the start of) the interrupt and trap table are located at safe addresses.

Safe non-SPRAM addresses are defined as any address except:

- `bit [15:14] = 11b` (TC1130, TC1115, TC1110 PMEM)
- `bit [14:13] = 11b` (TC1762, TC1764, TC1766 PMEM)
CPU_TC.104

Command line option

--silicon-bug=cpu-tc104

Description

To bypass this CPU functional problem, the C compiler inserts a NOP instruction before a double-word load instruction using circular addressing mode (LD.D instruction).

The assembler issues a warning when a double-word load instruction using circular addressing mode (LD.D instruction) is not preceded by a NOP instruction.
DMU_TC.001

Command line option

--silicon-bug=dmu-tc001

Description

To bypass this CPU functional problem, the C compiler avoids generation of the `ST.T`, `SWAP` and `LDMST` instructions. For direct `__bit` and bit-field operations, alternative instructions are used.

The assembler issues a warning for `SWAP`, `LDMST` and `ST.T` instructions.
PMI_TC.003

Command line option

--silicon-bug=pmi-tc003

Description

There is no C compiler bypass for this problem.

To bypass this CPU functional problem, the assembler adds a macro to the C startup code to set the TLB-A and TLB-B mappings to a page size of 16 kB. The SZA and SZB in the MMU_CON are set to 16 kB.
PMU_TC.004

Command line option

--silicon-bug=pmu-tc004

Description

There is no C compiler bypass for this problem.

To bypass this CPU functional problem, the assembler adds a macro to the C startup code to disable the split mode on the LMB bus. The SPLT bit of the SFR register LFI_CON is set to zero.
Chapter 17. MISRA-C Rules

This chapter contains an overview of the supported and unsupported MISRA C rules.

17.1. MISRA-C:1998

This section lists all supported and unsupported MISRA-C:1998 rules.

See also Section 4.7, C Code Checking: MISRA-C.

A number of MISRA-C rules leave room for interpretation. Other rules can only be checked in a limited way. In such cases the implementation decisions and possible restrictions for these rules are listed.

x means that the rule is not supported by the TASKING C compiler. (R) is a required rule, (A) is an advisory rule.

1. (R) The code shall conform to standard C, without language extensions
2. x (A) Other languages should only be used with an interface standard
3. (A) Inline assembly is only allowed in dedicated C functions
4. x (A) Provision should be made for appropriate run-time checking
5. (R) Only use characters and escape sequences defined by ISO C
6. x (R) Character values shall be restricted to a subset of ISO 10646-1
7. (R) Trigraphs shall not be used
8. (R) Multibyte characters and wide string literals shall not be used
9. (R) Comments shall not be nested
10. (A) Sections of code should not be "commented out"

In general, it is not possible to decide whether a piece of comment is C code that is commented out, or just some pseudo code. Instead, the following heuristics are used to detect possible C code inside a comment:

• a line ends with ‘;’, or
• a line starts with ‘{’, possibly preceded by white space

11. (R) Identifiers shall not rely on significance of more than 31 characters
12. (A) The same identifier shall not be used in multiple name spaces
13. (A) Specific-length typedefs should be used instead of the basic types
14. (R) Use 'unsigned char' or 'signed char' instead of plain 'char'
15. (A) Floating-point implementations should comply with a standard
16. (R) The bit representation of floating-point numbers shall not be used
   A violation is reported when a pointer to a floating-point type is converted to a pointer to an integer type.
17. (R) "typedef" names shall not be reused
18. (A) Numeric constants should be suffixed to indicate type
   A violation is reported when the value of the constant is outside the range indicated
   by the suffixes, if any.
19. (R) Octal constants (other than zero) shall not be used
20. (R) All object and function identifiers shall be declared before use
21. (R) Identifiers shall not hide identifiers in an outer scope
22. (A) Declarations should be at function scope where possible
   x 23. (A) All declarations at file scope should be static where possible
24. (R) Identifiers shall not have both internal and external linkage
   x 25. (R) Identifiers with external linkage shall have exactly one definition
26. (R) Multiple declarations for objects or functions shall be compatible
   x 27. (A) External objects should not be declared in more than one file
28. (A) The "register" storage class specifier should not be used
29. (R) The use of a tag shall agree with its declaration
30. (R) All automatics shall be initialized before being used
   This rule is checked using worst-case assumptions. This means that violations are
   reported not only for variables that are guaranteed to be uninitialized, but also for
   variables that are uninitialized on some execution paths.
31. (R) Braces shall be used in the initialization of arrays and structures
32. (R) Only the first, or all enumeration constants may be initialized
33. (R) The right hand operand of && or || shall not contain side effects
34. (R) The operands of a logical && or || shall be primary expressions
35. (R) Assignment operators shall not be used in Boolean expressions
36. (A) Logical operators should not be confused with bitwise operators
37. (R) Bitwise operations shall not be performed on signed integers
38. (R) A shift count shall be between 0 and the operand width minus 1 This violation will
   only be checked when the shift count evaluates to a constant value at compile time.
39. (R) The unary minus shall not be applied to an unsigned expression
40. (A) "sizeof" should not be used on expressions with side effects
   x 41. (A) The implementation of integer division should be documented
42. (R) The comma operator shall only be used in a "for" condition
43. (R) Don't use implicit conversions which may result in information loss
44. (A) Redundant explicit casts should not be used
45. (R) Type casting from any type to or from pointers shall not be used
46. (R) The value of an expression shall be evaluation order independent
   This rule is checked using worst-case assumptions. This means that a violation will
   be reported when a possible alias may cause the result of an expression to be
   evaluation order dependent.
47. (A) No dependence should be placed on operator precedence rules
Mixed arithmetic should use explicit casting
Tests of a (non-Boolean) value against 0 should be made explicit
F.P. variables shall not be tested for exact equality or inequality
Constant unsigned integer expressions should not wrap-around
There shall be no unreachable code
All non-null statements shall have a side-effect
A null statement shall only occur on a line by itself
Labels should not be used
The "goto" statement shall not be used
The "continue" statement shall not be used
The "break" statement shall not be used (except in a "switch")
An "if" or loop body shall always be enclosed in braces
All "if", "else if" constructs should contain a final "else"
Every non-empty "case" clause shall be terminated with a "break"
All "switch" statements shall contain a final "default" case
A "switch" expression should not represent a Boolean case
Every "switch" shall have at least one "case"
Floating-point variables shall not be used as loop counters
A "for" should only contain expressions concerning loop control
A violation is reported when the loop initialization or loop update expression modifies an object that is not referenced in the loop test.
Iterator variables should not be modified in a "for" loop
Functions shall always be declared at file scope
Functions with variable number of arguments shall not be used
Functions shall not call themselves, either directly or indirectly
A violation will be reported for direct or indirect recursive function calls in the source file being checked. Recursion via functions in other source files, or recursion via function pointers is not detected.
Function prototypes shall be visible at the definition and call
The function prototype of the declaration shall match the definition
Identifiers shall be given for all prototype parameters or for none
Parameter identifiers shall be identical for declaration/definition
Every function shall have an explicit return type
Functions with no parameters shall have a "void" parameter list
An actual parameter type shall be compatible with the prototype
The number of actual parameters shall match the prototype
The values returned by "void" functions shall not be used
Void expressions shall not be passed as function parameters
81. (A) "const" should be used for reference parameters not modified
82. (A) A function should have a single point of exit
83. (R) Every exit point shall have a "return" of the declared return type
84. (R) For "void" functions, "return" shall not have an expression
85. (A) Function calls with no parameters should have empty parentheses
86. (A) If a function returns error information, it should be tested
     A violation is reported when the return value of a function is ignored.
87. (R) #include shall only be preceded by other directives or comments
88. (R) Non-standard characters shall not occur in #include directives
89. (R) #include shall be followed by either <filename> or "filename"
90. (R) Plain macros shall only be used for constants/qualifiers/specifiers
91. (R) Macros shall not be #define'd and #undef'd within a block
92. (A) #undef should not be used
93. (A) A function should be used in preference to a function-like macro
94. (R) A function-like macro shall not be used without all arguments
95. (R) Macro arguments shall not contain pre-preprocessing directives
     A violation is reported when the first token of an actual macro argument is '#'.
96. (R) Macro definitions/parameters should be enclosed in parentheses
97. (A) Don't use undefined identifiers in pre-processing directives
98. (R) A macro definition shall contain at most one # or ## operator
99. (R) All uses of the #pragma directive shall be documented
     This rule is really a documentation issue. The compiler will flag all #pragma directives
     as violations.
100. (R) "defined" shall only be used in one of the two standard forms
101. (A) Pointer arithmetic should not be used
102. (A) No more than 2 levels of pointer indirection should be used
     A violation is reported when a pointer with three or more levels of indirection is
     declared.
103. (R) No relational operators between pointers to different objects
     In general, checking whether two pointers point to the same object is impossible. The
     compiler will only report a violation for a relational operation with incompatible pointer
     types.
104. (R) Non-constant pointers to functions shall not be used
105. (R) Functions assigned to the same pointer shall be of identical type
106. (R) Automatic address may not be assigned to a longer lived object
107. (R) The null pointer shall not be de-referenced
     A violation is reported for every pointer dereference that is not guarded by a NULL
     pointer test.
108. (R) All struct/union members shall be fully specified
109. (R) Overlapping variable storage shall not be used. A violation is reported for every 'union' declaration.

110. (R) Unions shall not be used to access the sub-parts of larger types. A violation is reported for a 'union' containing a 'struct' member.

111. (R) Bit-fields shall have type "unsigned int" or "signed int".

112. (R) Bit-fields of type "signed int" shall be at least 2 bits long.

113. (R) All struct/union members shall be named.

114. (R) Reserved and standard library names shall not be redefined.

115. (R) Standard library function names shall not be reused.

* 116. (R) Production libraries shall comply with the MISRA C restrictions.

* 117. (R) The validity of library function parameters shall be checked.

118. (R) Dynamic heap memory allocation shall not be used.

119. (R) The error indicator "errno" shall not be used.

120. (R) The macro "offsetof" shall not be used.

121. (R) <locale.h> and the "setlocale" function shall not be used.

122. (R) The "setjmp" and "longjmp" functions shall not be used.

123. (R) The signal handling facilities of <signal.h> shall not be used.

124. (R) The <stdio.h> library shall not be used in production code.

125. (R) The functions atof/atoi/atol shall not be used.

126. (R) The functions abort/exit/getenv/system shall not be used.

127. (R) The time handling functions of library <time.h> shall not be used.

17.2. MISRA-C:2004

This section lists all supported and unsupported MISRA-C:2004 rules.

See also Section 4.7, C Code Checking: MISRA-C.

A number of MISRA-C rules leave room for interpretation. Other rules can only be checked in a limited way. In such cases the implementation decisions and possible restrictions for these rules are listed.

* means that the rule is not supported by the TASKING C compiler. (R) is a required rule, (A) is an advisory rule.

Environment


1.2 (R) No reliance shall be placed on undefined or unspecified behavior.
Multiple compilers and/or languages shall only be used if there is a common defined interface standard for object code to which the languages/compilers/assemblers conform.

The compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers.

Floating-point implementations should comply with a defined floating-point standard.

### Language extensions

- **2.1** (R) Assembly language shall be encapsulated and isolated.
- **2.2** (R) Source code shall only use /* ... */ style comments.
- **2.3** (R) The character sequence /* shall not be used within a comment.
- **2.4** (A) Sections of code should not be "commented out". In general, it is not possible to decide whether a piece of comment is C code that is commented out, or just some pseudo code. Instead, the following heuristics are used to detect possible C code inside a comment: - a line ends with ';', or - a line starts with '}', possibly preceded by white space

### Documentation

- **3.1** (R) All usage of implementation-defined behavior shall be documented.
- **3.2** (R) The character set and the corresponding encoding shall be documented.
- **3.3** (A) The implementation of integer division in the chosen compiler should be determined, documented and taken into account.
- **3.4** (R) All uses of the #pragma directive shall be documented and explained. This rule is really a documentation issue. The compiler will flag all #pragma directives as violations.
- **3.5** (R) The implementation-defined behavior and packing of bit-fields shall be documented if being relied upon.
- **3.6** (R) All libraries used in production code shall be written to comply with the provisions of this document, and shall have been subject to appropriate validation.

### Character sets

- **4.1** (R) Only those escape sequences that are defined in the ISO C standard shall be used.
- **4.2** (R) Trigraphs shall not be used.

### Identifiers

- **5.1** (R) Identifiers (internal and external) shall not rely on the significance of more than 31 characters.
- **5.2** (R) Identifiers in an inner scope shall not use the same name as an identifier in an outer scope, and therefore hide that identifier.
5.3 (R) A typedef name shall be a unique identifier.
5.4 (R) A tag name shall be a unique identifier.
× 5.5 (A) No object or function identifier with static storage duration should be reused.
5.6 (A) No identifier in one name space should have the same spelling as an identifier in
another name space, with the exception of structure and union member names.
× 5.7 (A) No identifier name should be reused.

Types

6.1 (R) The plain char type shall be used only for storage and use of character values.
× 6.2 (R) signed and unsigned char type shall be used only for the storage and use of
numeric values.
6.3 (A) typedefs that indicate size and signedness should be used in place of the basic
types.
6.4 (R) bit-fields shall only be defined to be of type unsigned int or signed int.
6.5 (R) bit-fields of type signed int shall be at least 2 bits long.

Constants

7.1 (R) Octal constants (other than zero) and octal escape sequences shall not be used.

Declarations and definitions

8.1 (R) Functions shall have prototype declarations and the prototype shall be visible at both
the function definition and call.
8.2 (R) Whenever an object or function is declared or defined, its type shall be explicitly
stated.
8.3 (R) For each function parameter the type given in the declaration and definition shall be
identical, and the return types shall also be identical.
8.4 (R) If objects or functions are declared more than once their types shall be compatible.
8.5 (R) There shall be no definitions of objects or functions in a header file.
8.6 (R) Functions shall be declared at file scope.
8.7 (R) Objects shall be defined at block scope if they are only accessed from within a single
function.
× 8.8 (R) An external object or function shall be declared in one and only one file.
× 8.9 (R) An identifier with external linkage shall have exactly one external definition.
× 8.10 (R) All declarations and definitions of objects or functions at file scope shall have internal
linkage unless external linkage is required.
8.11 (R) The static storage class specifier shall be used in definitions and declarations of
objects and functions that have internal linkage.
When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.

Initialization

All automatic variables shall have been assigned a value before being used. This rule is checked using worst-case assumptions. This means that violations are reported not only for variables that are guaranteed to be uninitialized, but also for variables that are uninitialized on some execution paths.

Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.

In an enumerator list, the "=" construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.

Arithmetic type conversions

The value of an expression of integer type shall not be implicitly converted to a different underlying type if:

- it is not a conversion to a wider integer type of the same signedness, or
- the expression is complex, or
- the expression is not constant and is a function argument, or
- the expression is not constant and is a return expression.

The value of an expression of floating type shall not be implicitly converted to a different type if:

- it is not a conversion to a wider floating type, or
- the expression is complex, or
- the expression is a function argument, or
- the expression is a return expression.

The value of a complex expression of integer type may only be cast to a type that is narrower and of the same signedness as the underlying type of the expression.

The value of a complex expression of floating type may only be cast to a narrower floating type.

If the bitwise operators \(~\) and \(<<\) are applied to an operand of underlying type \(\text{unsigned char}\) or \(\text{unsigned short}\), the result shall be immediately cast to the underlying type of the operand.

A "U" suffix shall be applied to all constants of \(\text{unsigned}\) type.

Pointer type conversions

Conversions shall not be performed between a pointer to a function and any type other than an integral type.

Conversions shall not be performed between a pointer to object and any type other than an integral type, another pointer to object type or a pointer to void.

A cast should not be performed between a pointer type and an integral type.
11.4  (A) A cast should not be performed between a pointer to object type and a different pointer to object type.

11.5  (R) A cast shall not be performed that removes any \texttt{const} or \texttt{volatile} qualification from the type addressed by a pointer.

**Expressions**

12.1  (A) Limited dependence should be placed on C's operator precedence rules in expressions.

12.2  (R) The value of an expression shall be the same under any order of evaluation that the standard permits. This rule is checked using worst-case assumptions. This means that a violation will be reported when a possible alias may cause the result of an expression to be evaluation order dependent.

12.3  (R) The \texttt{sizeof} operator shall not be used on expressions that contain side effects.

12.4  (R) The right-hand operand of a logical \texttt{&&} or \texttt{||} operator shall not contain side effects.

12.5  (R) The operands of a logical \texttt{&&} or \texttt{||} shall be primary-expressions.

12.6  (A) The operands of logical operators (\texttt{&&}, \texttt{||}, \texttt{and} \texttt{!}) should be effectively Boolean. Expressions that are effectively Boolean should not be used as operands to operators other than (\texttt{&&}, \texttt{||}, \texttt{and} \texttt{!}).

12.7  (R) Bitwise operators shall not be applied to operands whose underlying type is signed.

12.8  (R) The right-hand operand of a shift operator shall lie between zero and one less than the width in bits of the underlying type of the left-hand operand. This violation will only be checked when the shift count evaluates to a constant value at compile time.

12.9  (R) The unary minus operator shall not be applied to an expression whose underlying type is unsigned.

12.10 (R) The comma operator shall not be used.

12.11 (A) Evaluation of constant unsigned integer expressions should not lead to wrap-around.

12.12 (R) The underlying bit representations of floating-point values shall not be used. A violation is reported when a pointer to a floating-point type is converted to a pointer to an integer type.

12.13 (A) The increment (\texttt{++}) and decrement (\texttt{--}) operators should not be mixed with other operators in an expression.

**Control statement expressions**

13.1  (R) Assignment operators shall not be used in expressions that yield a Boolean value.

13.2  (A) Tests of a value against zero should be made explicit, unless the operand is effectively Boolean.

13.3  (R) Floating-point expressions shall not be tested for equality or inequality.

13.4  (R) The controlling expression of a \texttt{for} statement shall not contain any objects of floating type.
The three expressions of a for statement shall be concerned only with loop control. A violation is reported when the loop initialization or loop update expression modifies an object that is not referenced in the loop test.

13.6 (R) Numeric variables being used within a for loop for iteration counting shall not be modified in the body of the loop.

13.7 (R) Boolean operations whose results are invariant shall not be permitted.

Control flow

14.1 (R) There shall be no unreachable code.

14.2 (R) All non-null statements shall either:
   a) have at least one side effect however executed, or
   b) cause control flow to change.

14.3 (R) Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment provided that the first character following the null statement is a white-space character.

14.4 (R) The goto statement shall not be used.

14.5 (R) The continue statement shall not be used.

14.6 (R) For any iteration statement there shall be at most one break statement used for loop termination.

14.7 (R) A function shall have a single point of exit at the end of the function.

14.8 (R) The statement forming the body of a switch, while, do ... while or for statement be a compound statement.

14.9 (R) An if (expression) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement.

14.10 (R) All if ... else if constructs shall be terminated with an else clause.

Switch statements

15.1 (R) A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.

15.2 (R) An unconditional break statement shall terminate every non-empty switch clause.

15.3 (R) The final clause of a switch statement shall be the default clause.

15.4 (R) A switch expression shall not represent a value that is effectively Boolean.

15.5 (R) Every switch statement shall have at least one case clause.

Functions

16.1 (R) Functions shall not be defined with variable numbers of arguments.
Functions shall not call themselves, either directly or indirectly. A violation will be reported for direct or indirect recursive function calls in the source file being checked. Recursion via functions in other source files, or recursion via function pointers is not detected.

Identifiers shall be given for all of the parameters in a function prototype declaration.

The identifiers used in the declaration and definition of a function shall be identical.

Functions with no parameters shall be declared with parameter type void.

The number of arguments passed to a function shall match the number of parameters.

A pointer parameter in a function prototype should be declared as pointer to const if the pointer is not used to modify the addressed object.

All exit paths from a function with non-void return type shall have an explicit return statement with an expression.

A function identifier shall only be used with either a preceding &, or with a parenthesized parameter list, which may be empty.

If a function returns error information, then that error information shall be tested. A violation is reported when the return value of a function is ignored.

Pointers and arrays

Pointer arithmetic shall only be applied to pointers that address an array or array element.

Pointer subtraction shall only be applied to pointers that address elements of the same array.

>, >=, <, <= shall not be applied to pointer types except where they point to the same array. In general, checking whether two pointers point to the same object is impossible. The compiler will only report a violation for a relational operation with incompatible pointer types.

Array indexing shall be the only allowed form of pointer arithmetic.

The declaration of objects should contain no more than 2 levels of pointer indirection. A violation is reported when a pointer with three or more levels of indirection is declared.

The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

Structures and unions

All structure or union types shall be complete at the end of a translation unit.

An object shall not be assigned to an overlapping object.

An area of memory shall not be reused for unrelated purposes.

Unions shall not be used.
Preprocessing directives

19.1 (A) `#include` statements in a file should only be preceded by other preprocessor directives or comments.

19.2 (A) Non-standard characters should not occur in header file names in `#include` directives.

19.3 (R) The `#include` directive shall be followed by either a `<filename>` or "filename" sequence.

19.4 (R) C macros shall only expand to a braced initializer, a constant, a parenthesized expression, a type qualifier, a storage class specifier, or a do-while-zero construct.

19.5 (R) Macros shall not be `#define`d or `#undef`d within a block.

19.6 (R) `#undef` shall not be used.

19.7 (A) A function should be used in preference to a function-like macro.

19.8 (R) A function-like macro shall not be invoked without all of its arguments.

19.9 (R) Arguments to a function-like macro shall not contain tokens that look like preprocessing directives. A violation is reported when the first token of an actual macro argument is '#'.

19.10 (R) In the definition of a function-like macro each instance of a parameter shall be enclosed in parentheses unless it is used as the operand of `#` or `##`.

19.11 (R) All macro identifiers in preprocessor directives shall be defined before use, except in `#ifdef` and `#ifndef` preprocessor directives and the `defined()` operator.

19.12 (R) There shall be at most one occurrence of the `#` or `##` preprocessor operators in a single macro definition.

19.13 (A) The `#` and `##` preprocessor operators should not be used.

19.14 (R) The `defined` preprocessor operator shall only be used in one of the two standard forms.

19.15 (R) Precautions shall be taken in order to prevent the contents of a header file being included twice.

19.16 (R) Preprocessing directives shall be syntactically meaningful even when excluded by the preprocessor.

19.17 (R) All `#else`, `#elif` and `#endif` preprocessor directives shall reside in the same file as the `#if` or `#ifdef` directive to which they are related.

Standard libraries

20.1 (R) Reserved identifiers, macros and functions in the standard library, shall not be defined, redefined or undefined.

20.2 (R) The names of standard library macros, objects and functions shall not be reused.

20.3 (R) The validity of values passed to library functions shall be checked.

20.4 (R) Dynamic heap memory allocation shall not be used.

20.5 (R) The error indicator `errno` shall not be used.
20.6 (R) The macro offsetof, in library <stddef.h>, shall not be used.
20.7 (R) The setjmp macro and the longjmp function shall not be used.
20.8 (R) The signal handling facilities of <signal.h> shall not be used.
20.9 (R) The input/output library <stdio.h> shall not be used in production code.
20.10 (R) The library functions atof, atoi and atol from library <stdlib.h> shall not be used.
20.11 (R) The library functions abort, exit, getenv and system from library <stdlib.h> shall not be used.
20.12 (R) The time handling functions of library <time.h> shall not be used.

Run-time failures

× 21.1 (R) Minimization of run-time failures shall be ensured by the use of at least one of:
   a) static analysis tools/techniques;
   b) dynamic analysis tools/techniques;
   c) explicit coding of checks to handle run-time faults.