## Table of Contents

1. C Language ........................................................................................................................................................................ 1  
   1.1. Data Types ..................................................................................................................................................................... 2  
      1.1.1. Half Precision Floating-Point .......................................................................................................................... 3  
   1.2. Changing the Alignment: __unaligned, __packed__ and __align() ................................................................. 3  
   1.3. Accessing Memory .......................................................................................................................................................... 4  
      1.3.1. Memory Qualifiers ................................................................................................................................. 5  
      1.3.2. Placing an Object at an Absolute Address: __at() ....................................................................................... 6  
   1.4. Accessing Hardware from C ....................................................................................................................................... 6  
   1.5. Multi-Core Support ....................................................................................................................................................... 7  
      1.5.1. Compile Time Core Association ...................................................................................................................... 8  
   1.6. Shift JIS Kanji Support ................................................................................................................................................. 16  
   1.7. Using Assembly in the C Source: __asm() .................................................................................................................. 17  
   1.8. Attributes ..................................................................................................................................................................... 23  
   1.9. Pragmas to Control the Compiler ............................................................................................................................. 27  
   1.10. Predefined Preprocessor Macros ............................................................................................................................ 33  
   1.11. Switch Statement ....................................................................................................................................................... 34  
   1.12. Functions ................................................................................................................................................................... 36  
      1.12.1. Calling Convention .............................................................................................................................................. 36  
      1.12.2. Inlining Functions: inline .............................................................................................................................. 37  
      1.12.3. Interrupt Functions / Exception Handlers ...................................................................................................... 39  
      1.12.4. Intrinsic Functions ............................................................................................................................................. 41  
   1.13. Compiler Generated Sections ..................................................................................................................................... 48  
      1.13.1. Rename Sections ............................................................................................................................................ 49  
      1.13.2. Influence Section Definition ......................................................................................................................... 50  
2. C++ Language ......................................................................................................................................................................... 53  
   2.1. C++ Language Extension Keywords ............................................................................................................................ 53  
   2.2. C++ Dialect Accepted ..................................................................................................................................................... 53  
      2.2.1. C++03 Mode ......................................................................................................................................................... 54  
      2.2.2. GNU C++ Mode .................................................................................................................................................. 55  
      2.2.3. Anachronisms Accepted .................................................................................................................................. 56  
      2.2.4. Atomic Operations ........................................................................................................................................... 57  
   2.3. Namespace Support ......................................................................................................................................................... 58  
   2.4. Template Instantiation .................................................................................................................................................... 60  
      2.4.1. Instantiation Modes .............................................................................................................................................. 61  
      2.4.2. Instantiation #pragma Directives ..................................................................................................................... 61  
      2.4.3. Implicit Inclusion ............................................................................................................................................... 63  
   2.5. Inlining Functions ........................................................................................................................................................... 64  
   2.6. Extern Inline Functions .................................................................................................................................................. 64  
   2.7. Pragmas to Control the C++ Compiler ......................................................................................................................... 65  
      2.7.1. C pragmas Supported by the C++ compiler ....................................................................................................... 65  
   2.8. Predefined Macros ......................................................................................................................................................... 66  
   2.9. Precompiled Headers ..................................................................................................................................................... 68  
      2.9.1. Automatic Precompiled Header Processing .................................................................................................. 70  
      2.9.2. Manual Precompiled Header Processing ....................................................................................................... 74  
      2.9.3. Other Ways to Control Precompiled Headers ............................................................................................... 74  
      2.9.4. Performance Issues ........................................................................................................................................... 74  
3. Assembly Language ................................................................................................................................................................. 77  
   3.1. Assembly Syntax ......................................................................................................................................................... 77
17. Linker Script Language (LSL) ................................................................. 809
  17.1. Structure of a Linker Script File ...................................................... 809
  17.2. Syntax of the Linker Script Language ........................................... 811
    17.2.1. Preprocessing ........................................................................... 811
    17.2.2. Lexical Syntax ......................................................................... 812
    17.2.3. Identifiers and Tags ................................................................. 813
    17.2.4. Expressions ............................................................................. 813
    17.2.5. Built-in Functions .................................................................... 814
  17.2.6. LSL Definitions in the Linker Script File .................................... 816

  16. Object File Formats ........................................................................... 803
    16.1. ELF/DWARF Object Format ....................................................... 803
    16.2. Intel Hex Record Format ............................................................. 803
    16.3. Motorola S-Record Format ......................................................... 806

  15. List File Formats ............................................................................... 793
    15.1. Assembler List File Format ......................................................... 793
    15.2. Linker Map File Format ............................................................... 794

  14. C Library Reentrancy ....................................................................... 745
    14.2.7. fcntl.h ..................................................................................... 749
    14.2.8. fenv.h ..................................................................................... 749
    14.2.9. float.h .................................................................................... 750
    14.2.10. inttypes.h and stdint.h ....................................................... 750
    14.2.11. io.h ...................................................................................... 751
    14.2.12. iso646.h .............................................................................. 751
    14.2.13. limits.h ............................................................................... 751
    14.2.14. locale.h ............................................................................... 751
    14.2.15. malloc.h .............................................................................. 752
    14.2.16. math.h and tgmath.h ........................................................... 752
    14.2.17. setjmp.h .............................................................................. 756
    14.2.18. signal.h ............................................................................... 757
    14.2.19. stdalign.h ........................................................................... 757
    14.2.20. stdarg.h ............................................................................... 757
    14.2.21. statomic.h ........................................................................... 758
    14.2.22. stdbool.h ............................................................................... 762
    14.2.23. stddef.h ............................................................................... 762
    14.2.24. stdint.h ............................................................................... 762
    14.2.25. stdio.h and wchar.h ........................................................... 762
    14.2.26. stdlib.h and wchar.h ........................................................... 770
    14.2.27. stdnoreturn.h ...................................................................... 773
    14.2.28. string.h and wchar.h .......................................................... 774
    14.2.29. time.h and wchar.h ............................................................. 775
    14.2.30. typeinfo.h ........................................................................... 778
    14.2.31. uchar.h ................................................................................. 778
    14.2.32. unistd.h ............................................................................... 778
    14.2.33. wchar.h ............................................................................... 779
    14.2.34. wctype.h .............................................................................. 780

  14.3. C Library Reentrancy ................................................................... 781
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 ARM® C compiler fully supports the ISO C99 standard and supports all mandatory language features of the C11 standard, and adds extra possibilities to program the special functions of the target. C11 is the default of the C compiler.

The TASKING C compiler meets and exceeds the minimum requirements in all cases, only limited by the amount of memory available to the compiler.

C11 language features

All mandatory ISO C11 language features are supported (ISO/IEC 9899:2011 section 6.10.8.1 Mandatory macros). Furthermore the C compiler supports the following conditional features (ISO/IEC 9899:2011 section 6.10.8.3 Conditional feature macros):

- atomic types (including the _Atomic type qualifier) and the <stdatomic.h> header file for ARMv7 architectures (__STDC_NO_ATOMICS__ is undefined for those architectures)
- complex types and the <complex.h> header file
- variable length arrays and variably modified types

Other conditional language features such as threads, as mentioned in section 6.10.8.3 Conditional feature macros and section 6.10.8.2 Environment macros of the ISO/IEC 9899:2011 standard, are not supported. __STDC_NO_THREADS__ is defined as 1.

Additional language features

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

- 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 TASKING C compiler for ARM supports the ISO C11 defined 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>_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>32</td>
<td>$[-2^{31}, 2^{31}-1]$</td>
</tr>
<tr>
<td>unsigned int</td>
<td>32</td>
<td>32</td>
<td>$[0, 2^{32}-1]$</td>
</tr>
<tr>
<td>enum</td>
<td>32</td>
<td>32</td>
<td>$[-2^{31}, 2^{31}-1]$</td>
</tr>
<tr>
<td>long</td>
<td>32</td>
<td>32</td>
<td>$[-2^{31}, 2^{31}-1]$</td>
</tr>
<tr>
<td>unsigned long</td>
<td>32</td>
<td>32</td>
<td>$[0, 2^{32}-1]$</td>
</tr>
<tr>
<td>long long</td>
<td>64</td>
<td>64</td>
<td>$[-2^{63}, 2^{63}-1]$</td>
</tr>
<tr>
<td>unsigned long long</td>
<td>64</td>
<td>64</td>
<td>$[0, 2^{64}-1]$</td>
</tr>
<tr>
<td>_Float16 (10-bit mantissa)²</td>
<td>16</td>
<td>16</td>
<td>$[-65504.0F, -6.103515625E-05]$ [+6.103515625E-05, +65504.0F]\</td>
</tr>
<tr>
<td>float (23-bit mantissa)</td>
<td>32</td>
<td>32</td>
<td>$[-3.402E+38, -1.175E-38]$ [+1.175E-38, +3.402E+38]\</td>
</tr>
<tr>
<td>double</td>
<td>64</td>
<td>64</td>
<td>$[-1.797E+308, -2.225E-308]$ [+2.225E-308, +1.797E+308]\</td>
</tr>
<tr>
<td>long double (52-bit mantissa)</td>
<td>64</td>
<td>64</td>
<td>$[-3.402E+38i, -1.175E-38i]$ [+1.175E-38i, +3.402E+38i]\</td>
</tr>
<tr>
<td>_Imaginary float</td>
<td>32</td>
<td>32</td>
<td>$[-1.797E+308i, -2.225E-308i]$ [+2.225E-308i, +1.797E+308i]\</td>
</tr>
<tr>
<td>_Imaginary double</td>
<td>64</td>
<td>64</td>
<td>real part + imaginary part</td>
</tr>
<tr>
<td>_Imaginary long double</td>
<td>128</td>
<td>64</td>
<td>real part + imaginary part</td>
</tr>
<tr>
<td>pointer to data or function</td>
<td>32</td>
<td>32</td>
<td>$[0, 2^{32}-1]$</td>
</tr>
</tbody>
</table>

² The C compiler supports half-precision (16-bit) floating-point via the _Float16 type using the binary16 interchange format. See also Section 1.1.1, Half Precision Floating-Point.
Atomic type qualifier

The __Atomic type qualifier is supported and designates an atomic type as specified in the C11 standard.

1.1.1. Half Precision Floating-Point

The TASKING C compiler supports half precision (16-bit) floating-point via the __Float16 type using the binary16 interchange format. The binary16 interchange format is defined in IEEE Std 754-2008 IEEE Standard for Floating-Point Arithmetic. The __Float16 type is defined in ISO/IEC TS 18661-3:2015(E) – 2015-06-10 WG14 N1945.

The __Float16 type with binary16 format can represent normalized values in the range of $2^{-14}$ to 65504. There are 11 bits of significant precision, approximately 3 decimal digits. Also subnormal values are supported, as defined by FLT16_HAS_SUBNORM in float.h.

__Float16 values are automatically promoted to float.

Note that all conversions from and to __Float16 involve an intermediate conversion to float. Because of rounding, this can sometimes produce a different result than a direct conversion.

The C compiler generates hardware floating-point data instructions (VCVTB) for conversions between __Float16 and float if the architecture (VFP) supports it (VFPv4 and later).

Language-level support for the __Float16 data type is independent of whether the C compiler generates code using hardware floating-point instructions or not. In cases where hardware support is not specified or not available for the selected architecture, the C compiler implements conversions between __Float16 and float values as run-time library calls (--fpu=none, .., --fpu=VFPv3). These run-time functions are called __aeabi_f2h and __aeabi_h2f.

__Float16 __aeabi_f2h( float f ); // single precision to half precision
float __aeabi_h2f( __Float16 f ); // half precision to single precision

1.2. Changing the Alignment: __unaligned, __packed__ and __align()

Normally data, pointers and structure members are aligned according to the table in the previous section.

Suppress alignment

With the type qualifier __unaligned you can specify to suppress the alignment of objects or structure members. This can be useful to create compact data structures. In this case the alignment will be one byte for objects and non-bit-field structure members.

At the left side of a pointer declaration you can use the type qualifier __unaligned to mark the pointer value as potentially unaligned. This can be useful to access externally defined data. However the compiler can generate less efficient instructions to dereference such a pointer, to avoid unaligned memory access.

You can always convert a normal pointer to an unaligned pointer. Conversions from an unaligned pointer to an aligned pointer are also possible. However, the compiler will generate a warning in this situation,
with the exception of the following case: when the logical type of the destination pointer is `char` or `void`, no warning will be generated.

Example:

```c
struct
{
    char c;
    __unaligned int i; /* aligned at offset 1 ! */
} s;

__unaligned int * up = & s.i;
```

### Packed structures

To prevent alignment gaps in structures, you can use the attribute `__packed__`. When you use the attribute `__packed__` directly after the keyword `struct`, all structure members are marked `__unaligned`. For example the following two declarations are the same:

```c
struct __packed__
{
    char c;
    int * i;
} s1;

struct
{
    char __unaligned c;
    int * __unaligned i; /* __unaligned at right side of '*' to pack pointer member */
} s2;
```

The attribute `__packed__` has the same effect as adding the type qualifier `__unaligned` to the declaration to suppress the standard alignment.

You can also use `__packed__` in a pointer declaration. In that case it affects the alignment of the pointer itself, not the value of the pointer. The following two declarations are the same:

```c
int * __unaligned p;
int * p __packed__;
```

### Change alignment

With the attribute `__align(n)` you can overrule the default alignment of objects or structure members to `n` bytes.

### 1.3. 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()`.  

### 1.3.1. 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 type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__tcm</code></td>
<td>Tightly-Coupled Memory (TCM)</td>
<td>0x00000000 for ITCM, 0x20000000 for DTCM (device specific)</td>
<td>16 KiB (equals memory size)</td>
<td>32-bit</td>
<td>.text for ITCM, .data or .bss for DTCM</td>
</tr>
<tr>
<td><code>__tcm0</code></td>
<td>Tightly-Coupled Memory (TCM) in core cm7_0</td>
<td>0x00000000 for ITCM, 0x20000000 for DTCM (device specific)</td>
<td>16 KiB (equals memory size)</td>
<td>32-bit</td>
<td>.text for ITCM, .data or .bss for DTCM</td>
</tr>
<tr>
<td><code>__tcm1</code></td>
<td>Tightly-Coupled Memory (TCM) in core cm7_1</td>
<td>0x00000000 for ITCM, 0x20000000 for DTCM (device specific)</td>
<td>16 KiB (equals memory size)</td>
<td>32-bit</td>
<td>.text for ITCM, .data or .bss for DTCM</td>
</tr>
</tbody>
</table>

The memory qualifiers `__tcm0` and `__tcm1` are like `__tcm`, but are for use in a virtual-core multi-core project. For more information see Section 1.5, Multi-Core Support.

Of course these memory qualifiers only make sense for devices that support this type of memory, such as cyt4xxxxx_m7 devices.

All these memory qualifiers 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**

```c
__tcm int i; /* 32-bit object in .bss.tcm */
int j; /* 32-bit object in .bss */
int __tcm * p; /* pointer to int in __tcm memory */
                  (pointer has 32-bit size) /*

You cannot use memory qualifiers in structure declarations:

```c
struct S {
    __tcm int i; /* put an integer in TCM */
```
1.3.2. Placing an Object at an Absolute Address: __at()

With the attribute __at() you can specify an absolute address.

The compiler checks the address range, the alignment and if an object crosses a page boundary.

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;
```

The variable i is placed at address 0x1000 and is initialized.

```c
void f(void) __at( 0xf0ff + 1 ) { }
```

The function f is placed at address 0xf100.

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. Otherwise the compiler generates an error.

- You can place only global variables at absolute addresses. Parameters of functions, or automatic variables within functions cannot be placed at absolute addresses. If they are, the compiler generates an error.

- A variable that is declared extern, is not allocated by the compiler in the current module. Hence you should not use the keyword __at() on an external variable. If you do, the compiler ignores the keyword __at() without generating an error. Use __at() at the definition of the variable.

- You cannot place structure members at an absolute address. If you do, the compiler ignores the keyword __at() and generates a warning.

- 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.

1.4. Accessing Hardware from C

It is easy to access Special Function Registers (SFRs) that relate to peripherals from C. The SFRs are defined in a special include file (*.h) as symbol names for use with the compiler.
The TASKING VX-toolset for ARM supports the Cortex Microcontroller Software Interface Standard (CMSIS). You can find details about this standard on www.arm.com.

The product includes a full set of CMSIS files in the `cmsis` directory under the product installation directory. This includes SFR files for the various Cortex cores. The organization of the CMSIS files in the product installation is as follows:

- `cmsis/Device/ARM`: CMSIS-CORE files for the supported ARM processors
- `cmsis/DSP_Lib`: DSP Library Collection
- `cmsis/Include`: CMSIS-CORE header files
- `cmsis/RTOS/Template`: Contains the template header file `cmsis_os.h` for a CMSIS-RTOS compliant RTOS

When you include CMSIS SFR file in your source you must set an include search path to the appropriate CMSIS directory.

Example of including an SFR file:

```c
#include "ARMCM3.h"

void main(void)
{
    NVIC->ISER[0] = 1;
}
```

Compiler invocation:

```
ccarm -CARMv7M -I"installation_dir\cmsis\Device\ARM\ARMCM3\Include"
-I"installation_dir\cmsis\Include" file.c
```

When you use Eclipse you can easily add the include search paths by using the option `Project » Properties for » C/C++ Build » Settings » C/C++ Compiler » Add CMSIS include paths.`

### 1.5. Multi-Core Support

The TASKING toolset for ARM supports development of separate applications for each core in a multi-core device. When multiple identical cores are available (2 Cortex-M7 cores in cyt4xxxxx devices), an alternative configuration is also available, where a single application can be built for all (2 of) those cores. All Traveo II CYT2/3/4 devices are multi-core in the sense that they have a CM0+ and a CM4 (CYT2) or a CM7. The CM7 itself can be single-core (CYT3) or dual-core (CYT4). Each core has its own memory. Memory can be shared among cores or can be specific to the ITCM/DTCM memory of a core.

#### multi-core project

A multi-core project is required if the cores share the same memory map. In a multi-core project each core of a multi-core processor can have its own single-core project, with one of the projects acting as the base project which references the other projects.
**single-core project**

Single-core is supported for Traveo II CYT2 or CYT3 devices. This means the application only runs on the core of the selected device. The core is automatically configured when you select a device marked with _M0, _M4 or _M7 in Eclipse.

If two identical CM7 cores on Traveo II CYT4 devices are not combined they each have their own single-core project. Core number suffixes are not allowed in qualifiers in this case. You can use the generic __tcm and __interrupt qualifiers and the linker will take care of allocation to the proper core. Note also that you cannot use the __clone qualifier in single-core projects.

In Eclipse select the specific core, Core CM7 0 (this is the default) or Core CM7 1. If you build your sources on the command line with the control program, you also have to specify control program option --lsl-core=cm7_n, where n is 0 or 1.

See also Section 7.5, Linking Core-Specific Projects into a Multi-Core Application.

**virtual-core project**

Some cores of a multi-core processor can also share a single multi-core project. In that case the cores are represented to the linker as a so-called virtual core. For the CYT2/3/4 the CM0 cannot be combined into a virtual core with the other core(s) because it has a different ARM architecture. For the CYT4, the two CM7 cores can optionally be combined into one virtual core. If they are combined they share the same virtual-core (or dual-core) project. Combining the two CM7 cores has the advantage that code (for example library code) and data can be shared (default, __share qualifier) by the two cores and does not have to be duplicated in memory. An exception to this are the TCM memories: in a virtual-core project code and data can be allocated to a specific TCM memory by means of the __tcm0 and __tcm1 qualifiers, or code and data can be allocated to the TCM of both cores by means of the __clone qualifier (see Section 1.5.1, Compile Time Core Association). Similarly, in a virtual-core project an interrupt handler can be allocated to the vector table of a specific core by means of the __interrupt0 and __interrupt1 qualifiers (see Section 1.12.3, Interrupt Functions / Exception Handlers).

In Eclipse you need to select “All cores”. If you build your sources on the command line with the control program, you have to specify control program option --lsl-core=vm7. The control program will call the C compiler, assembler and linker with the correct options.

### 1.5.1. Compile Time Core Association

Code and data in a virtual-core project can be shared, TCM or cloned:

- **Shared.** In the default situation all code and data are accessible by all cores. The symbols are located in shared memory.

- **TCM.** TCM means that the code and/or data is copied to, and accessed from, the local ITCM and/or DTCM memory of one particular core.

- **Cloned.** Cloned means that code or data is copied to the same location (address) in local ITCM or DTCM memory for each compatible core, or optionally for each specified compatible core. The core then treats the code/data as if it were TCM.

In a single-core project you can only use TCM when the core has ITCM or DTCM memory.
Compile time core association

You can determine at compile time, with memory qualifiers or pragmas or options, whether data or code objects are specific to TCM or cloned instead of shared for local RAM (ITCM/DTCM). This is explained in the following sections.

Link time core association

Instead of at compile time, you can determine in which memory objects should be located at link time. This is necessary when you want to restrict a clone section to a subset of the available cores. This is explained in Section 7.9, Locating in a Multi-core Processor Environment.

1.5.1.1. Data Core Association

The term "data core association" (DCA) is used to define:

- whether a data object is accessible from one or multiple cores
- the type of memory where the data will be allocated
- the number of memory instances of the data object

You can use a memory qualifier (__share, __tcm, __tcmn or __clone) or pragma (#pragma data_core_association) to qualify individual data objects, or you can use an option (C compiler option --data-core-association) to specify the default data core association. The default data core association is “share”. This means that when you do not explicitly specify a memory qualifier or a pragma, all data can be accessed by all cores.

<table>
<thead>
<tr>
<th>Data core association</th>
<th>Memory qualifier</th>
<th>Accessible from</th>
<th>Number of instances</th>
<th>Allocation in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Share</td>
<td>__share</td>
<td>All cores</td>
<td>One instance. The data object is shared between cores.</td>
<td>Global RAM</td>
</tr>
<tr>
<td>TCM single-core</td>
<td>__tcm</td>
<td>Specific core</td>
<td>One instance.</td>
<td>DTCM of specific core</td>
</tr>
<tr>
<td>TCM virtual-core</td>
<td>__tcmn</td>
<td>Core n only</td>
<td>One instance. For one specific core: core n.</td>
<td>DTCM of core n</td>
</tr>
<tr>
<td>Clone</td>
<td>__clone</td>
<td>All cores</td>
<td>Multiple instances. For each core one instance is allocated. All instances will be located at identical addresses.</td>
<td>DTCM of all cores</td>
</tr>
</tbody>
</table>

* Note that __tcmn data objects are private for core n from a compiler point of view, where n depends on the number of cores supported by the derivative. For example, for a derivative with two CM7 cores, n...
can be 0 or 1. You can use __tcm without a core number when you want to specify DTCM memory for a single-core project.

Instead of a memory qualifier you can also use a pragma:

```c
#pragma data_core_association share | tcm[n] | clone | default | restore
```

With default you switch back to the default behavior. With restore you restore the previous value of the pragma.

Based on the specified data core association the compiler stores the data object in a section with the following naming convention:

```
section_type_prefix.share|tcmn|clone
```

Note however that when you do not specify a memory qualifier or a pragma or when you use `#pragma data_core_association default`, the data can be accessed by all cores, but the resulting sections do not have .share in the section name. This is the default behavior. The .share is only added to the name if the section was explicitly designated as shared.

Example:

```c
#pragma data_core_association clone
int var_1; // var_1 ends up in section .bss.clone

#pragma data_core_association default
int var_2; // var_2 ends up in section .bss
```

For more information on section names see Section 1.13, Compiler Generated Sections.

The linker recognizes the section names, duplicates clone sections for each binary compatible core and locates core specific code and data in the local memory of each core, resulting in one absolute object file (ELF) for each binary compatible set of cores.

**1.5.1.2. Code Core Association**

The term "code core association" (CCA) is used to define:

- the core or cores that are allowed to execute a function
- the type of memory where the function will be allocated
- the number of instances that are copied to local scratchpad RAM, i.e. the number of entries in the copy table
- a restriction on the type of data (defined by a data core association) the code may access

You can use a memory qualifier (__share, __tcm, __tcmn or __clone) or pragma (#pragma code_core_association) to qualify individual functions, or you can use an option (C compiler option --code-core-association) to change the default code core association. The default code core association is "share". This means that when you do not explicitly specify a memory qualifier or a pragma, all code can be executed by all cores.
<table>
<thead>
<tr>
<th>Code core association</th>
<th>Memory qualifier</th>
<th>Executes on</th>
<th>Number of instances</th>
<th>Allowed access of DCA qualified data</th>
<th>Allocation in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Share</td>
<td>__share</td>
<td>Any core</td>
<td>One instance. The code is shared between cores.</td>
<td>Shared Cloned (of executing core)</td>
<td>Flash or ROM</td>
</tr>
<tr>
<td>TCM single-core</td>
<td>__tcm</td>
<td>Specific core</td>
<td>One instance.</td>
<td>Shared TCM Cloned</td>
<td>ITCM of specific core</td>
</tr>
<tr>
<td>TCM multi-core</td>
<td>__tcmn</td>
<td>Core n only</td>
<td>One instance.</td>
<td>Shared TCM Cloned</td>
<td>ITCM of core n</td>
</tr>
<tr>
<td>Clone</td>
<td>__clone</td>
<td>Any core</td>
<td>Multiple instances. Each code instance is executed by one core.</td>
<td>Shared Cloned</td>
<td>ITCM of all cores</td>
</tr>
</tbody>
</table>

*Note that __tcmn code is private for core n from a compiler point of view, where n depends on the number of cores supported by the derivative. For example, for a derivative with two CM7 cores, n can be 0 or 1. You can use __tcm without a core number when you want to specify ITCM memory for a single-core project.*

Instead of a memory qualifier you can also use a pragma:

```
#pragma code_core_association share | tcm[n] | clone | default | restore
```

With default you switch back to the default behavior. With restore you restore the previous value of the pragma.

Based on the specified code core association the compiler stores the code object in a section with the following naming convention:

```
section_type_prefix.share|tcm[n]|clone
```

Note however that when you do not specify a memory qualifier or a pragma or when you use #pragma code_core_association default, the code can be executed by all cores, but the resulting sections do not have .share in the section name. The .share is only added to the name if the section was explicitly designated as shared.

For more information on section names see Section 1.13, Compiler Generated Sections.

The linker recognizes the section names, duplicates clone sections for each binary compatible core and locates core specific code and data in the local memory of each core, resulting in one absolute object file (ELF) for each binary compatible set of cores.
1.5.1.3. Core Association Restrictions

The following restriction apply to core associations:

**Run-time bounds data is shared by all cores**

For run-time bounds checking, bounds data is generated in a section declared with a fixed section name 'bounds'. No data core association is applied to this section. The linker uses the default core association share. The bounds data is shared among all cores.

Example (bounds.c):

```
typedef struct
{
    int i;
}s_t;

s_t s;          // .bss.share
const int ci1  =55;  // .rodata.share
const int ci2  =55;  // .rodata.share

int main(void)   // .text.share
{
    s.i = 42;

    return ci1+ci2;
}
```

Use the following command to see the result:
```
ccarm bounds.c --runtime --processors="vendor_support_dir\etc\processors.xml"
   --cpu=cyt4bf8cebf_m7 -Wl-I"vendor_support_dir\include.lsl"
   --data-core-association=share --code-core-association=share
```

**Profile data is shared by all cores**

For dynamic profiling, profiling data is generated in sections declared with fixed section names. No data core association is applied to these sections. The linker uses the default core association share. The profiling data is shared among all cores.

Example (prof.c):

```
void f1(int i);
void f2(int i);
void f3(int i);

void f3(int i)
{
    if ( i )
    {
        f1(i-1);
        f2(i-1);
    }
```
```c
void f2(int i)
{
    if ( i )
    {
        f1(i-1);
        f2(i-1);
        f3(i-1);
    }
}

void f1(int i)
{
    if ( i )
    {
        f1(i-1);
        f2(i-1);
        f3(i-1);
    }
}

void main(void)
{
    f1(3);
    f2(3);
    f3(3);
}

When you compile with:

```
ccarm prof.c --profile --processors="vendor_support_dir\etc\processors.xml"
    --cpu=cyt4bf8ceb_m7 -Wl-I"vendor_support_dir\include.lsl"
    --data-core-association=share --code-core-association=share
```

This results in the following section declarations:

```
.section .bss
.align  4
_999001___prof_counter_0:  .type  object
.size   _999001___prof_counter_0,4
.ds     4
.endsec  ; End of section

.section .bss
.align  4
_999002___prof_counter_0:  .type  object
.size   _999002___prof_counter_0,4
.ds     4
```
Predefined identifier __func__ is shared by all cores

You cannot use the data core association symbol qualifiers or pragmas to associate a core with predefined identifier __func__. The linker uses the default core association share. __func__ is shared among all cores.

Example (func.c):

```c
char funcname[10];

void function( void )
{
    for(int i = 0; i < 6; i++) {
        funcname[i] = __func__[i];
    }
}
```

When you compile with:

```bash
ccarm func.c --processors="vendor_support_dir\etc\processors.xml"
 --cpu=cyt4bf8ceb_m7 --Wl-I"vendor_support_dir\include.lsl
 --data-core-association=share --code-core-association=share
```

This results in the following section declaration:

```assembly
.section .rodata
```

No core is associated to a section renamed with the attribute section

No section prefixing is supported on sections that are renamed with attribute section. The linker uses the default core association share. Of course it is still possible to use the core association section naming convention in the section attribute to do the core association manually.
Example (section.c):

```c
int function( void ) __section__("fixed_section_name")
{
  return 0;
}
```

When you compile with:

```
carm section.c --processors="vendor_support_dir\etc\processors.xml"
 --cpu=cyt4bf8ceb_m7 -Wl-I"vendor_support_dir\include.lsl
--data-core-association=share --code-core-association=share
```

This results in the following section declaration:

```assembly
.section  fixed_section_name code
```

Renaming with `#pragma section supports the normal section prefixing.`

```
#pragma section .text=myname

int function( void )
{
  return 0;
}
```

results in the following section declaration:

```assembly
.section  .text.share.myname
```

For more details see Section 1.13, *Compiler Generated Sections*.

### 1.5.1.4. Core Association and Function Calls

The code core association affects caller-callee relations. TCM private functions are not accessible by each core. Therefore, calling a TCM function is illegal unless it is guaranteed that the code that contains the call can only be executed by the core associated with the TCM function.

Both the C compiler and linker check for illegal function calls. However, the C compiler and linker cannot check indirect calls and the C compiler cannot check calls to external functions, due to lack of type information.

The following table shows the relation between function calls and code core associations.

<table>
<thead>
<tr>
<th>Code core association of caller</th>
<th>Code core association of callee</th>
<th>Issues</th>
</tr>
</thead>
<tbody>
<tr>
<td>Share</td>
<td>Share</td>
<td>No issues.</td>
</tr>
<tr>
<td>TCM</td>
<td></td>
<td>No issues.</td>
</tr>
<tr>
<td>Clone</td>
<td></td>
<td>No issues.</td>
</tr>
</tbody>
</table>
1.6. Shift JIS Kanji Support

In order to allow for Japanese character support on non-Japanese systems (like PCs), you can use the Shift JIS Kanji Code standard. This standard combines two successive ASCII characters to represent one Kanji character. A valid Kanji combination is only possible within the following ranges:

- First (high) byte is in the range 0x81-0x9f or 0xe0-0xef.
- Second (low) byte is in the range 0x40-0x7e or 0x80-0xfc

Compiler option `-Ak` enables support for Shift JIS encoded Kanji multi-byte characters in strings and (wide) character constants. Without this option, encodings with 0x5c as the second byte conflict with the use of the backslash (`\`) as an escape character. Shift JIS in comments is supported regardless of this option.

Note that Shift JIS also includes Katakana and Hiragana.

Example:

```c
// Example usage of Shift JIS Kanji
// Do not switch off option -Ak
// At the position of the italic text you can
// put your Shift JIS Kanji code
int i;  // put Shift JIS Kanji here
char c1;
char c2;
unsigned int ui;
const char mes[]="put Shift JIS Kanji here";
const unsigned int ar[5]=('K','a','n','j','i');
    // 5 Japanese array
void main(void)
{
    i=(int)c1;
    i++; /* put Shift JIS Kanji here\  
         continuous comment */
```
1.7. 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.

You need to tell the compiler exactly what happens in the inline assembly code because it uses that for code generation and optimization. The compiler needs to know exactly which registers are written and which registers are only read. For example, if the inline assembly writes to a register from which the compiler assumes that it is only read, the generated code after the inline assembly is based on the fact that the register still contains the same value as before the inline assembly. If that is not the case the results may be unexpected. Also, an inline assembly statement using multiple input parameters may be assigned the same register if the compiler finds that the input parameters contain the same value. As long as this register is only read this is not a problem.

General syntax of the __asm keyword

```
__asm( "instruction_template"
    [ : output_param_list
    [ : input_param_list
    [ : register_reserve_list]] )
```

- `instruction_template` is Assembly instructions that may contain parameters from the input list or output list in the form: `%parm_nr`
- `%parm_nr` is Parameter number in the range 0 .. 9.
- `output_param_list` is `[[ "="[&]constraint_char"(C_expression)]]`,...
- `input_param_list` is `[[ "constraint_char"(C_expression)]]`,...
- `&` says that an output operand is written to before the inputs are read, so this output must not be the same register as any input.
- `constraint_char` is Constraint character: the type of register to be used for the `C_expression`. See the table below.
- `C_expression` is Any C expression. For output parameters it must be an lvalue, that is, something that is legal to have on the left side of an assignment.
- `register_reserve_list` is `[["register_name"]`,...
- `register_name` is Name of the register you want to reserve. For example because this register gets clobbered by the assembly code. The compiler will not use this register for inputs or outputs. Note that reserving too many registers can make register allocation impossible.
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_reserve_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>r</td>
<td>general purpose register</td>
<td>r0 .. r11, lr</td>
<td>Thumb mode r0 .. r7</td>
</tr>
<tr>
<td>number</td>
<td>type of operand it is</td>
<td>same as %number</td>
<td>Input constraint only. The number must</td>
</tr>
<tr>
<td></td>
<td>associated with</td>
<td></td>
<td>refer to an output parameter. Indicates</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>that %number and number are the same</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>register.</td>
</tr>
</tbody>
</table>

If an input parameter is modified by the inline assembly then this input parameter must also be added to the list of output parameters (see Example 6). If this is not the case, the resulting code may behave differently than expected since the compiler assumes that an input parameter is not being changed by the inline assembly.

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. You can use numeric labels for these purposes.

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. Note that you can use standard C escape sequences. 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.

__asm( "nop\n"  
   "\tnop" );

Example 2: using output parameters

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

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

Generated assembly code:
```
mov r0,#0xff
ldr r1,.L2
str r0,[r1,#0]
bx lr
```
```
.L2: .dw out
```

**Example 3: using input parameters**

Assign a variable to a register. A register is chosen for the parameter because of the constraint \( r \); 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 R0,%0"
            : : "r" (in) );
}
```

Generated assembly code:
```
ldr r0,.L2
ldr r0,[r0,#0]
MOV R0,r0
bx lr
```
```
.L2: .dw in
```

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

Add two C variables and assign the result to a third C variable. Registers are used for the input and output parameters (constraint \( r, %0 \) 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 add32( void )
{
    __asm( "add %0, %1, %2"
          : "=r" (out)
          : "r" (in1), "r" (in2) );
}
```

Generated assembly code:

```asm
ldr     r0,.L2
ldr     r1,[r0,#0]
ldr     r0,[r0,#4]
add r0, r1, r0
ldr     r1,.L2
str     r0,[r1,#8]
bx      lr
.size   add32,$-add32
.align  4
.L2:
  .dw     in1
  .section .bss
  .global in1
  .align  4
in1: .type   object
      .size   in1,4
      .ds     4
      .global in2
      .align  4
in2: .type   object
      .size   in2,4
      .ds     4
      .global out
      .align  4
out: .type   object
     .size    out,4
     .ds      4
.endsec
```

**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 \( r0 \) is a reserved register. You can do this by adding a reserved register list (e.g., "r0"). As you can see in the generated assembly code, register \( r0 \) is not used (the first register used is \( r1 \)).

```c
int in1, in2, out;

void add32( void )
{
    __asm( "add %0, %1, %2"
        : "=r" (out)
        : "r" (in1), "r" (in2)
        : "r0" );
}
```

Generated assembly code:

```
ldr     r2,.L2
ldr     r2,[r1,#0]
ldr     r1,[r1,#4]
add r1, r2, r1
ldr     r0,.L2
str     r1,[r0,#8]
bx      lr
.size   add32,$-add32
.align  4
.L2:
    .dw     in1
```

### Example 6: use the same register for input and output

As input constraint you can use a number to refer to an output parameter. This tells the compiler that the same register can be used for the input and output parameter. When the input and output parameter are the same C expression, these will effectively be treated as if the input parameter is also used as output. In that case it is allowed to write to this register. For example:

```c
inline int foo(int par1, int par2, int * par3)
{
    int retvalue;

    __asm(
        "add %2,%1,%1,lsl #2\n\t"
        "mov %5,%2\n\t"
        "mov %0,%2"
        : "=&r" (retvalue), "=r" (par1), "=r" (par2)
        : "1" (par1), "2" (par2), "r" (par3)
    );
    return retvalue;
}
```

```c
int result, parm;
```
void func(void)
{
    result = foo(1000,1000,&parm);
}

In this example the "1" constraint for the input parameter par1 refers to the output parameter par1, and
similar for the "2" constraint and par2. In the inline assembly %1 (par1) and %2 (par2) are written. This
is allowed because the compiler is aware of this.

This results in the following generated assembly code:

    mov  r0,#1000
    mov  r1,r0
    ldr  r2,.L2

    add  r1,r0,r0,lsl #2
    mov  r2,r1
    mov  r3,r1

    1dr  r0,.L2+4
    str  r3,[r0,#0]

    bx   lr

.L2:
   .dw  parm
   .dw  result

However, when the inline assembly would have been as given below, the compiler would have assumed
that %1 (par1) and %2 (par2) were read-only. Because of the inline keyword the compiler knows that
par1 and par2 both contain 1000. Therefore the compiler can optimize and assign the same register to
%1 and %2. This would have given an unexpected result.

```c
__asm(  
    "add  %2,%1,%1,lsl #2\n\t"  
    "mov  %3,%2\n\t"  
    "mov  %0,%2"  
    : "=r" (retvalue)  
    : "r" (par1), "r" (par2), "r" (par3)  
);
```

Generated assembly code:

    ldr  r0,.L2
    mov  r1,#1000

    add  r1,r1,r1,lsl #2    ; same register, but is expected read-only
    mov  r0,r1
    mov  r2,r1

    1dr  r0,.L2+4
    str  r2,[r0,#0]          ; contains unexpected result
1.8. Attributes

You can use the keyword \__attribute\__ to specify special attributes on declarations of variables, functions, types, and fields.

Syntax:

\__attribute__((\(name,\ldots\)))

or:

\_\_\_name\_

The second syntax allows you to use attributes in header files without being concerned about a possible macro of the same name.

**alias("symbol")**

You can use \__attribute__((alias("symbol"))) to specify that the function declaration appears in the object file as an alias for another symbol. For example:

```c
void __f() { /* function body */; }
void f() __attribute__((weak, alias("__f")));
```

declares \f\ to be a weak alias for \f\.

**const**

You can use \__attribute__((\(const\))) to specify that a function has no side effects and will not access global data. This can help the compiler to optimize code. See also attribute \texttt{pure}.

The following kinds of functions should not be declared \__const__:

- A function with pointer arguments which examines the data pointed to.
- A function that calls a non-const function.

**export**

You can use \__attribute__((\(export\))) to specify that a variable/function has external linkage and should not be removed. During MIL linking, the compiler treats external definitions at file scope as if they were declared \texttt{static}. As a result, unused variables/functions will be eliminated, and the alias checking algorithm assumes that objects with static storage cannot be referenced from functions outside the current module. During MIL linking not all uses of a variable/function can be known to the compiler. For example
when a variable is referenced in an assembly file or a (third-party) library. With the `export` attribute the compiler will not perform optimizations that affect the unknown code.

```
int i __attribute__((export)); /* 'i' has external linkage */
```

**flatten**

You can use `__attribute__((flatten))` to force inlining of all function calls in a function, including nested function calls.

Unless inlining is impossible or disabled by `__attribute__((noinline))` for one of the calls, the generated code for the function will not contain any function calls.

**format(type,arg_string_index,arg_check_start)**

You can use `__attribute__((format(type,arg_string_index,arg_check_start)))` to specify that functions take `printf`, `scanf`, `strftime` or `strfmon` style arguments and that calls to these functions must be type-checked against the corresponding format string specification.

*`type`* determines how the format string is interpreted, and should be `printf`, `scanf`, `strftime` or `strfmon`.

*`arg_string_index`* is a constant integral expression that specifies which argument in the declaration of the user function is the format string argument.

*`arg_check_start`* is a constant integral expression that specifies the first argument to check against the format string. If there are no arguments to check against the format string (that is, diagnostics should only be performed on the format string syntax and semantics), `arg_check_start` should have a value of 0. For `strftime`-style formats, `arg_check_start` must be 0.

Example:

```
int foo(int i, const char * my_format, ...) __attribute__((format(printf, 2, 3)));
```

The format string is the second argument of the function `foo` and the arguments to check start with the third argument.

**leaf**

You can use `__attribute__((leaf))` to specify that a function is a leaf function. A leaf function is an external function that does not call a function in the current compilation unit, directly or indirectly. The attribute is intended for library functions to improve dataflow analysis. The attribute has no effect on functions defined within the current compilation unit.

**malloc**

You can use `__attribute__((malloc))` to improve optimization and error checking by telling the compiler that:

- The return value of a call to such a function points to a memory location or can be a null pointer.
• On return of such a call (before the return value is assigned to another variable in the caller), the memory location mentioned above can be referenced only through the function return value; e.g., if the pointer value is saved into another global variable in the call, the function is not qualified for the malloc attribute.

• The lifetime of the memory location returned by such a function is defined as the period of program execution between a) the point at which the call returns and b) the point at which the memory pointer is passed to the corresponding deallocation function. Within the lifetime of the memory object, no other calls to malloc routines should return the address of the same object or any address pointing into that object.

**noinline**

You can use `__attribute__((noinline))` to prevent a function from being considered for inlining. Same as keyword `__noinline` or `#pragma noinline`.

**always_inline**

With `__attribute__((always_inline))` you force the compiler to inline the specified function, regardless of the optimization strategy of the compiler itself. Same as keyword `inline` or `#pragma inline`.

**noreturn**

Some standard C function, such as abort and exit cannot return. The C compiler knows this automatically. You can use `__attribute__((noreturn))` to tell the compiler that a function never returns. For example:

```c
void fatal() __attribute__((noreturn));
void fatal( /* ... */ )
{
    /* Print error message */
    exit(1);
}
```

The function `fatal` cannot return. The compiler can optimize without regard to what would happen if `fatal` ever did return. This can produce slightly better code and it helps to avoid warnings of uninitialized variables.

**overloadable**

You can use `__attribute__((overloadable))` to define multiple functions with the same name, but with different prototypes. This provides a limited form of function overloading. Function overloading is restricted to direct calls.

It is not possible to have both a normal and an overloadable function of the same name. In that case, the normal function takes precedence. The overloadable attribute is ignored for functions without a prototype.
When calling a function for which only overloadable definitions are visible, the function with the best match is selected. The best match is the function with the correct number of arguments, requiring the least amount of argument conversions. When there are no matches, or when there are multiple ambiguous matches, an error is generated.

**protect**

You can use `__attribute__((protect))` to exclude a variable/function from the duplicate/unreferenced section removal optimization in the linker. When you use this attribute, the compiler will add the "protect" section attribute to the symbol's section. Example:

```c
int i __attribute__((protect));
```

Note that the protect attribute will not prevent the compiler from removing an unused variable/function (see the used symbol attribute).

This attribute is the same as `#pragma protect/endprotect`.

**pure**

You can use `__attribute__((pure))` to specify that a function has no side effects, although it may read global data. Such pure functions can be subject to common subexpression elimination and loop optimization. See also attribute `const`.

**section("section_name")**

You can use `__attribute__((section("name")))` to specify that a function must appear in the object file in a particular section. For example:

```c
extern void foobar(void) __attribute__((section("bar")));
```

puts the function foobar in the section named bar.

See also `#pragma section`.

**used**

You can use `__attribute__((used))` to prevent an unused symbol from being removed, by both the compiler and the linker. Example:

```c
static const char copyright[] __attribute__((used)) = "Copyright 2021 TASKING BV";
```

When there is no C code referring to the copyright variable, the compiler will normally remove it. The `__attribute__((used))` symbol attribute prevents this. Because the linker should also not remove this symbol, `__attribute__((used))` implies `__attribute__((protect))`.

**unused**

You can use `__attribute__((unused))` to specify that a variable or function is possibly unused. The compiler will not issue warning messages about unused variables or functions.
weak

You can use `__attribute__((weak))` to specify that the symbol resulting from the function declaration or variable must appear in the object file as a weak symbol, rather than a global one. This is primarily useful when you are writing library functions which can be overwritten in user code without causing duplicate name errors.

See also `#pragma weak`.

1.9. Pragmas to Control the Compiler

Pragmas are keywords in the C source that control the behavior of the compiler. Pragmas overrule compiler options. Put pragmas in your C source where you want them to take effect. Unless stated otherwise, a pragma is in effect from the point where it is included to the end of the compilation unit or until another pragma changes its status.

The syntax is:

```c
#pragma [label:]pragma-spec pragma-arguments [on | off | default | restore]
```
or:

```c
_Pragma( "[label:]pragma-spec pragma-arguments [on | off | default | restore]" )
```

Some pragmas can accept the following special arguments:

- **on** switch the flag on (same as without argument)
- **off** switch the flag off
- **default** set the pragma to the initial value
- **restore** restore the previous value of the pragma

Examples:

```c
// by default all warnings are shown

#pragma warning 535 // disable W535
#pragma warning 530 // also disable W530
const char var_1 = 0x5678; // W530 is not shown
var_2; // W535 is not shown
#pragma warning restore // restore one level, only W535 is disabled
const char var_3 = 0x56789; // W530 is shown
#pragma warning default // back to default, all warnings are shown
var_4; // W535 is shown
```

Label pragmas

Some pragmas support a label prefix of the form "label:" between `#pragma` and the pragma name. Such a label prefix limits the effect of the pragma to the statement following a label with the specified name.
The `restore` argument on a pragma with a label prefix has a special meaning: it removes the most recent definition of the pragma for that label.

You can see a label pragma as a kind of macro mechanism that inserts a pragma in front of the statement after the label, and that adds a corresponding `#pragma ... restore` after the statement.

Compared to regular pragmas, label pragmas offer the following advantages:

- The pragma text does not clutter the code, it can be defined anywhere before a function, or even in a header file. So, the pragma setting and the source code are uncoupled. When you use different header files, you can experiment with a different set of pragmas without altering the source code.

- The pragma has an implicit end: the end of the statement (can be a loop) or block. So, no need for `pragma restore / endoptimize` etc.

Example:

```c
#pragma lab1:optimize P

volatile int v;

void f( void )
{
    int i, a;
    a = 42;

    lab1: for( i=1; i<10; i++ )
    {
        /* the entire for loop is part of the pragma optimize */
        a += i;
    }
    v = a;
}
```

**Supported pragmas**

The compiler recognizes the following pragmas, other pragmas are ignored. Pragmas marked with (*) support a label prefix.

**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).

**boolean [on | off | default | restore] (*)**

This pragma is used to mark the macros "false" and "true" from the library header file stdbool.h as "essentially BOOLEAN", which is a concept from the MISRA C:2012 standard.
call {near | far | default | restore} (*)

By default, functions are called with 26-bit PC-relative calls. This *near call is directly coded into the instruction, resulting in higher execution speed and smaller code size. The destination address of a near call must be located within +/-32 MB from the program counter.

The other call mode is a 32-bit indirect call. With *far calls you can address the full range of memory. The address is first loaded into a register after which the call is executed. Instead of #pragma call far you can use the function qualifier _far on a specific function.

See C compiler option --call (-m).

code_core_association {value | default | restore} (*)

Switch to another code core association, where *value is one of share, tcmn (for core n) or clone. The code core association of this pragma is assigned to the functions declarations or definitions that follow.

This pragma is only supported for multi-core architectures.

See C compiler option --code-core-association.

compactmaxmatch {value | default | restore} (*)

With this pragma you can control the maximum size of a match.

See C compiler option --compact-max-size.

data_core_association {value | default | restore} (*)

Switch to another data core association, where *value is one of share, tcmn (for core n) or clone. The data core association of this pragma is assigned to the data declarations that follow.

See C compiler option --data-core-association.

This pragma is only supported for multi-core architectures.

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 (\texttt{.EXTERN} assembler directive), even when the symbol is not used in the module.

\begin{verbatim}
fp_negzero [on | off | default | restore] (*)
\end{verbatim}

With this pragma you can control the \texttt{+negzero} flag of C compiler option \texttt{--fp-model}.

\begin{verbatim}
fp_rewrite [on | off | default | restore] (*)
\end{verbatim}

With this pragma you can control the \texttt{+rewrite} flag of C compiler option \texttt{--fp-model}.

\begin{verbatim}
inline / noninline / smartinline [default | restore] (*)
\end{verbatim}

See Section 1.12.2, \textit{Inlining Functions: inline}.

\begin{verbatim}
inline_max_incr / inline_max_size \{value | default | restore\} (*)
\end{verbatim}

With these pragmas you can control the automatic function inlining optimization process of the compiler. It has only effect when you have enabled the inlining optimization (\texttt{--optimize=+inline (-Oi)}).

See C compiler options \texttt{--inline-max-incr} and \texttt{--inline-max-size}.

\begin{verbatim}
macro / nomacro [on | off | default | restore] (*)
\end{verbatim}

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

\begin{verbatim}
maxcalldepth \{value | default | restore\} (*)
\end{verbatim}

With this pragma you can control the maximum call depth. Default is infinite (-1).

See C compiler option \texttt{--max-call-depth}.

\begin{verbatim}
messag\textquoteleft \textquoteleft message\textquoteright \textldots
\end{verbatim}

Print the message string(s) on standard output.

\begin{verbatim}
nomisrac \{nr,...\} [default | restore] (*)
\end{verbatim}

Without arguments, this pragma disables MISRA C checking. Alternatively, you can specify a comma-separated list of MISRA C rules to disable.

See C compiler option \texttt{--misrac} and Section 4.6.2, \textit{C Code Checking: MISRA C}.

\begin{verbatim}
optimize \{flags | default | restore\} (*) / endoptimize
\end{verbatim}

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

See Section 4.5, \textit{Compiler Optimizations}. 
profile [flags | default | restore] (*) / endprofile

Control the profile settings. The pragma works the same as C compiler option --profile. Note that this pragma will only be checked at the start of a function. endprofile switches back to the previous profiling settings.

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 [type=]{suffix |-f|-m|-fm} [default | restore] (*) / endsection

Rename sections by adding a suffix to all section types specified with type, or restore default section naming. If you specify only a suffix (without a name), the suffix is added to all section names. See Section 1.13.1, Rename Sections, C compiler option --rename-sections and assembler directive .SECTION for more information.

section_code_init [on | off | default | restore] (*) / section_no_code_init

Copy or do not copy code sections from ROM to RAM at application startup.

section_const_init [on | off | default | restore] (*) / section_no_const_init

Copy or do not copy read-only data sections from ROM to RAM at application startup.

silicon_bug [bug,...] [default | restore] (*)

Without arguments, all silicon bug workarounds are enabled. Alternatively, you can specify a comma-separated list of silicon bug workarounds.

See C compiler option --silicon-bug and Chapter 19, CPU Problem Bypasses and Checks.
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.

linear_switch / jump_switch / binary_switch / smart_switch / tbb_switch / tbh_switch / no_tbh_switch

With these pragmas you can overrule the compiler chosen switch method:

linear_switch Force jump chain code. A jump chain is comparable with an if/else-if/else-if/else construction.

jump_switch Force jump table code. A jump table is a table filled with jump instructions for each possible switch value. The switch argument is used as an index to jump within this table.

binary_switch Force binary lookup table code. A binary search table is a table filled with a value to compare the switch argument with and a target address to jump to.

smart_switch Let the compiler decide the switch method used.

tbb_switch Force use of the tbb instruction. Uses a table of 8-bit jump offsets.

tbh_switch Force use of the tbh instruction. Uses a table of 8-bit jump offsets.

no_tbh_switch Same as smart_switch, but do not use the tbh instruction.

See Section 1.11, Switch Statement.

tradeoff {level | default | restore} (*)

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

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.

## 1.10. 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>ARM</strong></td>
<td>Expands to 1 for the ARM toolset, otherwise unrecognized as macro.</td>
</tr>
<tr>
<td><strong>BIG_ENDIAN</strong></td>
<td>Expands to 1 if big-endian mode is selected (option --endianness=big), otherwise unrecognized as macro.</td>
</tr>
<tr>
<td><strong>BUILD</strong></td>
<td>Identifies the build number of the compiler in the format yymmdqq (year, month, day and quarter in UTC). For example, January 22 2021 14:13 UTC corresponds to 21012257.</td>
</tr>
<tr>
<td><strong>CARM</strong></td>
<td>Expands to 1 for the ARM toolset, otherwise unrecognized as macro.</td>
</tr>
<tr>
<td><strong>CPU</strong></td>
<td>Expands to the ARM architecture name (option --cpu=arch). For example, if --cpu=ARMv7M is specified, the symbol <strong>CPU</strong> expands to ARMv7M. This is also the default when no --cpu is supplied.</td>
</tr>
<tr>
<td><strong>CPU_arch</strong></td>
<td>A symbol is defined depending on the option --cpu=arch. The arch is converted to uppercase. For example, if --cpu=ARMv7M is specified, the symbol <strong>CPU_ARCHV7M</strong> is defined. This is also the default when no --cpu is supplied.</td>
</tr>
<tr>
<td><strong>DATE</strong></td>
<td>Expands to the compilation date: “mmm dd yyyy”.</td>
</tr>
<tr>
<td><strong>DOUBLE_FP</strong></td>
<td>Expands to 1 if you used option --fp-model=-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_fpu</strong></td>
<td>A symbol is defined depending on the option --fpu=fpu. The fpu is converted to uppercase and the lowercase “v” and the ‘-’ will be removed. For example, if --fpu=VFPv3-sp is specified, the symbol <strong>FPU_VFP3SP</strong> is defined. When no --fpu is supplied, the symbol <strong>FPU_NONE</strong> is the default.</td>
</tr>
<tr>
<td><strong>FPU_VFP</strong></td>
<td>Expands to 1 if one the options --fpu=fpu is specified and fpu is not NONE.</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>LITTLE_ENDIAN</strong></td>
<td>Expands to 1 if little-endian mode is selected (option --endianness=little), otherwise unrecognized as macro. This is the default.</td>
</tr>
<tr>
<td><strong>PROF_ENABLE</strong></td>
<td>Expands to 1 if profiling is enabled, otherwise expands to 0.</td>
</tr>
</tbody>
</table>
### Macro Description

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>
<tr>
<td><strong>SINGLE_FP</strong></td>
<td>Expands to 1 if you used option --fp-model=+float (Treat ‘double’ as ‘float’), otherwise unrecognized as macro.</td>
</tr>
<tr>
<td><strong>STDC</strong></td>
<td>Identifies the level of ANSI standard. The macro expands to 1 if you set option --language (Control language extensions), otherwise expands to 0.</td>
</tr>
<tr>
<td><strong>STDC_HOSTED</strong></td>
<td>Always expands to 0, indicating the implementation is not a hosted implementation.</td>
</tr>
<tr>
<td><strong>STDC_NO_ATOMICS</strong></td>
<td>(C11 only) This macro is undefined for ARMv7, indicating that this implementation supports atomic types and the stdatomic.h header file.</td>
</tr>
<tr>
<td><strong>STDC_NO_THREADS</strong></td>
<td>(C11 only) Expands to 1 to indicate that this implementation does not support the threads.h header file.</td>
</tr>
<tr>
<td><strong>STDC_VERSION</strong></td>
<td>Identifies the ISO-C version number. Expands to 201112L for ISO C11, 199901L for ISO C99 or 199409L for ISO C90.</td>
</tr>
<tr>
<td><strong>TASKING</strong></td>
<td>Identifies the compiler as a TASKING compiler. Expands to 1 if a TASKING compiler is used.</td>
</tr>
<tr>
<td><strong>THUMB</strong></td>
<td>Expands to 1 if you used option --thumb, otherwise unrecognized as macro.</td>
</tr>
<tr>
<td><strong>TIME</strong></td>
<td>Expands to the compilation time: “hh:mm:ss”</td>
</tr>
<tr>
<td><strong>VERSION</strong></td>
<td>Identifies the version number of the compiler. For example, if you use version 3.0r1 of the compiler, <strong>VERSION</strong> expands to 3000 (dot and revision number are omitted, minor version number in 3 digits).</td>
</tr>
</tbody>
</table>

### Example

```c
#ifdef __CARM__
/* this part is only compiled for the ARM */
...
#endif
```

### 1.11. 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 binary search table.

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

#pragma smart_switch is the default of the compiler. 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 put more emphasis on speed than on memory size.
For a switch with a long type argument, only linear code is used.

For an int type argument, a jump table switch is only used when the table of cases is not too sparse.

Especially for large switch statements, the jump table approach executes faster than the binary search 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.

For ARMv7M a switch using the \texttt{tbh} instruction gets priority over a normal switch table implementation.

### How to overrule the default switch method

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

```c
#pragma linear_switch       force jump chain code
#pragma jump_switch         force jump table code
#pragma binary_switch       force binary search table code
#pragma smart_switch        let the compiler decide the switch method used
#pragma tbb_switch          force use of \texttt{tbb} instruction (uses a table of 8-bit jump offsets)
#pragma tbh_switch          force use of \texttt{tbh} instruction (uses a table of 16-bit jump offsets)
#pragma no_tbh_switch       same as \texttt{smart_switch}, but do not use \texttt{tbh} instruction
```

Using a pragma cannot overrule the restrictions as described earlier.

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

Example:

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

#pragma jump_switch

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

1.12.1. 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 r0, r1, r2 and r3.

<table>
<thead>
<tr>
<th>Parameter type</th>
<th>Registers used for parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>_Bool, char, short, int, long, float, 32-bit pointer, 32-bit struct</td>
<td>r0, r1, r2, r3</td>
</tr>
<tr>
<td>long long, double, 64-bit struct</td>
<td>r0r1, r1r2, r2r3</td>
</tr>
</tbody>
</table>

The parameters are processed from left to right. The first not used and fitting register is used. Registers are searched for in the order listed above. When a parameter is > 64 bit, or when all registers are used, parameter passing continues on the stack. The stack grows from higher towards lower addresses. The first parameter is pushed at the lowest stack address. The alignment on the stack depends on the data type as listed in Section 1.1, Data Types, with the exception that objects with a size >= 64 bits have a 32-bit alignment on the stack.

Examples:

```c
void func1( int a, char * b, char c );    /* r0 r1 r2 */
void func2( long long d, char e );         /* r0r1 r2 */
void func3( double f, long long g, char h );
    /* r0r1 r2r3 stack */
```

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>_Bool, char, short, int, long, float, 32-bit pointer, 32-bit struct</td>
<td>r0</td>
</tr>
<tr>
<td>long long, double, 64-bit struct</td>
<td>r0r1</td>
</tr>
</tbody>
</table>

Objects larger than 64 bits are returned via the stack.

**FPU calling convention: __fpu**

The default calling convention only uses normal registers, as described above. With the function qualifier __fpu, the FPU calling convention is used, which uses FPU registers for parameters and return values if their type is float/double. The FPU calling convention is only allowed when FPU usage is enabled with option --fpu=fpu
The following types are affected for the FPU calling convention:

<table>
<thead>
<tr>
<th>Type</th>
<th>Registers for parameters</th>
<th>Register for function return</th>
<th>FPU has double-precision FP operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>_Float16</td>
<td>s0, s1, ...</td>
<td>s0</td>
<td>N/A</td>
</tr>
<tr>
<td>float, _Imaginary float</td>
<td>s0, s1, ...</td>
<td>s0</td>
<td>N/A</td>
</tr>
<tr>
<td>double, _Imaginary double</td>
<td>d0, d1, ...</td>
<td>d0</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>s0/s1, s2/s3, ...</td>
<td>s0/s1</td>
<td>no</td>
</tr>
<tr>
<td>_Complex float</td>
<td>s0/s1, s2/s3, ...</td>
<td>s0/s1</td>
<td>N/A</td>
</tr>
<tr>
<td>_Complex double</td>
<td>d0/d1, d2/d3, ...</td>
<td>d0/d1</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>s0/s1/s2/s3, ...</td>
<td>s0/s1/s2/s3</td>
<td>no</td>
</tr>
</tbody>
</table>

Examples (compiled with --fpu=VFPv4):

```c
void __fpu func4( float i, double j, char k ); /* s0, d1, r0 */
void __fpu func5( double l, float m, _Complex double n ); /* d0, s2, d2/d3 */
```

**Function calling modes: __far**

Functions are by default called with 26-bit PC-relative calls. This near call is directly coded into the instruction, resulting in higher execution speed and smaller code size. The destination address of a near call must be located within +/-32 MB from the program counter. However, when you link the application and the target address appears to be out of reach, the linker generates an error. In this case you can use the __far function qualifier to force a less efficient 32-bit indirect call to the function. With far calls you can address the full range of memory. The address is first loaded into a register after which the call is executed.

```c
extern __far void foo( void );
```

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

**1.12.2. 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;
    if (val < 0) abs_val = -val;
    return abs_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;
}
#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 `-Oinline`.

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 1.12.4.3, Writing Your Own Intrinsic Function.
### 1.12.3. Interrupt Functions / Exception Handlers

The TASKING C compiler supports a number of function qualifiers and keywords to program exception handlers. An *exception handler* (or: interrupt function) is called when an exception occurs.

The exception model for the ARMv6-M and ARMv7-M (M-profile architectures) is described in detail in the corresponding *Architecture Reference Manual*. It also contains a list of the different exception types.

When an exception occurs, execution is forced from a fixed memory address corresponding to the type of exception. These fixed addresses are called the exception vectors. A table of vectors is used to indicate the exception entry points.

#### 1.12.3.1. Defining an Exception Handler: __interrupt Keywords

You can define an exception handler with the function qualifier `__interrupt(n)` for single-core projects or `__interruptc(n)` for virtual-core projects. Only one of these qualifiers is allowed per function. The parameter `n` designates the corresponding index in the vector table. The suffix `c` is a number (0 or 1) that associates the interrupt handler with the processor core `c`.

If a function with the `__interrupt` qualifier is also designated as "tcmc" by any of the code core association methods (see Section 1.5.1.2, *Code Core Association*), then the C compiler interprets that function as having the `__interruptc` qualifier instead.

Interrupt functions and other exception handlers cannot return anything and must have a `void` argument type list:

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

void __interruptc(n) isr2( void )
{
...
}
```

**Example**

```c
void __interrupt( 16 ) myisr( void )
{
...
}
```

**Vector table**

The vector table contains the initialization value for the stack pointer, and the entry point addresses of each exception handler. The exception number (vector number) also defines the order of entries in the vector table.
### Vector number

<table>
<thead>
<tr>
<th>Vector number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Initialization value of the stack pointer</td>
</tr>
<tr>
<td>1</td>
<td>Reset handler</td>
</tr>
<tr>
<td>( n )</td>
<td>Exception using that exception number ( n ). Numbers 16 and higher are for user interrupts.</td>
</tr>
</tbody>
</table>

### Vector symbols

When you use one or more of the `__interrupt(n)` or `__interruptc(n)` function qualifiers, the compiler generates a corresponding vector symbol to designate the start of an exception handler function. The linker uses this symbol to automatically generate the exception vector.

<table>
<thead>
<tr>
<th>Function type qualifier</th>
<th>Vector symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__interrupt(n)</code></td>
<td><code>_vector_n</code></td>
</tr>
<tr>
<td><code>__interruptc(n)</code></td>
<td><code>_vectorc_n</code></td>
</tr>
</tbody>
</table>

Note that the reset handler is designated by the symbol `Reset_Handler` instead of `_vector_1`.

### Function qualifier `__novector`

You can prevent the compiler from generating the `_vector_n` symbol by specifying the function qualifier `__novector`. This can be necessary if you have more than one interrupt handler for the same exception. Without the `__novector` function qualifier the compiler generates the `_vector_n` symbol multiple times, which results in a link error.

```c
void __interrupt(6) __novector another_handler( void )
{
    ... // used __novector to prevent multiple _vector_6 symbols
}
```

#### 1.12.3.2. Interrupt Frame: `__frame()`

With the function type qualifier `__frame()` you can specify which registers and SFRs must be saved for a particular interrupt function. Only the specified registers will be pushed and popped from the stack. If you do not specify the function qualifier `__frame()`, the C compiler determines which registers must be pushed and popped. The syntax is:

```c
void __interrupt(n)
    __frame(reg[, reg]...) isr( void )
{
    ...
}
```

where, `reg` can be any register defined as an SFR. The compiler generates a warning if some registers are missing which are normally required to be pushed and popped in an interrupt function prolog and epilog to avoid run-time problems.
Example

__interrupt(2) __frame(R4,R5,R6) void alarm( void )
{
  ... 
}

1.12.4. 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 (__).

On the command line you can see a list of intrinsic functions with the following command:

carm --help=intrinsics

The TASKING ARM C compiler recognizes the following intrinsic functions:

**Program address**

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void volatile __remap_pc( void );</td>
<td>Load the 'real' program address. This intrinsic is used in the startup code to assure that the reset handler is immune for any ROM/RAM remapping.</td>
</tr>
</tbody>
</table>

**Stack pointer**

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void volatile __setsp( __data void * stack );</td>
<td>Initialize the stack pointer with 'stack'.</td>
</tr>
</tbody>
</table>

**SPSR status register**

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned int volatile __getspsr( void );</td>
<td>Get the value of the SPSR status register. Returns the value of the status register SPSR.</td>
</tr>
</tbody>
</table>
Intrinsic Function | Description
--- | ---
unsigned int volatile __setspsr(int set, int clear); | Set or clear bits in the SPSR status register. Returns the new value of the SPSR status register.

Example:

```c
#define SR_F 0x00000040
#define SR_I 0x00000080

i = __setspsr(0, SR_F | SR_I);

if (i & (SR_F | SR_I))
{
    exit (6); /* Interrupt flags not correct */
}

if (__getspsr () & (SR_F | SR_I))
{
    exit (7); /* Interrupt flags not correct */
}
```

CPSR status register

Intrinsic Function | Description
--- | ---
unsigned int volatile __getcpsr(void); | Get the value of the CPSR status register. Returns the value of the status register CPSR.

unsigned int volatile __setcpsr(int set, int clear); | Set or clear bits in the CPSR status register. Returns the new value of the CPSR status register.

APSR status register

Intrinsic Function | Description
--- | ---
unsigned int volatile __getapsr(void); | Get the value of the APSR status register (ARMv6-M and ARMv7-M). Returns the value of the status register APSR.

unsigned int volatile __setapsr(int set, int clear); | Set or clear bits in the APSR status register (ARMv6-M and ARMv7-M). Returns the new value of the APSR status register.

IPSR status register

Intrinsic Function | Description
--- | ---
unsigned int volatile __getipsr(void); | Get the value of the IPSR status register (ARMv6-M and ARMv7-M). Returns the value of the status register IPSR.
## Supervisor call

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void volatile __svc(int number);</code></td>
<td>Generate a supervisor call (software interrupt). Number must be a constant value.</td>
</tr>
</tbody>
</table>

## Miscellaneous intrinsics

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void * volatile __alloc(__size_t size);</code></td>
<td>Allocate memory. Returns a pointer to space of <code>size</code> bytes on the stack of the calling function. Memory allocated through this function is freed when the calling function returns. This function is used internally for variable length arrays, it is not to be used by end users.</td>
</tr>
<tr>
<td><code>void volatile __free(void * p);</code></td>
<td>Deallocate the memory pointed to by <code>p</code>. <code>p</code> must point to memory earlier allocated by a call to <code>__alloc()</code>.</td>
</tr>
<tr>
<td><code>void __nop(void);</code></td>
<td>Generate a NOP instruction.</td>
</tr>
<tr>
<td><code>__codeptr volatile __get_return_address(void);</code></td>
<td>Used by the compiler for profiling when you compile with the option <code>--profile</code>. Returns the return address of a function.</td>
</tr>
</tbody>
</table>

## 1.12.4.1. Atomic Intrinsic Functions

### Initialization

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>volatile void __c11_atomic_init(volatile _Atomic type * obj, type value)</code></td>
<td>Initializes the atomic object pointed to by <code>obj</code> to the value <code>value</code>.</td>
</tr>
</tbody>
</table>

### Fences

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>volatile void __c11_atomic_thread_fence(memory_order order)</code></td>
<td>Depending on the memory order, this operation has either no effects, is an acquire fence, a release fence, both acquire and release fence, or sequentially consistent acquire and release fence.</td>
</tr>
<tr>
<td><code>volatile void __c11_atomic_signal_fence(memory_order order)</code></td>
<td>Equivalent to <code>__c11_atomic_thread_fence</code>, except that the resulting ordering constraints are established only between a thread and a signal handler executed in the same thread.</td>
</tr>
</tbody>
</table>

### Lock-free property

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>volatile _Bool __c11_atomic_is_lock_free(volatile _Atomic type * obj)</code></td>
<td>Indicates whether or not the object pointed to by <code>obj</code> is lock-free.</td>
</tr>
</tbody>
</table>
### Operations on atomic types

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>volatile void __c11_atomic_store(volatile _Atomic type * obj, type desired, memory_order order)</td>
<td>Atomically replaces the value pointed to by <code>obj</code> with the value of <code>desired</code>. Memory is affected according to the value of <code>order</code>.</td>
</tr>
<tr>
<td>volatile type __c11_atomic_load(volatile _Atomic type * obj, memory_order order)</td>
<td>Atomically returns the value pointed to by <code>obj</code>.</td>
</tr>
<tr>
<td>volatile type __c11_atomic_exchange(volatile _Atomic type * obj, type desired, memory_order order)</td>
<td>Atomically replaces the value pointed to by <code>obj</code> with the value of <code>desired</code>. Memory is affected according to the value of <code>order</code>. These operations are atomic read-modify-write operations.</td>
</tr>
<tr>
<td>volatile __Bool __c11_atomic_compare_exchange_strong(volatile _Atomic type * obj, type * expected, type desired, memory_order success, memory_order failure)</td>
<td>Atomically compares the value pointed to by <code>obj</code> for equality with that in <code>expected</code>. If the comparison is true, it replaces the value pointed to by <code>obj</code> with <code>desired</code>, and memory is affected according to the value of <code>success</code>. If the comparison is false, it updates the value in <code>expected</code> with the value pointed to by <code>obj</code>, and memory is affected according to the value of <code>failure</code>. These operations are atomic read-modify-write operations.</td>
</tr>
<tr>
<td>volatile __Bool __c11_atomic_compare_exchange_weak(volatile _Atomic type * obj, type * expected, type desired, memory_order success, memory_order failure)</td>
<td>Atomically replaces the value pointed to by <code>obj</code> with the result of the computation applied to the value pointed to by <code>obj</code> and the given operand. Memory is affected according to the value of <code>order</code>. The key values are: add (+, addition) sub (-, subtraction) or (</td>
</tr>
<tr>
<td>volatile type __c11_atomic_fetch_key(volatile _Atomic type * obj, type operand, memory_order order)</td>
<td></td>
</tr>
</tbody>
</table>

### Atomic flag type and operations

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>volatile __Bool __c11_atomic_flag_test_and_set(volatile atomic_flag * obj, memory_order order)</td>
<td>Atomically sets the value pointed to by <code>obj</code> to true. Memory is affected according to the value of <code>order</code>.</td>
</tr>
<tr>
<td>volatile void __c11_atomic_flag_clear(volatile atomic_flag * obj, memory_order order)</td>
<td>Atomically sets the value pointed to by <code>obj</code> to false. Memory is affected according to the value of <code>order</code>.</td>
</tr>
</tbody>
</table>

### 1.12.4.2. CMSIS Intrinsics

The TASKING VX-toolset for ARM supports the Cortex Micro-controller Software Interface Standard (CMSIS). You can find details about this standard on www.arm.com.
The required functions as defined in the CMSIS are supported by the compiler as intrinsic functions and do not have any implementation in the CMSIS-CORE files. The implemented intrinsic functions are:

### CMSIS core register access

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void volatile __enable_irq( void );</td>
<td>Global Interrupt enable (using the instruction CPSIE i).</td>
</tr>
<tr>
<td>void volatile __disable_irq( void );</td>
<td>Global Interrupt disable (using the instruction CPSID i).</td>
</tr>
<tr>
<td>void volatile __set_PRIMASK( unsigned int value );</td>
<td>Assign value to Priority Mask Register (using the instruction MSR).</td>
</tr>
<tr>
<td>unsigned int __get_PRIMASK( void );</td>
<td>Return Priority Mask Register (using the instruction MRS).</td>
</tr>
<tr>
<td>void volatile __enable_fault_irq( void );</td>
<td>Global Fault exception and Interrupt enable (using the instruction CPSIE f).</td>
</tr>
<tr>
<td>void volatile __disable_fault_irq( void );</td>
<td>Global Fault exception and Interrupt disable (using the instruction CPSID f).</td>
</tr>
<tr>
<td>void volatile __set_FAULTMASK( unsigned int value );</td>
<td>Assign value to Fault Mask Register (using the instruction MSR).</td>
</tr>
<tr>
<td>unsigned int __get_FAULTMASK( void );</td>
<td>Return Fault Mask Register (using the instruction MRS).</td>
</tr>
<tr>
<td>void volatile __set_BASEPRI( unsigned int value );</td>
<td>Set Base Priority (using the instruction MSR).</td>
</tr>
<tr>
<td>unsigned int __get_BASEPRI( void );</td>
<td>Return Base Priority (using the instruction MRS).</td>
</tr>
<tr>
<td>void volatile __set_CONTROL( unsigned int value );</td>
<td>Set CONTROL register value (using the instruction MSR).</td>
</tr>
<tr>
<td>unsigned int __get_CONTROL( void );</td>
<td>Return Control Register Value (using the instruction MRS).</td>
</tr>
<tr>
<td>void volatile __set_PSP( unsigned int value );</td>
<td>Set Process Stack Pointer value (using the instruction MSR).</td>
</tr>
<tr>
<td>unsigned int __get_PSP( void );</td>
<td>Return Process Stack Pointer (using the instruction MRS).</td>
</tr>
<tr>
<td>void volatile __set_MSP( unsigned int value );</td>
<td>Set Main Stack Pointer (using the instruction MSR).</td>
</tr>
<tr>
<td>unsigned int __get_MSP( void );</td>
<td>Return Main Stack Pointer (using the instruction MRS).</td>
</tr>
</tbody>
</table>

### CMSIS core instruction access

<table>
<thead>
<tr>
<th>Intrinsic Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void volatile __NOP( void );</td>
<td>No Operation. This instruction does nothing. You can use this instruction for code alignment purposes.</td>
</tr>
<tr>
<td>void volatile __WFI( void );</td>
<td>Wait for Interrupt. Suspends execution until one of a number of events occurs.</td>
</tr>
<tr>
<td>void volatile __WFE( void );</td>
<td>Wait for Event. Permits the processor to enter a low-power state until one of a number of events occurs.</td>
</tr>
<tr>
<td>void volatile __SEV( void );</td>
<td>Set Event. Causes an event to be signaled to the CPU.</td>
</tr>
<tr>
<td>Intrinsic Function</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>void volatile __YIELD(void);</td>
<td>Generates a YIELD hint instruction. This enables multithreading software to indicate to the hardware that it is performing a task, for example a spin-lock, that could be swapped out to improve overall system performance.</td>
</tr>
<tr>
<td>void volatile __DBG(/<em>constant</em>/ unsigned int opt);</td>
<td>Generates a DBG instruction. This provides a hint to debugging and related systems. The argument must be a constant integer from 0 to 15 inclusive. Only available for ARMv7 (compiler options --cpu=ARMv7M, --cpu=ARMv7EM).</td>
</tr>
<tr>
<td>void volatile __ISB( void );</td>
<td>Instruction Synchronization Barrier. Flushes the pipeline in the processor, so that all instructions following the ISB are fetched from cache or memory, after the instruction has been completed.</td>
</tr>
<tr>
<td>void volatile __DSB( void );</td>
<td>Data Synchronization Barrier. This function acts as a special kind of Data Memory Barrier. It completes when all explicit memory accesses before this instruction have been completed.</td>
</tr>
<tr>
<td>void volatile __DMB( void );</td>
<td>Data Memory Barrier. This function ensures the apparent order of the explicit memory operations before and after the instruction, without ensuring their completion.</td>
</tr>
<tr>
<td>unsigned int __REV( unsigned int value );</td>
<td>Reverse byte order in integer value.</td>
</tr>
<tr>
<td>unsigned int __REV16( unsigned short value );</td>
<td>Reverse byte order in unsigned short value.</td>
</tr>
<tr>
<td>signed int __REVSH( signed int value );</td>
<td>Reverse byte order in signed short value with sign extension to integer.</td>
</tr>
<tr>
<td>unsigned int __RBIT( unsigned int value );</td>
<td>Reverse bit order of value.</td>
</tr>
<tr>
<td>unsigned int __ROR( unsigned int op, unsigned int shift );</td>
<td>Rotate Right in unsigned value (32 bit). This function Rotate Right (immediate) provides the value of the contents of a register rotated by a variable number of bits.</td>
</tr>
<tr>
<td>unsigned volatile char __LDREXB( unsigned char * addr );</td>
<td>Load exclusive byte. This function performs an exclusive LDR command for 8-bit values.</td>
</tr>
<tr>
<td>unsigned volatile short __LDREXH( unsigned short * addr );</td>
<td>Load exclusive half-word. This function performs an exclusive LDR command for 16-bit values.</td>
</tr>
<tr>
<td>unsigned int volatile __LDREXW( unsigned int * addr );</td>
<td>Load exclusive word. This function performs an exclusive LDR command for 32-bit values.</td>
</tr>
<tr>
<td>unsigned int volatile __STREXB( unsigned char value, unsigned char * addr );</td>
<td>Store exclusive byte. This function performs an exclusive STR command for 8-bit values.</td>
</tr>
<tr>
<td>unsigned int volatile __STREXH( unsigned short value, unsigned short * addr );</td>
<td>Store exclusive half-word. This function performs an exclusive STR command for 16-bit values.</td>
</tr>
<tr>
<td>Intrinsic Function</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>unsigned int volatile __STREXW( unsigned int value, unsigned int * addr );</td>
<td>Store exclusive word. This function performs an exclusive STR command for 32-bit values.</td>
</tr>
<tr>
<td>void volatile __CLREX( void );</td>
<td>Remove the exclusive lock created by __LDREXB, __LDREXH, or __LDREXW.</td>
</tr>
<tr>
<td>unsigned int volatile __RRX( unsigned int value );</td>
<td>Rotate Right with Extend (32-bit). This function moves each bit of a bit string right by one bit. The carry input is shifted in at the left end of the bit string.</td>
</tr>
<tr>
<td>unsigned char volatile __LDRBT( unsigned char *addr );</td>
<td>LDRT Unprivileged (8-bit). This function performs an Unprivileged LDRT command for 8-bit values.</td>
</tr>
<tr>
<td>unsigned volatile short __LDRHT( unsigned short *addr );</td>
<td>LDRT Unprivileged (16-bit). This function performs an Unprivileged LDRT command for 16-bit values.</td>
</tr>
<tr>
<td>unsigned char volatile __LDRT( unsigned int *addr );</td>
<td>LDRT Unprivileged (32-bit). This function performs an Unprivileged LDRT command for 32-bit values.</td>
</tr>
<tr>
<td>void volatile __STRBT( unsigned char value, unsigned char *addr );</td>
<td>STRT Unprivileged (8-bit). This function performs an Unprivileged STRT command for 8-bit values.</td>
</tr>
<tr>
<td>void volatile __STRHT( unsigned short value, unsigned short *addr );</td>
<td>STRT Unprivileged (16-bit). This function performs an Unprivileged STRT command for 16-bit values.</td>
</tr>
<tr>
<td>void volatile __STRT( unsigned int value, unsigned int *addr );</td>
<td>STRT Unpriviledge (32-bit). This function performs an Unprivileged STRT command for 32-bit values.</td>
</tr>
<tr>
<td>void volatile __BKPT( unsigned short int );</td>
<td>Set Breakpoint. This function causes the processor to enter Debug state. Debug tools can use this to investigate system state when the instruction at a particular address is reached.</td>
</tr>
<tr>
<td>unsigned char __CLZ( unsigned int value );</td>
<td>Count leading zeros. This function counts the number of leading zeros of a data value.</td>
</tr>
<tr>
<td>int __SSAT( int value, unsigned int sat );</td>
<td>Signed Saturate. This function saturates a signed value.</td>
</tr>
<tr>
<td>unsigned int __USAT( unsigned int value, unsigned int sat );</td>
<td>Unsigned Saturate. This function saturates an unsigned value.</td>
</tr>
</tbody>
</table>

### 1.12.4.3. 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 1.7, 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 1.12.2, Inlining Functions: inline.

```c
inline int __my_pow( int base, int power )
{
    int result;

    __asm( "movs  %0,#1\n"
```
"1:\n\t"  
"subs  %2,%2,#1\n\t"  
"mul   %0,%0,%1\n\t"  
"bne   1p\n\t"  
: ":="&r"(result) 
: "r"(base), "r"(power) );

    return result;
    }

int main(void)
{  
    int result;

    // call to function __my_pow
    result = __my_pow(3,2);

    return result;
    }

Generated assembly code (compiled for ARMv7M):

main:  .type func
; __my_pow code is inlined here
    mov     r1,#2
    mov     r2,#3
    movs  r0,#1

1:
    subs  r1,r1,#1
    mul   r0,r0,r2
    bne   1p

As you can see, the generated assembly code for the function __my_pow is inlined rather than called. Numeric labels are used for the loop.

1.13. Compiler Generated Sections

The compiler generates code and data in several types of sections. By default the C compiler generates sections with the following names:

section_type_prefix[.core_association]

A core association, share, tcmn or clone, is only present for virtual-core projects and is only present when explicitly specified by a core association. The same applies to tcm for a single-core project when explicitly specified. See Section 1.5, Multi-Core Support. When no core association is specified it is also not part of the section name and the section is shared by default.

When you use a section renaming pragma, the compiler uses the following section naming convention:
The prefix depends on the type of the section and determines if the section is initialized, constant or uninitialized.

The following table lists the section types and name prefixes.

<table>
<thead>
<tr>
<th>Section type prefix</th>
<th>Description</th>
<th>Section type</th>
<th>Implied attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text</td>
<td>Code sections</td>
<td>code</td>
<td></td>
</tr>
<tr>
<td>.data</td>
<td>Initialized data</td>
<td>data</td>
<td>init</td>
</tr>
<tr>
<td>.bss</td>
<td>Uninitialized data (cleared)</td>
<td>data</td>
<td>noinit, clear</td>
</tr>
<tr>
<td>.rodata</td>
<td>ROM data (constants)</td>
<td>data</td>
<td>noinit, romdata</td>
</tr>
<tr>
<td>.debug</td>
<td>Debug sections. .debug_ is used as a prefix</td>
<td>debug</td>
<td></td>
</tr>
</tbody>
</table>

### 1.13.1. Rename Sections

You can change the default section names with one of the following pragmas. The naming convention for the renamed section is:

```c
section_type_prefix[.core_association][.module_name][.symbol_name][.pragma_value]
```

Note however that a symbol at an absolute address (`__at`) is located in a section that always uses the default section name.

#### #pragma section [section_type_prefix=[suffix [-f|-m|-fm]]

With this pragma you can rename sections by adding a `suffix` to all section names specified with `section_type_prefix`, or restore default section naming. If you specify only a `suffix` (without a section type prefix), the suffix is added to all section names. On the command line you can use the [C compiler option `--rename-sections`].

**Example**

```c
#pragma section .bss=where
```

After this pragma all `.bss` sections will have the name `.bss.where`. The core association stays part of the name. So, if there is a section named `.bss.share`, that section will be renamed to `.bss.share.where`

The `-f`, `-m`, `-fm` have a special meaning:

- `-f` Add function name as section name suffix (for functions only)
- `-m` Add module base name as section name suffix
- `-fm` or `-mf` Add module base name for data and add function name for functions

**Examples (module is `test.c`):**
You can also specify one of the following special suffixes:

- `{module}` Add module base name as section name suffix
- `{name}` Add the symbol name as generated in the assembly file, including compiler generated prefixes and suffixes
- `{cname}` Add to the symbol name as used in your C source. Compiler generated names will be cleaned up and prefixed by a '$'.

Examples showing the difference between `{name}` and `{cname}`:

```
.section .text.tm..cocofun_1  ;; {name}
.section .text.tm.$cocofun    ;; {cname}
.section .rodata.hs..1.str   ;; {name}
.section .rodata.hs.$str     ;; {cname}
.section .rodata.hs..2.ini   ;; {name}
.section .rodata.hs.$ini     ;; {cname}
```

#pragma section default

This will restore the default section naming to "section_type_prefix[.core_association]".

#pragma section restore

This will restore the previous setting of #pragma section.

Examples:

```
#pragma section rename_1
   // .text.rename_1
   // .data.rename_1

#pragma section .text=rename_2
   // .text.rename_2
   // .data.rename_1

#pragma section restore
   // .text.rename_1
   // .data.rename_1

#pragma section default
   // .text
   // .data
```

1.13.2. Influence Section Definition

The following pragmas also influence the section definition:
#pragma section code_init

#pragma section no_code_init

Copy or do not copy code sections from ROM to RAM at application startup.

#pragma section const_init

#pragma section no_const_init

Copy or do not copy read-only data sections from ROM to RAM at application startup.
Chapter 2. C++ Language

The TASKING C++ compiler (cparm) offers a new approach to high-level language programming for your ARM architecture. The C++ compiler accepts the C++ language as defined by the ISO/IEC 14882:2014 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


More information on the standards can be found at http://www.iso.org/

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.

pragmas

The C++ compiler supports the 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 by default accepts the complete C++ language as defined by the ISO/IEC 14882:2014 standard (C++ compiler option --c++=14).

With option --c++=03 the C++ compiler accepts the complete C++03 language as defined by the ISO/IEC 14882:2003 standard.

With option --c++=11 the C++ compiler accepts the complete C++11 language as defined by the ISO/IEC 14882:2011 standard.

Selecting C++11 or later also enables exceptions. For C++03 the STLport library is used, while for C++11 and C++14 the libc++ library is used.

You can combine the C++ dialects with the option for strict standard conformance (option --strict).
2.2.1. C++03 Mode

The following extensions are accepted in C++03 mode. Most of these are also accepted in any other C++
mode (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.

• A const qualified object with file scope or namespace scope and the __at() attribute will have external
linkage, unless explicitly declared static. Examples:

```cpp
const int i = 5;                         // internal linkage
const int j __at( 0x1234 ) = 10;         // external linkage
static const int k __at( 0x1236 ) = 15;  // internal linkage
```

Note that no warning is generated when ‘j’ is not used.

• 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.)

```c
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 C++03 mode (but not other non-C++11 modes), the friend class syntax is extended to allow non-class types as well as class types expressed through a `typedef` or without an elaborated type name. For example:

```c
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 C++03 mode, mixed string literal concatenations are accepted. (This is a feature carried over from C99 and also available in GNU modes.)

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

- In C++03 mode, variadic macros are accepted. (This is a feature carried over from C99 and also available in GNU modes.)

- In C++03 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):

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

- In C++03 mode, the C++11 feature `extern template` is supported, unless option `--strict` is applied.

- The `_Pragma` operator is always supported in all modes, also when option `--strict` is applied.

### 2.2.2. GNU C++ Mode

The C++ compiler can be configured to support GNU C++ (command line option `--g++`). In GNU C++ mode, many extensions provided by the GNU C++ compiler are accepted.
Because the GNU C++ compiler frequently changes behavior between releases, the C++ compiler provides an option (\texttt{--gnu_version}) to specify a specific version of GCC to emulate. Generally speaking, features and bugs are emulated to exactly match each known version of GCC, but occasionally the emulation is approximate and in such cases the C++ compiler is often a little more permissive than GCC on the principle that it is more important to accept source that GCC accepts than to diagnose every case that GCC diagnoses. The C++ compiler does not, however, attempt to emulate every GCC command line option; in particular, GCC options to be extra-permissive are not emulated (however, the severity of specific error diagnostics can sometimes be decreased to accept constructs that are not by default allowed in GNU emulation mode).

The following GNU extensions are \textit{not} 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
- Local structs with variable-length array fields.

\subsection*{2.2.3. Anachronisms Accepted}

The following anachronisms are accepted when anachronisms are enabled (with \texttt{--anachronisms}):

- \texttt{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 \texttt{delete} operation. The value is ignored.
- A single \texttt{operator++()} and \texttt{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 \texttt{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 \( f \):

\[
\begin{align*}
\text{int } f(\text{int}); \\
\text{int } f(x) \text{ char } x; \{ \text{return } x; \} 
\end{align*}
\]

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 \texttt{--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.

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

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

### 2.2.4. Atomic Operations

In C++11 mode (option \texttt{--c++=11}) or higher, the C++ compiler supports lock-free atomic operations in machine word size.

The atomic operations are implemented through the use of atomic operations provided by the ARM instruction set architecture. The C++ compiler does not use mutexes or other locking mechanisms to stall threads. The implementation is limited to atomic operations on variables with a type up to 32-bit only.

See the header file "installation_directory/carm/include.cxx/atomic" for an overview of the atomic operations that you can use.

Example (atomic.cpp)

```cpp
#include <atomic>
#include <iostream>

volatile int data_buffer;
std::atomic_int atom(1); // This variable is volatile by type definition

void thread_on_core1()
{
    data_buffer = 2;
    ++atom; // Making atom 2 informs thread_on_core2 that data_buffer is
```
void thread_on_core1()
{
    while ( atom != 3 )
    {
        /* empty */ // Wait until atom becomes 3,
        // set by thread 2, meaning data_buffer is updated
        std::cout << data_buffer << std::endl;
        // If data_buffer is not volatile, data_buffer would still be 2
    }
}

void thread_on_core2()
{
    while ( atom != 2 )
    {
        /* empty */ // Wait until atom becomes 2,
        // which means that data_buffer can be used
        std::cout << data_buffer << std::endl;
        data_buffer = 3; // Set data_buffer to 3 and
        ++atom; // Inform the other thread that data_buffer
        // can be used with new value
    }
}

int main(void)
{
    thread_on_core1();
    thread_on_core2();
    return 0;
}

Use the following command to build the example:

carm --c++=11 --cpu=ARMv7M atomic.cpp

Use the options --c++=11 and --cpu for one of the ARMv7 architectures to enable the use of atomic operations.

2.3. 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);
}
```

### 2.4. 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.\(^1\) 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, template variables, 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.

- 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.

2.4.1. 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=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 `ccarm` command is given a single file to compile and link, e.g.,

```
ccarm 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.

2.4.2. 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 member declaration `int A<int>::i`

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

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

• a template variable name `x<int>

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.

### 2.4.3. 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. Inlining Functions

The C++ compiler supports a minimal form of function inlining. When the C++ compiler encounters a call of a function declared \textit{inline} it can replace the call with the body of the function with the parameters replaced by the corresponding arguments. When a function call occurs as a statement, the statements of the function body are inserted in place of the call. When the function call occurs within an expression, the body of the function is rewritten as one large expression and that expression is inserted in the proper place in the containing expression. It is not always possible to do this sort of inlining: there are certain constructs (e.g. loops and inline assembly) that cannot be rendered in expression form. Even when inlining is done at the statement level, there are certain constructs that are not practical to inline. Calls that cannot be inlined are left in their original call form, and an out-of-line copy of the function is used. When enabled, a remark is issued.

When the C++ compiler decides not to inline a function, the keyword \textit{inline} is passed to the generated C file. This allows for the C compiler to decide again whether to inline a function or not.

A function is disqualified for inlining immediately if any of the following are true:

- The function has local static variables.
- The function has local constants.
- The function has local types.
- The function has block scopes.
- The function includes pragmas.
- The function has a variable argument list.

2.6. Extern Inline Functions

Depending on the way in which the C++ compiler is configured, out-of-line copies of \textit{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.
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:

```cpp
#pragma pragma-spec
```

or:

```cpp
Pragma("pragma-spec")
```

The C++ compiler supports the following pragmas:

**instantiate / do_notInstantiate / canInstantiate**

These are template instantiation pragmas. They are described in detail in Section 2.4.2, *Instantiation #pragma Directives*.

**hdrstop / no_pch**

These are precompiled header pragmas. They are described in detail in Section 2.9, *Precompiled Headers*.

**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 `#pragma once` at the start of a header file, it will skip over it if the file is #included again.

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

```cpp
#pragma once    // optional
#ifndef FILE_H
#define FILE_H
... body of the header file ...
#endif
```

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

2.7.1. C pragmas Supported by the C++ compiler

The C++ compiler supports the following C pragmas with the restriction that the `#pragma` can only be used if it is placed in C++ code adjacent to the variable or function it is meant for:
message "message" ...

Print the message string(s) on standard output.

section [name=][suffix |-f|-m|-fm} [default | restore] (*) / endsection

Rename sections by adding a suffix to all section names specified with name, or restore default section naming. If you specify only a suffix (without a name), the suffix is added to all section names. See C compiler option --rename-sections and assembler directive .SECTION for more information.

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><strong>BIG_ENDIAN</strong></td>
<td>Expands to 1 if big-endian mode is selected (option --endianness=big), otherwise unrecognized as macro.</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 in the format yymmddqq (year, month, day and quarter in UTC).</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>__CHAR16_T_AND_CHAR32_T</td>
<td>Defined when char16_t and char32_t are keywords. These keywords are enabled when you use C++ compiler option --uliterals.</td>
</tr>
<tr>
<td><strong>CHAR16_TYPE</strong></td>
<td>Defined as the underlying type for char16_t when option --g++ or --gnu-version is used. (GNU version 40400 and above, see --gnu-version).</td>
</tr>
<tr>
<td><strong>CHAR32_TYPE</strong></td>
<td>Defined as the underlying type for char32_t when option --g++ or --gnu-version is used. (GNU version 40400 and above, see --gnu-version).</td>
</tr>
<tr>
<td><strong>CPARM</strong></td>
<td>Identifies the C++ compiler. You can use this symbol to flag parts of the source which must be recognized by the cparm 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 ARM architecture name supplied to the option --cpu=arch. For example, if --cpu=ARMv7M is specified, the symbol <strong>CPU_ARMV7M</strong> is defined. This is also the default when no --cpu is supplied.</td>
</tr>
<tr>
<td><strong>CPU_arch</strong></td>
<td>A symbol is defined depending on the option --cpu=arch. The arch is converted to uppercase. For example, if --cpu=ARMv7M is specified, the symbol <strong>CPU_ARMV7M</strong> is defined. This is also the default when no --cpu is supplied.</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>Macro</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</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 C++11 mode and GNU C++ mode.</td>
</tr>
<tr>
<td><strong>GNUC</strong></td>
<td>Defined to the GNU C++ major version when option --g++ or --gnu-version is used. When the GNU C++ version is x.y.z, this macro is defined to x.</td>
</tr>
<tr>
<td><strong>GNUC_MINOR</strong></td>
<td>Defined to the GNU C++ minor version when option --g++ or --gnu-version is used. When the GNU C++ version is x.y.z, this macro is defined to y.</td>
</tr>
<tr>
<td><strong>GNUC_PATCHLEVEL</strong></td>
<td>Defined to the GNU C++ patch level version when option --g++ or --gnu-version is used. When the GNU C++ version is x.y.z, this macro is defined to z.</td>
</tr>
<tr>
<td><strong>GNUC_STDCC_INLINE</strong></td>
<td>Defined when option --g++ or --gnu-version is used and if functions declared inline will be handled according to the ISO C99 or later standards.</td>
</tr>
<tr>
<td><strong>GNUC_VERSION</strong></td>
<td>Defined to a string with the GNU C++ version when option --g++ or --gnu-version is used. When the GNU C++ version is x.y.z, this macro is defined to TASKING g++ x.y.z mode.</td>
</tr>
<tr>
<td><strong>GNUC</strong></td>
<td>Defined to the GNU C++ major version when option --g++ or --gnu-version is used. Testing it is equivalent to testing (<strong>GNUC</strong> &amp;&amp; __cplusplus).</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><strong>LITTLE_ENDIAN</strong></td>
<td>Expands to 1 if little-endian mode is selected (option --endianness=little), otherwise unrecognized as macro. This is the default.</td>
</tr>
<tr>
<td>Macro</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>__NAMESPACE</td>
<td>Defined when namespaces are supported (this is the default, you can disable support for namespaces with --no-namespaces).</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><strong>PTRDIFF_TYPE</strong></td>
<td>Defined to be the type of ptrdiff_t.</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>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>SIZE_TYPE</strong></td>
<td>Defined to be the type of size_t.</td>
</tr>
<tr>
<td><strong>STDC</strong></td>
<td>Always expands to 0.</td>
</tr>
<tr>
<td>_STLP_NO_IOSTREAMS</td>
<td>Defined when option --io-streams is not used. This disables I/O stream functions in the STLport C++ library.</td>
</tr>
<tr>
<td><strong>TASKING</strong></td>
<td>Always defined for the TASKING C++ compiler.</td>
</tr>
</tbody>
</table>
### Macro Description

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>THUMB</strong></td>
<td>Expands to 1 if you used option --thumb, otherwise unrecognized as macro.</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>__VARIADIC_TEMPLATES</td>
<td>Defined when C++11 variadic templates are supported (option --variadic-templates).</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>__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,

```c
#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 recompileions, 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:

```
#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:

```
#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 ARM and contains a detailed description of all built-in assembly functions and assembler directives. For a complete overview of the architecture you are using and a description of the assembly instruction set, refer to the target's core reference manual (for example the ARM Architecture Reference Manual ARM DDI 0100I [2005, ARM Limited]).

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, directives and other keywords 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. The label can also be a *number*. 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.

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

Examples:

LAB1:  
LAB1  
1: b 1p

; This label is followed by a colon and can be prefixed by whitespace
; This label has to start at the beginning
; of a line
; This is an endless loop
; using numeric labels
An instruction consists of a mnemonic and zero, one or more operands. It must not start in the first column.

All instructions of the ARM Unified Assembler Language (UAL) are supported. With assembler option --old-syntax you can specify to use the pre-UAL syntax. VFP instructions are only supported in the UAL syntax.

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

The instruction can also be a so-called 'generic instruction'. Generic instructions are pseudo instructions (no instructions from the instruction set). Depending on the situation in which a generic instruction is used, the assembler replaces the generic instruction with appropriate real assembly instruction(s). For a complete list, see Section 3.11, Generic Instructions.

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.

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

Comment, preceded by a ; (semicolon).

You can use empty lines or lines with only comments.

### 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>Immediate addressing</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 <strong>Section 3.4, Symbol Names</strong>. Symbols can also occur in expressions.</td>
</tr>
<tr>
<td>register</td>
<td>Any valid register as listed in <strong>Section 3.5, Registers</strong>.</td>
</tr>
<tr>
<td>expression</td>
<td>Any valid expression as described in <strong>Section 3.6, Assembly Expressions</strong>.</td>
</tr>
<tr>
<td>address</td>
<td>A combination of <em>expression</em>, <em>register</em> and <em>symbol</em>.</td>
</tr>
</tbody>
</table>

### Addressing modes

The ARM assembly language has several addressing modes. These are described in detail in the target's core Architecture Reference 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 or starts at the first column.
Reserved symbols

Symbol names and other identifiers starting with a period (.) are reserved for the system (for example for directives or section names). Identifiers starting with an at sign (@) are reserved for built-in assembler functions. Instructions are also reserved. The case of these built-in symbols is insignificant.

Examples

Valid symbol names:

```markdown
loop_1
ENTRY
_a_B_c
_aBC
```

Invalid symbol names:

```markdown
1_loop ; starts with a number
.DEFINExx ; 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.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ASARM</strong></td>
<td>Expands to 1 for the ARM toolset, otherwise unrecognized as macro.</td>
</tr>
<tr>
<td><strong>BUILD</strong></td>
<td>Identifies the build number of the assembler in the format yymmddqq (year, month, day and quarter in UTC).</td>
</tr>
<tr>
<td><strong>REVISION</strong></td>
<td>Expands to the revision number of the assembler. 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><strong>TASKING</strong></td>
<td>Identifies the assembler as a TASKING assembler. Expands to 1 if a TASKING assembler is used.</td>
</tr>
<tr>
<td><strong>VERSION</strong></td>
<td>Identifies the version number of the assembler. For example, if you use version 2.1r1 of the assembler, <strong>VERSION</strong> expands to 2001 (dot and revision number are omitted, minor version number in 3 digits).</td>
</tr>
</tbody>
</table>

Example

```markdown
.if @defined('__ASARM__')
  ; this part is only for the asarm assembler
...
.endif
```
3.5. Registers

The following register names, either uppercase or lowercase, should not be used for user-defined symbol names in an assembly language source file:

R0 .. R15  (general purpose registers)
IP (alias for R12)
SP (alias for R13)
LR (alias for R14)
PC (alias for R15)

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
- symbol
- expression binary_operator expression
- unary_operator expression
- (expression)
- function call

All types of expressions are explained in separate sections.
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 lowercase or uppercase.

<table>
<thead>
<tr>
<th>Base</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>A \texttt{0b} prefix followed by binary digits (0,1). Or use \texttt{a b} suffix.</td>
<td>\texttt{0B1101} \texttt{11001010b}</td>
</tr>
<tr>
<td>Hexadecimal</td>
<td>A \texttt{0x} prefix followed by hexadecimal digits (0-9, A-F, a-f). Or use \texttt{a h} suffix.</td>
<td>\texttt{0x12FF} \texttt{0x45} \texttt{0fa10h}</td>
</tr>
<tr>
<td>Decimal integer</td>
<td>Decimal digits (0-9).</td>
<td>12 \texttt{1245}</td>
</tr>
<tr>
<td>Decimal floating-point</td>
<td>Decimal digits (0-9), includes a decimal point, or an 'E' or 'e' followed by the exponent.</td>
<td>\texttt{6E10} .6 \texttt{3.14} \texttt{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 \texttt{.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 8 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 \texttt{.DB} assembler directive; in that case all characters result in a constant value of the specified size. Null strings have a value of 0.

**Examples**

\begin{verbatim}
'ABCD'                      ; (0x44434241)
''  '79'                    ; to enclose a quote double it
"A""BC"                     ; or to enclose a quote escape it
'AB'+1                      ; (0x4341) string used in expression
' '                        ; null string
.DW  'abcdef'               ; (0x64636261) 'ef' are ignored
; warning: string value truncated
'ab'++'cd'                  ; you can concatenate
; two strings with the '++' operator.
; This results in 'abcd'
\end{verbatim}
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></td>
<td>-</td>
<td>subtraction</td>
<td>Yields the difference of its operands.</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 Operators

<table>
<thead>
<tr>
<th>Type</th>
<th>Operator</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&lt;</td>
<td>less than</td>
<td>Returns an integer 1 if the indicated condition is TRUE or an integer 0 if</td>
</tr>
<tr>
<td></td>
<td>&lt;=</td>
<td>less than or</td>
<td>the indicated condition is FALSE.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>equal</td>
<td>For example, if D has a value of 3 and E has a value of 5, then the result</td>
</tr>
<tr>
<td></td>
<td>&gt;</td>
<td>greater than</td>
<td>of the expression ( D &lt; E ) is 1, and the result of the expression ( D &gt; E ) is 0.</td>
</tr>
<tr>
<td></td>
<td>&gt;=</td>
<td>greater than or</td>
<td>Use tests for equality involving floating-point values with caution, since</td>
</tr>
<tr>
<td></td>
<td></td>
<td>equal or</td>
<td>rounding errors could cause unexpected results.</td>
</tr>
<tr>
<td></td>
<td>!=</td>
<td>not equal</td>
<td></td>
</tr>
</tbody>
</table>

Use tests for equality involving floating-point values with caution, since rounding errors could cause unexpected results.

### Bitwise Operators

<table>
<thead>
<tr>
<th>Type</th>
<th>Operator</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&amp;</td>
<td>AND</td>
<td>Integer only. Yields the bitwise AND function of</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>its operand.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>OR</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>its operand.</td>
</tr>
<tr>
<td></td>
<td>^</td>
<td>exclusive OR</td>
<td>Integer only. Yields the bitwise exclusive OR</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>function of its operands.</td>
</tr>
</tbody>
</table>

### Logical Operators

<table>
<thead>
<tr>
<th>Type</th>
<th>Operator</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>&amp;&amp;</td>
<td>logical AND</td>
<td>Returns an integer 1 if both operands are non-zero;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>otherwise, it returns an integer 0.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>non-zero; otherwise, it returns an integer 1</td>
</tr>
</tbody>
</table>

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. The compiler automatically generates sections. If you program in assembly you have to define sections yourself.

For more information about locating sections see Section 7.8.8, *The Section Layout Definition: Locating Sections*.

### Section definition

Sections are defined with the `.SECTION/.ENDSEC` directive and have a name. The names have a special meaning to the locating process and have to start with a predefined name, optionally extended by a dot
'.' and a user defined name. Optionally, you can specify the `at()` attribute to locate a section at a specific address.

```
.SECTION name[,at(address)]
; instructions etc.
.ENDSEC
```

See the description of the `.SECTION` directive for more information.

**Examples**

```
.SECTION .data               ; Declare a .data section
 ; ...
.ENDSEC

.SECTION .data.abs, at(0x0)  ; Declare a .data.abs section at
 ; an absolute address
 ; ...
.ENDSEC
```

### 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.

The names of assembly functions are case insensitive.

**Overview of assembly functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>@ALUPCREL(expr,group[,check])</code></td>
<td>PC-relative ADD/SUB with operand split</td>
</tr>
<tr>
<td>`@ARG('symbol'</td>
<td>expr)`</td>
</tr>
<tr>
<td><code>@BIGENDIAN()</code></td>
<td>Test if assembler generates code for big-endian mode</td>
</tr>
<tr>
<td><code>@CNT()</code></td>
<td>Return number of macro arguments</td>
</tr>
<tr>
<td><code>@CPU('architecture')</code></td>
<td>Test if current CPU matches <code>architecture</code></td>
</tr>
<tr>
<td>`@DEFINED('symbol'</td>
<td>symbol)`</td>
</tr>
<tr>
<td><code>@LSB(expr)</code></td>
<td>Least significant byte of the expression</td>
</tr>
<tr>
<td><code>@LSH(expr)</code></td>
<td>Least significant half word of the absolute expression</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>@LSW(expr)</td>
<td>Least significant word of the expression</td>
</tr>
<tr>
<td>@MSB(expr)</td>
<td>Most significant byte of the expression</td>
</tr>
<tr>
<td>@MSH(expr)</td>
<td>Most significant half word of the absolute expression</td>
</tr>
<tr>
<td>@MSW(expr)</td>
<td>Most significant word of the expression</td>
</tr>
<tr>
<td>@PRE_UAL()</td>
<td>Test if the assembler runs in pre-UAL syntax mode or in UAL syntax mode by default (option (--old-syntax))</td>
</tr>
<tr>
<td>@STRCAT(str1, str2)</td>
<td>Concatenate (str1) and (str2)</td>
</tr>
<tr>
<td>@STRCMP(str1, str2)</td>
<td>Compare (str1) with (str2)</td>
</tr>
<tr>
<td>@STRLEN(string)</td>
<td>Return length of string</td>
</tr>
<tr>
<td>@STRPOS(str1, str2[, start])</td>
<td>Return position of (str2) in (str1)</td>
</tr>
<tr>
<td>@STRSUB(str, expr1, expr2)</td>
<td>Return substring</td>
</tr>
<tr>
<td>@THUMB()</td>
<td>Test if the assembler runs in Thumb mode or in ARM mode by default (option (--thumb))</td>
</tr>
</tbody>
</table>

**Detailed Description of Built-in Assembly Functions**

**@ALUPCREL(expression, group[, check])**

This function is used internally by the assembler with the generic instructions ADR, ADRL and ADRLL. This function returns the PC-relative address of the expression for use in these generic instructions. `group` is 0 for ADR, 1 for ADRL or 2 for ADRLL.

With `check` you can specify to check for overflow (1 means true, 0 means false). If `check` is omitted, the default is 1.

Example:

```
; The instruction "ADRAL R1,label" expands to
ADRAL R1,PC,@ALUPCREL(label,0,1)
```

**@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:

```
.IOF @ARG('TWIDDLE') ;is argument twiddle present?
.IOF @ARG(1)           ;is first argument present?
```
@BIGENDIAN()  
Returns 1 if the assembler generates code for big-endian mode, returns 0 if the assembler generates code for little-endian mode (this is the default).

@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

@CPU('architecture')  
Returns 1 if architecture corresponds to the architecture that was specified with the option --cpu=architecture; 0 otherwise. See also assembler option --cpu (Select architecture).

Example:

.IF @CPU('ARMv7EM') ; true if you specified option --cpu=ARMv7EM  
... ; code for the ARMv7EM  
.ELIF @CPU('ARMv6M') ; true if you specified option --cpu=ARMv6M  
... ; code for the ARMv6-M  
.ELSE  
... ; code for other architectures  
.ENDIF

@DEFINED('symbol | symbol')  
Returns 1 if symbol has been defined, 0 otherwise. 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, macro or label.

Example:

.IF @DEFINED('ANGLE') ; is symbol ANGLE defined?  
.IF @DEFINED(ANGLE) ; does label ANGLE exist?

@LSB(expression)  
Returns the least significant byte of the result of the expression. The result of the expression is calculated as 16 bits.

Example:

.DB @LSB(0x1234) ; stores 0x34  
.DB @MSB(0x1234) ; stores 0x12
@LSH(expression)

Returns the least significant half word (bits 0..15) of the result of the absolute expression. The result of the expression is calculated as a word (32 bits).

@LSW(expression)

Returns the least significant word (bits 0..31) of the result of the expression. The result of the expression is calculated as a double-word (64 bits).

Example:

```
.DW @LSW(0x12345678) ; stores 0x5678
.DW @MSW(0x123456) ; stores 0x0012
```

@MSB(expression)

Returns the most significant byte of the result of the expression. The result of the expression is calculated as 16 bits.

@MSH(expression)

Returns the most significant half word (bits 16..31) of the result of the absolute expression. The result of the expression is calculated as a word (32 bits). @MSH(expression) is equivalent to ((expression>>16) & 0xffff).

@MSW(expression)

Returns the most significant word of the result of the expression. The result of the expression is calculated as a double-word (64 bits).

@PRE_UAL()

Returns 1 if the assembler runs in pre-UAL syntax mode by default, or 0 if the assembler runs in UAL syntax mode (default). This function reflects the setting of the assembler option --old-syntax.

Example:

```
.IF @PRE_UAL() ; true if you specified option --old-syntax
... ; old code
.ELSE
... ; new code, UAL syntax
.ENDIF
```

@STRCAT(string1,string2)

Concatenates string1 and string2 and returns them as a single string. You must enclose string1 and string2 either with single quotes or with double quotes.

Example:
.DEFINE ID "@STRCAT('TAS','KING')" ; ID = 'TASKING'

@STRCMP(string1,string2)

Compares string1 with string2 by comparing the characters in the string. The function returns the difference between the characters at the first position where they disagree, or zero when the strings are equal:

<0 if string1 < string2
0 if string1 == string2
>0 if string1 > string2

Example:

.IF (@STRCMP(STR,'MAIN'))==0 ; does STR equal 'MAIN'?

@STRLEN(string)

Returns the length of string as an integer.

Example:

SLEN .SET @STRLEN('string') ; SLEN = 6

@STRPOS(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 start you can specify the starting position of the search. If you do not specify start, the search is started from the beginning of string1.

Example:

ID .set @STRPOS('TASKING','ASK') ; ID = 1
ID .set @STRPOS('TASKING','BUG') ; ID = 7

@STRSUB(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:

.DEFINFE ID "@STRSUB('TASKING',3,4)" ;ID = 'KING'
@THUMB()

Returns 1 if the assembler runs in Thumb mode by default or 0 if the assembler runs in ARM mode (default). This function reflects the setting of the assembler option --thumb. So, it does not depend on the .CODE16, .CODE32, .ARM or .THUMB directive.

If you are in a .CODE32 part and you specified --thumb, @THUMB() still returns 1.

3.9. Assembler Directives

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. Unlike other directives, preprocessor directives can start in the first column.

• 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 directives .NOLIST and .LIST you overrule this option for a part of the code that you do not want to appear in the list file. Directives of this kind sometimes are called controls.

Each assembler directive has its own syntax. 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). You can use assembler directives in the assembly code as pseudo instructions. The assembler recognizes both uppercase and lowercase for directives.
3.9.1. Overview of Assembler Directives

The following tables provide an overview of all assembler directives. For a detailed description of these directives, refer to Section 3.9.2, Detailed Description of Assembler Directives.

Overview of assembly control directives

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.END</td>
<td>Indicates the end of an assembly module</td>
</tr>
<tr>
<td>.INCLUDE</td>
<td>Include file</td>
</tr>
<tr>
<td>.MESSAGE</td>
<td>Programmer generated 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>.SECTION, .ENDSEC</td>
<td>Start a new section</td>
</tr>
<tr>
<td>.SET</td>
<td>Set temporary value to a symbol</td>
</tr>
<tr>
<td>.SIZE</td>
<td>Set size of symbol in the ELF symbol table</td>
</tr>
<tr>
<td>.SOURCE</td>
<td>Specify name of original C source file</td>
</tr>
<tr>
<td>.TYPE</td>
<td>Set symbol type in the ELF symbol table</td>
</tr>
<tr>
<td>.WEAK</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>.ALIGN</td>
<td>Align location counter</td>
</tr>
<tr>
<td>.BS, .BSB, .BSH, .BSW, .BSD</td>
<td>Define block storage (initialized)</td>
</tr>
<tr>
<td>.DB</td>
<td>Define byte</td>
</tr>
<tr>
<td>.DH</td>
<td>Define half word (16 bits)</td>
</tr>
<tr>
<td>.DW</td>
<td>Define word (32 bits)</td>
</tr>
<tr>
<td>.DD</td>
<td>Define double-word (64 bits)</td>
</tr>
<tr>
<td>.DOUBLE</td>
<td>Define a 64-bit floating-point constant</td>
</tr>
<tr>
<td>.DS, .DSB, .DSH, .DSW, .DSD</td>
<td>Define storage</td>
</tr>
<tr>
<td>.FLOAT</td>
<td>Define a 32-bit floating-point constant</td>
</tr>
</tbody>
</table>
Overview of macro preprocessor directives

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.OFFSET</td>
<td>Move location counter forwards</td>
</tr>
<tr>
<td>.DEFINE</td>
<td>Define substitution string</td>
</tr>
<tr>
<td>.BREAK</td>
<td>Break out of current macro expansion</td>
</tr>
<tr>
<td>.FOR,.ENDFOR</td>
<td>Repeat sequence of source lines n times</td>
</tr>
<tr>
<td>.IF,.ELIF,.ELSE</td>
<td>Conditional assembly directive</td>
</tr>
<tr>
<td>.ENDIF</td>
<td>End of conditional assembly directive</td>
</tr>
<tr>
<td>.MACRO,.ENDM</td>
<td>Define macro</td>
</tr>
<tr>
<td>.REPEAT,.ENDREP</td>
<td>Repeat sequence of source lines</td>
</tr>
<tr>
<td>.UNDEF</td>
<td>Undefine .DEFINE symbol or macro</td>
</tr>
</tbody>
</table>

Overview of listing control directives

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.LIST,.NOLIST</td>
<td>Print / do not print source lines to list file</td>
</tr>
<tr>
<td>.PAGE</td>
<td>Set top of page/size of page</td>
</tr>
<tr>
<td>.TITLE</td>
<td>Set program title in header of assembly list file</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>.COMPILER_INVOCATION</td>
<td>Pass C compiler invocation</td>
</tr>
<tr>
<td>.COMPILER_NAME</td>
<td>Pass C compiler name</td>
</tr>
<tr>
<td>.COMPILER_VERSION</td>
<td>Pass C compiler version header</td>
</tr>
<tr>
<td>.MISRAC</td>
<td>Pass MISRA C information</td>
</tr>
</tbody>
</table>

Overview of ARM specific directives

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.CODE16,.CODE32</td>
<td>Treat instructions as Thumb or ARM instructions using pre-UAL syntax</td>
</tr>
<tr>
<td>.THUMB,.ARM</td>
<td>Treat instructions as Thumb or ARM instructions using UAL syntax</td>
</tr>
<tr>
<td>.LTORG</td>
<td>Assemble current literal pool immediately</td>
</tr>
</tbody>
</table>

3.9.2. Detailed Description of Assembler Directives
.ALIAS

Syntax

alias-name .ALIAS symbol-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

Related information

Pragma alias
.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

.SECTION .text
.ALIGN 4 ; the assembler aligns
instruction ; this instruction at 4 MAUs and
; fills the 'gap' with NOP instructions.
.ENDSEC

.SECTION .text
.ALIGN 3 ; WRONG: not a power of two, the
instruction ; assembler aligns this instruction at
; 4 MAUs and issues a warning.
.ENDSEC
.BREAK

Syntax

.BREAK

Description

The .BREAK directive causes immediate termination of a macro expansion, a .FOR loop expansion or a .REPEAT loop expansion. In case of nested loops or macros, the .BREAK directive returns to the previous level of expansion.

The .BREAK directive is, for example, useful in combination with the .IF directive to terminate expansion when error conditions are detected.

The assembler does not allow a label with this directive.

Example

.FOR MYVAR IN 10 TO 20
    ... ;
    ... ; assembly source lines
    ...
    .IF MYVAR > 15
        .BREAK
        .ENDIF
    .ENDFOR
.BS, .BSB, .BSH, .BSW, .BSD

Syntax

[<label>] .BS count[,<value>]
[<label>] .BSB count[,<value>]
[<label>] .BSH count[,<value>]
[<label>] .BSW count[,<value>]
[<label>] .BSD count[,<value>]

Description

With the .BS directive the assembler reserves a block of memory. The reserved block of memory is initialized to the value of value, or zero if omitted.

With count you specify the number of minimum addressable units (MAUs) you want to reserve, and how much the location counter will advance. The expression must be an integer greater than zero and cannot contain any forward references to address labels (labels that have not yet been defined).

With value you can specify a value to initialize the block with. Only the least significant MAU of value is used. If you omit value, the default is zero.

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

You cannot initialize of a block of memory in sections with prefix .bss. In those sections, the assembler issues a warning and only reserves space, just as with .DS.

The .BSB, .BSH, .BSW and .BSD directives are variants of the .BS directive. The difference is the number of bits that are reserved for the count argument:

<table>
<thead>
<tr>
<th>Directive</th>
<th>Reserved bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>.BSB</td>
<td>8</td>
</tr>
<tr>
<td>.BSH</td>
<td>16</td>
</tr>
<tr>
<td>.BSW</td>
<td>32</td>
</tr>
<tr>
<td>.BSD</td>
<td>64</td>
</tr>
</tbody>
</table>

Example

The .BSB directive is for example useful to define and initialize an array that is only partially filled:

```
.section .data
.DB 84,101,115,116 ; initialize 4 bytes
.BSB 96,0xFF        ; reserve another 96 bytes, initialized with 0xFF
.endsec
```
Related Information

.DB (Define Memory)

.DS (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.

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. Use the .CALLS directive in hand-coded assembly when the assembly code calls a C function. If you manually add .CALLS directives, make sure they connect to the compiler generated .CALLS directives: the name of the caller must also be named as a callee in another directive.

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.
.CODE16, .CODE32, .THUMB, .ARM

Syntax

.CODE16
.CODE32
.THUMB
.ARM

Description

With the .CODE16 directive you instruct the assembler to interpret subsequent instructions as 16-bit Thumb instructions using the pre-UAL syntax until it encounters another mode directive or till it reaches the end of the active section. This directive causes an implicit alignment of two bytes.

The .THUMB directive is the same as the .CODE16 directive except that the UAL syntax is expected.

With the .CODE32 directive you instruct the assembler to interpret subsequent instructions as 32-bit ARM instructions using the pre-UAL syntax until it encounters another mode directive or till it reaches the end of the active section. This directive causes an implicit alignment of four bytes. The assembler issues an error message if .CODE32 is used in combination with option --cpu=ARMv7M.

The .ARM directive is the same as the .CODE32 directive except that the UAL syntax is expected.

These directives are useful when you have files that contain both ARM and Thumb instructions. The directive must appear before the instruction change and between a .SECTION/.ENDSEC. The default instruction set at the start of a section depends on the use of assembler option --thumb.

Example

.section .text
.code32
  ;following instructions are ARM instructions
  ;
.endsec

Related Information

Assembler option --thumb (Treat input as Thumb instructions)
.COMPILER_INVOCATION, .COMPILER_NAME, .COMPILER_VERSION

Syntax

.COMPILER_VERSION "version_header"
.COMPILER_INVOCATION "invocation"
.COMPILER_NAME "name"

Description

The C compiler generates information about itself and the invocation at the start of the assembly source. This way you can always see how the assembly source file was generated. When you assemble the source file, this information will appear in .note sections in the object file.

A label is not allowed before these directives.

Example

.COMPILER_VERSION "TASKING VX-toolset for ARM vx.yrz - ARM C compiler Build yymmddqq"
.COMPILER_INVOCATION "carm -CARMv7M test.c"
.COMPILER_NAME "carm"
.DB, .DH, .DW, .DD

Syntax

[label] .DB argument[,argument]...
[label] .DH argument[,argument]...
[label] .DW argument[,argument]...
[label] .DD argument[,argument]...

Description

With these directives you can define memory. With each directive the assembler allocates and initializes one or more bytes 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 must be separated by commas with no intervening spaces. Empty arguments are stored as 0 (zero).

The following table shows the number of bits initialized.

<table>
<thead>
<tr>
<th>Directive</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>.DB</td>
<td>8</td>
</tr>
<tr>
<td>.DH</td>
<td>16</td>
</tr>
<tr>
<td>.DW</td>
<td>32</td>
</tr>
<tr>
<td>.DD</td>
<td>64</td>
</tr>
</tbody>
</table>

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 half word / word / double-word, the assembler issues a warning and truncates the value.

String constants

Single-character strings are stored in a byte whose lower seven bits represent the ASCII value of the character, for example:

```
.DB 'R'        ; = 0x52
```

Multiple-character strings are stored in consecutive byte addresses, as shown below. The standard C language escape characters like \n are permitted.

```
.DB 'AB',,'C'  ; = 0x41420043 (second argument is empty)
```

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:
HTBL: .DH 'ABC',,'D' ; results in 0x424100004400 , the 'C' is truncated
WTBL: .DW 'ABC' ; results in 0x43424100

Related Information

.BS (Block Storage)
.DS (Define Storage)
.DEFINE

Syntax

```
.DEFINEm 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":

```
.DEFINEM LEN "32"
```

Then you can use the symbol `LEN` for example as follows:

```
.DSM LEN
.MESSAGE I "The length is: LEN"
```

The assembler preprocessor replaces `LEN` with "32" and assembles the following lines:

```
.DSM 32
.MESSAGE I "The length is: 32"
```

Related Information

`.UNDEF` (Undefine a .DEFINE symbol)

`.MACRO, .ENDM` (Define a macro)
.DS, .DSB, .DSH, .DSW, .DSD

Syntax

[label] .DS expression
[label] .DSB expression
[label] .DSH expression
[label] .DSW expression
[label] .DSD expression

Description

With the .DS directive the assembler reserves a block in memory. The reserved block of memory is not initialized to any value.

With the expression you specify the number of MAUs (Minimal Addressable Units) that you want to reserve, 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).

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

The .DSB, .DSH, .DSW and .DSD directives are variants of the .DS directive. The difference is the number of bits that are reserved per expression argument:

<table>
<thead>
<tr>
<th>Directive</th>
<th>Reserved bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>.DSB</td>
<td>8</td>
</tr>
<tr>
<td>.DSH</td>
<td>16</td>
</tr>
<tr>
<td>.DSW</td>
<td>32</td>
</tr>
<tr>
<td>.DSD</td>
<td>64</td>
</tr>
</tbody>
</table>

Example

```
.section .bss
RES: .DS 5+3 ; allocate 8 bytes
.endsec
```

Related Information

.BS (Block Storage)

.DB (Define Memory)
.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

    .section .text
    ; source lines
    .endsec
    .END                   ; End of assembly module
.EQU

Syntax

symbol .EQU expression

Description

With the .EQU directive you assign the value of expression to symbol permanently. The expression can be relative or absolute. 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 0x400

You cannot redefine the symbol MYSymbol after this.

Related Information

.SET (Set temporary value to a symbol)
.EXTERN

Syntax

.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

.EXTERN  AA,CC,DD       ;defined elsewhere

Related Information

.GLOBAL (Declare global section symbol)
.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

.DS (Define Storage)
.FOR, .ENDFOR

Syntax

\[
\text{[label]} \quad .\text{FOR var IN expression[,expression]}\ldots
\]
\[
\ldots
\]
\[
.\text{ENDFOR}
\]

or:

\[
\text{[label]} \quad .\text{FOR var IN start TO end [STEP step]}\]
\[
\ldots
\]
\[
.\text{ENDFOR}
\]

Description

With the .FOR/.ENDFOR directive you can repeat a block of assembly source lines with an iterator. As shown by the syntax, you can use the .FOR/.ENDFOR in two ways.

1. In the first method, the block of source statements is repeated as many times as the number of arguments following IN. If you use the symbol var in the assembly lines between .FOR and .ENDFOR, for each repetition the symbol var is substituted by a subsequent expression from the argument list. If the argument is a null, then the block is repeated with each occurrence of the symbol var removed. If an argument includes an embedded blank or other assembler-significant character, it must be enclosed with single quotes.

2. In the second method, the block of source statements is repeated using the symbol var as a counter. The counter passes all integer values from start to end with a step. If you do not specify step, the counter is increased by one for every repetition.

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

Example

In the following example the block of source statements is repeated 4 times (there are four arguments). With the .DB directive you allocate and initialize a byte of memory for each repetition of the loop (a word for the .DW directive). Effectively, the preprocessor duplicates the .DB and .DW directives four times in the assembly source.

\[
.\text{FOR VAR1 IN 1,2+3,4,12}
\]
\[
.\text{DB VAR1}
\]
\[
.\text{DW (VAR1*VAR1)}
\]
\[
.\text{ENDFOR}
\]

In the following example the loop is repeated 16 times. With the .DW directive you allocate and initialize four bytes of memory for each repetition of the loop. Effectively, the preprocessor duplicates the .DW directive 16 times in the assembled file, and substitutes VAR2 with the subsequent numbers.

\[
.\text{FOR VAR2 IN 1 to 0x10}
\]
\[
.\text{DW (VAR1*VAR1)}
\]
\[
.\text{ENDFOR}
\]
Related Information

.REPEAT, .ENDREP (Repeat sequence of source lines)
.GLOBAL

Syntax

.GLOBAL symbol[,symbol]...

Description

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

**Related Information**

*Assembler option --define* (Define preprocessor macro)
**.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 you omit a filename extension, the assembler assumes the extension `.asm`.

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 `ASARMINC` when the product was installed.

4. The default include directory in the installation directory.

The assembler does not allow a label with this directive.

The state of the assembler is not changed when an include file is processed. The lines of the include file are inserted just as if they belong to the file where it is included.

**Example**

Suppose that your assembly source file `test.src` contains the following line:

```
.INCLUDE "c:\myincludes\myinc.inc"
```

The assembler issues an error if it cannot find the file at the specified location.

```
.INCLUDE "myinc.inc"
```

The assembler searches the file `myinc.inc` according to the rules described above.

**Related Information**

Assembler option `--include-directory` (Add directory to include file search path)
LIST, Nolist

Syntax

.NLIST
.
.  ; assembly source lines
.
.LIST

Description

If you generate a list file with the assembler option --list-file, you can use the directives .LIST and .NLIST to specify which source lines the assembler must write to the list file. Without the assembler option --list-file these directives have no effect. The directives take effect starting at the next line.

The assembler prints all source lines to the list file, until it encounters a .NLIST directive. The assembler does not print the .NLIST directive and subsequent source lines. When the assembler encounters the .LIST directive, it resumes printing to the list file.

It is possible to nest the .LIST/.NLIST directives.

Example

Suppose you assemble the following assembly code with the assembler option --list-file:

```assembly
.SECTION .text
  ... ; source line 1
.NLIST
  ... ; source line 2
.LIST
  ... ; source line 3
.ENDSEC
```

The assembler generates a list file with the following lines:

```assembly
.SECTION .text
  ... ; source line 1
  ... ; source line 3
.ENDSEC
```

Related Information

Assembler option --list-file (Generate list file)
.LTORG

Syntax

.LTORG

Description

With this directive you force the assembler to generate a literal pool (data pocket) at the current location.

All literals from the LDR= pseudo-instructions (except those which could be translated to MOV or MVN instructions) between the previous literal pool and the current location will be assembled in a new literal pool using .DW directives.

By default, the assembler generates a literal pool at the end of a code section, i.e. the .ENDSEC directive at the end of a code section causes an implicit .LTORG directive. However, the default literal pool may be out-of-reach of one or more LDR= pseudo-instructions in the section. In that case the assembler issues an error message and you should insert .LTORG directives at proper locations in the section.

Example

.section .text

; LDR r1,=0x12345678
; code
.ltorg         ; literal pool contains the literal &0x12345678
;
; .endsec       ; default literal pool is empty

Related Information

LDR= ARM generic instruction

LDR= Thumb generic instruction
.MACRO, .ENDM

Syntax

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

Description

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

The 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).

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:

```
macro_a .MACRO  arg1,arg2
  .db arg1
;header
;body
```

Assembly Language
The macro call:

```assembly
.section far
macro_a 2,3
.endsec
```

The macro expands as follows:

```assembly
.db 2
.dw (2*3)
```

**Related Information**

Section 3.10, *Macro Operations*

`.DEFINE` (Define a substitution string)
.MESSAGE

Syntax

.MESSagE type taxonomy[{str|exp}[,{str|exp}]...]

Description

With the .MESSAGE directive you tell the assembler to print a message to stderr during the assembling process.

With type you can specify the following types of messages:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>Information message. Error and warning counts are not affected and the assembler continues the assembling process.</td>
</tr>
<tr>
<td>W</td>
<td>Warning message. Increments the warning count and the assembler continues the assembling process.</td>
</tr>
<tr>
<td>E</td>
<td>Error message. Increments the error count and the assembler continues the assembling process.</td>
</tr>
<tr>
<td>F</td>
<td>Fatal error message. The assembler immediately aborts the assembling process and generates no object file or list file.</td>
</tr>
</tbody>
</table>

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

.MESSagE I 'Generating tables'

ID .EQU 4
    .MESSagE E 'The value of ID is', ID

.DEFINe LONG "SHORT"
    .MESSagE I 'This is a LONG string'
    .MESSagE I "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
.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,63,87ff7,6ff3,4'

Related Information

Section 4.6.2, C Code Checking: MISRA C

C compiler option --misrac
.OFFSET

Syntax

.OFFSET expression

Description

With the .OFFSET directive you tell the assembler to give the location counter a new offset relative to the start of the section.

When the assembler encounters the .OFFSET directive, it moves the location counter forwards to the specified address, relative to the start of the section, and places the next instruction on that address. If you specify an address equal to or lower than the current position of the location counter, the assembler issues an error.

A label is not allowed with this directive.

Example

.SECTION .text
nop
nop
nop
.OFFSET 0x20 ; the assembler places
nop ; this instruction at address 0x20
; relative to the start of the section.
.ENDSEC

.SECTION .text
nop
nop
nop
.OFFSET 0x02 ; WRONG: the current position of the
nop ; location counter is 0x0C.
.ENDSEC

Related Information

.SECTION (Start a new section)
**.PAGE**

**Syntax**

```
.PAGE [pagewidth[, pagelength[, blankleft[, blanktop[, blankbtm]]]]]
```

**Default**

```
.PAGE 132, 72, 0, 0, 0
```

**Description**

If you generate a list file with the assembler option --list-file, you can use the directive .PAGE 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>blankleft</td>
<td>Number of blank columns at the left of the page. The default is 0,</td>
</tr>
<tr>
<td></td>
<td>the minimum is 0, and the maximum must maintain the relationship: blankleft &lt; pagewidth.</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>
</tbody>
</table>

If you use the .PAGE directive without arguments, it causes a 'formfeed': the next source line is printed on the next page in the list file. The .PAGE directive itself is not printed.

You can omit an argument by using two adjacent commas. If the remaining arguments after an argument are all empty, you can omit them.

**Example**

```
.PAGE ; formfeed, the next source line is printed
        ; on the next page in the list file.

.PAGE 96 ; set page width to 96. Note that you can
          ; omit the last four arguments.

.PAGE ,,,3,3 ; use 3 line top/bottom margins.
```

**Related Information**

`.TITLE` (Set program title in header of assembler list file)
Assembler option --list-file
.REPEAT, .ENDREP

Syntax

[label] .REPEAT expression
....
 .ENDREP

Description

With the .REPEAT/.ENDREP directive you can repeat a sequence of assembly source lines. With expression you specify the number of times the loop is repeated. 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 .REPEAT 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 3 times. Effectively, the preprocessor repeats the source lines (.DB 10) three times, then the assembler assembles the result:

.REPEAT 3
 .DB 10 ; assembly source lines
 .ENDREP

Related Information

.FOR, .ENDFOR (Repeat sequence of source lines n times)
**.SECTION, .ENDSEC**

**Syntax**

```
.SECTION name [, attribute ]... [,at(address)]
....
.ENDSEC
```

**Description**

With the `.SECTION` directive you define a new section. Each time you use the `.SECTION` directive, a new section is created. It is possible to create multiple sections with exactly the same name.

If you define a section, you must always specify the section `name`. The names have a special meaning to the locating process and have to start with a predefined name, optionally extended by a dot `.' and a user defined name. The predefined section name also determines the type of the section (code, data or debug). Optionally, you can specify the `at()` attribute to locate a section at a specific address.

You can use the following predefined section names:

<table>
<thead>
<tr>
<th>Section name</th>
<th>Description</th>
<th>Section type</th>
<th>Implied attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>.text</td>
<td>Code sections</td>
<td>code</td>
<td></td>
</tr>
<tr>
<td>.data</td>
<td>Initialized data</td>
<td>data</td>
<td>init</td>
</tr>
<tr>
<td>.bss</td>
<td>Uninitialized data (cleared)</td>
<td>data</td>
<td>noinit, clear</td>
</tr>
<tr>
<td>.rodata</td>
<td>ROM data (constants)</td>
<td>data</td>
<td>noinit, romdata</td>
</tr>
<tr>
<td>.debug</td>
<td>Debug sections. .debug_ is used as a prefix</td>
<td>debug</td>
<td></td>
</tr>
</tbody>
</table>

The section attributes are case insensitive. The defined attributes are:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Allowed on type</th>
</tr>
</thead>
<tbody>
<tr>
<td>at( address )</td>
<td>Locate the section at the given address.</td>
<td>code, data</td>
</tr>
<tr>
<td>clear</td>
<td>Sections are zeroed at startup. Clear can only be used in combination with noinit.</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, debug</td>
</tr>
<tr>
<td>concat</td>
<td>Concatenate sections. Used by the linker to merge sections with the same name. The assembler removes the default 'separate' ELF section attribute.</td>
<td>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>
</tbody>
</table>
**Attribute** | **Description** | **Allowed on type**
---|---|---
max | 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. Only works on sections with both attributes noinit and noclear set. | data |
noclear | 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. | data |
noinit | Defines that the section contains no initialization data. | code, data |
protect | Tells the linker to exclude a section from unreferenced section removal and duplicate section removal. | code, data |
romdata | Section contains data to be placed in ROM. This ROM area is not executable. | data |

Sections of a specified type are located by the linker in a memory space. The space names are defined in a so-called 'linker script file' (files with the extension .lsl) delivered with the product in the directory `installation-dir\include.lsl`.

**Example**

```
.SECTION .data                           ; Declare a .data section
    ;
.ENDSEC

.SECTION .rodata, cluster('5group_var')  ; Declare a section in ROM
    ;
.ENDSEC

.SECTION .data.abs, at(0x0)             ; Declare a .data.abs section at an absolute address
    ;
.ENDSEC
```

**Related Information**

`.OFFSET` *(Move location counter forwards)*
.SET

Syntax

symbol .SET  expression

.SET  symbol  expression

Description

With the .SET directive you assign the value of expression to symbol temporarily. If a symbol was defined with the .SET directive, you can redefine that symbol in another part of the assembly source, using the .SET directive again. Symbols that you define with the .SET directive are always local: you cannot define the symbol global with the .GLOBAL directive.

The .SET directive is useful in establishing temporary or reusable counters within macros. expression must be absolute and cannot include a symbol that is not yet defined (no forward references are allowed).

Example

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

Related Information

.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

```
.section    .text
.global    main
.arm
   .align    4

; Function main
main:    .type    func
   ;
   .SIZE    main,$-main
.endsec
```

Related Information

(TYPE (Set symbol type)
.SOURCE

Syntax

.SOURCE string

Description

With the .SOURCE 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

.SOURCE "main.c"
.TITLE

Syntax

.TITLE ["string"]

Default

.TITLE ""

Description

If you generate a list file with the assembler option --list-file, you can use the .TITLE directive to specify the program title which is printed at the top of each page in the assembler list file.

If you use the .TITLE directive without the argument, the title becomes empty. This is also the default. The specified title is valid until the assembler encounters a new .TITLE directive.

The .TITLE directive 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

.PAGE (Format the assembler list file)

Assembler option --list-file
.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 undefine a non-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**

`.DEFINE` (Define a substitution string)

`.MACRO, .ENDM` (Define a macro)
.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)
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.

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,

- **label** An optional label that corresponds to the value of the location counter at the start of the macro expansion.
- **macro_name** The name of the macro. This may not start in the first column.
argument One or more optional, substitutable arguments. Multiple arguments must be separated by commas.

comment 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 ?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: Argument Concatenation Operator - \**

Consider the following macro definition:

```assembly
MAC_A .MACRO reg,val
    sub  r\reg,r\reg,#val
.ENDM
```

The macro is called as follows:

```assembly
MAC_A 2,1
```

The macro expands as follows:

```assembly
    sub r2,r2,#1
```

The macro preprocessor substitutes the character '2' for the argument `reg`, and the character '1' for the argument `val`. The concatenation operator (`\`) indicates to the macro preprocessor that the substitution characters for the arguments are to be concatenated with the characters 'r'.

Without the `\` operator the macro would expand as:

```assembly
    sub rreg,rreg,#1
```

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 `MAC_A` after the argument `AVAL` has been set to 1.

```assembly
AVAL .SET 1
    MAC_A 2,AVAL
```

If you want to replace the argument `val` with the value of `AVAL` rather than with the literal string 'AVAL', you can use the `?` operator and modify the macro as follows:

```assembly
MAC_A .MACRO reg,val
    sub  r\reg,r\reg,#?val
.ENDM
```

**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:

```
STR_MAC .MACRO STRING
  .DB "STRING"
.ENDM
```

The macro is called as follows:

```
STR_MAC ABCD
```

The macro expands as follows:

```
  .DB '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:

```
  .DEFINE LONG 'short'
STR_MAC .MACRO STRING
  .MESSAGE I 'This is a LONG STRING'
  .MESSAGE I "This is a LONG STRING"
.ENDM
```

If the macro is called as follows:

```
STR_MAC sentence
```
it expands as:

```
.MESSAGE I 'This is a LONG STRING'
.MESSAGE I '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 LOCAL__M_L000001).

The macro ^-operator prevents name mangling on macro local labels.

Consider the following macro definition:

```
INIT .MACRO addr
LOCAL:  ldr  r0,^addr
.ENDM
```

The macro is called as follows:

```
LOCAL:
    INIT LOCAL
```

The macro expands as:

```
LOCAL__M_L000001: ldr  r0,LOCAL
```

If you would not have used the ^ operator, the macro preprocessor would choose another name for LOCAL because the label already exists. The macro would expand like:

```
LOCAL__M_L000001: ldr  r0,LOCAL__M_L000001
```

3.11. Generic Instructions

The assembler supports so-called 'generic instructions'. Generic instructions are pseudo instructions (no instructions from the instruction set). Depending on the situation in which a generic instruction is used, the assembler replaces the generic instruction with appropriate real assembly instruction(s).

3.11.1. ARM Generic Instructions

The ARM assembler recognizes the following generic instructions in ARM mode:

**ADR, ADRL, ADRLL ARM generics**

Load a PC-relative address into a register. The address is specified as a target label. The assembler generates one (ADR), two (ADRL) or three (ADRLL) generic DPR instruction (called ADR) and one, two or three PC-relative relocation types for the target label. The linker evaluates the relocation types (calculate the PC-relative offset) and generates one, two or three add or sub instructions each with an 8-bit immediate operand plus a 4-bit rotation. If the offset cannot be encoded the linker generates an error message.
### 3.11.2. ARM and Thumb-2 32-bit Generic Instructions

#### LDR= ARM and Thumb-2 generic

Load an address or a 32-bit constant value into a register. If the constant or its bitwise negation can be encoded, then the assembler will generate a MOV or a MVN instruction. Otherwise the assembler places the constant or the address in a literal pool and generates a PC-relative LDR instruction that loads the value from the literal pool.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Replacement</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR\texttt{cond} \texttt{Rd,=} \texttt{expr}</td>
<td>MOV\texttt{cond} \texttt{Rd, #expr}</td>
<td>If \texttt{expr} can be encoded</td>
</tr>
<tr>
<td></td>
<td>MVN\texttt{cond} \texttt{Rd,#@LSW(\neg(expr))}</td>
<td>If \texttt{\neg(expr)} can be encoded</td>
</tr>
<tr>
<td></td>
<td>LDR\texttt{cond} \texttt{Rd,\texttt{ltpool}; code\ltpool: .DW expr}</td>
<td>If \texttt{expr} is external or PC-relative, or cannot be encoded</td>
</tr>
</tbody>
</table>

The PC-relative offset from the LDR instruction to the value in the literal pool must be positive and less than 4 kB. By default the assembler will place a literal pool at the end of each code section. If the default literal pool is out-of-range you will have to ensure that there is another literal pool within range by means of the \texttt{.LTORG} directive.

#### VLDR= ARM and Thumb-2 generic

Load a 32-bit or 64-bit floating-point constant value into a register. The assembler places the constant in a literal pool and generates a PC-relative VLDR instruction that loads the value from the literal pool.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>VLDR\texttt{cond Sd,=} \texttt{expr}</td>
<td>VLDR\texttt{cond Sd,\texttt{ltpool}; code\ltpool: .FLOAT expr}</td>
</tr>
<tr>
<td>VLDR\texttt{cond Dd,=} \texttt{expr}</td>
<td>VLDR\texttt{cond Dd,\texttt{ltpool}; code\ltpool: .DOUBLE expr}</td>
</tr>
</tbody>
</table>
### MOV32 ARM and Thumb-2 generic

Load an address or a 32-bit constant value into a register.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Replacement</th>
<th>Remarks</th>
</tr>
</thead>
</table>
| MOV32cond Rd,=expr | MOVWcond Rd, #@LSH(expr)  
MOVTcond Rd, #@MSH(expr) | If expr is internal and absolute |
| MOVWcond Rd, #expr  
MOVTcond Rd, #expr | If expr is external or relocatable |

### ARM and Thumb-2 generic DPR inversions for immediate operands

For data processing instructions (DPR) which operate on an immediate operand, the operand value must be encoded as an 8-bit value plus a 4-bit even rotation value. If a value does not fit in such an encoding, it could be possible that the negated value (-value) or the bitwise negated value (~value) does fit in such an encoding. In that case the assembler will replace the DPR instruction by its inverse DPR instruction operating on the negated value.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Replacement (if #-imm or #~imm can be encoded)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDcond Rd,Rn,#imm32</td>
<td>SUBcond Rd,Rn,#-(imm32)</td>
</tr>
<tr>
<td>ADDcondS Rd,Rn,#imm32</td>
<td>SUBcondS Rd,Rn,#-(imm32)</td>
</tr>
<tr>
<td>ADDWcond Rd,Rn,#imm12</td>
<td>SUBWcond Rd,Rn,#-(imm12)</td>
</tr>
<tr>
<td>SUBcond Rd,Rn,#imm32</td>
<td>ADDcond Rd,Rn,#-(imm32)</td>
</tr>
<tr>
<td>SUBcondS Rd,Rn,#imm32</td>
<td>ADDcondS Rd,Rn,#-(imm32)</td>
</tr>
<tr>
<td>SUBWcond Rd,Rn,#imm12</td>
<td>ADDWcond Rd,Rn,#-(imm12)</td>
</tr>
<tr>
<td>ADCcond Rd,Rn,#imm32</td>
<td>SBCcond Rd,Rn,#-(imm32)</td>
</tr>
<tr>
<td>ADCcondS Rd,Rn,#imm32</td>
<td>SBCcondS Rd,Rn,#-(imm32)</td>
</tr>
<tr>
<td>SBCcond Rd,Rn,#imm32</td>
<td>ADCcond Rd,Rn,#-(imm32)</td>
</tr>
<tr>
<td>SBCcondS Rd,Rn,#imm32</td>
<td>ADCcondS Rd,Rn,#-(imm32)</td>
</tr>
<tr>
<td>ANDcond Rd,Rn,#imm32</td>
<td>BICcond Rd,Rn,#@LSW(-(imm32))</td>
</tr>
<tr>
<td>ANDcondS Rd,Rn,#imm32</td>
<td>BICcondS Rd,Rn,#@LSW(-(imm32))</td>
</tr>
<tr>
<td>BICcond Rd,Rn,#imm32</td>
<td>ANDcond Rd,Rn,#@LSW(-(imm32))</td>
</tr>
<tr>
<td>BICcondS Rd,Rn,#imm32</td>
<td>ANDcondS Rd,Rn,#@LSW(-(imm32))</td>
</tr>
<tr>
<td>CMNcond Rn,#imm32</td>
<td>CMPcond Rn,#-(imm)</td>
</tr>
<tr>
<td>CMPcond Rn,#imm32</td>
<td>CMNcond Rn,#-(imm)</td>
</tr>
<tr>
<td>MOVcond Rd,#imm32</td>
<td>MVNcond Rd,#@LSW(-(imm32))</td>
</tr>
<tr>
<td>MOVcondS Rd,#imm32</td>
<td>MVNcondS Rd,#@LSW(-(imm32))</td>
</tr>
<tr>
<td>MVNcond Rd,#imm32</td>
<td>MOVcond Rd,#@LSW(-(imm32))</td>
</tr>
<tr>
<td>MVNcondS Rd,#imm32</td>
<td>MOVcondS Rd,#@LSW(-(imm32))</td>
</tr>
</tbody>
</table>
Note that the built-in function @LSW() must be used on the bitwise negated immediate value because all values are interpreted by the assembler as 64-bit signed values. The @LSW() function returns the lowest 32 bits.

### 3.11.3. Thumb 16-bit Generic Instructions

The ARM assembler recognizes the following generic instructions in Thumb mode:

#### ADR Thumb 16-bit generic

Load a PC-relative address into a low register. The address is specified as a target label. The PC-relative offset must be less than 1 kB. The target label must be defined locally, must be word-aligned and must be in the same code section as the instruction. The assembler will not emit a relocation type for the target label. If the offset is out-of-range or the target label is external or in another section, then the assembler generates an error message.

#### LDR= Thumb 16-bit generic

Load an address or a 32-bit constant value into a low register. If the constant is in the range \([0,255]\) the assembler will generate a MOV instruction. Otherwise the assembler places the constant or the address in a literal pool and generates a PC-relative LDR instruction that loads the value from the literal pool.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Replacement</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR (Rd,=expr)</td>
<td>MOV (Rd, #expr)</td>
<td>If (expr) is in range</td>
</tr>
<tr>
<td></td>
<td></td>
<td>If (expr) is external or PC-relative, or not in range</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LDR (Rd,ltpool)</td>
<td>; code ltpool: .DW expr</td>
<td></td>
</tr>
</tbody>
</table>

The PC-relative offset from the LDR instruction to the value in the literal pool must be positive and less than 1 kB. By default the assembler will place a literal pool at the end of each code section. If the default literal pool is out-of-range you will have to ensure that there is another literal pool within range by means of the .LTORG directive.

#### Bcond inversion Thumb 16-bit generic

The PC-relative conditional branch instruction has a range of \((-256,+255)\) bytes. The unconditional version has a range of \((-2048,+2047)\) bytes. If the conditional branch target is out-of-range, the assembler will rewrite the conditional branch instruction with an inversed conditional branch and an unconditional branch.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Replacement</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bcond label</td>
<td>Binv_cond ~1 B label ~1:</td>
<td>If target (label) out-of-range</td>
</tr>
</tbody>
</table>
**ADD, SUB inversions Thumb 16-bit generic**

For the following six instructions the assembler accepts negative values for the immediate operand. If a negative value is specified, the assembler inverts the instruction from ADD to SUB or vice versa. For example: ADD R1,#-4 will be rewritten as SUB R1,#4.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD Rd,Rn,#imm</td>
<td>SUB Rd,Rn,#{imm}</td>
</tr>
<tr>
<td>ADD Rd,#imm</td>
<td>SUB Rd,#{imm}</td>
</tr>
<tr>
<td>ADD SP,#imm</td>
<td>SUB SP,#{imm}</td>
</tr>
<tr>
<td>SUB Rd,Rn,#imm</td>
<td>ADD Rd,Rn,#{imm}</td>
</tr>
<tr>
<td>SUB Rd,#imm</td>
<td>ADD Rd,#{imm}</td>
</tr>
<tr>
<td>SUB SP,#imm</td>
<td>ADD SP,#{imm}</td>
</tr>
</tbody>
</table>
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 ARM under Eclipse uses the TASKING makefile generator and make utility 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:

\[
\begin{align*}
\text{C source file} & \quad \text{.c} \\
\text{assembly file} & \quad \text{.src} \\
\text{compiler intermediate file} & \quad \text{.mil}
\end{align*}
\]

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 11.2, 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 in the ISO/IEC 9899:2011(E) standard.
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. When there are not enough physical registers, virtual registers are spilled to the stack. Intermediate results of any optimization can live, for some time, on the stack or in physical registers.

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 ARM under Eclipse uses the TASKING makefile generator and make utility 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 for** dialog.
Building a project under Eclipse

You have several ways of building your project:

* Build Selected File(s) (**1**). 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 (**2**).

To build individual projects incrementally, select Project » Build project.

* Rebuild Project (**3**). 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 Clean.

* Build Automatically. This performs a build of all projects 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. In order for this option to work, you must also enable option Build on resource save (Auto build) on the Behavior tab of the C/C++ Build page of the Project » Properties for dialog.

See also Chapter 12, Influencing the Build Time.

Select a target processor (core)

Processor options affect the invocation of all tools in the toolset. In Eclipse you need to set them for each configuration. 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.

You can specify the target processor when you create a new project with the New C/C++ Project wizard (File » New » TASKING ARM C/C++ Project), but you can always change the processor in the project properties dialog.

1. From the Project menu, select Properties for

   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 Configuration list, select a configuration or select [ All configurations ].

4. From the Processor selection list, select a processor.
To access the C/C++ compiler options

1. From the Project menu, select Properties for

   The Properties dialog appears.

2. In the left pane, expand C/C++ Build and select Settings.

   In the right pane the Settings appear.

3. From the Configuration list, select a configuration or select [ All configurations ].

4. On the Tool Settings tab, select C/C++ Compiler.

5. 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.

Note that when you click Restore Defaults to restore the default tool options, as a side effect the processor is also reset to its default value on the Processor page (C/C++ Build » Processor).

You can find a detailed description of all C compiler options in Section 11.2, C Compiler Options.

Invocation syntax on the command line:

carm [ [option]... [file]... ]...

4.3. 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 option --include-directory (-I)). If the option Add CMSIS include paths is enabled, this path is search first.
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 CARMINC.

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:

carm -Imyinclude test.c

First the compiler looks for the file stdio.h in the directory myinclude relative to the current directory. If it was not found, the compiler searches in the environment variable CARMINC 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 CARMINC and then in the default include directory.

4.4. 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 for

   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 4.5, Compiler Optimizations.

Invocation syntax on the command line

The invocation syntax on the command line is:

carm -g file.c

4.5. Compiler Optimizations

The compiler has a number of optimizations which you can enable or disable.

1. From the Project menu, select Properties for

   The Properties dialog appears.

2. In the left pane, expand C/C++ Build and select Settings.

   In the right pane the Settings appear.

3. From the Configuration list, select a configuration or select [ All configurations ].


5. 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 ...

#pragma optimize c  /* Enable common expression
... elimination. Expression
... C source ...
... simplification still enabled */

#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.

**4.5.1. 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).

A CSE can live in a register, on stack or can be recomputed when required.

**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 subscripting).

**Constant propagation (option -Op/-OP)**

A variable with a known value is replaced by that value.
Automatic function inlining (option -Oi/-O1)

Small functions that are not too often called, are inlined. This reduces execution time at the cost of code size.

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.

MIL linking (Control program option --mil-link)

The frontend phase performs its optimizations on the MIL code. When all C modules and/or MIL modules of an application are given to the C compiler in a single invocation, the C compiler will link MIL code of the modules to a complete application automatically. Next, the frontend will run its optimizations again with application scope. After this, the MIL code is passed on to the backend, which will generate a single .src file for the whole application. Linking with the run-time library, floating-point library and C library is still necessary. Linking with the C library is required because this library contains some hand-coded assembly functions, that are not linked in at MIL level.

In the ISO C99 standard a "translation unit" is a preprocessed source file together with all the headers and source files included via the preprocessing directive #include. After MIL linking the compiler will
treat the linked sources files as a single translation unit, allowing global optimizations to be performed, that otherwise would be limited to a single module.

**MIL splitting (option --mil-split)**

When you specify that the C compiler has to use MIL splitting, the C compiler will first link the application at MIL level as described above. However, after rerunning the optimizations the MIL code is not passed on to the backend. Instead the frontend writes a `.ms` file for each input file or library. A `.ms` file has the same format as a `.mil` file. Only `.ms` files that really change are updated. The advantage of this approach is that it is possible to use the make utility to translate only those parts of the application to a `.src` file that really have changed. MIL splitting is therefore a more efficient build process than MIL linking. The penalty for this is that the code compaction optimization in the backend does not have application scope. As with MIL linking, it is still required to link with the normal libraries to build an ELF file.
To read more about how MIL linking influences the build process of your application, see Section 12.1, MIL Linking.

Note that with both options some extra strict type checking is done that can cause building to fail in a way that is unforeseen and difficult to understand. For example, when you use one of these options in combination with option \(--schar\) and you link the MIL library, you might get the following error:

```
carm E289: ["..\..\..\..\strlen.c" 14/1] "strlen" redeclared with a different type
carm I802: ["installation-dir\include\string.h" 44/17]
    previous declaration of "strlen"
1 errors, 0 warnings
```

This is caused by the fact that the MIL library is built without \(--schar\). You can workaround this problem by rebuilding the MIL libraries.

4.5.2. 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 and code size.

Interprocedural register optimization (option \(-Ob/-OB\))

Register allocation is improved by taking note of register usage in functions called by a given function.
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.

Instruction Scheduler (option -Ok/-OK)

The instruction scheduler is a backend optimization that acts upon the generated instructions. 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.

First the instruction stream is partitioned into basic blocks. A new basic block starts at a label, or right after a jump instruction. Unschedulable instructions and, when -Av is enabled, instructions that access volatile objects, each get their own basic block. Next, the scheduler searches the instructions within a basic block, looking for places where the pipeline stalls. After identifying these places it tries to rebuild the basic block using the existing instructions, while avoiding the pipeline stalls. In this process data dependencies between instructions are honoured.

Note that the function inlining optimization happens in the frontend of the compiler. The instruction scheduler has no knowledge about the origin of the instructions.

Unroll small loops (option -Ou/-OU)

To reduce the number of branches, short loops are eliminated by replacing them with a number of copies.

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.

Code compaction (reverse inlining) (option -Or/-OR)

Compaction is the opposite of inlining functions: chunks of code that occur more than once, are transformed into a function. This reduces code size at the cost of execution speed. The size of the chunks of code to be inlined depends on the setting of the C compiler option --tradeoff (-t). See the subsection Code Compaction in Section 4.5.3, Optimize for Code Size or Execution Speed.

Generic assembly optimizations (option -Og/-OG)

A set of target independent optimizations that increase speed and decrease code size.

Cluster global variables (option -O+cluster/-O-cluster)

Global variables are accessed by first loading their address into a register and then accessing them via this register. Each address will result in an entry in the constant pool. By clustering global variables it is possible to access multiple variables using the same base register, which means we can lower the amount of entries in the constant pool. It also means that potentially we need less base registers. Clustering ensures that the linker locates the global variables together.
4.5.3. Optimize for Code Size or Execution 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 focuses 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.

Optimization hint: Optimizing for size has a speed penalty and vice versa. The advice is to optimize for size by default and only optimize those areas for speed that are critical for the application with respect to speed. Using the tradeoff options -t0, -t1 and -t2 globally for the application is not recommended.

To specify the size/speed trade-off optimization level:

1. From the Project menu, select Properties for
   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>
</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>
<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
ldr  r2,.L4
b    .L2         ;; unconditional jump to loop test at bottom
.L3:
    ldr  r3,[r2,#0]
    add  r1,r1,#1
    add  r3,r3,#1
    str  r3,[r2,#0]
.L2:       ;; loop entry point
```
Coded as a top loop (compiled with --tradeoff=0) is:

```assembly
    cmp   r1,r0       ;; test for at least one loop iteration
    ldr   r2,.L4      ;; can be omitted when number of iterations is known
    ldr   r3,[r2,#0]
    bge   .L2
    sub   r0,r0,r1
    .L3:                      ;; loop entry point
        subs  r0,r0,#1
        add   r0,r0,#1
        bgt   .L3
    .L2:
        str   r3,[r2,#0]
```

**Automatic Function Inlining**

You can enable automatic function inlining with the option `--optimize=+inline (-Oi)` or by using `#pragma optimize +inline`. This option is also part of the `-O3` predefined option set.

When automatic inlining is enabled, you can use the options `--inline-max-incr` and `--inline-max-size` (or their corresponding pragmas `inline_max_incr / inline_max_size`) to control automatic inlining. By default their values are set to -1. This means that the compiler will select a value depending upon the selected trade-off level. The defaults are:

<table>
<thead>
<tr>
<th>Trade-off value</th>
<th>inline-max-incr</th>
<th>inline-max-size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>100</td>
<td>50</td>
</tr>
<tr>
<td>1</td>
<td>50</td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

For example with trade-off value 1, the compiler inlines all functions that are smaller or equal to 25 internal compiler units. After that the compiler tries to inline even more functions as long as the function will not grow more than 50%.

When these options/pragmas are set to a value >= 0, the specified value is used instead of the values from the table above.

Static functions that are called only once, are always inlined, independent of the values chosen for `inline-max-incr` and `inline-max-size`.

**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.

See C compiler option --compact-max-size

4.6. Static Code Analysis

Static code analysis (SCA) is a relatively new feature in compilers. Various approaches and algorithms exist to perform SCA, each having specific pros and cons.

SCA Implementation Design Philosophy

SCA is implemented in the TASKING compiler based on the following design criteria:

• An SCA phase does not take up an excessive amount of execution time. Therefore, the SCA can be performed during a normal edit-compile-debug cycle.

• SCA is implemented in the compiler front-end. Therefore, no new makefiles or work procedures have to be developed to perform SCA.

• The number of emitted false positives is kept to a minimum. A false positive is a message that indicates that a correct code fragment contains a violation of a rule/recommendation. A number of warnings is issued in two variants, one variant when it is guaranteed that the rule is violated when the code is executed, and the other variant when the rules is potentially violated, as indicated by a preceding warning message.

For example see the following code fragment:

```c
extern int some_condition(int);
void f(void)
{
    char buf[10];
    int i;

    for (i = 0; i <= 10; i++)
    {
        if (some_condition(i))
        {
            buf[i] = 0; /* subscript may be out of bounds */
        }
    }
}
```
As you can see in this example, if \( i=10 \) the array \( \text{buf}[i] \) might be accessed beyond its upper boundary, depending on the result of \( \text{some\_condition}(i) \). If the compiler cannot determine the result of this function at run-time, the compiler issues the warning "subscript is \textit{possibly} out of bounds" preceding the CERT warning "ARR35: do not allow loops to iterate beyond the end of an array". If the compiler can determine the result, or if the \( \text{if} \) statement is omitted, the compiler can guarantee that the "subscript is out of bounds".

- The SCA implementation has real practical value in embedded system development. There are no real objective criteria to measure this claim. Therefore, the TASKING compilers support well known standards for safety critical software development such as the MISRA guidelines for creating software for safety critical automotive systems and secure "CERT C Secure Coding Standard" released by CERT. CERT is founded by the US government and studies internet and networked systems security vulnerabilities, and develops information to improve security.

**Effect of optimization level on SCA results**

The SCA implementation in the TASKING compilers has the following limitations:

- Some violations of rules will only be detected when a particular optimization is enabled, because they rely on the analysis done for that optimization, or on the transformations performed by that optimization. In particular, the constant propagation and the CSE/PRE optimizations are required for some checks. It is preferred that you enable these optimizations. These optimizations are enabled with the default setting of the optimization level (-O2).

- Some checks require cross-module inspections and violations will only be detected when multiple source files are compiled and linked together by the compiler in a single invocation.

### 4.6.1. C Code Checking: CERT C

The CERT C Secure Coding Standard provides rules and recommendations for secure coding in the C programming language. The goal of these rules and recommendations is to eliminate insecure coding practices and undefined behaviors that can lead to exploitable vulnerabilities. The application of the secure coding standard will lead to higher-quality systems that are robust and more resistant to attack.

For details about the standard, see the CERT C Secure Coding Standard web site. For general information about CERT secure coding, see [www.cert.org/secure-coding](http://www.cert.org/secure-coding).

**Versions of the CERT C standard**

Version 1.0 of the CERT C Secure Coding Standard is available as a book by Robert C. Seacord [Addison-Wesley]. Whereas the web site is a wiki and reflects the latest information, the book serves as a fixed point of reference for the development of compliant applications and source code analysis tools.

The rules and recommendations supported by the TASKING compiler reflect the version of the CERT web site as of June 1 2009.

The following rules/recommendations implemented by the TASKING compiler, are not part of the book: PRE11-C, FLP35-C, FLP36-C, MSC32-C

For a complete overview of the supported CERT C recommendations/rules by the TASKING compiler, see Chapter 20, *CERT C Secure Coding Standard*. 
Priority and Levels of CERT C

Each CERT C rule and recommendation has an assigned priority. Three values are assigned for each rule on a scale of 1 to 3 for

- severity - how serious are the consequences of the rule being ignored
  1. low (denial-of-service attack, abnormal termination)
  2. medium (data integrity violation, unintentional information disclosure)
  3. high (run arbitrary code)

- likelihood - how likely is it that a flaw introduced by ignoring the rule could lead to an exploitable vulnerability
  1. unlikely
  2. probable
  3. likely

- remediation cost - how expensive is it to comply with the rule
  1. high (manual detection and correction)
  2. medium (automatic detection and manual correction)
  3. low (automatic detection and correction)

The three values are then multiplied together for each rule. This product provides a measure that can be used in prioritizing the application of the rules. These products range from 1 to 27. Rules and recommendations with a priority in the range of 1-4 are level 3 rules (low severity, unlikely, expensive to repair flaws), 6-9 are level 2 (medium severity, probable, medium cost to repair flaws), and 12-27 are level 1 (high severity, likely, inexpensive to repair flaws).

The TASKING compiler checks most of the level 1 and some of the level 2 CERT C recommendations/rules.

For a complete overview of the supported CERT C recommendations/rules by the TASKING compiler, see Chapter 20, CERT C Secure Coding Standard.

To apply CERT C code checking to your application

1. From the Project menu, select Properties for
2. In the left pane, expand C/C++ Build and select Settings.
3. On the Tool Settings tab, select C/C++ Compiler » CERT C Secure Coding.
4. Make a selection from the CERT C secure code checking list.
5. If you selected Custom, expand the Custom CERT C entry and enable one or more individual recommendations/rules.

On the command line you can use the option --cert.

carm --cert={all | name [-name],...}

With --diag=cert you can see a list of the available checks, or you can use a three-letter mnemonic to list only the checks in a particular category. For example, --diag=pre lists all supported checks in the preprocessor category.

### 4.6.2. 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 and MISRA C: 2012, the latest version of MISRA C. You can select the version with the following C compiler option:

```
--misrac-version=1998
--misrac-version=2004
--misrac-version=2012
```

In your C source files you can check against the MISRA C version used. For example:

```
#if __MISRAC_VERSION__ == 1998
    ...
#elif __MISRAC_VERSION__ == 2004
    ...
#else __MISRAC_VERSION__ == 2012
    ...
#endif
```

For a complete overview of all MISRA C rules, see Chapter 21, *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 mandatory rules, 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:

--misrac-mandatory-warnings
--misrac-required-warnings
--misrac-advisory-warnings

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 for

   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 1998, Custom 2004 or Custom 2012 entry, specify the individual rules.

On the command line you can use the option --misrac.

carm --misrac={all | number [-number],...}

4.7. 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 for 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 » TASKING » 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:

carm --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 C++ compiler (cparm) is intermediate C, which can be translated with the C compiler (carm).

The C++ compiler is part of a complete toolset, the TASKING VX-toolset for ARM. For details about the C compiler see Chapter 4, Using the C Compiler.

The C++ compiler takes the following files for input and output:

C++ source file

\(.cc\)

\(. CPP\)

C++ compiler

 intermediate C file

\(.ic\)

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. 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. 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 for dialog.
Building a project under Eclipse

You have several ways of building your project:

• **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 **Clean**.

• **Build Automatically**. This performs a build of all projects 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. In order for this option to work, you must also enable option **Build on resource save (Auto build)** on the **Behavior** tab of the **C/C++ Build** page of the **Project » Properties for** dialog.

Select a target processor (core)

Processor options affect the invocation of all tools in the toolset. In Eclipse you need to set them for each configuration. 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 for**

   *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 **Configuration** list, select a configuration or select [ **All configurations** ].

4. From the **Processor selection** list, select a processor.

To access the C/C++ compiler options

1. From the **Project** menu, select **Properties for**

   *The Properties dialog appears.*
2. In the left pane, expand **C/C++ Build** and select **Settings**.  
   *In the right pane the Settings appear.*

3. From the **Configuration** list, select a configuration or select [ All configurations ].

4. On the Tool Settings tab, select **C/C++ Compiler**.

5. 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.

   Note that when you click **Restore Defaults** to restore the default tool options, as a side effect the processor is also reset to its default value on the **Processor** page (**C/C++ Build » Processor**).

You can find a detailed description of all C++ compiler options in **Section 11.3, C++ Compiler Options**.

**Invocation syntax on the command line:**

cparm [ [option]... [file]... ]...

**5.2. 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 CPARMINC.
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:

```c
#include <stdio.h>
#include "myinc.h"
```

You can call the C++ compiler as follows:

cparm -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 CPARMINC 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 CPARMINC and then in the default include.cpp and include directories.

5.3. 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 for 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 for 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 TASKING. Please include a small C++ program causing the error.

Message format

By default, diagnostics are written in a form like the following:

cparm E0020: ["test.cc" 3] identifier "name" is undefined

With the command line option --error-file=file you can redirect messages to a file instead of stderr.

Note that the message identifies the file and line involved. Long messages are wrapped to additional lines when necessary.

With the option C/C++ Build » Settings » Tool Settings » Global Options » Treat warnings as errors (option --warnings-as-errors) you can change the severity of warning messages to errors.

With the command line option --diag you can see a list of all messages.

For some messages, a list of entities is useful; they are listed following the initial error message:

cparm 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)
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:

```
cparm E0265: ["test.cc" 7] "A::A()" is inaccessible  
             detected during implicit generation of "B::B()" at line 7
```

Without the context information, it is very hard to figure out what the error refers to.

**Termination Messages**

The C++ compiler writes sign-off messages to `stderr` (the Problems view in Eclipse) if errors are detected. For example, one of the following forms of message

- `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".

is written to indicate the detection of errors in the compilation. No message is written if no errors were detected. The following message

```
Error limit reached.
```

is written when the count of errors reaches the error limit (see the option `--error-limit`); compilation is then terminated. The message

```
Compilation terminated.
```

is written at the end of a compilation that was prematurely terminated because of a catastrophic error.

The message

```
Compilation aborted
```

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 TASKING. Please include a small C++ program causing the error.
Chapter 6. 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:

```
assembly file (hand coded) .asm
```

```
assembly file .src
```

```
assembler
```

```
list file .lst
```

```
error messages .err
```

```
relocatable object file .obj
```

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 11.4, Assembler Options.
- How to generate a list file.
- Types of assembler messages.

6.1. Assembly Process

The assembler generates relocatable output files with the extension .obj. 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
- 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.
6.2. Assembler Versions

The TASKING VX-toolset for ARM consists of a set of three assemblers. Depending on the architecture and the selection of the Thumb or mixed ARM/Thumb instruction set Eclipse and the control program select the correct assembler, which results in faster build times.

<table>
<thead>
<tr>
<th>Assembler</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>asarm</td>
<td>supports both ARM and Thumb/Thumb-2 instruction set (full assembler)</td>
</tr>
<tr>
<td>asarma</td>
<td>supports ARM instruction set only</td>
</tr>
<tr>
<td>asarmt</td>
<td>supports Thumb/Thumb-2 instruction set only</td>
</tr>
</tbody>
</table>

All command line options are the same for all three assemblers.

Also see control program option --thumb.

6.3. Calling the Assembler

The TASKING VX-toolset for ARM under Eclipse uses the TASKING makefile generator and make utility 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 for dialog.

Building a project under Eclipse

You have several ways of building your project:

* 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 Clean.

* Build Automatically. This performs a build of all projects 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
Select a target processor (core)

Processor options affect the invocation of all tools in the toolset. In Eclipse you need to set them for each configuration. 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 for**
   
   *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 **Configuration** list, select a configuration or select `[ All configurations ]`.

4. From the **Processor selection** list, select a processor.

To access the assembler options

1. From the **Project** menu, select **Properties for**
   
   *The Properties dialog appears.*

2. In the left pane, expand **C/C++ Build** and select **Settings**.
   
   *In the right pane the Settings appear.*

3. From the **Configuration** list, select a configuration or select `[ All configurations ]`.

4. On the Tool Settings tab, select **Assembler**.

5. 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.**

   **Note that when you click Restore Defaults to restore the default tool options, as a side effect the processor is also reset to its default value on the Processor page (C/C++ Build » Processor).**

You can find a detailed description of all assembler options in Section 11.4, **Assembler Options**.

Invocation syntax on the command line:

```
asarm  [ [option]... [file]... ]...```
The input file must be an assembly source file (.asm or .src).

6.4. 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 option --include-directory (-I)).

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 ASARMINC.

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:

.INCLUDE 'myinc.inc'

You can call the assembler as follows:

asarm -Imyinclude 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 ASARMINC and then in the default include directory.

6.5. 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 for

   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.

**Example on the command line**

The following command generates the list file `test.lst`:

```
asarm -l test.asm
```

See Section 15.1, *Assembler List File Format*, for an explanation of the format of the list file.

### 6.6. 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 for menu (assembler option **--no-warnings**).

**Display detailed information on diagnostics**

1. From the **Window** menu, select **Show View » Other » TASKING » 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:

```
asarm --diag=[format:]{all | number,...}
```
Chapter 7. 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 (.obj files, generated by the assembler), and libraries into a single relocatable linker object file (.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:

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 Section 11.5, 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.

7.1. Linking Process

The linker combines and transforms relocatable object files (.obj) 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 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</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>
### Using the Linker

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relocation information</td>
<td>Information about how the linker must modify the machine code instructions when it relocates addresses.</td>
</tr>
<tr>
<td>Section</td>
<td>A group of instructions and/or data objects that occupy a contiguous range of addresses.</td>
</tr>
<tr>
<td>Section attributes</td>
<td>Attributes that define how the section should be linked or located.</td>
</tr>
<tr>
<td>Target</td>
<td>The hardware board on which an application is executing. A board contains at least one processor. However, a complex target may contain multiple processors and external memory and may be shared between processors.</td>
</tr>
<tr>
<td>Unresolved reference</td>
<td>A reference to a symbol for which the linker did not find a definition yet.</td>
</tr>
</tbody>
</table>

### 7.1.1. 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 (.obj) or libraries (.lib) 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.

7.1.2. 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 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 format (.abs) 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 7.8, Controlling the Linker with a Script.

### 7.2. 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 for dialog.

**Building a project under Eclipse**

You have several ways of building your project:

- Build Individual Project (⌘B).
  
  To build individual projects incrementally, select Project » Build project.

- Rebuild Project (⌘R). 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 of all projects 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. In order for this option to work, you must also enable option Build on resource save (Auto build) on the Behavior tab of the C/C++ Build page of the Project » Properties for dialog.

To access the linker options

1. From the Project menu, select Properties for

   The Properties dialog appears.

2. In the left pane, expand C/C++ Build and select Settings.

   In the right pane the Settings appear.

3. From the Configuration list, select a configuration or select [ All configurations ].

4. On the Tool Settings tab, select Linker.

5. Select the sub-entries and set the options in the various pages.

   Note that when you click Restore Defaults to restore the default tool options, as a side effect the processor is also reset to its default value on the Processor page (C/C++ Build » Processor).

You can find a detailed description of all linker options in Section 11.5, Linker Options.

Invocation syntax on the command line:

\[ \text{larm} \ [ \text{option}]\ldots \ [\text{file}]\ldots \ ]\ldots \]

When you are linking multiple files, either relocatable object files (.obj) or libraries (.lib), it is important to specify the files in the right order. This is explained in Section 7.3, Linking with Libraries.

Example:

\[ \text{larm} \ -darm.lsl \ test.obj \]

This links and locates the file test.obj and generates the file test.abs.

7.3. Linking with Libraries

There are two kinds of libraries: system libraries and user libraries.

System library

System libraries are stored in the directories:
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>carm[\s].lib</td>
<td>C libraries for ARM and Thumb instructions respectively</td>
</tr>
<tr>
<td>cthumb[\s].lib</td>
<td>Optional letter: s = single precision floating-point (control program option</td>
</tr>
<tr>
<td></td>
<td>--fp-model=+float)</td>
</tr>
<tr>
<td>fparm.lib</td>
<td>Floating-point libraries for ARM and Thumb</td>
</tr>
<tr>
<td>fpthumb.lib</td>
<td>Run-time library for ARM and Thumb</td>
</tr>
<tr>
<td>rtarm.lib</td>
<td>Profiling libraries for ARM and Thumb</td>
</tr>
<tr>
<td>rtthumb.lib</td>
<td>pb = block/function counter</td>
</tr>
<tr>
<td></td>
<td>pc = call graph</td>
</tr>
<tr>
<td></td>
<td>pct = call graph and timing</td>
</tr>
<tr>
<td></td>
<td>pd = dummy</td>
</tr>
<tr>
<td></td>
<td>pt = function timing</td>
</tr>
<tr>
<td>dspthumb[s].lib</td>
<td>CMSIS DSP libraries</td>
</tr>
<tr>
<td></td>
<td>Optional letter: s = single precision floating-point</td>
</tr>
<tr>
<td>cparm[s</td>
<td>x].lib</td>
</tr>
<tr>
<td>cpthumb[s</td>
<td>x].lib</td>
</tr>
<tr>
<td></td>
<td>x = exception handling</td>
</tr>
<tr>
<td>stlarmx.lib</td>
<td>STLport C++ libraries (exception handling variants only)</td>
</tr>
<tr>
<td>stlthumbx.lib</td>
<td></td>
</tr>
</tbody>
</table>

**To link the default C (system) libraries**

1. From the **Project** menu, select **Properties for**

   *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 » Libraries**.

4. Enable the option **Link default libraries**.

   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 **carm.lib**, type:
To use the CMSIS DSP library in your Eclipse project

Part of the CMSIS standard is a DSP library. The CMSIS DSP library is included in the TASKING product's cmsis folder. The library is also available as a pre-built library file in the lib folders for v6M, v7M and v7EM. The libraries are dsphthumb.lib (double precision floating point) and dsptthumbs.lib (single precision floating point). MIL libraries are present also: dsptthumb.ma and dsptthumbs.ma.

1. From the Project menu, select Properties for
   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 » Include Paths.
4. Enable the option Add CMSIS include paths.
5. On the Tool Settings tab, select Linker » Libraries.
6. Enable the option Link CMSIS DSP library.

This passes the option --dsp-library of the control program (ccarm) for compilation of C/C++ files and for linking. With this option the control program sets the compiler macro ARM_MATH_CM0, ARM_MATH_CM0PLUS, ARM_MATH_CM3, ARM_MATH_CM4 or ARM_MATH_CM7, depending on the selected processor. These macros are required for the CMSIS arm_math.h header file to operate correctly. With --dsp-library the control program also selects the appropriate library.

User library

You can create your own libraries. Section 9.4, 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 for
   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:
Using the Linker

lkarm start.obj mylib.lib

If the library resides in a sub-directory, specify that directory with the library name:

lkarm start.obj mylibs\mylib.lib

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:

lkarm --first-library-first a.lib test.obj b.lib

If the file `test.obj` calls a function which is both present in `a.lib` and `b.lib`, normally the function in `b.lib` would be extracted. With this option the linker first tries to extract the symbol from the first library `a.lib`.

Note that routines in `b.lib` that call other routines that are present in both `a.lib` and `b.lib` are now also resolved from `a.lib`.

7.3.1. 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 option `--library-directory (-L)`). 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 variable `LIBARM`.

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.
7.3.2. How the Linker Extracts Objects from Libraries

A library built with the TASKING archiver ararm 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:

```bash
lkarm mylib.lib
```

nothing is linked and no output file will be produced, because there are no unresolved symbols when the linker searches through mylib.lib.

It is possible to force a symbol as external (unresolved symbol) with the option --extern (-e):

```bash
lkarm --extern=main mylib.lib
```

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.lib. This process repeats until no new unresolved symbols are found.

7.4. Incremental Linking

With the TASKING linker it is possible to link incrementally. Incremental linking means that you link some, but not all .obj 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 .obj files as the .out file you had created with the first invocation.

Incremental linking is only possible on the command line.

```bash
lkarm -darm.lsl --incremental test1.obj -otest.out
lkarm -darm.lsl test2.obj test.out
```
This links the file `test1.obj` and generates the file `test.out`. This file is used again and linked together with `test2.obj` to create the file `test.abs` (the default name if no output filename is given in the default ELF/DWARF format).

With incremental linking it is normal to have unresolved references in the output file until all `.obj` files are linked and the final `.out` or `.abs` file has been reached. The option `--incremental (-r)` for incremental linking also suppresses warnings and errors because of unresolved symbols.

### 7.5. Linking Core-Specific Projects into a Multi-Core Application

The TASKING toolset for ARM has support for support for systems where multiple ARM cores share memory. To build an application for such a multi-core processor you must specify the correct processor in Eclipse and select a multi-core configuration or you need to specify the processor to the control program (control program option `--cpu`) and specify an LSL core (control program option `--lsl-core=core`).

**To create a single-core project**

If you want to build an application for a specific core (CM0, CM4 or CM7, CM7_0 or CM7_1) with its own code and data (a separate namespace), you need to select the specific core in Eclipse (for CM7 Core CM7_0 is the default).

1. From the Project menu, select Properties for

   *The Properties dialog appears.*

2. In the left pane, expand C/C++ Build and select Processor.

   *In the right pane the Processor settings appear.*

3. From the Processor Selection list, select a processor.

4. From the Multi-core configuration list, select an ARM core.

If you build your sources on the command line with the control program, apart from `--cpu` you also have to specify control program option `--lsl-core=core`. Your main project should always be a CM0 project that has project references to core-specific projects.

**To create a virtual-core project**

1. From the Project menu, select Properties for

   *The Properties dialog appears.*

2. In the left pane, expand C/C++ Build and select Processor.

   *In the right pane the Processor settings appear.*

3. From the Processor Selection list, select a multi-core processor.

4. From the Multi-core configuration list, select All cores.
With the control program apart from \texttt{-cpu} you also need to specify control program option \texttt{--lsl-core=vm7}.

The macro \texttt{__VM7} is defined for a virtual-core CM7 project.

**Add the core-specific projects to the main CM0 project**

1. In the C/C++ Projects view, right-click on the name of main CM0 ARM project and select Properties. The Properties dialog appears.

2. In the left pane, select Project References.

3. In the right pane, select the core-specific projects (single-core or virtual-core) that must be part of the ARM project and click Apply and Close.

**Build the project**

1. Make the CM0 project the active project.

2. From the Project menu, select Build project.

Eclipse will create linker .out files for the core-specific projects other than CM0. They will be linked to the main CM0 project to produce the final ELF absolute object file.

When you build your project, the linker is called with linker option \texttt{--core=ARM:core}.

**Using the control program**

With the control program you need to use control program option \texttt{--lsl-core=core} and control program option \texttt{--new-task}.

1. Build the core projects other than CM0 to .out files. For example,
   
   
   
   ```
   ccarm --processors="vendor_support_dir\etc\processors.xml"
   --cpu=cyt4bf8ceb_m7 -Wl-I"vendor_support_dir\include.lsl"
   --lsl-core=cm7_0 source_cm7_0.c --link-only -t -o cm7_0.out
   ```

2. Build the CM0 project with link tasks for each core. For example,
   
   ```
   ccarm --processors="vendor_support_dir\etc\processors.xml"
   --cpu=cyt4bf8ceb_m0 -Wl-I"vendor_support_dir\include.lsl"
   --lsl-core=cm0 mtproject.c --new-task=cm7_0,cm7_0.out
   ```

**7.6. 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.
Add a data object in Eclipse

1. Select Linker » Data Objects.
   
   The Data objects box shows the list of object files that are imported.

2. To add a data object, click on the Add button in the Data objects box.

3. Type or select a binary file (including its path).

On the command line you can add raw data to your application with the linker option --import-object.

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:

```c
#include <stdio.h>
extern char   _lc_ub_my_mp3; /* linker labels */
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;
    for (i=0;i<size;i++)
        putchar(mp3[i]);
}
```

If you want to use the export functionality of Eclipse, the binary file has to be part of your project.

7.7. 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 for
   
   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 unreferenced sections from the resulting object file.

This optimization considers a section referenced if either of the following two conditions is true:

1. The section is protected from unreferenced section removal, which can be one of:
   - the section is assigned an absolute address, either in the object file or in LSL
   - the section is selected by exact name in LSL (no wildcard pattern)
   - a symbol defined in the section is referenced in LSL
   - the section has the 'protected' section flag set, either in the object file or in LSL

2. The section is referenced via a relocation by another section that is considered referenced.

   If multiple sections of a specific name are created by using section renaming, all of these sections are protected against unreferenced section removal. With a selection using wildcards, matching sections are selected, but matching sections that are unreferenced may be removed. See Selecting sections for a group in Section 17.8.2, Creating and Locating Groups of Sections.

**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.

Note that this optimization only affects unrestricted sections that require an initialization action in the copy table. The affected sections get a clustered restriction. Unrestricted sections are sections that do not have their absolute location or their relative location to other sections restricted. See also Define the mutual order of sections in an LSL group in Section 17.8.2, Creating and Locating Groups of Sections.

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.

Compress ROM sections of copy table items (option -Oz/-OZ)

Reduces the size of the application's ROM image by compressing the ROM image of initialized data sections. At application startup time the ROM image is decompressed and copied to RAM.

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.

7.8. 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.

7.8.1. 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 TASKING 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 Chapter 17, *Linker Script Language (LSL)*.

### 7.8.2. 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 » TASKING ARM 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.
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 » Linker Script File (LSL).**

**To change the Linker Script File in Eclipse**

There are two ways of changing the LSL file in Eclipse.

- You can change the LSL file directly in an editor.
  1. Double-click on the file *project_name.lsl.*

   *The project LSL file opens in the editor area.*
2. You can edit the LSL file directly in the project_name.lsl editor.

   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 also make changes to the property pages Memory and Stack/Heap.

   1. From the Project menu, select Properties for

      The Properties dialog appears.

   2. In the left pane, expand C/C++ Build and select Memory or Stack/Heap.

      In the right pane the corresponding property page appears.

   3. Make changes to memory and/or stack/heap and click Apply and Close.

      The project LSL file is updated automatically according to the changes you make in the pages.

You can quickly navigate through the LSL file by using the Outline view (Window » Show View » Outline).

### 7.8.3. 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 vector table.

This specification is normally written by TASKING. TASKING 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.

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.

**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 ARM architecture, its external memory and how sections are located in memory, may have the following skeleton:

```plaintext
architecture ARM
{
    // Specification of the ARM core architecture.
    // Written by TASKING.
}

derivative X  // derivative name is arbitrary
{
    // Specification of the derivative.
    // Written by TASKING.
    core vm7  // always specify the core
    {
        architecture = ARM;
    }

    bus local_bus  // local bus
    {
        // maps to bus "local_bus" in "ARM" core
    }

    // internal memory
}

processor ARM  // 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 :vm7:linear  // section layout
```
Overview of LSL files delivered by TASKING

TASKING 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>arm_arch.lsl</td>
<td>Defines the base architecture (ARM) for all cores.</td>
</tr>
<tr>
<td>arm_mc_arch.lsl</td>
<td>Defines the base architecture (ARM) for all multi-core architectures.</td>
</tr>
<tr>
<td>arm.lsl</td>
<td>It includes the file <code>arm_arch.lsl</code> and contains a default specification of the external memory attached to the target processor.</td>
</tr>
<tr>
<td>default.lsl</td>
<td>Default LSL file. This file includes the file <code>arm.lsl</code> or <code>device.lsl</code> if <code>__DEVICE_LSL_FILE</code> is defined.</td>
</tr>
<tr>
<td>device.lsl</td>
<td>This file includes a processor specific LSL file based on the selected processor. See control program option <code>--cpu</code>.</td>
</tr>
<tr>
<td>processor.lsl</td>
<td>Processor specific LSL file with a specification of the external memory attached to the target processor. It includes the file <code>arm_arch.lsl</code>.</td>
</tr>
<tr>
<td>template.lsl</td>
<td>This file is used by Eclipse as a template for the project LSL file. It includes the file <code>device.lsl</code>.</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)`.

7.8.4. The Architecture Definition

Although you will probably not need to write an 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, separate
spaces for code and data. 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 or 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 ARM as defined in arm_arch.lsl.

<table>
<thead>
<tr>
<th>Space</th>
<th>Id</th>
<th>MAU</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>linear</td>
<td>1</td>
<td>8</td>
<td>Linear address space.</td>
</tr>
</tbody>
</table>

The ARM architecture in LSL notation

The best way to write the architecture definition, is to start with a drawing. The following figure shows a part of the ARM architecture:

The figure shows one address space called linear. The address space has attributes like a number that identifies the logical space (id), a MAU and an alignment. In LSL notation the definition of this address space looks as follows:

```lsl
space linear
{  
id = 1;
  mau = 8;

  map (size=4G, dest=bus:local_bus);
}
```

The keyword `map` corresponds with the arrows in the drawing. You can map:

- address space => address space (not shown in the drawing)
- address space => bus
- memory => bus (not shown in the drawing)
Next the internal bus, named `local_bus` must be defined in LSL:

```idl
bus local_bus
{
    mau = 8;
    width = 32;  // there are 32 data lines on the bus
}
```

This completes the LSL code in the architecture definition. Note that all code above goes into the architecture definition, thus between:

```idl
architecture ARM
{
    // All code above goes here.
}
```

### 7.8.5. The Derivative Definition

Although you will probably not need to write a derivative definition (unless you are using multiple cores that both access the same memory device) 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

### 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).

A link task (resolving symbols from object files and libraries) is associated with one core in the LSL file(s). In a multi-core environment you can combine multiple cores with the same architecture into a single link task. This is done by importing one or more cores into a root core with an `import` statement. By importing a core the hardware resources of that core are made available to the link task associated with the core that contains the `import` statement. The imported cores share a single symbol namespace. The address spaces in each imported core must have a unique ID in the link task. For each imported core is specified that the space IDs of the imported core start at a specific offset.

The following example is part of `cyt4bf.lsl` delivered with the product.

```idl
core cm0
{
    architecture = ARM;
}
```
core vm7
{
    architecture = ARM;
    import cm7_0;
    import cm7_1;
}

core cm7_0
{
    architecture = ARM;
    space_id_offset = 10; // add 10 to all address space IDs
        // in the architecture definition
}

core cm7_1
{
    architecture = ARM;
    space_id_offset = 20; // add 20 to all address space IDs
        // in the architecture definition
}

Bus

Each derivative can contain a bus definition for connecting external memory. In this example, the bus
local_bus maps to the bus local_bus defined in the architecture definition of cores cm0, vm7, cm7_0
and cm7_1:

bus local_bus
{
    mau = 8;
    width = 32;
    map (size=4G-64k, src_offset=64k, dest_offset=64k, dest=bus:cm0:local_bus);
    map (size=4G-64k, src_offset=64k, dest_offset=64k, dest=bus:vm7:local_bus);
    map (size=4G-64k, src_offset=64k, dest_offset=64k, dest=bus:cm7_0:local_bus);
    map (size=4G-64k, src_offset=64k, dest_offset=64k, dest=bus:cm7_1:local_bus);
}

Memory

Memory is usually described in a separate memory definition, but you can specify on-chip memory for a
derivative. For example:

memory internal_code_rom
{
    mau = 8;
    type = rom;
    size = 2k;
    map( dest=bus:cm0:local_bus, size = 2k, dest_offset = 0x00100000);
        // src_offset is zero by default
}
This completes the LSL code in the derivative definition. Note that all code above goes into the derivative definition, thus between:

```plaintext
derivative X    // name of derivative
{
    // All code above goes here
}
```

### 7.8.6. 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.

```plaintext
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.

### 7.8.7. 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:

```plaintext
memory name
{
    // memory definitions
}
```

Suppose your embedded system has 512kB of external ROM, named `simrom`, 512kB of external RAM, named `simram` and 32kB of external NVRAM, named `my_nvram` (see figure above.) All memories are connected to the bus `local_bus`. In LSL this looks like follows:

```plaintext
memory simrom
{
    mau = 8;
}
```
TASKING VX-toolset for ARM User Guide

```c
memory simram
{
    mau = 8;
    type = ram;
    size = 512k;
    map ( size = 512k, dest_offset=512k, dest=bus:X:local_bus);
}

memory my_nvram
{
    mau = 8;
    size = 32k;
    type = ram;
    map ( size = 32k, dest_offset=1M, dest=bus:X:local_bus);
}
```

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. From the **Project** menu, select **Properties for**

   *The Properties dialog appears.*

2. In the left pane, expand **C/C++ Build** and select **Memory**.

   *In the right pane the Memory page appears.*

3. Open the **Memory** tab and click on the **Add...** button.

   *The Add new memory dialog appears.*

4. Enter the memory name (for example `my_nvram`), type (for example `nvram`) and size.

5. Click on the **Add...** button.

   *The Add new mapping dialog appears.*

6. You have to specify at least one mapping. Enter the mapping name (optional), address, size and destination and click **OK**.

   *The new mapping is added to the list of mappings.*

7. Click **OK**.
The new memory is added to the list of memories (user memory).

8. Click **Apply and Close** to close the Properties dialog.

The updated settings are stored in the project LSL file.

If you make changes to the on-chip memory as defined in the architecture LSL file, the memory is copied to your project LSL file and the line `#define __MEMORY` is added. If you remove all the on-chip memory from your project LSL file, also make sure you remove this define.

**7.8.8. 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 "non_volatile"
int battery_backup_tag;
int battery_backup_invok;
#pragma endsection

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++);
}
```

Using the Linker
The compiler assigns names and attributes to sections. With the `#pragma section` and `#pragma endsection` the compiler's default section naming convention is overruled and a section with the name `non_volatile` appended is defined. In this section the battery back-upped data is stored.

As a result of the `#pragma section "non_volatile"`, the data objects between the pragma pair are placed in a section with the name `".bss.non_volatile"`. Note that `".bss"` sections are cleared at startup. However, battery back-upped sections should not be cleared and therefore we will change this section attribute using the LSL.

### 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_nvram` from the example in Section 7.8.7, *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 `linear`:

```plaintext
section_layout ::linear
{
    // 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 `.bss.non_volatile`. All other sections are located using the defaults specified in the architecture definition. Section `.bss.non_volatile` should be placed in non-volatile ram. To achieve this, the run address refers to our non-volatile memory called `my_nvram`. Furthermore, the section should not be cleared and therefore the attribute `s` (scratch) is assigned to the group:

```plaintext
group ( run_addr = mem:my_nvram, attributes = rws )
{
    select " .bss.non_volatile";
}
```

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 17, *Linker Script Language (LSL)*.

### 7.9. Locating in a Multi-core Processor Environment

#### Locating in core local RAM with link time core association

For ARM derivatives that have multi-core support, instead of determining the core at compile time, for example by using `__tcm0` in your C source, you can use link time core association: omit the keyword in your C source and locate the section directly at the correct location in RAM as follows:
section_layout ARM:vm7:linear  // core vm7 containing all address spaces
{                                      // of all cores
    group itcm70 ( run_addr = mem:ARM:CYT4BF_CM7_0_ITCM, copy ) // cm7_0 memory ITCM
    {                      // select sections for itcm0
        select "*.itcm70";
    }
}

The `copy` keyword tells to copy the section from ROM to RAM at program startup.

**Locating clone sections**

Instead of using the `__clone` keyword in C you can create a clone section in LSL with the `section_setup` keyword in combination with `modify input`. For more details about these LSL keywords, see Section 17.7.1, Setting up a Section. The following example shows a definition for a multi-core ARM with two cores with clone sections in address space `linear` in both cores.

```
section_setup :vm7:linear
{                                  // modify input (space = ARM:vm7:ARM_cm7_0_linear|ARM_cm7_1_linear)
    {                              // select ".text.file_1.func_1"
    }
}
```

In order to locate this section e.g. at a dedicated address in the core local memory use an entry like:

```
section_layout :vm7:ARM_cm7_0_linear|ARM_cm7_1_linear
{                              // group MY_CLONE_FUNCTION (ordered, run_addr = 0xC0000800)
    {                      // select ".text.file_1.func_1"
    }
}
```

### 7.10. 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></td>
</tr>
<tr>
<td>_lc_ue_name</td>
<td>End of section name. Also used to mark the end of the stack or heap. It points to the section address + section size, in other words the first MAU behind the section.</td>
</tr>
<tr>
<td>_lc_e_name</td>
<td></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. It points to the first MAU behind the last section in the group. This label appears in the output file even if no reference to the label exists in the input file.</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.

**Example: refer to a label with section name with dots from C**

Suppose a section has the name .text. When you want to refer to the begin of this section you have to replace all dots in the section name by underscores:

```c
#include <stdio.h>
extern char _lc_ub__text[];

void main(void)
{
    printf("The function main is located at %p\n",
            _lc_ub__text);
}
```

**Example: refer to the stack**

Suppose in an LSL file a stack section is defined with the name "stack" (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_stack[];
extern char _lc_ue_stack[];
void main()
{
    printf( "Size of stack is %d\n",
```
From assembly you can refer to the end of the stack with:

```c
.extern _lc_ue_stack  ; end of user stack
```

### 7.11. 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.

When the linker works on more than one task, a map file can be created for each of the tasks. There is also an option to create one global map file that includes information for all tasks involved. Use the linker option `--global-map-file` to generate the global map file. This map file format is very similar to that of the map file for a single task.

**To generate a map file**

1. From the **Project** menu, select **Properties for**
   
   *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**

The following command generates the map file `test.map`:

```bash
lkarm --map-file test.obj
```

With this command the map file `test.map` is created.

See Section 15.2, *Linker Map File Format* for an explanation of the format of the map file.

### 7.12. 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 for 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 » TASKING » 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:

```
lkarm --diag=[format:][all | number,...]
```
Chapter 8. Run-time Environment

This chapter describes the startup code used by the TASKING VX-toolset for ARM C Compiler, the vector table, the stack layout and the heap.

8.1. Startup Code

You need the run-time startup code to build an executable application. The default startup code consists of the following components:

- **Initialization code.** This code is executed when the program is initiated and before the function `main()` is called.

- **Exit code.** This controls the close down of the application after the program's main function terminates.

The startup code is part of the C library, and the source is present in the file `cstart.c` (Thumb2 specific) in the directory `lib\src`.

The startup code is generic code. It uses linker generated symbols which you can give target specific or application specific values. These symbols are defined in the linker script file `include.lsl\arm_arch.lsl` and you can specify their values in Eclipse or on the command line with linker option **--define**. If the default run-time startup code does not match your configuration, you need to make a copy of the startup file, modify it and add it to your project. A typical example for doing this is when `main()` has arguments, typically argc/argv. In this case cstart needs to be recompiled with the macro **__USE_ARGC_ARGV** set. When necessary you can use the macro **__ARGCV_BUFSIZE** to define the size of the buffer used to pass arguments to `main()`.

The entry point of the startup code (reset handler) is label `Reset_Handler`. This global label should not be removed, since the linker uses it in the linker script file. It is also used as the default start address of the application.

**Initialization code**

The following initialization actions are executed before the application starts:

- Load the ‘real’ program address. This assures that the reset handler is immune for any ROM/RAM re-mapping.

- Initialize the stack pointer. The stack pointer is loaded in memory by the stack address located at a linker generated symbol `_lc_ub_stack`. This symbol is defined in the linker script file. See Section 8.4, **Stack and Heap**, for detailed information on the stack.

- Call a user function which initializes hardware. The startup code calls the function `SystemInit`. This function has an empty implementation in the C library, which you should overrule if certain hardware initializations, such as ROM/RAM re-mapping or MMU configuration, are required before calling the main application.

- Copy initialized sections from ROM to RAM, using a linker generated table (also known as the ‘copy table’) and clear uninitialized data sections in RAM.
• Switch to the user-defined application mode as defined through the symbol \_APPLICATION\_MODE\_ in the LSL file. This symbol is used to set the value of the CPSR status register before calling the function \texttt{main}.

• Initialize profiling if profiling is enabled. For an extensive description of profiling refer to Chapter 13, \textit{Profiling}.

• Initialize the \texttt{argc} and \texttt{argv} arguments.

• Call the entry point of your application with a call to function \texttt{main()}.  

\textbf{Exit code}

When the C application 'returns', which is not likely to happen in an embedded environment, the program ends with a call to the library function \texttt{exit()}.  

\textbf{Macro preprocessor symbols}

A number of macro preprocessor symbols are used in the startup code. These are enabled when you use a particular option or you can enable or disable them using the assembler option \texttt{--define} with the following syntax:

\texttt{--define=\textit{symbol} [=\textit{value}]}

In the startup file (\texttt{cstart.c}) the following macro preprocessor symbols are used:

\begin{tabular}{|c|l|}
\hline
\textbf{Define} & \textbf{Description} \\
\hline
\_PROF\_ENABLE\_ & If defined, initialize profiling. \\
\_POSIX\_ & If defined, call posix\_main instead of \texttt{main}. \\
\_USE\_ARGC\_ARGV & If defined, pass arguments to \texttt{main}: \texttt{int main ( int argc, char *argv[] )}.
\_ARGCV\_BUFSIZE & Define buffer size for \texttt{argv}. (default: 256 bytes) \\
\hline
\end{tabular}

\section*{8.2. Reset Handler and Vector Table}

\textbf{Reset handler}

As explained in the previous section the entry point of the startup code (reset handler) is label \texttt{Reset\_Handler}. The reset handler can have a fixed ROM address (run address). If the reset handler is called from the vector table, you do not need the specify a fixed address. In this case the linker determines the address and patches the vector table. There are however situations were you have to specify a fixed ROM address:

• If \texttt{Reset\_Handler} is the entry point upon reset. Typically you would set the ROM address to the address which is mapped at address 0x00000000. Your initialization code remaps this address during startup. Note that the reset handler in the run-time library is immune to this remapping because the first instruction in the startup code sets the program counter to the actual ROM address.
Reset handler on fixed ROM address (all architectures)

To force the reset handler on a fixed ROM address, you need to define the symbol `__START`. This symbol is used in the linker script file `arm_arch.lsl`. By default, `__START` is not defined.

To define a symbol for the linker script file

1. From the **Project** menu, select **Properties for**
   
   *The Properties dialog appears.*

2. In the left pane, expand **C/C++ Build** and select **Settings**.
   
   *In the right pane the Settings appear.*

3. Select **Linker » Script File**.
   
   *The Defined symbols box shows the symbols that are currently defined.*

4. To define a new symbol, click on the **Add** button in the **Defined symbols** box.

5. Type the symbol definition (for example, `__START=0x0`).

The following table contains an overview of the defines you can set. The defines are used in `arm_arch.lsl`.

<table>
<thead>
<tr>
<th>Define</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__START</code></td>
<td>Reset handler ROM address</td>
</tr>
<tr>
<td><code>__NO_AUTO_VECTORS</code></td>
<td>If defined, the vector table will not be generated. Define this macro when you do not want to generate a vector table automatically.</td>
</tr>
<tr>
<td><code>__NO_DEFAULT_AUTO_VECTORS</code></td>
<td>If defined, the vector table will not be generated. Define this macro when you define your own vectors.</td>
</tr>
<tr>
<td><code>__NR_OF_VECTORS</code></td>
<td>Number of vectors (default 48).</td>
</tr>
<tr>
<td><code>__SET_VTOR_BIT29</code></td>
<td>Define this macro if bit 29 of the VTOR SFR indicates that the vector table is in RAM.</td>
</tr>
<tr>
<td><code>__VECTOR_TABLE_ROM_ADDR</code></td>
<td>Address of the vector table in ROM (default 0x00000000).</td>
</tr>
<tr>
<td><code>__MTB_SIZE</code></td>
<td>Size of the Micro Trace Buffer (MTB). This allows tracing on Cortex-M0+ devices that support MTB. To use this debugging facility, define this macro to a value that is a power of 2 and at least 32.</td>
</tr>
</tbody>
</table>

**Vector table**

By default the linker can generate a vector table, unless you define the symbol `__NO_AUTO_VECTORS` or `__NO_DEFAULT_AUTO_VECTORS`. 
The vector table contains the initialization value for the stack pointer, and the entry point addresses of each exception handler. The exception number (vector number) also defines the order of entries in the vector table.

<table>
<thead>
<tr>
<th>Vector number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Initialization value of the stack pointer</td>
</tr>
<tr>
<td>1</td>
<td>Reset handler</td>
</tr>
<tr>
<td>( n )</td>
<td>Exception using that exception number ( n ). Numbers 16 and higher are for user interrupts.</td>
</tr>
</tbody>
</table>

The linker will look for specific symbols designating the start of a handler function. These symbols are generated by the compiler when one of the following function qualifiers is used:

<table>
<thead>
<tr>
<th>Function type qualifier</th>
<th>Vector symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>_interrupt ( (n) )</td>
<td>_vector ( n )</td>
</tr>
<tr>
<td>_interruptc ( n )</td>
<td>_vectorc ( n )</td>
</tr>
</tbody>
</table>

Note that the reset handler is designated by the symbol `Reset_Handler` instead of `\_vector\_1`.

Note that if you have more than one handler for the same exception and you are using the \_interrupt \( (n) \) or \_interruptc \( n \) function qualifier of the compiler, you will need to specify the \_novector attribute in order to prevent the compiler from generating the \_vector\_\( n \) symbol multiple times, as this would lead to a link error.

See also Section 1.12.3.1, Defining an Exception Handler: \_interrupt Keywords.

**Vector table size (M-profile architectures)**

The vector table size for M-profile architectures is calculated as 4 times the number of vectors. The default number of vectors is 48, but you can specify another value by defining the symbol \_NR\_OF\_VECTORS.

**Vector table ROM address (all architectures)**

The ROM address of the vector table is usually address 0x00000000. You have to specify an address if the vector table will be copied from ROM to RAM (address 0x00000000 is mapped to RAM) or if the hardware uses high vectors at address 0xFFFF0000. If you forced the reset handler on address 0x00000000 then you also have to specify a vector table ROM address to prevent overlapping address ranges.

By default, the symbol \_VECTOR\_TABLE\_ROM\_ADDR is defined as 0x00000000.
8.3. CMSIS Support

The interrupt vector table, required for CMSIS, is defined in device specific LSL files. These LSL files are available in Eclipse (and can be updated by the Eclipse update feature) or can be downloaded as a vendor support archive from the TASKING website (www.tasking.com).

Device LSL files are similarly named as the CMSIS header files. For example when you use `stm32f10x.h` the LSL file `stm32f10x.lsl` is available. The device LSL files include the file `arm_arch.lsl`. You can control the allocated amount of flash and SRAM by using defines for the linker. The names of these defines vary per device.

The following table contains an overview of the defines you can set.

### Vector table defines

<table>
<thead>
<tr>
<th>Define</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>___NO_DEFAULT_AUTO_VECTORS</td>
<td>If defined, the default vector table will not be generated.</td>
</tr>
<tr>
<td>___CMSIS_VECTORS</td>
<td>If defined, the CMSIS vector table will be generated.</td>
</tr>
<tr>
<td>___NR_OF_VECTORS</td>
<td>Number of vectors.</td>
</tr>
<tr>
<td>___COPY_VECTOR_TABLE</td>
<td>If defined, the vector table is copied from ROM to RAM at startup.</td>
</tr>
<tr>
<td>___VECTOR_TABLE_ROM_ADDR</td>
<td>Address of the vector table in ROM.</td>
</tr>
</tbody>
</table>

### Memory defines

<table>
<thead>
<tr>
<th>Define</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>___ROM_SIZE</td>
<td>Size of ROM memory to be allocated.</td>
</tr>
<tr>
<td>___RAM_SIZE</td>
<td>Size of RAM memory to be allocated.</td>
</tr>
<tr>
<td>___FLASH_SIZE</td>
<td>Size of the flash memory to be allocated.</td>
</tr>
<tr>
<td>___SRAM_SIZE</td>
<td>Size of the SRAM memory to be allocated.</td>
</tr>
<tr>
<td>___CPU_SRAM_SIZE</td>
<td>Size of the SRAM memory to be allocated.</td>
</tr>
<tr>
<td>___AHB_SRAM0_SIZE</td>
<td>Size of the AHB SRAM bank 0 memory. The memory is not allocated if this macro is not defined.</td>
</tr>
<tr>
<td>___AHB_SRAM1_SIZE</td>
<td>Size of the AHB SRAM bank 1 memory. The memory is not allocated if this macro is not defined.</td>
</tr>
</tbody>
</table>

An example of the invocation of the linker (using the control program):

```
ccarm -CARMv7M -d"vendor_support_dir\include.lsl\stm32f10x.lsl"
    -Wl-D___FLASH_SIZE=128k -Wl-D___SRAM_SIZE=20k file.obj
```

When you create a new project in Eclipse the LSL template file will be copied to the project. Eclipse will pass device specific macro definitions to the linker, depending on the device selected in the Project » Properties for » C/C++ Build » Processor properties page. This way the project LSL file will include the appropriate device LSL file and memories are mapped as required for the selected device.
8.4. Stack and Heap

The stack is used for local automatic variables and function parameters. The following diagram shows the structure of a stack frame.

All ARM M-profile architectures have one stack pointer. This stack pointer should be initialized at run-time. This is taken care of by the startup code in the run-time library, by means of the linker-generated symbol `_lc_ub_stack` defined in the LSL file. See Section 8.1, Startup Code.

You can define the values of these symbols in Eclipse as follows.

1. From the Project menu, select Properties for
   
   The Properties dialog appears.

2. In the left pane, expand C/C++ Build and select Stack/Heap.

   In the right pane the Stack/Heap property page appears.
3. Make your changes and click **Apply and Close**.

   *The project LSL file is updated automatically according to the changes you make.*

You can specify the size and location of the stacks.

The stack size is defined in the linker script file *(arm_arch.lsl in directory include.lsl)* with macros:

<table>
<thead>
<tr>
<th>Define</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__STACK</td>
<td>Minimum size of the stack.</td>
</tr>
<tr>
<td>__STACK_FIXED</td>
<td>Define this macro if you do not want to let the stack section grow to the maximum available memory.</td>
</tr>
<tr>
<td>__STACKADDR</td>
<td>Start address of the stack. If not defined the stack will be located freely.</td>
</tr>
<tr>
<td>__STACK_THREADS</td>
<td>Number of threads used to calculate the estimated stack usage which is listed in the map file.</td>
</tr>
<tr>
<td>__STACK_ENTRY_POINTS</td>
<td>The entry points for stack estimation. The default is &quot;__START&quot;.</td>
</tr>
</tbody>
</table>

**Heap allocation**

The heap is only needed when you use one or more of the dynamic memory management library functions: `malloc()`, `calloc()`, `free()` and `realloc()`. The heap is a reserved area in memory. Only if you use one of the memory allocation functions listed above, the linker automatically allocates a heap, as specified in the linker script file with the keyword `heap`.

A special section called `heap` is used for the allocation of the heap area. The size of the heap is defined in the linker script file *(arm_arch.lsl in directory include.lsl)* with the macro `__HEAP`, which results in a section called `heap`. The linker defined labels `_lc_ub_heap` and `_lc_ue_heap` (begin and end of heap) are used by the library function `sbrk()`, which is called by `malloc()` when memory is needed from the heap.

The following heap macros are used in `arm_arch.lsl`:

<table>
<thead>
<tr>
<th>Define</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__HEAP</td>
<td>Size of heap.</td>
</tr>
<tr>
<td>__HEAP_FIXED</td>
<td>Defined if you do not expand the heap if space is left.</td>
</tr>
<tr>
<td>__HEAPADDR</td>
<td>Heap start address.</td>
</tr>
</tbody>
</table>
Chapter 9. Using the Utilities

The TASKING VX-toolset for ARM comes with a number of utilities:

**ccarm**  
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.

**amk**  
A make utility 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. It supports parallelism which utilizes the multiple cores found on modern host hardware.

**eclipsec**  
The Eclipse console utility. You can use it to perform a headless build or generate makefiles from the command line without starting the IDE.

**ararm**  
An archiver. With this utility you create and maintain library files with relocatable object modules (.obj) generated by the assembler.

**hldumparm**  
A high level language (HLL) object dumper. With this utility you can dump information about an absolute object file (.abs). Key features are a disassembler with HLL source intermixing and HLL symbol display and a HLL symbol listing of static and global symbols.

**expirearm**  
A utility to limit the size of the cache by removing all files older than a few days or by removing older files until the total size of the cache is smaller than a specified size.

### 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:

```
ccarm [ [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.
- Files with a `.c` suffix are interpreted as C source programs and are passed to the compiler.
- Files with a `.asm` suffix are interpreted as hand-written assembly source files which have to be passed to the assembler.
- Files with a `.src` suffix are interpreted as compiled assembly source files. They are directly passed to the assembler.
- Files with a `.lib` suffix are interpreted as library files and are passed to the linker.
Files with a .obj 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 \texttt{--pass-*} (\texttt{-Wcp}, \texttt{-Wc}, \texttt{-Wa}, \texttt{-WI}) to pass arguments directly to tools.

For a complete list and description of all control program options, see Section 11.6, \textit{Control Program Options}.

Example with verbose output

\texttt{ccarm --verbose test.c}

The control program calls all tools in the toolset and generates the absolute object file \texttt{test.abs}. With option \texttt{--verbose (-v)} you can see how the control program calls the tools:

\begin{verbatim}
+ "path\carm" -o cc3248a.src test.c
+ "path\asarm" -o cc3248b.obj cc3248a.src
+ "path\lkarm" cc3248b.obj -o test.abs --map-file
    -lcarm -lfparm -lrtarm"
\end{verbatim}

The control program produces unique filenames for intermediate steps in the compilation process (such as \texttt{cc3248a.src} and \texttt{cc3248b.obj} in the example above) which are removed afterwards, unless you specify command line option \texttt{--keep-temporary-files (-t)}.

Example with argument passing to a tool

\texttt{ccarm --pass-c=-Oc test.c}

The option \texttt{-Oc} is directly passed to the compiler.
9.2. Make Utility amk

amk is a make utility that you can use to maintain, update, and reconstruct groups of programs. amk features parallelism which utilizes the multiple cores found on modern host hardware, hardening for path names with embedded white space and it has an (internal) interface to provide progress information for updating a progress bar. It does not use an external command shell (/bin/sh, cmd.exe) but executes commands directly.

The primary purpose of any make utility is to speed up the edit-build-test cycle. To avoid having to build everything from scratch even when only one source file changes, it is necessary to describe dependencies between source files and output files and the commands needed for updating the output files. This is done in a so called "makefile".

9.2.1. Makefile Rules

A makefile dependency rule is a single line of the form:

[target ...] : [prerequisite ...]

where target and prerequisite are path names to files. Example:

test.obj : test.c

This states that target test.obj depends on prerequisite test.c. So, whenever the latter is modified the first must be updated. Dependencies accumulate: prerequisites and targets can be mentioned in multiple dependency rules (circular dependencies are not allowed however). The command(s) for updating a target when any of its prerequisites have been modified must be specified with leading white space after any of the dependency rule(s) for the target in question. Example:

```plaintext
test.obj :
    ccarm test.c   # leading white space
```

Command rules may contain dependencies too. Combining the above for example yields:

```plaintext
test.obj : test.c
    ccarm test.c
```

White space around the colon is not required. When a path name contains special characters such as ‘:’, ‘#’ (start of comment), ‘=’ (macro assignment) or any white space, then the path name must be enclosed in single or double quotes. Quoted strings can contain anything except the quote character itself and a newline. Two strings without white space in between are interpreted as one, so it is possible to embed single and double quotes themselves by switching the quote character.

When a target does not exist, its modification time is assumed to be very old. So, amk will try to make it. When a prerequisite does not exist possibly after having tried to make it, it is assumed to be very new. So, the update commands for the current target will be executed in that case. amk will only try to make targets which are specified on the command line. The default target is the first target in the makefile which does not start with a dot.
Static pattern rules

Static pattern rules are rules which specify multiple targets and construct the prerequisite names for each target based on the target name.

```
[target ...] : target-pattern : [prerequisite-patterns ...]
```

The `target` specifies the targets the rules applies to. The `target-pattern` and `prerequisite-patterns` specify how to compute the prerequisites of each target. Each target is matched against the `target-pattern` to extract a part of the target name, called the `stem`. This stem is substituted into each of the `prerequisite-patterns` to make the prerequisite names (one from each `prerequisite-pattern`).

Each pattern normally contains the character `%` just once. When the `target-pattern` matches a target, the `%` can match any part of the target name; this part is called the `stem`. The rest of the pattern must match exactly. For example, the target `foo.obj` matches the pattern `%.obj`, with `foo` as the stem. The targets `foo.c` and `foo.abs` do not match that pattern.

The prerequisite names for each target are made by substituting the stem for the `%` in each prerequisite pattern.

`Example:`

```
objects = test.obj filter.obj

all: $(objects)

$(objects): %.obj: %.c
    ccarm -c $< -o $@
    echo the stem is $*
```

Here `$<` is the automatic variable that holds the name of the prerequisite, `$@` is the automatic variable that holds the name of the target and `$*` is the stem that matches the pattern. Internally this translates to the following two rules:

```
test.obj: test.c
    ccarm -c test.c -o test.obj
    echo the stem is test

filter.obj: filter.c
    ccarm -c filter.c -o filter.obj
    echo the stem is filter
```

Each target specified must match the target pattern; a warning is issued for each target that does not.
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>.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>.INIT</td>
<td>The rules following this target are executed before any other targets are built.</td>
</tr>
<tr>
<td>.PHONY</td>
<td>The prerequisites of this target are considered to be phony targets. A phony target is a target that is not really the name of a file. The rules following a phony target are executed unconditionally, regardless of whether a file with that name exists or what its last-modification time is. For example: .PHONY: clean clean: rm *.obj With amk clean, the command is executed regardless of whether there is a file named clean.</td>
</tr>
</tbody>
</table>

9.2.2. Makefile Directives

Directives inside makefiles are executed while reading the makefile. When a line starts with the word "include" or "-include" then the remaining arguments on that line are considered filenames whose contents are to be inserted at the current line. "-include" will silently skip files which are not present. You can include several files. Include files may be nested.

Example:

include makefile2 makefile3

White spaces (tabs or spaces) in front of the directive are allowed.

9.2.3. 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 lowercase or uppercase characters, uppercase is an accepted convention. When a line does not start with white space and contains the assignment operator '=' or ':=' or '+' then the line is interpreted as a macro definition. White space around the assignment operator and white space at the end of the line is discarded. Single character macro evaluation happens by prefixing the name with '$'. To evaluate macros with names longer than one character put the name between parentheses '{}' or curly braces '{}'. Macro names may contain anything, even white space or other macro evaluations.

Example:

DINNER = $(FOOD) and $(BEVERAGE)
FOOD = pizza
BEVERAGE = sparkling water
FOOD += with cheese

With the += operator you can add a string to an existing macro. An extra space is inserted before the added string automatically.

Macros are evaluated recursively. Whenever $(DINNER) or $(DINNER) is mentioned after the above, it will be replaced by the text "pizza with cheese and sparkling water". The left hand side in a macro definition is evaluated before the definition takes place. Right hand side evaluation depends on the assignment operator:

= Evaluate the macro at the moment it is used.
:= Evaluate the replacement text before defining the macro.

Subsequent ‘+=' assignments will inherit the evaluation behavior from the previous assignment. If there is none, then ‘+=’ is the same as ‘=’. The default value for any macro is taken from the environment. Macro definitions inside the makefile overrule environment variables. Macro definitions on the amk command line will be evaluated first and overrule definitions inside the makefile.

Predefined macros

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>
<tr>
<td>@</td>
<td>The name of the current target. When a rule has multiple targets, then it is the name of the target that caused the rule commands to be run.</td>
</tr>
<tr>
<td>*</td>
<td>The basename (or stem) of the current target. The stem is either provided via a static pattern rule or is calculated by removing all characters found after and including the last dot in the current target name. If the target name is 'test.c' then the stem is 'test' (if the target was not created via a static pattern rule).</td>
</tr>
<tr>
<td>&lt;</td>
<td>The name of the first prerequisite.</td>
</tr>
<tr>
<td>MAKE</td>
<td>The amk path name (quoted if necessary). Optionally followed by the options -n and -s.</td>
</tr>
<tr>
<td>ORIGIN</td>
<td>The name of the directory where amk is installed (quoted if necessary).</td>
</tr>
<tr>
<td>SUBDIR</td>
<td>The argument of option -G. If you have nested makes with -G options, the paths are combined. This macro is defined in the environment (i.e. default macro value).</td>
</tr>
</tbody>
</table>

The @, * and < macros may be suffixed by 'D' to specify the directory component or by 'F' to specify the filename component. $(@D) evaluates to the directory name holding the file$(@F). $(@D)/$(@F) is equivalent to @$@. Note that on MS-Windows most programs accept forward slashes, even for UNC path names.

The result of the predefined macros @, * and < and 'D' and 'F' variants is not quoted, so it may be necessary to put quotes around it.

Note that stem calculation can cause unexpected values. For example:
Macro string substitution

When the macro name in an evaluation is followed by a colon and equal sign as in

\$\{(MACRO:string1=string2)\}

then \texttt{amk} will replace \textit{string1} at the end of every word in \$\{(MACRO)\} by \textit{string2} during evaluation. When \$\{(MACRO)\} contains quoted path names, the quote character must be mentioned in both the original string and the replacement string\footnote{Internally, \texttt{amk} tokenizes the evaluated text, but performs substitution on the original input text to preserve compatibility here with existing make implementations and POSIX.}. For example:

\$\{(MACRO:.obj"=\'.d\')\}

\textbf{9.2.4. Makefile Functions}

A function not only expands but also performs a certain operation. The following functions are available:

\$\{(\texttt{filter} \ pattern \ ..., \item \ ...)\}

The \texttt{filter} function filters a list of items using a pattern. It returns \texttt{items} that do match any of the \texttt{pattern} words, removing any items that do not match. The patterns are written using \texttt{'\%'}:

\$\{(\texttt{filter} \ %.c \ %.h, \texttt{test.c \ test.h \ test.obj \ readme.txt \ .project \ output.c})\}

results in:

\texttt{test.c \ test.h \ output.c}

\$\{(\texttt{filter-out} \ pattern \ ..., \item \ ...)\}

The \texttt{filter-out} function returns all \texttt{items} that do not match any of the \texttt{pattern} words, removing the items that do match one or more. This is the exact opposite of the \texttt{filter} function.

\$\{(\texttt{filter-out} \ %.c \ %.h, \texttt{test.c \ test.h \ test.obj \ readme.txt \ .project \ output.c})\}

results in:

\texttt{test.obj \ readme.txt \ .project}

\$\{(\texttt{foreach} \ \texttt{var-name}, \ \item \ ...; \ \texttt{action})\}

The \texttt{foreach} function runs through a list of items and performs the same \texttt{action} for each \texttt{item}. The \texttt{var-name} is the name of the macro which gets dynamically filled with an item while iterating through the \texttt{item} list. In the \texttt{action} you can refer to this macro. For example:
results in:

```
test.c test.h filter.c filter.h output.c output.h
```

### 9.2.5. 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. White spaces (tabs or spaces) in front of preprocessing directives are allowed.

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 to any level.

You can also add tests based on strings. With `ifeq` the result is true if the two strings match, with `ifneq` the result is true if the two strings do not match. They are used in the following way:

```
ifeq(string1,string2)
  if-lines
else
  else-lines
endif
```

### 9.2.6. Makefile Parsing

`amk` reads and interprets a makefile in the following order:

1. When the last character on a line is a backslash (\) (i.e. without trailing white space) then that line and the next line will be concatenated, removing the backslash and newline.

2. The unquoted `#` character indicates start of comment and may be placed anywhere on a line. It will be removed in this phase.

   ```
   # this comment line is continued\
   on the next line
   ```
3. Trailing white space is removed.

4. When a line starts with white space and it is not followed by a directive or preprocessing directive, then it is interpreted as a command for updating a target.

5. Otherwise, when a line contains the unquoted text ‘=’, ‘+=’ or ‘:=’ operator, then it will be interpreted as a macro definition.

6. Otherwise, all macros on the line are evaluated before considering the next steps.

7. When the resulting line contains an unquoted ‘:’ the line is interpreted as a dependency rule.

8. When the first token on the line is "include" or "-include" (which by now must start on the first column of the line), amk will execute the directive.

9. Otherwise, the line must be empty.

Macros in commands for updating a target are evaluated right before the actual execution takes place (or would take place when you use the -n option).

9.2.7. Makefile Command Processing

A line with leading white space (tabs or spaces) without a (preprocessing) directive is considered as a command for updating a target. When you use the option -j or -J, amk will execute the commands for updating different targets in parallel. In that case standard input will not be available and standard output and error output will be merged and displayed on standard output only after the commands have finished for a target.

You can precede a command by one or more of the following characters:

- @ Do not show the command. By default, commands are shown prior to their output.
- - Continue upon error. This means that amk ignores a non-zero exit code of the command.
- + Execute the command, even when you use option -n (dry run).
- | Execute the command on the foreground with standard input, standard output and error output available.

**Built-in commands**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>This command does nothing. Arguments are ignored.</td>
</tr>
<tr>
<td>false</td>
<td>This command does nothing, except failing with exit code 1. Arguments are ignored.</td>
</tr>
<tr>
<td>echo arg</td>
<td>Display a line of text.</td>
</tr>
<tr>
<td>exit code</td>
<td>Exit with defined code. Depending on the program arguments and/or the extra rule options '-' this will cause amk to exit with the provided code. Please note that 'exit 0' has currently no result.</td>
</tr>
</tbody>
</table>
Create an argument file suitable for the `--option-file (-f)` option of all the other tools. The first `argfile` argument is the name of the file to be created. Subsequent arguments specify the contents. An existing argument file is not modified unless necessary. So, the argument file itself can be used to create a dependency to options of the command for updating a target.

Remove the specified file(s). The following options are available:

- `-r, --recursive` Remove directories and their contents recursively.
- `-f, --force` Force deletion. Ignore non-existent files, never prompt.
- `-i, --interactive` Interactive. Prompt before every removal.
- `-v, --verbose` Verbose mode. Explain what is being done.
- `-m file` Read options from `file`.
- `-?, --help` Show usage.

### 9.2.8. Calling the amk Make Utility

The invocation syntax of `amk` is:

```
amk [option]... [target]... [macro=def]...
```

For example:

```
amk test.abs
```

**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 `amk` invocation. It overrides any regular definitions for the specified macro within the makefiles and from the environment. It is not inherited by subordinate `amk`

**option** For a complete list and description of all `amk` make utility options, see Section 11.7, *Parallel 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.
9.3. Eclipse Console Utility

eclipse is the console executable variant of Eclipse. Unlike the other utilities it is part of Eclipse and therefore it is present in the eclipse sub-directory of the product (installation-path\eclipse) instead of in the bin directory. You can use it to start Eclipse from a Windows Command Prompt. This section describes how to use this utility to perform the following actions:

- Build a project from the command line without starting the Eclipse IDE.
- Generate makefiles from the command line without starting the Eclipse IDE.

The general invocation syntax of eclipse is:

installation-dir\eclipse\eclipse [option]...

Note that in this manual we only describe the options that are of interest to perform the desired actions.

9.3.1. Headless Build

The Eclipse workbench (IDE) is referred to as 'head'. Headless build essentially means running builds from the command line without starting the Eclipse IDE.

To perform a headless build

Use the following invocation syntax in a Windows Command Prompt:

installation-dir\eclipse\eclipse -nosplash -data workspace-location -application com.tasking.managedbuilder.headlessbuild -build {project | all}

Explanation of the options used:

- **-nosplash**
  Do not generate the Eclipse splash screen on startup.

- **-data workspace-location**
  The location of your workspace. If you used the default settings when installing the product, the workspace location is C:\Users\user\workspace_arm_version (Windows 7 or higher).

- **-application**
  com.tasking.managedbuilder.headlessbuild is the application to use for a headless build.

- **-build {project | all}**
  Perform a headless build. When you specify a project name, Eclipse builds the active configuration of the specified project. With all, Eclipse builds all active configurations of all projects in the specified workspace.

For example, to build the active configuration of myproject, enter:

eclipse -nosplash -data "C:\Users\User\workspace_arm_vx.yrz" -application com.tasking.managedbuilder.headlessbuild -build myproject
Output similar to the following appears on the console:

**** Build of configuration Debug for project myproject ****

"C:\\Program Files\\TASKING\\carm vx.yrz\\bin\\amk" -j4 all -a
Compiling myproject.c
Linking to myproject.abs

Time consumed: 397 ms
**** End of build ****

9.3.2. Generating Makefiles from the Command Line

You can use *eclipsec* to generate makefiles for one or more of your projects, without starting the Eclipse IDE. This can be useful for testing and batch processing.

**To generate makefiles on the command line**

Use the following invocation syntax in a Windows Command Prompt:

```
installation-dir\eclipse\eclipsec -nosplash -data workspace-location
   -application com.tasking.managedbuilder.headlessbuild
   -generateMakefile {project[/configuration] | all}
```

Explanation of the options used:

- **-nosplash**
  Do not generate the Eclipse splash screen on startup.

- **-data workspace-location**
  The location of your workspace. If you used the default settings when installing the product, the workspace location is `C:\Users\user\workspace_arm_version` (Windows 7 or higher).

- **-application**
  `com.tasking.managedbuilder.headlessbuild` is the application to use for generating makefiles on the command line.

- **-generateMakefile {project[/configuration] | all}**
  Generate makefile(s). When you specify a project name, Eclipse generates makefiles for all configurations of the specified project. When you also specify a configuration, Eclipse generates makefiles for the specified configuration of the project. With all, Eclipse generates makefiles for all active configurations of all projects in the specified workspace. With .*.* Eclipse generates makefiles for all configurations of all projects in the specified workspace.

For example, to generate makefiles for all configurations of myproject, enter:

```
eclipsec -nosplash -data "C:\Users\User\workspace_arm_vx.yrz"
   -application com.tasking.managedbuilder.headlessbuild
   -generateMakefile myproject
```

Output similar to the following appears on the console:
**** Build Makefile of configuration Debug for project myproject ****

**** Build Makefile of configuration Release for project myproject ****

The Debug and Release directory of myproject now contain updated makefiles.
The archiver **ararm** is a program to build and maintain your own library files. A library file is a file with extension `.lib` and contains one or more object files (.obj) 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.

### 9.4.1. 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 an ARM 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.

1. From the **File** menu, select **New » TASKING ARM C/C++ Project**.
   
   *The New C/C++ Project wizard appears.*

2. Enter a project name.

3. In the **Project type** box, select **TASKING ARM 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**.

   *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:

```
arm 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><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 -n -u -v</td>
</tr>
<tr>
<td>Extract an object module from the library</td>
<td>-x</td>
<td>-o -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 name</td>
<td>-a name</td>
<td></td>
</tr>
<tr>
<td>Append or move new modules before existing module name</td>
<td>-b name</td>
<td></td>
</tr>
<tr>
<td>Suppress the message that is displayed when a new library is created</td>
<td>-c</td>
<td></td>
</tr>
<tr>
<td>Create a new library from scratch</td>
<td>-n</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>
</tbody>
</table>
### Archiver Options

<table>
<thead>
<tr>
<th>Description</th>
<th>Option</th>
<th>Sub-option</th>
</tr>
</thead>
<tbody>
<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 description of one or more diagnostic messages</td>
<td>--diag</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 file</td>
<td></td>
</tr>
<tr>
<td>Suppress warnings above level n</td>
<td>-wn</td>
<td></td>
</tr>
</tbody>
</table>

For a complete list and description of all archiver options, see [Section 11.8, Archiver Options](#).

### 9.4.2. 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.lib` and add the object modules `cstart.obj` and `calc.obj` to it:

```bash
ararm -r mylib.lib cstart.obj calc.obj
```

#### 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.)

```bash
ararm -r mylib.lib mod3.obj
```

#### Print a list of object modules in the library

To inspect the contents of the library:

```bash
ararm -t mylib.lib
```

The library has the following contents:

- `cstart.obj`
- `calc.obj`
- `mod3.obj`

#### Move an object module to another position

To move `mod3.obj` to the beginning of the library, position it just before `cstart.obj`:

```bash
ararm -mb cstart.obj mylib.lib mod3.obj
```
Delete an object module from the library

To delete the object module `cstart.obj` from the library `mylib.lib`:

```
ar -d mylib.lib cstart.obj
```

Extract all modules from the library

Extract all modules from the library `mylib.lib`:

```
ar -x mylib.lib
```
9.5. HLL Object Dumper

The high level language (HLL) dumper hldumparm is a program to dump information about an absolute object file (.abs). Key features are a disassembler with HLL source intermixing and HLL symbol display and a HLL symbol listing of static and global symbols.

9.5.1. Invocation

Command line invocation

You can call the HLL dumper from the command line. The invocation syntax is:

hldumparm [option]... file...

The input file must be an ELF file with or without DWARF debug info (.abs).

The HLL dumper can process multiple input files. Files and options can be intermixed on the command line. Options apply to all supplied files. If multiple files are supplied, the disassembly of each file is preceded by a header to indicate which file is dumped. For example:

========== file.abs ==========

For a complete list and description of all options, see Section 11.9, HLL Object Dumper Options. With hldumparm --help you will see the options on stdout.

9.5.2. HLL Dump Output Format

The HLL dumper produces output in text format by default, but you can also specify the XML output format with --output-type=xml. The XML output is mainly for use in the Eclipse editor. The output is printed on stdout, unless you specify an output file with --output=filename.

The parts of the output are dumped in the following order:

1. Module list
2. Section list
3. Section dump (disassembly)
4. HLL symbol table
5. Assembly level symbol table
6. Note sections
7. Debug control flow section

With the option --dump-format=flag you can control which parts are shown. By default, all parts are shown, except for part 7.
Example

Suppose we have a simple "Hello World" program in a file called `hello.c`. We call the control program as follows:

```
ccarm -g -t --control-flow-info hello.c
```

Option `-g` tells to include DWARF debug information. Option `-t` tells to keep the intermediate files. Option `--control-flow-info` adds control flow information to the output file. This command results (among other files) in the file `hello.abs` (the absolute object file).

We can dump information about the object file with the following command:

```
hldumparm -F3 hello.abs
```

Option `-F3` enables all parts. A possible output could be (just a fraction of the actual output is shown):

```
---------- Module list ----------

Name        Full path
hello.c     hello.c

---------- Section list ----------

<table>
<thead>
<tr>
<th>Address</th>
<th>Size</th>
<th>Align</th>
<th>Type</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>000001a0</td>
<td>20</td>
<td>4</td>
<td>text</td>
<td>.text</td>
</tr>
<tr>
<td>0000018c</td>
<td>6</td>
<td>4</td>
<td>romdata</td>
<td>.rodata</td>
</tr>
<tr>
<td>00040184</td>
<td>4</td>
<td>4</td>
<td>bss</td>
<td>.data</td>
</tr>
<tr>
<td>00000194</td>
<td>11</td>
<td>4</td>
<td>romdata</td>
<td>.rodata</td>
</tr>
</tbody>
</table>

---------- Section dump ----------

```
[section .data, '[.data]', at(0x00000188)]
.db 8c,01,00,00                        ; ....
.endsec

[section .data, '.rodata', at(0x0000018c)]
.db 77,6f,72,6c,64,00                    ; world.
.endsec

[section .data, '.rodata', at(0x00000194)]
.db 48,65,6c,6c,6f,20,25,73,21,0a,00             ; Hello %s!..
.endsec

000001a0  02 48     main:    ldr    r0, [pc, #0x8]
000001a2  01 68     ldr    r1, [r0, #0x0]
000001a4  02 48     ldr    r0, [pc, #0x8]
000001a6  00 f0 d5 b8   b      printf
000001aa  c0 46     mov    r8, r8
000001ac  84 01        .dh 0184
```

Using the Utilities
--- HLL symbol table ---

<table>
<thead>
<tr>
<th>Address</th>
<th>Size</th>
<th>HLL Type</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>000001a0</td>
<td>10</td>
<td>void</td>
<td>main()</td>
</tr>
<tr>
<td>000001b4</td>
<td>48</td>
<td>void</td>
<td>Reset_Handler()</td>
</tr>
<tr>
<td>000001f4</td>
<td>2</td>
<td>void</td>
<td>SystemInit()</td>
</tr>
<tr>
<td>00000354</td>
<td>48</td>
<td>int</td>
<td>printf(const char * restrict format, ...)</td>
</tr>
<tr>
<td>00040000</td>
<td>24</td>
<td>struct</td>
<td>_dbg_request [dbg.c]</td>
</tr>
<tr>
<td>00040018</td>
<td>80</td>
<td>char</td>
<td>stdin_buf[80] [iob.c]</td>
</tr>
<tr>
<td>00040068</td>
<td>80</td>
<td>char</td>
<td>stdout_buf[80] [iob.c]</td>
</tr>
<tr>
<td>000400b8</td>
<td>4</td>
<td>unsigned int</td>
<td>_Vectors [cstart.c]</td>
</tr>
<tr>
<td>000400bc</td>
<td>200</td>
<td>struct _iobuf</td>
<td>_iob[10] [iob.c]</td>
</tr>
<tr>
<td>00040184</td>
<td>4</td>
<td>char</td>
<td>* world [hello.c]</td>
</tr>
</tbody>
</table>

--- Assembly level symbol table ---

<table>
<thead>
<tr>
<th>Address</th>
<th>Size</th>
<th>Type</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td></td>
<td>.data.libc</td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td></td>
<td>.data</td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td></td>
<td>_vector_0</td>
<td>hello.c</td>
</tr>
<tr>
<td>000001a0</td>
<td>20</td>
<td>code</td>
<td>main</td>
</tr>
<tr>
<td>000001b4</td>
<td>64</td>
<td>code</td>
<td>Reset_Handler</td>
</tr>
<tr>
<td>00000354</td>
<td>56</td>
<td>code</td>
<td>printf</td>
</tr>
<tr>
<td>00040184</td>
<td>4</td>
<td>data</td>
<td>world</td>
</tr>
</tbody>
</table>

--- .note sections ---

Section .note, section 101:
00000000 type: TASKING LINKER NAME
0000000c name: TASKING
00000014 desc: lkarm

--- Debug control flow section ---

start offset: 0
start address: 0x000001a1
code size: 0
#entries: 1
dest. offset: 10

start offset: 10
start address: 0x000001a1
code size: 10
#entries: 0
Module list

This part lists all modules (C/C++ files) found in the object file(s). It lists the filename and the complete path name at the time the module was built.

Section list

This part lists all sections found in the object file(s).

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>The start address of the section. Hexadecimal, 8 digits, 32-bit.</td>
</tr>
<tr>
<td>Size</td>
<td>The size (length) of the section in bytes. Decimal, filled up with spaces.</td>
</tr>
<tr>
<td>Align</td>
<td>The alignment of the section in number of bytes. Decimal, filled up with spaces.</td>
</tr>
<tr>
<td>Type</td>
<td>The section type.</td>
</tr>
<tr>
<td>Name</td>
<td>The name of the section.</td>
</tr>
</tbody>
</table>

With option `--sections=name[,name]...` you can specify a list of sections that should be dumped.

Section dump

This part contains the disassembly. It consists of the following columns:

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>address column</td>
<td>Contains the address of the instruction or directive that is shown in the disassembly. If the section is relocatable the section start address is assumed to be 0. The address is represented in hexadecimal and has a fixed width. The address is padded with zeros. No 0x prefix is displayed. For example, on a 32-bit architecture, the address 0x32 is displayed as 00000032.</td>
</tr>
<tr>
<td>encoding column</td>
<td>Shows the hexadecimal encoding of the instruction (code sections) or it shows the hexadecimal representation of data (data sections). The encoding column has a maximum width of eight digits, i.e. it can represent a 32-bit hexadecimal value. The encoding is padded to the size of the data or instruction. For example, a 16-bit instruction only shows four hexadecimal digits. The encoding is aligned left and padded with spaces to fill the eight digits.</td>
</tr>
<tr>
<td>label column</td>
<td>Displays the label depending on the option `--symbols=[hll</td>
</tr>
<tr>
<td>disassembly column</td>
<td>For code sections the instructions are disassembled. Operands are replaced with labels, depending on the option `--symbols=[hll</td>
</tr>
</tbody>
</table>

With option `--hex`, no directives will be generated for ROM data sections and no disassembly dump will be done for code sections. Instead a hex dump is done with the following format:
where,

\( A = \) Address (8 digits, 32-bit)

\( H_x = \) Hex contents, one byte (16 bytes max)

\( R = \) ASCII representation (16 characters max)

For example:

```
section 7 (.rodata):
00000094 48 65 6c 6c 6f 20 25 73 21 0a 00
```

Hello %s!

With option `--hex`, RAM sections will be represented with only a start address and a size indicator:

```
AAAAAAA Space: 48 bytes
```

With option `--disassembly-intermix` you can intermix the disassembly with HLL source code.

### HLL symbol table

This part contains a symbol listing based on the HLL (DWARF) symbols found in the object file(s). The symbols are sorted on address.

- **Address**: The start address of the symbol. Hexadecimal, 8 digits, 32-bit.
- **Size**: The size of the symbol from the DWARF info in bytes.
- **HLL Type**: The HLL symbol type.
- **Name**: The name of the HLL symbol.

HLL arrays are indicated by adding the size in square brackets to the symbol name. For example:

```
00040018 80 static char stdin_buf[80] [__iob.c]
```

With option `--expand-symbols=+basic-types` HLL struct and union symbols are listed including all fields. Array members are expanded in one array member per line regardless of the HLL type. For example:

```
00040018 80 static char stdin_buf[80] [__iob.c]
00040018 1 char
00040019 1 char
0004001a 1 char
...
00040067 1 char
```

HLL struct and union symbols are listed by default without fields. For example:

```
00040000 24 struct _dbg_request [dbg.c]
```
With option `--expand-symbols` all struct, union and array fields are included as well. For the fields the types and names are indented with two spaces. For example:

```
00040000  24 struct  _dbg_request [dbg.c]
00040000   4 int      _errno
00040004   4 enum      nr
00040008  16 union     u
00040008   4 struct    exit
00040008   4 int       status
00040008  16 union     open
00040008   4      const char * pathname
0004000c   2     unsigned short int  flags
...
```

Functions are displayed with the full function prototype. Size is the size of the function. HLL Type is the return type of the function. For example:

```
00000474  64 int      printf(const char * restrict format, ...)
```

The local and static symbols get an identification between square brackets. The filename is printed and if a function scope is known the function name is printed between the square brackets as well. If multiple files with the same name exist, the unique part of the path is added. For example:

```
00040100   4 int      count [file.c, somefunc()]
00040104   4 int      count [x\a.c]
00040108   4 int      count [y\a.c, foo()]
```

Global symbols do not get information in square brackets.

**Assembly level symbol table**

This part contains a symbol listing based on the assembly level (ELF) symbols found in the object file(s). The symbols are sorted on address.

<table>
<thead>
<tr>
<th>Address</th>
<th>The start address of the symbol. Hexadecimal, 8 digits, 32-bit.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>The size of the symbol from the ELF info in bytes. If this field is empty, the size is zero.</td>
</tr>
<tr>
<td>Type</td>
<td>Code or Data, depending on the section the symbol belongs to. If this field is empty, the symbol does not belong to a section.</td>
</tr>
<tr>
<td>Name</td>
<td>The name of the ELF symbol.</td>
</tr>
</tbody>
</table>

**Debug control flow section**

When control flow information is present in the ELF file (control program option `--control-flow-info`), this part shows information about the basic blocks and their relation.

<table>
<thead>
<tr>
<th>start offset</th>
<th>The start seek offset in bytes from the beginning of the section.</th>
</tr>
</thead>
<tbody>
<tr>
<td>start address</td>
<td>The start address of the basic block.</td>
</tr>
<tr>
<td>code size</td>
<td>The code size of the basic block.</td>
</tr>
</tbody>
</table>
#entries  The number of successor basic blocks. This value can be 0 if there are no successors.

dest. offset  The destination offset in bytes to the first, second, ... successor from the beginning of the section.
9.6. Expire Cache Utility

With the utility `expirearm` you can limit the size of the cache (C compiler option `--cache`) by removing all files older than a few days or by removing older files until the total size of the cache is smaller than a specified size. See also Section 12.5, Compiler Cache.

The invocation syntax is:

```
expirearm [option]... cache-directory
```

The compiler cache is present in the directory `carmcache` under the specified `cache-directory`.

For a complete list and description of all options, see Section 11.10, Expire Cache Utility Options. With `expirearm --help` you will see the options on stdout.

**Examples**

To remove all files older than seven days, enter:

```
expirearm --days=7 "installation-dir\mproject\.cache"
```

To reduce the compiler cache size to 4 MB, enter:

```
expirearm --megabytes=4 "installation-dir\mproject\.cache"
```

Older files are removed until the total size of the cache is smaller than 4 MB.

To clear the compiler cache, enter:

```
expirearm --megabytes=0 "installation-dir\mproject\.cache"
```
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.

If you want to debug on a target board, you have to create a custom debug configuration for your target board, otherwise you have to create a debug launch configuration for the TASKING simulator.

Create or customize your debug configuration

To create or change a debug configuration follow the steps below.

1. From the Debug menu, select Debug Configurations...
   
   *The Debug Configurations dialog appears.*

2. Select TASKING C/C++ Debugger and click the New launch configuration button ( ).
3. In the Name field enter the name of the configuration. By default, this is the name of the project, but you can give your configuration any name you want to distinguish it from the project name. For example enter myproject.simulator to identify the simulator debug configuration.

4. On the Target tab, select the ARM Simulator or any of the target boards.

The dialog shows several tabs.

**Target tab**

On the Target tab you can select on which target the application should be debugged. An application can run on an external evaluation board, or on a simulator using your own PC. On this tab you can also select the connection settings. The information in this tab is based on the Debug Target Configuration (DTC) files as explained in Chapter 18, *Debug Target Configuration Files*. If you select Show all targets you can see an overview of all supported evaluation boards.

**Initialization tab**

On the 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.

• **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.

• **Reduce target state polling**

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.

• **Initialize Target Board**
Some target boards contain a power supply chip which needs to be initialized every time after power-on, before hardware debugging is started. If enabled, the target board is initialized automatically before the start of a debug session.

This option is not supported for ARM.

**Initialization tab: Flash settings**

- **Use default flash settings (recommended)**

  By default, the flash settings are derived from the `.dtc` file for the chosen target processor. So, when you change processors the flash settings change automatically. If you do not want that, you can specify your own flash settings. You can click **Restore Defaults** to restore the default flash settings.

- **Monitor file**

  Filename of the monitor, usually an Intel Hex or S-Record file.

- **Sector buffer size**

  Specifies the buffer size for buffering a flash sector.

- **Workspace address**

  The address of the workspace of the flash programming monitor.

**Project tab**

On the **Project** tab, you can set the properties for the debug configuration such as a name for the project and the application binary file(s) which are used when you choose this configuration.
• 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 **Binary files** group box, you can choose one or more binary files to debug. The file `myproject.abs` is automatically selected from the active project.

The order of the binary files matters. Use the **Up** and **Down** buttons to change the order. If there are multiple files, the application start address is taken from the first file that defines one. An ELF file always defines one, whereas Hex files may not.

Note that conflicts between symbols could arise, for example when you download two ELF files that both contain the function `main()`. When you download multiple files, we recommend that the first binary file is an ELF file that contains the startup code and `main()` and that the other files are auxiliary Hex files.

**To add a binary file**

1. Click **Add...** to add a binary file.

   *The Add Binary File dialog appears.*

   ![Add Binary File dialog](image)

2. Specify the binary file, use the **Search...** button to select one from the active project, or use the **Browse...** button to search the file system.

3. Optionally, specify an address offset. The value will be added to all target addresses in the binary file.

   Note that the address offset will be applied only to code, data and the start address, not to debug information. Specifying a non-zero offset is not recommended for an ELF/DWARF file. If the offset causes an address to underflow or overflow an error occurs.

**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.
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.

Miscellaneous tab

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 kernel-aware debugging, specify the name of a Kernel Debug Interface (KDI) compatible KSM module (shared library) in the appropriate edit box. The toolset comes with KSM modules suitable for RTOS kernels. If you wish to use one of these, browse for the file `posix_ksm_arm.dll` (POSIX) or `tx_ksm_arm.dll` (ThreadX) in the `carm\bin` directory of the product installation directory. 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 TASKING support personnel.

• **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.

• **Launch in background**
When this option is disabled you will see a progress bar when the debugger starts. If you do not want to see the progress bar and want that the debugger launches in the background you can enable this option.

- **Use linker/locator memory map file (.mdf) for memory map**

You can use this option 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.

### 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>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>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>

### Device specific problem with WFE/WFI

Depending on the device, the clock feeding the debug interface may shut down when the processor executes a WFE or WFI instruction, causing debugging to fail. Try to avoid the __WFE() and __WFI() intrinsic functions in this case.
10.4. 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><img src="image" alt="Icon" /></td>
<td>Launch instance</td>
<td>Launch configuration name and launch type</td>
</tr>
</tbody>
</table>
### 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 main toolbar.

### Controlling debug sessions

<table>
<thead>
<tr>
<th>Icon</th>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="https://example.com/icon.png" alt="Icon" /></td>
<td>Remove all</td>
<td>Removes all terminated launches.</td>
</tr>
<tr>
<td><img src="https://example.com/icon.png" alt="Icon" /></td>
<td>Reset target system</td>
<td>Resets the target system.</td>
</tr>
<tr>
<td><img src="https://example.com/icon.png" alt="Icon" /></td>
<td>Restart</td>
<td>Resets the target system and restarts the application. The application stops executing when it reaches the first C instruction in the function <code>main()</code>.</td>
</tr>
<tr>
<td><img src="https://example.com/icon.png" alt="Icon" /></td>
<td>Resume</td>
<td>Resumes the application after it was suspended (manually, breakpoint, signal).</td>
</tr>
<tr>
<td><img src="https://example.com/icon.png" alt="Icon" /></td>
<td>Suspend</td>
<td>Suspends the application (pause). Use the Resume button to continue.</td>
</tr>
<tr>
<td><img src="https://example.com/icon.png" alt="Icon" /></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="https://example.com/icon.png" alt="Icon" /></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="https://example.com/icon.png" alt="Icon" /></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="https://example.com/icon.png" alt="Icon" /></td>
<td>Terminate all</td>
<td>Right-click menu. As terminate. Ends all debug sessions.</td>
</tr>
<tr>
<td><img src="https://example.com/icon.png" alt="Icon" /></td>
<td>Terminate and remove</td>
<td>Right-click menu. Ends the debug session and removes it from the Debug view.</td>
</tr>
</tbody>
</table>
## Using the Debugger

### Stepping through the application

<table>
<thead>
<tr>
<th>Icon</th>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>![step into]</td>
<td>Step into</td>
<td>Steps to the next source line or instruction.</td>
</tr>
<tr>
<td>![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>![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>
<tr>
<td>![instruction stepping]</td>
<td>Instruction stepping</td>
<td>Toggle. If enabled, the stepping functions are performed on instruction level instead of on C source line level.</td>
</tr>
<tr>
<td>![interrupt aware stepping]</td>
<td>Interrupt aware stepping</td>
<td>Toggle. If an interrupt source continues generating interrupts while the target is stopped (either manually or by hitting a breakpoint), a following single step will always enter the Interrupt Service Routine (ISR). This can lead to some problems during single stepping. With interrupt aware stepping enabled, interrupts are temporarily disabled after the target has stopped. When execution resumes the interrupts are restored.</td>
</tr>
</tbody>
</table>

### Miscellaneous

<table>
<thead>
<tr>
<th>Icon</th>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>![copy stack]</td>
<td>Copy Stack</td>
<td>Right-click menu. Copies the stack as text to the windows clipboard. You can paste the copied selection as text in, for example, a text editor.</td>
</tr>
<tr>
<td>![edit project]</td>
<td>Edit project...</td>
<td>Right-click menu. Opens the debug configuration dialog to let you edit the current debug configuration.</td>
</tr>
<tr>
<td>![edit source lookup]</td>
<td>Edit Source Lookup...</td>
<td>Right-click menu. Opens the Edit Source Lookup Path window to let you edit the search path for locating source files.</td>
</tr>
</tbody>
</table>

### 10.4.2. Breakpoints View

You can add, disable and remove breakpoints by clicking in the marker bar (left margin) of the Editor view. This is explained in the Getting Started manual.

**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 specific type. You can set the following types of breakpoints:

• File breakpoint

![File breakpoint dialog](image)

If a debug session is active, the File drop-down box is filled with all source files as present in the debug information in the ELF file. This can include files not present in the Eclipse project (for example from libraries). If a file could be matched to a file in the active Eclipse project it will show as an Eclipse project relative filename.

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. It is also possible that on some files no line breakpoints can be set because the debugger lacks line information.
The **Function** drop-down box is filled with all functions from the debug information and the symbol table (if not already in the debug information). You can use the **File** drop-down box to filter the list of functions. If you select `<all>` you will see the filenames (between parentheses) behind each entry in the **Function** drop-down box. Functions marked with `function_name [section]` originate from the symbol table. These functions are normally not associated with a filename and will therefore be included if `<unknown>` is selected in the **File** drop-down box. Functions marked `'filename':function_name` are static functions.

The target halts when it reaches the first line of the specified function. Note that function breakpoints generally will not work on inlined instances of a function.
• **Code Address**

![Breakpoints](image)

Select breakpoint type

- No address specified.

<table>
<thead>
<tr>
<th>File</th>
<th>Function</th>
<th>Code Address</th>
<th>Data</th>
<th>Data Address</th>
<th>Stack</th>
<th>Instruction</th>
<th>Cycle</th>
<th>Timer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Method:  
  - Hardware breakpoint  
  - Software breakpoint  
  - No preference

- Condition:
- Ignore count:

The target halts when it reaches the specified instruction address.

• **Data**

![Breakpoints](image)

Select breakpoint type

- Create data breakpoint

<table>
<thead>
<tr>
<th>File</th>
<th>Function</th>
<th>Code Address</th>
<th>Data</th>
<th>Data Address</th>
<th>Stack</th>
<th>Instruction</th>
<th>Cycle</th>
<th>Timer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Variable: `queens.c:chess_board`
- Type:  
  - Break on read access  
  - Break on write access  
  - Break on read or write access

- Condition:
- Ignore count:

The **Variable** drop-down box is filled with all variables from the debug information and the symbol table (if not already in the debug information), but you can also enter text yourself. If a label is filled in, the size will be 1 MAU. You can use the **File** drop-down box to filter the list of variables. If you select `<all>` you will see the filenames (between parentheses) behind each entry in the **Variables** drop-down box. Variables marked with `variable_name [section]` originate from the symbol table. These variables are normally not associated with a filename and will therefore be included if `<unknown>` is selected in the **File** drop-down box. Variables marked `'filename':variable_name` are static.
The target halts when the given variable is read or written to, as specified.

- **Data Address**

The target halts when the given memory range (specified in terms of an absolute Address and a Length in MAUs) is read or written to, as specified.

- **Stack**

The target halts when it reaches the specified stack level.
• **Instruction**

The target halts when the given number of instructions (Count) has been executed.

• **Cycle**

The target halts when the given number of clock cycles (Count) has elapsed.
• **Timer**

![Timer GUI](image)

The target halts when the given amount of **Time** elapsed. The value entered is interpreted by the debug instrument.

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 `<ARM installation path>\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 14.2.4, 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.

When you use MIL linking/splitting the C library is translated along with your application. Therefore you need to build your application with debug information generation enabled when FSS support is needed.

**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 if you click the Show Source button ( ).

The left part of the Disassembly view shows the addresses, opcodes and/or function offsets depending on what you selected in the right-click menu.

The right part of the Disassembly view shows the disassembly instructions and/or the C/C++ source code and symbols depending on what you selected in the right-click menu. Right-click in the right part and select Preferences to open the Disassembly Preferences dialog.

If you are debugging a multi-core project, you can display each thread in its own Disassembly view. To open a thread specific Disassembly view, select a stack frame in a thread in the Debug view and click the Open New View button ( ) in the Disassembly view and then click the Pin to Debug Context button ( ). The selected thread is mentioned in the view.

To view the contents of a specific memory location, type the address in the Enter location here field.

**10.4.5. Expressions View**

The Expressions view allows you to evaluate and watch regular C expressions.

To add an 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"

3. Click OK to add the expression.

If you have added one or more expressions to watch, the right-click menu provides options to Remove and Edit Watch Expression 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, unsigned integer or traditional. You can add or remove renderings per monitor. Though you cannot change a rendering, you can add or remove them:

1. Click the New Renderings... tab in the Memory Renderings pane.  
   The Add Memory Rendering dialog appears.
2. Select the rendering you want (Traditional, Floating Point, Hex Integer, Hex, ASCII, Signed Integer or Unsigned Integer) and click Add Rendering(s).

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, or your application overwrote parts of the memory.

- 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
char * string = (char *) calloc(100, sizeof(char*));
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 ARM ships with a POSIX KSM and a KSM for the ThreadX kernel. You have to create your own Run Time Interface (ORTI) and specify this file on the Miscellaneous tab while configuring a customized debug configuration (see also Section 10.2, Creating a Customized Debug Configuration):

1. From the Debug menu, select Debug Configurations...

   The Debug Configurations dialog appears.

2. In the left pane, select the configuration you want to change, for example, TASKING C/C++ Debugger » myproject.

   Or: click the New launch configuration button (⋯) to add a new configuration.

3. Open the Miscellaneous tab

4. In the ORTI file field, specify the name of your own ORTI file.

5. If you want to use one of the supplied KSM modules suitable for RTOS kernels, in the KSM module field, browse for the file posix_ksm_arm.dll (POSIX) or tx_ksm_arm.dll (ThreadX) in the carm\bin directory of the product installation directory.

The debugger supports ORTI specifications v2.0 and v2.1.

10.4.11. Registers View

In the Registers view you can examine the value of registers while stepping through your application. If you are debugging a multi-core project, you can display each thread in its own Registers view.

The registers are organized in a number of register groups, which together contain all known registers. You can expand a group to see which registers it contains. This view has a number of features:

• While you step through the application, the registers that are changed turn yellow. If you scroll in the view or switch groups, some registers may appear on a lighter yellow background, indicating that the
debugger does not know whether the registers have changed because the debugger did not read the registers before the step began.

Registers view:

Thread specific Registers view (Pin to Debug Context):
• To open a thread specific Registers view, select a stack frame in a thread in the Debug view and click the **Open New View** button (🚀) in the Registers view and then click the **Pin to Debug Context** button (📌). The selected thread is mentioned in the view.

• You can change each register's value.

• For some registers you can view the internal fields of the register. For example, the XPSR register from the Core group may be shown as follows:

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.

• You can search for a specific register: right-click on a register and from the popup menu select **Find**.... Enter a group or register name filter, click the register you want to see and click **OK**. The register of your interest will be shown in the view.

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:

• Right-click in the Trace view and 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.

When you use the simulator, this works as is. Otherwise, for Cortex-M0+ devices, tracing only works when the processor has a Micro Trace Buffer (MTB). See **Section 10.4.12.1, Micro Trace Buffer Support**.
For M3 devices and up, tracing only works when the processor has a Trace Port Interface Unit (TPIU) and this is supported. See Section 10.4.12.2, Trace Port Interface Unit Support.

When tracing is enabled note the following:

• If the condition associated with a conditional instruction was false, the instruction will still be shown in the trace, with no distinct visual indication. This applies to conditional branches, but also to instructions inside an IT block, for example.

• If the target halted due to a software breakpoint, then technically a BKPT instruction was executed and hence traced. However, for convenience this will not actually be shown in the Trace view.

• For bandwidth and performance reasons, the trace shown may be shorter than what would be expected considering the amount of data the debug probe can provide.

• The Raw tab shows the trace in its compressed form, i.e. as MTB packets (Cortex-M0+) or as a data stream compliant with the Embedded Trace Macrocell (ETM) signal protocol as defined by ARM Ltd. (Cortex-M3 and up). Usually, you will not need this information, but it may be useful if you suspect problems with the target communication or the decompression.

10.4.12.1. Micro Trace Buffer Support

The Micro Trace Buffer (MTB) stores the data packets that it generates within the processor's own SRAM, from where the debugger reads it every time the processor halts. You therefore need to reserve space for this data when you build your project. You can do this by defining the LSL macro __MTB_SIZE (see linker option --define), which defines the size in bytes. The value must be a power of 2 and at least 32. Be aware that because of how the hardware works, the start address of the reserved space needs to be a multiple of __MTB_SIZE, but the linker takes care of this automatically.

The larger the value of __MTB_SIZE, the longer and hence the more useful the trace will be. There is no exact relationship between the macro's value and the number of instructions traced, however, because it depends on the kind of code being executed.

10.4.12.2. Trace Port Interface Unit Support

Many modern ARM-based processors have a Trace Port Interface Unit (TPIU), which provides a means to communicate trace and other debug related information to a suitable debug probe, such as a SEGGER J-Trace.

Currently, the TASKING debugger can process TPIU output only for STMicroelectronics STM32F processors and only in conjunction with a SEGGER J-Trace probe. Note that not all evaluation boards actually allow connecting a J-Trace.

You need to be aware of the following:

• The TPIU data is output via five pins named TRACECK and TRACED0...3 or similar. Typically, these pins are shared with other chip peripherals and hence you must take care not to use these pins for other purposes while using TPIU-related debugger facilities.

• At the moment, the TPIU is used only for tracing purposes (see Section 10.4.12, Trace View). In this case especially the amount of TPIU data is relatively large and therefore, it is important that the TPIU output frequency is large enough relative to the core's, because otherwise overflow may occur.
default settings are such that no overflow will occur, but if you increase the core clock frequency, it is recommended that you increase the TPIU output frequency by the same factor.

If overflow does occur, the debugger will try to detect and/or work around it, but this may not always succeed. This may manifest itself as a trace which is inconsistent with actual program execution, for example.

### 10.5. Multi-core Hardware Debugging

The TASKING debugger supports multi-core hardware debugging. When you start the debugger for a multi-core device, you can debug all cores of the selected device.

The following picture shows an example multi-core application and is debugged on an Cypress CYTVII-B-H-8M-320BGA Board.

For each of the cores CM0, CM7 0 and CM7 1 a separate thread is started. The example shows the situation where Thread [Core CM0] is suspended, Thread [Core CM7 0] is running and Thread [Core CM7 1] is suspended.

### Suspend or resume a thread/core

You can suspend or resume each of the threads independent from the other threads.

1. Select the thread you want to suspend or resume.
2. Click the Suspend ( ) or Resume ( ) button.

In the example above Thread [Core CM7 0] has been selected (highlighted color). When you click the Suspend button or Resume button this only effects Thread [Core CM7 0].
Update views

When you select a function in a thread, the Source view, the Register view and the Disassembly view are updated to the contents of the thread (either running or suspended - with the latter situation up-to-date information is shown).

Suspend or resume all threads/cores simultaneously

When you select the board, in the picture above Cypress CYTVII-B-H-8M-320BGA Board, you can suspend or resume all cores simultaneously. However, when one of the threads is running and the others are suspended, you must suspend the running thread before you can resume all threads at once via the board selection method. The same applies for suspending all threads at once.

10.6. Programming a Flash Device

With the TASKING debugger you can download an application file to flash memory. Before you download the file, you must specify the type of flash devices you use in your system and the address range(s) used by these devices.

To program a flash device the debugger needs to download a flash programming monitor to the target to execute the flash programming algorithm (target-target communication). This method uses temporary target memory to store the flash programming monitor and you have to specify a temporary data workspace for interaction between the debugger and the flash programming monitor.

Two types of flash devices can exist: on-chip flash devices and external flash devices.

Setup an on-chip flash device

When you specify a target configuration board using the New C/C++ wizard, as explained in the Getting Started manual, any on-chip flash devices are setup automatically.

Setup an external flash device

1. From the Project menu, select Properties for

   The Properties for project dialog appears.

2. In the left pane, expand Run/Debug Settings and select Flash Programming.

   The Flash Programming pane appears.
3. Click **Add...** to specify an external flash device.

*The Select a New Flash Device dialog appears.*
4. In the **Device type** box, expand the name of the manufacturer of the device and select a device.

   The **Sector map displays the memory layout of the flash device(s). Each sector has a size and**

5. In the **Base address** field enter the start address of the memory range that will be covered by the flash device. Any following addresses separated by commas are considered mirror addresses. This allows the flash device to be programmed through its mirror address before switching the flash to its base address.

6. In the **Chip width** field select the width of the flash device.

7. In the **Number of chips** field, enter the number of flash devices that are located in parallel. For example, if you have two 8-bit devices in parallel attached to a 16-bit data bus, enter 2.

8. Fill in the **Number of unused address lines** field, if necessary.

   The flash memory is added to the linker script file automatically with the tag "flash=flash-id".

---

**To program a flash device**

1. From the **Debug** menu, select **Debug Configurations...**

   The **Debug Configurations dialog appears.**

2. In the left pane, select the configuration you want to change, for example, **TASKING C/C++ Debugger » myproject.board.**

3. Open the **Initialization** tab

   The **Flash settings group box should be active.**

4. Enable the option **Use default flash settings (recommended)**

   By default, the flash settings are derived from the .dtc file for the chosen target processor. So, when you change processors the flash settings change automatically. If you do not want that, you can specify your own flash settings. In that case perform steps 5-7, otherwise skip to step 8. You can click **Restore Defaults** to restore the default flash settings.

5. In the **Monitor file** field, specify the filename of the flash programming monitor, usually an Intel Hex or S-Record file.

6. In the **Sector buffer size** field, specify the buffer size for buffering a flash sector.

7. Specify the data **Workspace address** used by the flash programming monitor. This address may not conflict with the addresses of the flash devices.

8. Click **Debug** to program the flash device and start debugging.
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 for
   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.

The following tables give an overview of all tool options on the Tool Settings tab in Eclipse with hyperlinks to the corresponding command line options (if available).

Global Options

<table>
<thead>
<tr>
<th>Eclipse option</th>
<th>Description or option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use global 'product directory' preference</td>
<td>Directory where the TASKING toolset is installed</td>
</tr>
<tr>
<td>Treat warnings as errors</td>
<td>Control program option --warnings-as-errors</td>
</tr>
<tr>
<td>Keep temporary files</td>
<td>Control program option --keep-temporary-files (-t)</td>
</tr>
<tr>
<td>Verbose mode of control program</td>
<td>Control program option --verbose (-v)</td>
</tr>
<tr>
<td>Endianness</td>
<td>Control program option --endianness</td>
</tr>
</tbody>
</table>

C/C++ Compiler

<table>
<thead>
<tr>
<th>Eclipse option</th>
<th>Description or option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preprocessing</td>
<td></td>
</tr>
<tr>
<td>Include CMSIS device register definition header file</td>
<td>C compiler option --include-file</td>
</tr>
<tr>
<td>Description or option</td>
<td>Control program option</td>
</tr>
<tr>
<td>--------------------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Store preprocessor output in <code>&lt;file&gt;.pre</code></td>
<td><code>--preprocess (-E) / --no-preprocessing-only</code></td>
</tr>
<tr>
<td>Keep comments in preprocessor output</td>
<td><code>--preprocess=+comments</code></td>
</tr>
<tr>
<td>Keep #line info in preprocessor output</td>
<td><code>--preprocess=+noline</code></td>
</tr>
<tr>
<td>Defined symbols</td>
<td><code>--define</code></td>
</tr>
<tr>
<td>Pre-include files</td>
<td><code>--include-file</code></td>
</tr>
<tr>
<td><strong>Include Paths</strong></td>
<td></td>
</tr>
<tr>
<td>Add CMSIS include paths</td>
<td><code>--include-directory</code></td>
</tr>
<tr>
<td>Include paths</td>
<td><code>--include-directory</code></td>
</tr>
<tr>
<td><strong>Precompiled C++ Headers</strong></td>
<td></td>
</tr>
<tr>
<td>Automatically use/create precompiled header file</td>
<td><code>--pch</code></td>
</tr>
<tr>
<td>Create precompiled header file</td>
<td><code>--create-pch</code></td>
</tr>
<tr>
<td>Use precompiled header file</td>
<td><code>--use-pch</code></td>
</tr>
<tr>
<td>Precompiled header file directory</td>
<td><code>--pch-dir</code></td>
</tr>
<tr>
<td><strong>Language</strong></td>
<td></td>
</tr>
<tr>
<td>Comply to C++ standard</td>
<td><code>--c++</code></td>
</tr>
<tr>
<td>Comply to C standard</td>
<td><code>--iso</code></td>
</tr>
<tr>
<td>Allow GNU C extensions</td>
<td><code>--language=+gcc</code></td>
</tr>
<tr>
<td>Allow // comments in ISO C90 mode</td>
<td><code>--language=+comments</code></td>
</tr>
<tr>
<td>Check assignment of string literal to non-'const' string pointer</td>
<td><code>--language=-strings</code></td>
</tr>
<tr>
<td>Treat 'char' variables as signed</td>
<td><code>--schar</code></td>
</tr>
<tr>
<td>Allow optimization across volatile access</td>
<td><code>--language=-volatile</code></td>
</tr>
<tr>
<td>Allow Shift JIS Kanji in strings</td>
<td><code>--language=+kanji</code></td>
</tr>
<tr>
<td>Comply to embedded C++ subset</td>
<td><code>--embedded-c++</code></td>
</tr>
<tr>
<td>Support for C++ I/O streams</td>
<td><code>--io-streams</code></td>
</tr>
<tr>
<td>Support for C++ exception handling</td>
<td><code>--exceptions</code></td>
</tr>
<tr>
<td>Support for C++ RTTI (run-time type information)</td>
<td><code>--rtti</code></td>
</tr>
<tr>
<td>Allow the 'wchar_t' keyword (C++)</td>
<td><code>--no-wchar_t-keyword</code></td>
</tr>
<tr>
<td>Allow non-ANSI/ISO C++ features</td>
<td><code>--strict</code></td>
</tr>
<tr>
<td>C++ anachronisms</td>
<td><code>--anachronisms</code></td>
</tr>
<tr>
<td>Allow GNU C++ extensions</td>
<td><code>--g++</code></td>
</tr>
<tr>
<td><strong>Floating-Point</strong></td>
<td></td>
</tr>
<tr>
<td>Floating-point model</td>
<td><code>--fp-model</code></td>
</tr>
<tr>
<td>Eclipse option</td>
<td>Description or option</td>
</tr>
<tr>
<td>---------------------------------------------------------</td>
<td>------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Code Generation</strong></td>
<td></td>
</tr>
<tr>
<td>Use Thumb instruction set</td>
<td>C compiler option <strong>--thumb</strong></td>
</tr>
<tr>
<td>Compile for ARM/Thumb interworking</td>
<td>C compiler option <strong>--interwork</strong></td>
</tr>
<tr>
<td>Use FPU</td>
<td>C compiler option <strong>--fpu</strong></td>
</tr>
<tr>
<td>Alignment for composite types</td>
<td>C compiler option <strong>--align-composites</strong></td>
</tr>
<tr>
<td>Select call mode</td>
<td>C compiler option <strong>--call</strong></td>
</tr>
<tr>
<td><strong>Optimization</strong></td>
<td></td>
</tr>
<tr>
<td>Optimization level</td>
<td>C compiler option <strong>--optimize</strong></td>
</tr>
<tr>
<td>Trade-off between speed and size</td>
<td>C compiler option <strong>--tradeoff</strong></td>
</tr>
<tr>
<td>Maximum size for code compaction</td>
<td>C compiler option <strong>--compact-max-size</strong></td>
</tr>
<tr>
<td>Maximum call depth for code compaction</td>
<td>C compiler option <strong>--max-call-depth</strong></td>
</tr>
<tr>
<td>Always inline function calls</td>
<td>C compiler option <strong>--inline</strong></td>
</tr>
<tr>
<td>Maximum size increment when inlining (in %)</td>
<td>C compiler option <strong>--inline-max-incr</strong></td>
</tr>
<tr>
<td>Maximum size for functions to always inline</td>
<td>C compiler option <strong>--inline-max-size</strong></td>
</tr>
<tr>
<td>Build for application wide optimizations (MIL linking)</td>
<td>Control program option <strong>--mil-link</strong> / <strong>--mil-split</strong></td>
</tr>
<tr>
<td>Application wide optimization mode</td>
<td>Control program option <strong>--mil-link</strong> / <strong>--mil-split</strong></td>
</tr>
<tr>
<td>Custom Optimization</td>
<td>C compiler option <strong>--optimize</strong></td>
</tr>
<tr>
<td>Compilation Speed</td>
<td>C compiler option <strong>--cache</strong></td>
</tr>
<tr>
<td><strong>Debugging</strong></td>
<td></td>
</tr>
<tr>
<td>Generate symbolic debug information</td>
<td>C compiler option <strong>--debug-info</strong></td>
</tr>
<tr>
<td>Static profiling</td>
<td>C compiler option <strong>--profile=+static</strong></td>
</tr>
<tr>
<td>Generate profiling information for block counters</td>
<td>C compiler option <strong>--profile=+block</strong></td>
</tr>
<tr>
<td>Generate profiling information to build a call graph</td>
<td>C compiler option <strong>--profile=+callgraph</strong></td>
</tr>
<tr>
<td>Generate profiling information for function counters</td>
<td>C compiler option <strong>--profile=+function</strong></td>
</tr>
<tr>
<td>Generate profiling information for function timers</td>
<td>C compiler option <strong>--profile=+time</strong></td>
</tr>
<tr>
<td>Exclude time spent in interrupt functions</td>
<td>C compiler option <strong>--profile=+time,+interrupt</strong></td>
</tr>
<tr>
<td>Generate code for bounds checking</td>
<td>C compiler option <strong>--runtime=+bounds</strong></td>
</tr>
<tr>
<td>Generate code to detect unhandled case in a switch</td>
<td>C compiler option <strong>--runtime=+case</strong></td>
</tr>
<tr>
<td>Generate code for malloc consistency checks</td>
<td>C compiler option <strong>--runtime=+malloc</strong></td>
</tr>
<tr>
<td>Generate code for stack overflow checks (allowed for USR and SYS mode only)</td>
<td>C compiler option <strong>--runtime=+stack</strong></td>
</tr>
<tr>
<td>Generate code for division by zero checks</td>
<td>C compiler option <strong>--runtime=+zero</strong></td>
</tr>
<tr>
<td><strong>MISRA C</strong></td>
<td></td>
</tr>
<tr>
<td>MISRA C checking</td>
<td>C compiler option <strong>--misrac</strong></td>
</tr>
<tr>
<td>MISRA C version</td>
<td>C compiler option <strong>--misrac-version</strong></td>
</tr>
</tbody>
</table>
### Eclipse option

<table>
<thead>
<tr>
<th>Description or option</th>
<th>C compiler option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warnings instead of errors for mandatory rules</td>
<td>--misrac-mandatory-warnings</td>
</tr>
<tr>
<td>Warnings instead of errors for required rules</td>
<td>--misrac-required-warnings</td>
</tr>
<tr>
<td>Warnings instead of errors for advisory rules</td>
<td>--misrac-advisory-warnings</td>
</tr>
<tr>
<td>Custom 1998 / Custom 2004 / Custom 2012</td>
<td>--misrac</td>
</tr>
</tbody>
</table>

### CERT C Secure Coding

<table>
<thead>
<tr>
<th>Description or option</th>
<th>C compiler option</th>
</tr>
</thead>
<tbody>
<tr>
<td>CERT C secure code checking</td>
<td>--cert</td>
</tr>
<tr>
<td>Warnings instead of errors</td>
<td>--warnings-as-errors=700-715</td>
</tr>
<tr>
<td>Custom CERT C</td>
<td>--cert</td>
</tr>
</tbody>
</table>

### Diagnostics

<table>
<thead>
<tr>
<th>Description or option</th>
<th>C++ compiler option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Suppress C++ compiler warnings</td>
<td>--no-warnings=\textit{num}</td>
</tr>
<tr>
<td>Suppress C compiler warnings</td>
<td>--no-warnings=\textit{num}</td>
</tr>
<tr>
<td>Suppress all warnings</td>
<td>--no-warnings</td>
</tr>
<tr>
<td>Suppress C++ compiler “used before set” warnings</td>
<td>--no-use-before-set-warnings</td>
</tr>
<tr>
<td>Issue remarks on C++ code</td>
<td>--remarks</td>
</tr>
<tr>
<td>Perform global type checking on C code</td>
<td>--global-type-checking</td>
</tr>
<tr>
<td>Maximum number of emitted errors</td>
<td>--error-limit</td>
</tr>
</tbody>
</table>

### Miscellaneous

<table>
<thead>
<tr>
<th>Description or option</th>
<th>C compiler option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge C source code with generated assembly</td>
<td>--source</td>
</tr>
<tr>
<td>Force definition of virtual function tables (C++)</td>
<td>--force-vtbl</td>
</tr>
<tr>
<td>Suppress definition of virtual function tables (C++)</td>
<td>--suppress-vtbl</td>
</tr>
<tr>
<td>Implicit inclusion of source files for finding templates</td>
<td>--implicit-include</td>
</tr>
<tr>
<td>Minimal inlining of function calls (C++)</td>
<td>--no-inlining</td>
</tr>
<tr>
<td>Instantiation mode of external template entities</td>
<td>--instantiate</td>
</tr>
<tr>
<td>Generated options</td>
<td>C compiler options, Control program options</td>
</tr>
</tbody>
</table>

### Assembler

<table>
<thead>
<tr>
<th>Description or option</th>
<th>Assembler option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preprocessing</td>
<td>--preprocessor-type</td>
</tr>
</tbody>
</table>
### Eclipse option | Description or option
--- | ---
Defined symbols | Assembler option \(--define\)  
Pre-include files | Assembler option \(--include-file\)  
Include Paths |  
Include paths | Assembler option \(--include-directory\)  
Symbols |  
Generate symbolic debug | Assembler option \(--debug-info\)  
Case insensitive identifiers | Assembler option \(--case-insensitive\)  
Emit local EQU symbols | Assembler option \(--emit-locals=+equ\)  
Emit mapping symbols ($a,$t,$d) | Assembler option \(--emit-locals=+mappings\)  
Emit local non-EQU symbols | Assembler option \(--emit-locals=+symbols\)  
Set default symbol scope to global | Assembler option \(--symbol-scope\)  
List File |  
Generate list file | Control program option \(--list-files\)  
List … | Assembler option \(--list-format\)  
List section summary | Assembler option \(--section-info=+list\)  
Diagnostics |  
Suppress warnings | Assembler option \(--no-warnings=num\)  
Suppress all warnings | Assembler option \(--no-warnings\)  
Display section summary | Assembler option \(--section-info=+console\)  
Maximum number of emitted errors | Assembler option \(--error-limit\)  
Miscellaneous |  
Use full assembler for mixed ARM and Thumb instructions | Control program option \(--mixed-arm-thumb\)  
Assemble Thumb instructions by default | Control program option \(--thumb\)  
Allow 2-operand form for 3-operand instructions | Assembler option \(--relaxed\)  
UAL syntax mode | Assembler option \(--old-syntax\)  
Allow instruction inversions | Assembler option \(--inversions\)  
Allow Shift JIS Kanji in strings | Assembler option \(--kanji\)  
Additional options | Assembler options  

### Linker

### Eclipse option | Description or option
--- | ---
Output Format |  
Generate Intel Hex format file | Linker option \(--output=\textit{file}:IHEX\)  
Generate S-records file | Linker option \(--output=\textit{file}:SREC\)  
Generate binary file | Linker option \(--chip-output=:\textit{BIN}:0\)
<table>
<thead>
<tr>
<th><strong>Eclipse option</strong></th>
<th><strong>Description or option</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Create file for each memory chip</td>
<td>Linker option --chip-output</td>
</tr>
<tr>
<td>Size of addresses (in bytes) for Intel Hex records</td>
<td>Linker option --output=file:IHEX:size</td>
</tr>
<tr>
<td>Size of addresses (in bytes) for Motorola S records</td>
<td>Linker option --output=file:SREC:size</td>
</tr>
<tr>
<td>Emit start address record</td>
<td>Linker option --hex-format=s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Libraries</strong></th>
<th><strong>Control program option</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Link default libraries</td>
<td>--no-default-libraries</td>
</tr>
<tr>
<td>Link CMSIS DSP library</td>
<td>--dsp-library</td>
</tr>
<tr>
<td>Rescan libraries to solve unresolved externals</td>
<td>--no-rescan</td>
</tr>
<tr>
<td>Libraries</td>
<td>The libraries are added as files on the command line.</td>
</tr>
<tr>
<td>Library search path</td>
<td>--library-directory</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Data Objects</strong></th>
<th><strong>Linker option</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Data objects</td>
<td>--import-object</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Script File</strong></th>
<th><strong>Linker option</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Defined symbols</td>
<td>--define</td>
</tr>
<tr>
<td>Linker script file (.lsl)</td>
<td>--lsl-file</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Optimization</strong></th>
<th><strong>Linker option</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Delete unreferenced sections</td>
<td>--optimize=c</td>
</tr>
<tr>
<td>Use a 'first-fit decreasing' algorithm</td>
<td>--optimize=l</td>
</tr>
<tr>
<td>Compress copy table</td>
<td>--optimize=t</td>
</tr>
<tr>
<td>Delete duplicate code</td>
<td>--optimize=x</td>
</tr>
<tr>
<td>Delete duplicate data</td>
<td>--optimize=y</td>
</tr>
<tr>
<td>Compress ROM sections of copy table items</td>
<td>--optimize=z</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Map File</strong></th>
<th><strong>Control program option</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Generate map file (.map)</td>
<td>--no-map-file</td>
</tr>
<tr>
<td>Generate XML map file format (.mapxml) for map file viewer</td>
<td>--map-file=file.mapxml:XML</td>
</tr>
<tr>
<td>Include ...</td>
<td>--map-file-format</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Diagnostics</strong></th>
<th><strong>Linker option</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Suppress warnings</td>
<td>--no-warnings=</td>
</tr>
<tr>
<td>Suppress all warnings</td>
<td>--no-warnings</td>
</tr>
<tr>
<td>Maximum number of emitted errors</td>
<td>--error-limit</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Miscellaneous</strong></th>
<th><strong>Linker option</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Strip symbolic debug information</td>
<td>--strip-debug</td>
</tr>
<tr>
<td>Link case insensitive</td>
<td>--case-insensitive</td>
</tr>
<tr>
<td>Eclipse option</td>
<td>Description or option</td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td>Do not use standard copy table for initialization</td>
<td>Linker option <code>--user-provided-initialization-code</code></td>
</tr>
<tr>
<td>Show link phases during processing</td>
<td>Linker option <code>--verbose</code></td>
</tr>
<tr>
<td>Generate long-branch veneers</td>
<td>Linker option <code>--long-branch-veneers</code></td>
</tr>
<tr>
<td>Application is not romable</td>
<td>Linker option <code>--non-romable</code></td>
</tr>
<tr>
<td>Additional options</td>
<td>Linker options</td>
</tr>
</tbody>
</table>
11.1. Configuring the Command Line Environment

If you want to use the tools on the command line (using a Command Prompt), you can set environment variables.

You can set the following environment variables:

<table>
<thead>
<tr>
<th>Environment variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASARMINC</td>
<td>With this variable you specify one or more additional directories in which the assembler looks for include files. See Section 6.4, <em>How the Assembler Searches Include Files</em>.</td>
</tr>
<tr>
<td>CARMINC</td>
<td>With this variable you specify one or more additional directories in which the C compiler looks for include files. See Section 4.3, <em>How the Compiler Searches Include Files</em>.</td>
</tr>
<tr>
<td>CPARMINC</td>
<td>With this variable you specify one or more additional directories in which the C++ compiler looks for include files. See Section 5.2, <em>How the C++ Compiler Searches Include Files</em>.</td>
</tr>
<tr>
<td>CCARMBIN</td>
<td>When this variable is set, the control program prepends the directory specified by this variable to the names of the tools invoked.</td>
</tr>
<tr>
<td>LIBCARM</td>
<td>With this variable you specify the directory in which the executables reside. This allows you to call the executables when you are not in the <code>bin</code> directory. Usually your system already uses the PATH variable for other purposes. To keep these settings, you need to add (rather than replace) the path. Use a semicolon (<code>;</code>) to separate path names.</td>
</tr>
<tr>
<td>PATH</td>
<td>With this variable you specify the location where programs can create temporary files. Usually your system already uses this variable. In this case you do not need to change it.</td>
</tr>
</tbody>
</table>

See the documentation of your operating system on how to set environment variables.
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 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 for**

   *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:

carm -Oac test.c
carm --optimize=+coalesce,+cse test.c

When you do not specify an option, a default value may become active.
C compiler option: --align-composites

Menu entry
2. Select the Alignment for composite types: Natural alignment or Optimal alignment.

Command line syntax

```
--align-composites=alignment
```

You can specify the following alignments:

```
 n   Natural alignment (default)
o   Optimal alignment
```

Description

With this option you can set the alignment for composite types (structs, unions and arrays).

Natural alignment (n) uses the natural alignment of the most-aligned member of the composite type.

Optimal alignment (o) sets the alignment to 8, 16, or 32 bits depending on the size of the composite type.

Example

Consider the following structure of three chars.

```
struct s
{
    char a;
    char b;
    char c;
} s_var;
```

With natural alignment this results in a size of s_var of three bytes.

```
s_var .type object
    .size s_var,3
    .ds 3
```

With --align-composites=o, s_var is padded with one extra byte, so that the contents of the struct is aligned at 4 bytes.

```
s_var .type object
    .size s_var,4
    .ds 4
```
Related information

-
C compiler option: --cache

Menu entry
1. Select C/C++ Compiler » Optimization » Compilation Speed.
2. Enable the option Cache generated code to improve the compilation speed.
3. In the Directory for cached files field, enter the name for the location of the cache.

Command line syntax

--cache[=directory]

Default on command line: . (current directory)

Default in Eclipse: .cache directory under project directory

Description

This option enables a cache for output files in the specified directory. When the source code after preprocessing and relevant compiler options and the compiler version are the same as in a previous invocation, the previous result is copied to the output file. The cache only works when there is a single C input file and a single output file (no --mil-split).

You can also enable the cache and specify the cache directory with the environment variable CARMCACHE. This option takes precedence over the environment variable.

The cache directory may be shared, for instance by placing it on a network drive.

The compiler creates a directory carmcache in the directory specified with the option --cache or the environment variable CARMCACHE. The directory is only created when it does not yet exist. The cache files are stored in this directory.

Example

To improve the compilation speed and put cached files in directory .cache, enter:

carm --cache=.cache test.c

Related information

Section 12.5, Compiler Cache

Section 9.6, Expire Cache Utility
C compiler option: --call (-m)

Menu entry
2. Set the option Select call mode to Use PC-relative calls (default) or to Use 32-bit indirect calls.

Command line syntax
```
--call={far|near}
-m{f|n}
```

Description
To address the memory of the ARM, you can use two different call modes:

- **far**: 32-bit indirect calls. Though you can address the full range of memory, the address is first loaded into a register after which the call is executed.
- **near**: 26-bit PC-relative call. The PC-relative call is directly coded into the B instruction. This way of calling results in higher execution speed. However, not the full range of memory can be addressed with near calls.

If you compile your C source with near calls but the called address cannot be reached with a near call, the linker will generate an error.

It is recommended to use the near addressing mode unless your application needs calls to addresses that fall outside a 256 MB region.

Example
Consider the following function q which calls function p:

```c
extern int p(int a, int b, int c, int d)
{
    return a+b+c+d + a*b*c*d;
}

extern int iii;
void q(void)
{
    iii=p(42,43,44,45);
}
```

With **--call=near**, this results into

```
q:   .type   func
```
With --call=far, this results into

q: .type func

stmfd sp!,{r4,lr}

ldr r4,.L3 ; address loaded in register
mov r3,#45
mov r2,#44
mov r1,#43
mov r0,#42
mov lr,pc
bx r4 ; indirect call
ldr r4,.L3+4
str r0,[r1,#0]
ldmfd sp!,{r4,pc}
.align 4

.L3:
.dw p
.dw iii
.size q,$-q

.calls '__INDIRECT__','p' ; indirect calls
.calls 'q','__INDIRECT__'
.calls 'p','__INDIRECT__'
.calls 'q','' ,8
.calls 'q','' ,8
.extern iii
.extern __INDIRECT__
Related information

Function qualifier __far

Pragma call {near | far}

Linker option --long-branch-veeers
C compiler option: --cert

Menu entry

1. Select C/C++ Compiler » CERT C Secure Coding.
2. Make a selection from the CERT C secure code checking list.
3. If you selected Custom, expand the Custom CERT C entry and enable one or more individual recommendations/rules.

Command line syntax

```
--cert={all | name[-name],...}
```

Default format: all

Description

With this option you can enable one or more checks for CERT C Secure Coding Standard recommendations/rules. When you omit the argument, all checks are enabled. name is the name of a CERT recommendation/rule, consisting of three letters and two digits. Specify only the three-letter mnemonic to select a whole category. For the list of names you can use, see Chapter 20, CERT C Secure Coding Standard.

On the command line you can use `--diag=cert` to see a list of the available checks, or you can use a three-letter mnemonic to list only the checks in a particular category. For example, `--diag=pre` lists all supported preprocessor checks.

Example

To enable the check for CERT rule STR30-C, enter:

```
carm --cert=str30 test.c
```

Related information

Chapter 20, CERT C Secure Coding Standard

C compiler option `--diag` (Explanation of diagnostic messages)
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: --code-core-association

Menu entry
1. Select C/C++ Compiler » Allocation.
2. Select a Code core association.

Command line syntax
--code-core-association=value

You can specify the following arguments:

- **clone**: Multiple code instances, each executed by one core
- **tcm**: One code instance executed by core in single-core configuration
- **tcmn**: One code instance executed by core $n$ in virtual-core configuration. $n$ can be 0 or 1.
- **share**: One code instance shared between cores.

Description
You can use this option to specify the default core association for functions.

Example
To specify that code instances can only be executed by core 0, enter:

```bash
carm test.c --processors="vendor_support_dir\etc\processors.xml"
   --cpu=cyt4bf8ceb_m7 --code-core-association=tcm0
```

Related information
Section 1.5.1.2, Code Core Association
C compiler option --data-core-association (Data core association)
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:

carm --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: --control-flow-info

Menu entry
1. Select C/C++ Compiler » Debugging.
2. Enable the option Generate control flow information.

Command line syntax
--control-flow-info

Description
With this option the compiler adds control flow information to the output file. The compiler generates a .debug_control_flow section which describes the basic blocks and their relations. This information can be used for code coverage analysis on optimized code.

Example
carm --control-flow-info test.c

Related information
Section 9.5.2, HLL Dump Output Format
C compiler option --debug-info (Debug information)
C compiler option: --cpu (-C)

Menu entry

1. Expand C/C++ Build and select Processor.
2. From the Processor selection list, make a selection by Architecture, Core or one of the manufacturers.

Command line syntax

--cpu=architecture

-Carchitecture

You can specify the following architectures:

- ARMv6M Compile for ARMv6-M architecture
- ARMv7M Compile for ARMv7-M architecture
- ARMv7EM Compile for ARMv7E-M architecture

Description

With this option you specify the ARM architecture for which you create your application. The ARM target supports more than one architecture and therefore you need to specify for which architecture the compiler should compile. The architecture determines which instructions are valid and which are not.

You choose one of the following architectures: ARMv6-M, ARMv7-M or ARMv7E-M. The compiler sets the Thumb instruction set implicitly (option --thumb).

The macro __CPU_arch__ is defined in the C source file. The arch is converted to uppercase.

When you call the compiler from the command line, make sure you specify the same core type to the assembler to avoid conflicts!

Example

To compile the file test.c for the ARMv7E-M processor type, enter the following on the command line:

carm --cpu=ARMv7EM test.c

The compiler compiles for the chosen processor type.

Related information

Control program option --cpu (Select architecture)
Assembler option --cpu (Select architecture)
C compiler option: --data-core-association

Menu entry
1. Select C/C++ Compiler » Allocation.
2. Select a Data core association.

Command line syntax

```bash
--data-core-association=value
```

You can specify the following arguments:

- **clone**: Multiple data instances, each accessed by one core
- **tcm**: One data instance accessed by core in single-core configuration
- **tcmn**: One data instance accessed by core n in virtual-core configuration. n can be 0 or 1.
- **share**: One data instance shared between cores.

Description

You can use this option to specify the default core association for data objects.

Example

To specify that data instances can only be accessed by core 1, enter:

```bash
ccarm test.c --processors="vendor_support_dir\etc\processors.xml"
--cpu=cyt4bf8ceb_m7 --data-core-association=tcm1
```

Related information

Section 1.5.1.1, Data Core Association

C compiler option **--code-core-association** (Code core association)
C compiler option: --debug-info (-g)

Menu entry

1. Select C/C++ Compiler » Debugging.

2. To generate symbolic debug information, select Default, Small set 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>small</td>
<td>Emit small set of debug information.</td>
</tr>
<tr>
<td>default</td>
<td>Emit default symbolic debug information.</td>
</tr>
<tr>
<td>all</td>
<td>Emit full 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.

The DWARF debug format allows for a flexible approach as to how much symbolic information is included, as long as the structure is valid. Adding all possible DWARF data for a program is not practical. The amount of DWARF information per compilation unit can be huge. And for large projects, with many object modules the link time can grow unacceptably long. That is why the compiler has several debug information levels. In general terms one can say, the higher the level the more DWARF information is produced.

The DWARF data in an object module is not only used for debugging. The toolset can also do "type checking" of the whole application. In that case the linker will use the DWARF information of all object modules to determine if every use of a symbol is done with the same type. In other words, if the application is built with type checking enabled then the compiler will add DWARF information too.

Small set of debug information

With this suboption only DWARF call frame information and type information are generated. This enables you to inspect parameters of nested functions. The type information improves debugging. You can perform a stack trace, but stepping is not possible because debug information on function bodies is not generated. You can use this suboption, for example, to compact libraries.
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 suboption extra debug information is generated about unused typedefs and DWARF "lookup table sections". Under normal circumstances this extra debug information is not needed to debug the program. Information about unused typedefs concerns all typedefs, even the ones that are not used for any variable in the program. (Possibly, these unused typedefs are listed in the standard include files.) With this suboption, the resulting assembler/object file will increase significantly.

In the following table you see in more detail what DWARF information is included for the debug option levels.

<table>
<thead>
<tr>
<th>Feature</th>
<th>-g1</th>
<th>-g2</th>
<th>-g3</th>
<th>type check</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>basic info</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>info such as symbol name and type</td>
</tr>
<tr>
<td>call frame</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>this is information for a debugger to compute a stack trace when a program has stopped at a breakpoint</td>
</tr>
<tr>
<td>symbol lifetime</td>
<td></td>
<td>+</td>
<td>+</td>
<td></td>
<td>this is information about where symbols live (e.g. on stack at offset so and so, when the program counter is in this range)</td>
</tr>
<tr>
<td>line number info</td>
<td></td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>file name, line number, column number</td>
</tr>
<tr>
<td>&quot;lookup tables&quot;</td>
<td></td>
<td></td>
<td>+</td>
<td></td>
<td>DWARF sections ... this is an optimization for the DWARF data, it is not essential</td>
</tr>
<tr>
<td>unused typedefs</td>
<td></td>
<td></td>
<td></td>
<td>+</td>
<td>in the C/G++ code of the program there can be (many) typedefs that are not used for any variable. Sometimes this can cause enormous expansion of the DWARF data and thus it is only included in -g3.</td>
</tr>
</tbody>
</table>

Related information
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, 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 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.

Make sure you do not use a reserved keyword as a macro name, as this can lead to unexpected results.

Example
Consider the following C program with conditional code to compile a demo program and a real program:

```
void main( void )
{
    #if DEMO
        demo_func();    /* compile for the demo program */
    #else
        real_func();    /* compile for the real program */
    #endif
}
```
You can now use a macro definition to set the DEMO flag:

```
carm --define=DEMO test.c
carm --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.

```
carm --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

Eclipse uses this option in the background to create a file with extension .d (one for every input file).

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

```
carm --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`. For example:

```
test.obj :   test.c
  test.c :
  test.obj :   <install-dir>/carm/include/stdio.h
<install-dir>/carm/include/stdio.h :
  test.obj :   <install-dir>/carm/include/stdarg.h
<install-dir>/carm/include/stdarg.h :
```

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 » TASKING » 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

```bash
--diag=[format:]{all | msg[-msg],...}
```

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. The compiler does not compile any files. 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 (except for the CERT checks). If you want the description of one or more selected error messages, you can specify the error message numbers, separated by commas, or you can specify a range.

With **--diag=cert** you can see a list of the available CERT checks, or you can use a three-letter mnemonic to list only the checks in a particular category. For example, **--diag=pre** lists all supported preprocessor checks.

Example

To display an explanation of message number 282, enter:

```
carm --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:

carm --diag=html:all > cerrors.html

Related information

Section 4.7, C Compiler Error Messages

C compiler option --cert (Enable individual CERT checks)
C compiler option: --endianness

Menu entry
1. Select Global Options.
2. Specify the Endianness: Little-endian mode or Big-endian mode.

Command line syntax

--endianness=endianness

-B
--big-endian

You can specify the following endianness:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>big</td>
<td>b</td>
</tr>
<tr>
<td>little</td>
<td>l</td>
</tr>
</tbody>
</table>

Description

By default, the compiler generates code for a little-endian target (least significant byte of a word at lowest byte address). With --endianness=big the compiler generates code for a big-endian target (most significant byte of a word at lowest byte address). -B is an alias for option --endianness=big.

Related information

-
C compiler option: --error-file

Menu entry
-

Command line syntax

--error-file [=file]

Description
With this option the compiler redirects diagnostic messages to a file. If you do not specify a filename, the error file will be named after the output file with extension .err.

Example
To write diagnostic messages to errors.err instead of stderr, enter:

carm --error-file=errors.err test.c

Related information
-

C compiler option: --error-limit

Menu entry

1. Select C/C++ Compiler » 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 limit the number of error messages in one compiler run to the specified number. When the limit is exceeded, the compiler aborts with fatal error message F105. Warnings and informational messages are not included in the count. When 0 (zero) or a negative number is specified, the compiler emits all errors. Without this option the maximum number of errors is 42.

Related information

Section 4.7, C Compiler Error Messages
C compiler option: --fp-model

Menu entry
1. Select C/C++ Compiler » Floating-Point.
2. Make a selection from the Floating-point model list.
3. If you selected Custom, enable one or more individual options.

Command line syntax
--fp-model=flags

You can set the following flags:

+/-float f/F treat 'double' as 'float'
+/-rewrite r/R allow expression rewriting
+/-negzero z/Z ignore sign of -0.0

1 alias for --fp-model=Frz
2 alias for --fp-model=Fr
3 alias for --fp-model=frz

Default: --fp-model=Frz

Description
With this option you select the floating-point execution model.

With --fp-model=+float you tell the compiler to treat variables and constants of 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. Make sure you specify the corresponding libraries to the linker.

With --fp-model=+rewrite you allow the compiler to rewrite expressions by reassociating. This might result in rounding differences and possibly different exceptions. An example is to rewrite (a*c)+(b*c) as (a+b)*c.

With --fp-model=+negzero you allow the compiler to ignore the sign of -0.0 values. An example is to replace (a-a) by zero.

Related information
Pragmas fp_negzero and fp_rewrite in Section 1.9, Pragmas to Control the Compiler.
C compiler option: --fpu

Menu entry
2. Enable the option Use FPU.

Command line syntax

--fpu=fp

You can specify the following arguments:

- FPv4-sp alias for VFPv4-sp
- FPv5 alias for VFPv5
- FPv5-sp alias for VFPv5-sp
- VFPv2 alias for VFPv3
- VFPv3 Compile for VFPv3 architecture
- VFPv3-sp Compile for VFPv3-sp architecture
- VFPv4-sp Compile for VFPv4-sp architecture
- VFPv5 Compile for VFPv5 architecture
- VFPv5-sp Compile for VFPv5-sp architecture
- none Compile for software FPU library (default)

Description

With this option you define the kind of FPU support with which you create your application. The v4 versions generate fused multiply-add operations instead of the regular compound operations. The v5 versions add VSEL, VMINNM/VMAXNM, several conversion and rounding instructions. The "sp" suffix denotes single precision floating-point only.

The macro __FPU_fpu__ is defined in the C source file. The fpu is converted to uppercase and the lowercase "v" and the '-' will be removed. Also when "none" is not used, the macro __FPU_VFP__ is defined in the C source file, otherwise the macro __FPU_NONE__ is defined.

Example

To compile the file test.c for the VFPv3-sp architecture, enter the following on the command line:

carm --fpu=VFPv3-sp test.c

This defines the symbols __FPU_VFP__ and __FPU_VFP3SP__.

Related information

__fpu function qualifier
C compiler option: --global-type-checking

Menu entry

1. Select C/C++ Compiler » Diagnostics.

2. Enable the option Perform global type checking on C code.

Command line syntax

--global-type-checking

Description

The C compiler already performs type checking within each module. Use this option when you want the linker to perform type checking between modules.

Related information

-
C compiler option: --help (-?)

Menu entry
-

Command line syntax

--help [=item]
-

You can specify the following arguments:

- `intrinsic`   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:

```
carm -?
carm --help
carm
```

The following invocation displays a list of the available pragmas:

```
carm --help=pragmas
```

Related information
-

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.
4. Optionally enable the option Add CMSIS include paths.

   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 or paths that are specified with this option. Multiple paths/options are handled by the C compiler from left to right. From Eclipse, the CMSIS include paths, if enabled, are the first option, so they are searched first.
3. The path that is specified in the environment variable CARMINC when the product was installed.
4. The default directory $(PRODDIR)\include (unless you specified option --no-stdinc).

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:

carm --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** `--no-stdinc` (Skip standard include files directory)
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.
4. (Optional) Enable the option Include CMSIS device register definition header 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,...
-Hfile,...

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

carm --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: --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:

carm --optimize=+compact --inline test.c

Related information
C compiler option --optimize=+compact (Optimization: code compaction)

Section 1.12.2, Inlining Functions: inline
C compiler option: --inline-max-incr / --inline-max-size

Menu entry
1. Select C/C++ Compiler » Optimization.
2. In the Maximum size increment when inlining field, enter a value (default -1).
3. In the Maximum size for functions to always inline field, enter a value (default -1).

Command line syntax

--inline-max-incr=percentage (default: -1)
--inline-max-size=threshold (default: -1)

Description

With these options you can control the automatic function inlining optimization process of the compiler. These options only have 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 -1, which means that the threshold depends on the option --tradeoff.

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. The default value is -1, which means that the value depends on the option --tradeoff.

Example

carm --optimize=+inline --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 1.12.2, Inlining Functions: inline
Section 4.5.3, Optimize for Code Size or Execution Speed
C compiler option: --interwork

Menu entry

2. Enable the option Compile for ARM/Thumb interworking.

Command line syntax

--interwork

Description

With this option the compiler generates code which supports calls between functions with the ARM and Thumb instruction set.

Use this option if your program consists of both ARM and Thumb functions.

By default this option is disabled, since it produces slightly larger code.

Related information

C compiler option --thumb (Use Thumb instruction set)
C compiler option: --iso (-c)

Menu entry
1. Select C/C++ Compiler » Language.
2. From the Comply to C standard list, select ISO C99, ISO C11, or ISO C90.

Command line syntax
--iso=90|99|11
-c90|99|11

Default: --iso=11

Description

Example
To select the ISO C99 standard on the command line:
carm --iso=99 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

carm --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 of string literal to non-'const' string pointer
   • Allow optimization across volatile access
   • Allow Shift JIS Kanji in strings

Command line syntax

--language=flags
-Aflags

You can set the following flags:

+/-gcc g/G enable a number of gcc extensions
+/-kanji k/K support for Shift JIS Kanji in strings
+/-comments p/P // comments in ISO C90 mode
+/-volatile v/V don't optimize across volatile access
+/-strings x/X relaxed const check for string literals

Default: -AGKpVx
Default (without flags): -AGKPVX

Description

With this option you control the language extensions the compiler can accept. By default the ARM 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.
TASKING VX-toolset for ARM User Guide

- Alternative syntax for designated initializers.
- Allow zero sized arrays.
- Allow empty struct/union.
- Allow unnamed struct/union fields.
- 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.

For a more complete description of these extensions, you can refer to the UNIX gcc info pages (info gcc).

Shift JIS Kanji support

With `--language=+kanji (-Ak)` you tell the compiler to support Shift JIS encoded Kanji multi-byte characters in strings, (wide) character constants and // comments. Without this option, encodings with 0x5c as the second byte conflict with the use of the backslash as an escape character. Shift JIS in /*...*/ comments is supported regardless of this option. Note that Shift JIS also includes Katakana and Hiragana.

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 later ISO C versions, this style of comments is always accepted.

Check assignment of string literal to non-const string pointer

With `--language=+strings (-Ax)` you disable warnings about discarded const qualifiers when a string literal is assigned to a non-const pointer.

```c
char *p;
int main( void )
{
    p = "hello"; // with -AX the compiler issues warning W525
```
Optimization across volatile access

With the --language=+volatile (-Av) option, the compiler will block optimizations when reading or writing a volatile object, by executing all memory and (SFR) register accesses before the access of the volatile object. The volatile access acts as a memory barrier. With this option you can prevent for example that code below the volatile object is optimized away to somewhere above the volatile object.

Example:

```c
extern unsigned int variable;
extern volatile unsigned int access;

void TestFunc( unsigned int flag )
{
    access = 0;
    variable |= flag;
    if( variable == 3 )
    {
        variable = 0;
    }
    variable |= 0x8000;
    access = 1;
}
```

Result with --language=-volatile (default):

```
TestFunc:   .type   func
str     lr,[sp,#-4]!
ldr     r1,.L3
ldr     lr,.L3+4
ldr     r2,[r1,#0]       ; <== Moved across volatile access
mov     r3,#0
orr     r0,r2,r0
cmp     r0,#3
str     r3,[lr,#0]       ; <== Volatile access
bne     .L2
orr     r0,r0,#32768
mov     r2,#1
str     r2,[lr,#0]       ; <== Volatile access
str     r0,[r1,#0]       ; <== Moved across volatile access
ldr     pc,[sp],#4
.size   TestFunc,$-TestFunc
.align   4
.L2:
    .call
.L3:
    .dw     variable
    .dw     access
```
Result with --language=+volatile:

```
TestFunc: .type func
    str lr,[sp,#-4]!
    ldr r3,.L3
    ldr r2,.L3+4
    ldr lr,[r3,#0]
    mov r1,#0
    orr r0,lr,r0
    cmp r0,#3
    str r1,[r2,#0] ; <= Volatile access
    str r0,[r3,#0] ; <= Not moved
    bne .L2
    str r1,[r3,#0]
.L2:
    ldr r0,[r3,#0]
    orr r0,r0,#32768
    str r0,[r3,#0] ; <= Not moved
    mov r0,#1
    str r0,[r2,#0] ; <= Volatile access
    ldr pc,[sp],#4
.size TestFunc,$-TestFunc
.align 4
.L3:
    .dw variable
    .dw access
```

Note that the volatile behavior of the compiler with option --language=-volatile or --language=+volatile is ISO C compliant in both cases.

**Related information**

- **C compiler option --iso** (ISO C standard)

  Section 1.6, *Shift JIS Kanji Support*
C compiler option: --make-target

Menu entry

-  

Command line syntax

--make-target=\textit{name}

Description

With this option you can overrule the default target name in the make dependencies generated by the options \texttt{--preprocess=+make (-Em)} and \texttt{--dep-file}. The default target name is the basename of the input file, with extension \texttt{.obj}.

Example

carm \texttt{--preprocess=+make --make-target=mytarget.obj test.c}

The compiler generates dependency lines with the default target name \texttt{mytarget.obj} instead of \texttt{test.obj}.

\texttt{mytarget.obj : test.c}
\texttt{test.c :}

Related information

\textbf{C compiler option} \texttt{--preprocess=+make} (Generate dependencies for make)

\textbf{C compiler option} \texttt{--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
1. Select C/C++ Compiler » Optimization.
2. Enable the option Build for application wide optimizations (MIL linking).
3. Select Optimize less/Build faster or Optimize more/Build slower.

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. Application wide code compaction is not possible in this case.

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.

Note that with both options some extra strict type checking is done that can cause building to fail in a way that is unforeseen and difficult to understand. For example, when you use one of these options in combination with option --schar and you link the MIL library, you might get the following error:

```
carm E289: ["..\..\.\..\strlen.c" 14/1] "strlen" redeclared with a different type
carm I802: ["installation-dir\include\string.h" 44/17]
    previous declaration of "strlen"
1 errors, 0 warnings
```

This is caused by the fact that the MIL library is built without --schar. You can workaround this problem by rebuilding the MIL libraries.

Build for application wide optimizations (MIL linking) and Optimize less/Build faster

This option is standard MIL linking and splitting. Note that you can control the optimizations to be performed with the optimization settings.
Optimize more/Build slower

When you enable this option, the compiler's frontend does not split the MIL stream in separate modules, but feeds it directly to the compiler's backend, allowing the code compaction to be performed application wide.

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 1998, Custom 2004 or Custom 2012 entry and enable one or more individual rules.

Command line syntax

--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

carm --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 4.6.2, C Code Checking: MISRA C

C compiler option --misrac-mandatory-warnings

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 / --misrac-mandatory-warnings

Menu entry

1. Select C/C++ Compiler » MISRA C.
2. Make a selection from the MISRA C checking list.
3. Enable one or more of the options:
   - Warnings instead of errors for mandatory rules
   - Warnings instead of errors for required rules
   - Warnings instead of errors for advisory rules.

Command line syntax

--misrac-advisory-warnings
--misrac-required-warnings
--misrac-mandatory-warnings

Description

Normally, if an advisory rule, required rule or mandatory 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 4.6.2, 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|2012}

Default: 2004

Description

MISRA C rules exist in three versions: MISRA C:1998, MISRA C:2004 and MISRA C:2012. By default, the C source is checked against the MISRA C:2004 rules. With this option you can select which version to use.

Related information

Section 4.6.2, C Code Checking: MISRA C

C compiler option --misrac
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

C compiler option --include-directory (Add directory to include file search path)

Section 4.3, How the Compiler Searches Include Files
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 C compiler warnings box.

3. Enter the numbers, separated by commas or as a range, 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[-number],...]
-w[number[-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 or a range, only the specified warnings are suppressed. You can specify the option --no-warnings=number multiple times.

Example

To suppress warnings 537 and 538, enter:

carm test.c --no-warnings=537,538

Related information

C compiler option --warnings-as-errors (Treat warnings as errors)

Pragma warning
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]
-flags

You can set the following flags:

+/-coalesce a/A Coalescer: remove unnecessary moves
+/-ipro b/B Interprocedural register optimizations
+/-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
+/-sign j/J Sign extend elimination
+/-schedule k/K Instruction scheduler
+/-loop l/L Loop transformations
+/-forward o/O Forward store
+/-propagate p/P Constant propagation
+/-compact r/R Code compaction (reverse inlining)
+/-script s/S Subscript strength reduction
+/-unroll u/U Unroll small loops
+/-peehole y/Y Peephole optimizations
+/-cluster Cluster global variables

Use the following options for predefined sets of flags:

--optimize=0 -O0 No optimization
Alien for -OaBCFGIJKLOPRSUY,-cluster

No optimizations are performed except for the coalescer (to allow better debug information). 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.
Optimize
alias for -OabceFGiJKLOPRSUy,-cluster

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.

--optimize=2 -O2
Optimize more (default)
alias for -OabcefgIJklopRsUy,-cluster

Enables more optimizations to reduce code size and/or execution time. This is the default optimization level.

--optimize=3 -O3
Optimize most
alias for -OabcefgiJkloprsuy,+cluster

This is the highest optimization level. Use this level to decrease execution time to meet your real-time requirements.

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:

carm test.c

carm --optimize=2 test.c
carm -O2 test.c

carm --optimize test.c
carm -O test.c

carm -OabcefgIJKloprsUy test.c
carm --optimize=+coalesce,+ipro,+cse,+expression,+flow,+glo,
    -inline,-sign,+schedule,+loop,+forward,+propagate,
    +compact,+subscript,-unroll,+peephole,-cluster test.c
Related information

C compiler option --tradeoff (Trade off between speed and size)

Pragma optimize/endoptimize

Section 4.5, 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,...
-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 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:

```plaintext
--debug-info
--define=DEMO=1
test.c
```

Specify the option file to the compiler:

```plaintext
carm --option-file=myoptions
```

This is equivalent to the following command line:

```plaintext
carm --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:
carm --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 preprocessor output in <file>.pre.
3. (Optional) Enable the option Keep comments in preprocessor output.
4. (Optional) Enable the option Keep #line info in preprocessor output.

Command line syntax

```
--preprocess [=flags]
-E[flags]
```

You can set the following flags:

```
+/-comments c/C  keep comments
+/-includes i/I  generate a list of included source files
+/-list l/L  generate a list of macro definitions
+/-make m/M  generate dependencies for make
+/-noline p/P  strip #line source position information
```

Default: -ECILMP

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=+includes the compiler will generate a list of all included source files. The preprocessor output is discarded.

With --preprocess=+list the compiler will generate a list of all macro definitions. The preprocessor output is discarded.

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 .obj. 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**

carm --preprocess=+comments,+includes,-list,-make,-noline test.c --output=test.pre

The compiler preprocesses the file test.c and sends the output to the file test.pre. Comments and a list of all included source files are included but no list of macro definitions and 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
4. Optionally, when you selected function timers, enable option Exclude time spent in interrupt functions.

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-info 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: -pBcFStI

You can set the following flags:

+-block  b/B  block counters
+-callgraph  c/C  call graph
+-function  f/F  function counters
+-interrupt  i/I  exclude interrupt time
+-static  s/S  static profile generation
+-time  t/T  function timers

Default: -pBcFStI
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 13, Profiling.

You can obtain the following profiling data (see flags above):

**Block counters (not in combination with Call graph or Function timers)**

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.

**Function timers (not in combination with Block counters/Function counters)**

This will instrument the code to measure the time spent in a function. This includes the time spent in all sub functions (callees). Also the time spent in interrupt functions is taken into account, unless you enable option *Exclude time spent in interrupt functions* (-pti).

**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 7.3, 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:

carm --profile=+block test.c

In this case you must link the library libpb.a.
Related information

Chapter 13, 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=[type=]suffix|-f|-m|-fm
-R[type=]suffix|-f|-m|-fm
```

Description
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. You can then use this unique section name in the linker script file for locating. Because sections have reserved names, the compiler will not actually change the section name, but will add a suffix to the name. Any core association stays part of the name.

With the section type `type` you select which sections are renamed. With `suffix` you specify the suffix part which will be attached to the existing name. You can specify a user defined name, or you can use one of the following suffix specifiers:

- `{module}` expands to the module name
- `{name}` expands to the symbol name as generated in the assembly file, including compiler generated prefixes and suffixes
- `{cname}` expands to the symbol name as used in your C source. Compiler generated names will be cleaned up and prefixed by a '$'.

The `-f`, `-m`, `-fm` have a special meaning:

- `-f` Add function name as section name suffix (for functions only)
- `-m` Add module base name as section name suffix
- `-fm` or `-mf` Add module base name for data and add function name for functions

If you do not specify a section type, all sections will receive the specified suffix.

Example
To change all sections with type `.data` into `.data.NEW`, enter:

```
carm --rename-sections=.data=NEW test.c
```

To add the name of the current module as suffix to all data sections, resulting in `.data.test`, enter:

```
carm --rename-sections=.data=-m test.c
```
The following examples show the difference when using `--rename-sections={name}` or `--rename-sections={cname}`.

Generated labels:

```
.section .text.tm..cocofun_1 ;; {name}
.section .text.tm.$cocofun ;; {cname}
.section .rodata.hs..1.str ;; {name}
.section .rodata.hs.$str ;; {cname}
.section .rodata.hs..2.ini ;; {name}
.section .rodata.hs.$ini ;; {cname}
```

Statics within a function:

```
.section .data.hs._999001_my_local ;; {name}
.section .data.hs.my_local ;; {cname}
.section .data.hs._999002_my_local ;; {name}
.section .data.hs.my_local ;; {cname}
```

Several modules with static functions of the same name:

```
.section .text.hs1.f1 ;; {name}
.section .text.hs1.f1 ;; {cname}
.section .text.hs2.f1.1 ;; {name}
.section .text.hs2.f1 ;; {cname}
```

Related information

Section 1.13.1, Rename Sections

Assembler directive `.SECTION`
**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
   - Generate code for stack overflow checks (allowed for USR and SYS mode only)
   - Generate code for division by zero checks

**Command line syntax**

```
--runtime [=flag,...]
-r [flags]
```

You can set the following flags:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+/-bounds</td>
<td>bounds checking</td>
</tr>
<tr>
<td>+/-case</td>
<td>report unhandled case in a switch</td>
</tr>
<tr>
<td>+/-malloc</td>
<td>malloc consistency checks</td>
</tr>
<tr>
<td>+/-stack</td>
<td>check for stack overflow</td>
</tr>
<tr>
<td>+/-zero</td>
<td>check for divide by zero checks</td>
</tr>
</tbody>
</table>

Default (without flags): `-rbcmSZ`

**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 checking**

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`).
Report unhandled case in a switch

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 consistency checks

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.

Stack overflow check

The compiler generates extra code within the function prolog that will check the available stack size before allocating. This is only useful when the processor runs in USR or SYS mode.

Division by zero check

The compiler generates a call to specific run-time functions for additional division by zero checks. If this situation occurs, an abort signal is issued. Without this check, a division by zero could lead to unpredictable results.

Related information

Pragma runtime
C compiler option: --schar

Menu entry
1. Select C/C++ Compiler » Language.
2. Enable the option Treat 'char' variables as signed.

Command line syntax
--schar

Description
By default char is the same as specifying unsigned char as required by the ARM ABI. With this option char is the same as signed char.

Note that this option can cause conflicts when you use it in combination with MIL linking. With MIL linking some extra strict type checking is done that can cause building to fail in a way that is unforeseen and difficult to understand. For example, when you use option --mil in combination with option --schar and you link the MIL library, you might get the following error:

carm E289: ["..\..\..\strlen.c" 14/1] "strlen" redeclared with a different type
carm I802: ["installation-dir\include\string.h" 44/17]
   previous declaration of "strlen"
1 errors, 0 warnings

This is caused by the fact that the MIL library is built without --schar. You can workaround this problem by rebuilding the MIL libraries.

Related information
Section 1.1, Data Types
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[=bug,...]

Description
With this option you specify for which hardware problems the compiler should generate workarounds. Please refer to Chapter 19, CPU Problem Bypasses and Checks for the numbers and descriptions. Silicon bug numbers are specified as a comma separated list. When you use this option without arguments, all silicon bug workarounds are enabled.

Example
To enable workarounds for problem 602117, enter:
carm --silicon-bug=602117 test.c

Related information
Chapter 19, 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: --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: --thumb

Menu entry
2. Enable the option Use Thumb instruction set.

Command line syntax
--thumb

Description
With this option you tell the compiler to generate Thumb or Thumb-2 instructions, depending on the architecture.

When you specify the ARMv6-M, ARMv7-M or ARMv7E-M architecture (option --cpu), the compiler automatically selects the Thumb-2 instruction set.

Related information
C compiler option --cpu (Select architecture)
C compiler option --interwork (Generate interworking code)
**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=4`

**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 optimizes for code size (`--tradeoff=4`).

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:

```
carm --tradeoff=2 test.c
```

The compiler uses the default *Optimize more* optimization level and balances speed and size while optimizing.

**Related information**

C compiler option `--optimize` (Specify optimization level)

Section 4.5.3, *Optimize for Code Size or Execution Speed*
C compiler option: --unaligned-access

Menu entry

1. Select C/C++ Compiler » Miscellaneous.
2. Add the option --unaligned-access to the Additional options field.

Command line syntax

--unaligned-access

Description

With this option you tell the compiler to generate more efficient instructions to access unaligned 16-bit or larger data. Halfword or word load and store instructions are used instead of byte instructions.

This option is only useful for cores that have support for unaligned access.

Related information

-
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

-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:

__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__:

carm --undefine=__TASKING__ test.c

Related information

C compiler option --define (Define preprocessor macro)

Section 1.10, Predefined Preprocessor Macros
C compiler option: --verbose (-v)

Menu entry
-

Command line syntax

--verbose
-v

Description
With this option you put the C compiler in verbose mode. The C compiler performs its tasks while it prints the steps it performs to stdout.

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.

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[-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 not suppressed by option --no-warnings (or #pragma warning) 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 limit this option to specific warnings by specifying a comma-separated list of warning numbers or ranges. In this case, this option takes precedence over option --no-warnings (and #pragma warning).

Related information
C compiler option --no-warnings ( Suppress some or all warnings)
Pragma warning
11.3. 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 for

   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:

   cparm -Ecp test.cc
   cparm --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:

```
cparm --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: --auto-type**

**Menu entry**

- 

**Command line syntax**

--auto-type

--no-auto-type

Default: auto is a storage class specifier.

**Description**

Enable or disable auto as a type specifier where the actual type is deduced from an initializer that follows. This feature is implicitly enabled in C++11 mode and later modes.

**Related information**

C++ compiler option --no-auto-storage (Do not allow auto as storage specifier)

C++ compiler option --c++=11 (C++ standard)
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++

Menu entry

1. Select C/C++ Compiler » Language.
2. From the Comply to C++ standard list, select the ISO C++ standard.

Command line syntax

--c++=year

You can specify one of the following years: 03, 11 or 14.

Default: --c++=14

Description


For C++03 the STLport library is used, whereas C++11 and C++14 use libc++.

Related information

C++ compiler option --exceptions (Enable exceptions)
C++ compiler option: --c++11-sfinae

Menu entry
-

Command line syntax

--c++11-sfinae
--no-c++11-sfinae

Description

Enable or disable template deduction in the style dictated by the C++11 standard (SFINAE rules of document N2634), i.e., where general expressions are allowed in deduction contexts and they undergo the full usual semantic checking. This type of deduction is necessary to get the full power of the decltype feature in return types. “SFINAE” refers to the initials of the phrase “Substitution Failure Is Not An Error”, which is the guiding principle for template deduction, and by extension a name for the process of deduction.

This feature is implicitly enabled in C++11 mode and later modes, and is implicitly disabled in GNU modes.

Related information

C++ compiler option --c++=11 (C++ standard)
C++ compiler option: --c++11-sfinae-ignore-access

Menu entry

Command line syntax

--c++11-sfinae-ignore-access
--no-c++11-sfinae-ignore-access

Default: --no-c++11-sfinae-ignore-access

Description

When C++11 SFNAE is enabled (option --c++11-sfinae), option --c++11-sfinae-ignore-access indicates that access errors are not counted as errors that cause deduction failures. In document N2634 access errors are ignored, but the standards committee changed its mind about that later. So, the default (option --no-c++11-sfinae-ignore-access) is that access errors cause a deduction failure.

Related information

C++ compiler option --c++11-sfinae (C++11 SFINAE rules)
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: --check-concatenations

Menu entry

Command line syntax

--check-concatenations

Description

With this option the preprocessor will generate a diagnostic when a macro concatenation (such as a ## b) does not result in a valid token.

Related information
C++ compiler option: --compound-literals

Menu entry

- 

Command line syntax

--compound-literals

Description

Allow compound literals in expressions. A compound literal looks like a cast containing an initializer. Its value is an object of the type specified in the cast, containing the elements specified in the initializer; it is an lvalue.

For example:

```c
static int i[] = (int []) {1, 2, 3};
```

Related information

-
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:

cparm --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, make a selection by Architecture, Core or one of the manufacturers.

Command line syntax
--cpu=architecture
-Carchitecture

You can specify the following architectures:

- ARMv6M Compile for ARMv6-M architecture
- ARMv7M Compile for ARMv7-M architecture
- ARMv7EM Compile for ARMv7E-M architecture

Description
With this option you specify the ARM architecture for which you create your application. The ARM target supports more than one architecture and therefore you need to specify for which architecture the compiler should compile. The architecture determines which instructions are valid and which are not.

The effect of this option is that the C++ compiler uses the appropriate instruction set. You choose one of the following architectures: ARMv6-M, ARMv7-M or ARMv7E-M.

The macro __CPU__ is set to the name of the architecture.

Example
To compile the file test.cc for the ARMv7E-M architecture, enter the following on the command line:

cparm --cpu=ARMv7EM test.cc

Related information
C compiler option --cpu (Select architecture)
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

```bash
--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:

```bash
cparm --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: --default-nocommon-tentative-definitions

Menu entry
-

Command line syntax
--default-nocommon-tentative-definitions

Description
In GNU C++ mode, this option prevents tentative definitions to be placed in common storage.

Related information
-
C++ compiler option: --defer-parse-function-templates

Menu entry
-

Command line syntax

--defer-parse-function-templates
--no-defer-parse-function-templates

Default: --defer-parse-function-templates in GNU C++ mode.

Description

Enable or disable deferral of prototype instantiations until the first actual instantiation of a function. This is used to permit the compilation of programs that contain definitions of unusable function templates.

It is enabled by default in GNU C++ mode.

Related information
-

368
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.

Make sure you do not use a reserved keyword as a macro name, as this can lead to unexpected results.

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 */
```
You can now use a macro definition to set the DEMO flag:

```bash
cparm --define=DEMO test.cc
cparm --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.

```bash
cparm --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: --delegating-constructors

Menu entry
-

Command line syntax

--delegating-constructors

Description
Enable delegating constructors (a C++11 feature).

Related information
-
C++ compiler option: --dep-file

Menu entry

Command line syntax

--dep-file[=file]

Description

With this option you tell the C++ 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

cparm --dep-file=test.dep test.cc

The C++ compiler compiles the file test.cc, which results in the output file test.ic, and generates dependency lines in the file test.dep.

Related information

C++ compiler option --preprocess=+make (Generate dependencies for make)
C++ compiler option: --deprecated-string-conv

Menu entry
-

Command line syntax
--deprecated-string-conv

Description
With this option the deprecated conversion from a string literal to char * is allowed. This is only meaningful when string literals are const.

Related information
-
C++ compiler option: --diag

Menu entry

Command line syntax

--diag

Description

With this option the C++ compiler displays a list of all diagnostic messages on stdout (usually your screen). The C++ compiler does not compile any files.

Related information

Section 5.3, C++ Compiler Error Messages
C++ compiler option: --digit-separators

Menu entry
-

Command line syntax
--digit-separators

Description
Enable support for digit separators (a C++14 feature). When this feature is enabled, an apostrophe can appear between the digits of a numeric literal and is ignored, e.g., 123’456’789.

This feature is implicitly enabled in C++14 mode and later modes.

Related information
C++ compiler option --c++=14 (C++ standard)
C++ compiler option: --dollar

Menu entry

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 Comply to embedded C++ subset.

Command line syntax
--embedded-c++

Description
The "Embedded C++" subset does not support templates, exceptions, namespaces, new-style casts, RTTI, multiple inheritance, virtual base classes, and the mutable keyword. Select this option when you want the C++ compiler to give an error when you use any of them in your C++ source.

Related information
-
C++ compiler option: --endianness

Menu entry

1. Select Global Options.
2. Specify the Endianness: Little-endian mode or Big-endian mode.

Command line syntax

```--endianness=endianness```

-B
```--big-endian```

You can specify the following endianness:

<table>
<thead>
<tr>
<th>big</th>
<th>b</th>
<th>Big endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>little</td>
<td>l</td>
<td>Little endian (default)</td>
</tr>
</tbody>
</table>

Description

By default, the C++ compiler generates code for a little-endian target (least significant byte of a word at lowest byte address). With `--endianness=big` the C++ compiler generates code for a big-endian target (most significant byte of a word at lowest byte address). -B is an alias for option `--endianness=big`.

The macro `__BIG_ENDIAN__` is defined when this option is specified, otherwise the macro `__LITTLE_ENDIAN__` is defined.
C++ compiler option: --error-file

Menu entry
-

Command line syntax
--error-file[=file]

Description
With this option the C++ compiler redirects diagnostic messages to a file. If you do not specify a filename, the error file will be named after the output file with extension .ecp.

Example
To write diagnostic messages to errors.ecp instead of stderr, enter:
cparm --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:

cparm --error-limit=10 test.cc

Related information

-
C++ compiler option: --exc-spec-in-func-type

Menu entry
-

Command line syntax

--exc-spec-in-func-type
--no-exc-spec-in-func-type

Default: --no-exc-spec-in-func-type

Description

Enable or disable exceptions being part of a function type (this is a C++17 feature).

Related information
-
C++ compiler option: --exceptions (-x)

Menu entry
1. Select C/C++ Compiler » Language.
2. Enable or disable the option Support for C++ exception handling.

Command line syntax
--exceptions
-x

--no-exceptions
Default: --no-exceptions

Default in C++11 mode and later modes: --exceptions

Description
With this option you enable or disable support for exception handling in the C++ compiler.

The macro __EXCEPTIONS is defined when exception handling support is enabled.

This feature is implicitly enabled in C++11 mode and later modes.

Related information
C++ compiler option --c++=11 (C++ standard)
C++ compiler option: --extended-variadic-macros

Menu entry
-

Command line syntax

--extended-variadic-macros
--no-extended-variadic-macros

Description

Enable or disable support for 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 (C++).

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 Allow GNU C++ extensions.

Command line syntax

--g++

Description

Enable GNU C++ compiler language extensions.

This option sets the predefined macros __GNUC__, __GNUC_MINOR__, __GNUC_PATCHLEVEL__, __GNUC_VERSION__, __GNUC_STDC_INLINE__, __GNUG__, __CHAR16_TYPE__ and __CHAR32_TYPE__.

Related information

Section 2.2.2, GNU C++ Mode

Section 2.8, Predefined Macros
C++ compiler option: --generate-move-operations

Menu entry
-

Command line syntax
--generate-move-operations

Description
With this option the C++ compiler generates move constructors and move assignment operators according to standard C++11 rules.

Related information
-
C++ compiler option: --gnu-version

Menu entry

Command line syntax

--gnu-version=version

Default: 40800 (version 4.8.0)

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.

This option implicitly enables GNU C++ mode and sets the predefined macros __GNUC__, __GNUC_MINOR__, __GNUC_PATCHLEVEL__, __GNUC_VERSION__, __GNUC_STDC_INLINE__, __GNUG__, __CHAR16_TYPE__ and __CHAR32_TYPE__.

Example

To specify version 4.7.1 of the GNU C++ compiler, enter:

cparm --g++ --gnu-version=40701 test.cc

Related information

Section 2.2.2, GNU C++ Mode
Section 2.8, Predefined Macros
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:

cparm -?
cparm --help
cparm

cparm

The following invocation displays an extended list of the available options:

cparm --help=options

Related information

-
C++ compiler option: --ignore-std

Menu entry
-

Command line syntax
--ignore-std

Description
Enable a GNU C++ compatibility feature that makes the std namespace a synonym for the global namespace.

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

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

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 2.4, Template Instantiation
C++ compiler option: --implicit-noexcept

Menu entry

Command line syntax

--implicit-noexcept

Description

With this option destructors and deallocation functions (i.e., operator delete and operator delete[]) have implicit noexcept specifications if no explicit exception specification is provided.

Related information
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:

cparm --incl-suffixes=h:stdh test.cc

Related information

C++ compiler option --include-file (Include file at the start of a compilation)

Section 5.2, 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:

```
cparm --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 5.2, *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 **Pre-include files** box.

3. Type the full path and file name or select a file.

4. (Optional) Enable the option **Include CMSIS device register definition header 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**

```
cparm --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 5.2, *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 5.2, 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

cparm --init-priority=101 test.cc

Related information

-
C++ compiler option: --inline-statement-limit

Menu entry

Command line syntax

--inline-statement-limit=number

Default: 100

Description

With this option you can set the maximum number of statements that a routine can have and still be inlinable. The setting is somewhat arbitrary, but a limit is needed to prevent iterative inlining from exhausting memory. A value of zero allows an inlined routine to be any number of statements.

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

-t mode

You can specify the following modes:

- **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:

- **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

Section 2.4, Template Instantiation
**C++ compiler option: --io-streams**

**Menu entry**
1. Select **C/C++ Compiler » Language**.
2. Enable the option **Support for C++ I/O streams**.

**Command line syntax**

`--io-streams`

**Description**

As I/O streams require substantial resources they are disabled by default for C++03. Use this option to enable I/O streams support in the C++03 library. So, when you use standard I/O, e.g. `cout`, or STL templates in your C++03 source, you need to specify this option.

For C++11 and later, I/O streams features are always available.

This option also enables exception handling.

**Related information**

C++ compiler option **--exceptions** (Enable exceptions)
C++ compiler option: --lambdas

Menu entry

Command line syntax

--lambdas

--no-lambdas

Default: --no-lambdas

Default in C++11 mode and later modes: --lambdas

Description

Enable or disable support for C++11 lambdas.

This feature is implicitly enabled in C++11 mode and later modes.

Related information

C++ compiler option --c++=11 (C++ standard)
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=\textit{file}

-\textit{L}file

Default: -1

Description

Generate raw listing information in the $\textit{file}$. 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:

\begin{itemize}
  \item \textbf{N} A normal line of source; the rest of the line is the text of the line.
  \item \textbf{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.
  \item \textbf{S} A line of source skipped by an \texttt{#if} or the like; the rest of the line is text. Note that the \texttt{#else}, \texttt{#elif}, or \texttt{#endif} that ends a skip is marked with an N.
  \item \textbf{L} An indication of a change in source position. The line has a format similar to the \texttt{#} line-identifying directive output by the C preprocessor, that is to say

\begin{verbatim}
  L line_number "file-name" [key]
\end{verbatim}

where \texttt{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 \texttt{#}line directives (key is omitted). L lines indicate the source position of the following source line in the raw listing file.
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 lowercase 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:

```
cparm --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: --lossy-conversion-warning

Menu entry

Command line syntax

--lossy-conversion-warning

Description

With this option a warning is issued whenever a conversion occurs from one arithmetic type to a smaller one or from a floating type to an integral type.

Related information
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 .obj.

Example

cparm --preprocess=+make --make-target=mytarget.obj test.cc

The compiler generates dependency lines with the default target name mytarget.obj instead of test.obj.

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-cost-constexpr-call

Menu entry

Command line syntax

```
--max-cost-constexpr-call=value
```

Default: 50000

Description

With this option you can set the maximum "cost" for folding a top-level constexpr function or constructor call. In C++11 mode, the "cost" is the number of calls expanded from a single top-level call. In C++14 mode or later modes, the "cost" is a combination of the number of calls interpreted and the number of loop iterations performed during the interpretation of a top-level call.

Related information

-
C++ compiler option: --max-depth-constexpr-call

Menu entry
-

Command line syntax

--max-depth-constexpr-call=value

Default: 1000

Description

With this option you can set the maximum nesting depth of constexpr function and constructor calls.

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++ compliance tests)
C++ compiler option --using-std (Implicit use of the std namespace)
Section 2.3, 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-storage

Menu entry

Command line syntax

--no-auto-storage

Default: auto is a storage class specifier.

Description

Disable the traditional meaning of auto keyword as a storage class specifier.

Related information

C++ compiler option --auto-type (auto can be used as type specifier)
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
-

Tool Options
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 » Floating-Point.
2. Select Custom from the Floating-point model list.
3. 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
cparm --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-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-anonymous-unions

Menu entry

Command line syntax

--nonstd-anonymous-unions

Description

Enable extensions to the C++ anonymous union feature (including of anonymous "structs" or "classes").

Related information

-
C++ compiler option: --nonstd-default-arg-deduction

Menu entry
-

Command line syntax
--nonstd-default-arg-deduction

Description
Controls whether default arguments are retained as part of deduced function types. The C++ standard requires that default arguments not be part of deduced function types. This option changes that behavior.

Related information
-
C++ compiler option: --nonstd-instantiation-lookup

Menu entry

Command line syntax

--nonstd-instantiation-lookup

Description

Controls whether the lookup of names during template instantiation should, instead of the normal lookup rules, use rules that were part of the C++98 working paper for some time during the development of the standard. In this mode, names are looked up in both the namespace of the template definition and in the namespace in which a template entity was first referenced in a way that would require an instantiation.

Related information
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**

cparm --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-stdarg-builtin

Menu entry

Command line syntax

--no-stdarg-builtin

Description

Disable special treatment of the stdarg.h header. When enabled, the stdarg.h header is treated as a built-in, and references to its macros ("va_start" et al) are passed through as such in generated C code.

Related information
C++ compiler option: --no-stdinc / --no-stdcppinc

Menu entry

1. Select C/C++ Compiler » Miscellaneous.
2. Add the option --no-stdinc or --no-stdcppinc to the Additional options field.

Command line syntax

--no-stdinc

--no-stdcppinc

Description

With option --no-stdinc you tell the C++ compiler not to look in the default include, include.cpp, include.stl and include.cxx directories relative to the installation directory, when searching for standard include files.

With option --no-stdcppinc you tell the C++ compiler not to look in the default include.stl and include.cxx directories relative to the installation directory, when searching for standard C++03 and C++11/C++14 include files respectively.

This way the C++ compiler only searches in the include file search paths you specified.

Related information

Section 5.2, How the C++ Compiler Searches Include Files

C++ compiler option --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.
   *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 C++ compiler warnings box.
3. Enter the numbers, separated by commas or as a range, of the warnings you want to suppress (for example 611, 2709). 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. Error messages are still issued.

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 one or more numbers, only the specified warnings are suppressed. You can specify the option --no-warnings=number multiple times.
- Leave out any leading zeros.

Example

To suppress warning W0611, enter:

cparm --no-warnings=611 test.cc

Related information

C++ compiler option --warnings-as-errors (Treat warnings as errors)
C++ compiler option: --no-wchar_t-keyword

Menu entry
1. Select C/C++ Compiler » Language.
2. Disable the option Allow the 'wchar_t' keyword (C++)

Command line syntax

--no-wchar_t-keyword

Default: wchar_t is recognized as a keyword.

Description

With this option you disable 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: --nullptr

Menu entry

Command line syntax

--nullptr
--no-nullptr

Default: --no-nullptr
Default in C++11 mode and later modes: --nullptr

Description

Enable or disable support for the C++11 nullptr keyword.

This feature is implicitly enabled in C++11 mode and later modes.

Related information

C++ compiler option --c++=11 (C++ standard)
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:
cparm --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 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:

```bash
--embedded-c++
--define=DEMO=1
```
```
test.cc
```

Specify the option file to the C++ compiler:

```
cparm --option-file=myoptions
cparm --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:

cparm --output=output.ic test.cc

To use the file my.pre as the preprocessing output file, enter:

cparm --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 `c:\usr\include\pch` to automatically create precompiled header files, enter:

```
cparm --pch-dir=c:\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**

```
cparm --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:

cparm --pending-instantiations=32 test.cc

Related information

Section 2.4, Template Instantiation
C++ compiler option: --preprocess (-E)

Menu entry
1. Select C/C++ Compiler » Preprocessing.
2. Enable the option Store preprocessor output in <file>.pre.
3. (Optional) Enable the option Keep comments in preprocessor output.
4. (Optional) Enable the option Keep #line info in preprocessor output.

Command line syntax
--preprocess [=flags]
-E[flags]

You can set the following flags:

+/-comments  c/C  keep comments
+/-includes   i/l  generate a list of included source files
+/-make       m/M  generate dependencies for make
+/-noline     p/P  strip #line source position information

Default: --ECIMP

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=+includes the C++ compiler will generate a list of all included source files. The preprocessor output is discarded.

With --preprocess=+make the C++ 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 .obj. With the option --make-target you can specify a target name which overrules the default target name.

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**

cparm --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 **--dep-file** (Generate dependencies in a file)

C++ compiler option **--make-target** (Specify target name for -Em output)
C++ compiler option: --preserve-lvalues-with-same-type-casts

Menu entry

- 

Command line syntax

--preserve-lvalues-with-same-type-casts

Description

With this option a cast of an lvalue to its own type results in an lvalue (rather than an rvalue, as required by the C++ standard).

Related information

-
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 5.3, 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 C++ 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: --rvalue-ctor-is-not-copy-ctor

Menu entry

Command line syntax

--rvalue-ctor-is-not-copy-ctor

Description

Do not treat an rvalue (or "move") constructor as a copy constructor. In the default situation, where rvalue constructors are treated as copy constructors, a user-declared rvalue constructor will inhibit the implicit generation of a traditional copy constructor.

Related information
C++ compiler option: --rvalue-refs

Menu entry

Command line syntax

--rvalue-refs
--no-rvalue-refs

Default: --no-rvalue-refs

Default in C++11 mode and later modes: --rvalue-refs

Description

Enable or disable support for rvalue references.

This feature is implicitly enabled in C++11 mode and later modes.

Related information

C++ compiler option --c++=11 (C++ standard)
C++ compiler option: --schar (-s)

Menu entry
1. Select C/C++ Compiler » Language.
2. Enable the option Treat 'char' variables as signed.

Command line syntax

--schar

-s

Description
By default char is the same as specifying unsigned char. 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++. All ANSI/ISO violations are issued as errors.

Example
To enable strict ANSI/ISO mode, with error diagnostic messages, enter:

cparm --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:

cparm --strict-warnings test.cc

Related information

C++ compiler option --strict (Strict ANSI/ISO mode with errors)
**C++ compiler option: --suppress-vtbl**

**Menu entry**

1. Select C/C++ Compiler » Miscellaneous.
2. Enable the option *Suppress definition of virtual function tables (C++)*.  

**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. The `--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 c:\proj\include to the system include file search path, enter:

cparm --sys-include=c:\proj\include test.cc

Related information

C++ compiler option --include-directory (Add directory to include file search path)

Section 5.2, How the C++ Compiler Searches Include Files
C++ compiler option: --template-typedefs-in-diagnostics

Menu entry

Command line syntax

--template-typedefs-in-diagnostics

Default: typedefs declared in template classes are not replaced with their underlying type.

Description

Enable the replacement of typedefs declared in template classes with their underlying type. Diagnostic messages are often more useful when such typedefs are replaced.

Related information

-
C++ compiler option: --thumb

Menu entry
2. Enable the option Use Thumb instruction set.

Command line syntax
--thumb

Description
Generate code in Thumb mode. The Thumb instruction set is a subset of the ARM instruction set which is encoded using 16-bit instructions instead of 32-bit instructions.

The macro __THUMB__ is defined when the Thumb mode is enabled.

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

cparm --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

cparm --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. Disable the option Treat 'char' variables as signed.

Command line syntax

--uchar
-u

Description

With this option char is the same as unsigned char. This is the default.

Related information

C++ compiler option --schar (Plain char is signed)

Section 1.1, Data Types
C++ compiler option: --uliterals

Menu entry
-

Command line syntax
--uliterals

Default: U-literals are not recognized.

Description
Enable recognition of U-literals (string literals of the forms U"..." and u"...", and character literals of the forms U'...' and u'...'). In C++ mode, this also enables the C++11 char16_t and char32_t keywords.

Related information
-
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
-Umacro_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:

```
cparm --undefine=__cplusplus test.cc
```

Related information

C++ compiler option --define (Define preprocessor macro)

Section 2.8, Predefined Macros
C++ compiler option: --unrestricted-unions

Menu entry
-

Command line syntax
--unrestricted-unions

Description
Accept C++11-style "unrestricted unions". For example, non-static data members of unions can have nontrivial constructors and destructors if you use this option.

Related information
-
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:

cparm --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: --user-defined-literals

Menu entry
-

Command line syntax
--user-defined-literals

Description
Allow C++11-style "user-defined literals", i.e., literals with a suffix designating an operator function that is to be called to produce the value of the literal, e.g., 12.34_km, which calls operator ""_km(12.34).

Related information
-
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 2.3, Namespace Support
C++ compiler option: --utf8-char-literals

Menu entry
-

Command line syntax
--utf8-char-literals

Description
Enable C++17-style UTF-8 character literals (e.g., u8’a’).

Related information
-
C++ compiler option: --variadic-macros

Menu entry

Command line syntax

--variadic-macros
--no-variadic-macros

Default: --no-variadic-macros
Default in C++11 mode and later modes: --variadic-macros

Description

Allow or do not allow macros with a variable number of arguments.
This feature is implicitly enabled in C++11 mode and later modes.

Related information

C++ compiler option --extended-variadic-macros (Allow extended variadic macros)
C++ compiler option --c++=11 (C++ standard)
C++ compiler option: --variadic-templates

Menu entry
-

Command line syntax
--variadic-templates

Description
Allow "variadic templates", a C++11 feature that allows declaration of templates with a variable number of arguments.

Related information
-
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.
C++ compiler option: --vla

Menu entry

- 

Command line syntax

--vla

Description

Enable support for variable length arrays. This allows the declaration and use of arrays of automatic storage duration with dimensions that are fixed at run-time.
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 C++ compiler encounters an error, it stops compiling. When you use this option without arguments, 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.

You can limit this option to specific warnings by specifying a comma-separated list of warning numbers.

Related information

C++ compiler option --no-warnings (Suppress all warnings)
**C++ compiler option: --xref-file (-X)**

**Menu entry**

- 

**Command line syntax**

```bash
--xref-file=file
-X file
```

**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**

-
11.4. Assembler Options

This section lists all assembler options. All options are the same for all three assemblers, asarm (mixed ARM/Thumb), asarma (ARM only) and asarmt (Thumb only). In the examples we only use asarm.

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 for

   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:

```bash
asarm -l -LeM test.src
asarm --list-file --list-format=+symbol,-macro 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 uppercase and lowercase characters. By default the assembler considers uppercase and lowercase 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.

asarm --case-insensitive test.src

Related information

-
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: --cpu (-C)

Menu entry
1. Expand C/C++ Build and select Processor.
2. From the Processor selection list, make a selection by Architecture, Core one of the manufacturers.

Command line syntax
--cpu=architecture
-C architecture

You can specify the following architectures:

- ARMv6M Assemble for ARMv6-M architecture
- ARMv7M Assemble for ARMv7-M architecture
- ARMv7EM Assemble for ARMv7E-M architecture

Description
With this option you specify the ARM architecture for which you create your application. The architecture determines which instructions are valid and which are not. The default architecture is ARMv7M and the complete list of supported architectures is: ARMv6-M, ARMv7-M or ARMv7E-M.

Assembly code can check the value of the option by means of the built-in function @CPU().

To avoid conflicts, make sure you specify the same architecture as you did for the compiler (Eclipse and the control program do this automatically).

Related information
Assembly function @CPU()
Control program option --cpu (Select architecture)
C compiler option --cpu (Select architecture)
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:

+/asm   a/A Assembly source line information
+/hll    h/H Pass high level language debug information (HLL)
+/local  l/L Assembler local symbols debug information
+/smart  s/S Smart debug information

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 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.

Make sure you do not use a reserved keyword as a macro name, as this can lead to unexpected results.
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:

```
asarm --define=DEMO test.src
asarm --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: --dep-file**

**Menu entry**

- 

**Command line syntax**

`--dep-file[=file]`

**Description**

With this option you tell the assembler to generate dependency lines that can be used in a Makefile. 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`. When you specify a filename, all dependencies will be combined in the specified file.

**Example**

`asarm --dep-file=test.dep test.src`

The assembler assembles the file `test.src`, which results in the output file `test.obj`, and generates dependency lines in the file `test.dep`.

**Related information**

Assembler option `--make-target` (Specify target name for `--dep-file` output)
Assembler option: --diag

Menu entry

1. From the Window menu, select Show View » Other » TASKING » 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:

```
asarm --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 `aserrors.html`, use redirection and enter:

```
 asarm --diag=html:all > aserrors.html
```

**Related information**

Section 6.6, *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 mapping symbols ($a, $t, $d)
   - Emit local non-EQU symbols

Command line syntax

--emit-locals [=flag,...]

You can set the following flags:

```
 +/-equs    e/E      emit local EQU symbols
 +/-mappings m/M     emit mapping symbols ($a, $t, $d)
 +/-symbols s/S     emit local non-EQU symbols
```

Default: --emit-locals=+mappings, +symbols

Description

With the option --emit-locals=+equs the assembler also emits local EQU symbols to the object file. Normally, only global symbols, mapping symbols and non-EQU local symbols are emitted. Having local symbols in the object file can be useful for debugging.

Mapping symbols are local symbols inside code sections which mark the start of a range of ARM instructions ($a), a range of Thumb instructions ($t), or a literal pool a.k.a. data pocket ($d). Also, data sections start with a $d symbol. The mapping symbol names are made unique with a’.’ character suffix followed by a unique integer, for example: $a.1, $t.2 and $d.3. This option only takes effect if local labels are emitted as well (default).

Related information

Assembler directive .EQU
Assembler option: --endianness

Menu entry
1. Select Global Options.
2. Specify the Endianness:Little-endian mode or Big-endian mode.

Command line syntax

```bash
--endianness=endianness
-B
--big-endian
```

You can specify the following endianness:

<table>
<thead>
<tr>
<th>big</th>
<th>b</th>
<th>Big endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>little</td>
<td>l</td>
<td>Little endian (default)</td>
</tr>
</tbody>
</table>

Description

By default, the assembler generates object files with instructions and data in little-endian format (least significant byte of a word at lowest byte address). With `--endianness=big` the assembler generates object files in big-endian format (most significant byte of a word at lowest byte address). -B is an alias for option `--endianness=big`.

The endianness is reflected in the list file.

Assembly code can check the setting of this option by means of the built-in assembly function `@BIGENDIAN()`.

Related information

Assembly function `@BIGENDIAN()`
Assembler option: --error-file

Menu entry

- 

Command line syntax

--error-file[=file]

Description

With this option the assembler redirects diagnostic messages to a file. If you do not specify a filename, the error file will be named after the output file with extension .ers.

Example

To write diagnostic messages to errors.ers instead of stderr, enter:

asarm --error-file=errors.ers test.src

Related information

Section 6.6, 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 6.6, *Assembler Error Messages*
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:

    asarm -?
    asarm --help
    asarm

To see a detailed description of the available options, enter:

    asarm --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 ASARMINC 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:

asarm --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 Pre-include files 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 **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 (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**

```
asarm --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: --inversions

Menu entry
1. Select Assembler » Miscellaneous.
2. Enable the option Allow instruction inversions.

Command line syntax
--inversions

Description
With this option you tell the assembler to try to invert some data processing instructions with an immediate operand. Inversions are available for MOV/MVN, CMP/CMN, AND/BIC, ADC/SBC, and ADD/SUB.

Example
With this option enabled, you can write

    add r1,r2,#-4

and the assembler will generate

    sub r1,r2,#4

and instead of

    mov r1,0xFFFFFFFF

the assembler will generate

    mvn r1,0

Related information
-
Assembler option: --kanji

Menu entry
1. Select Assembler » Miscellaneous.
2. Enable the option Allow Shift JIS Kanji in strings.

Command line syntax
--kanji

Description
With this option you tell the assembler to support Shift JIS encoded Kanji multi-byte characters in strings. Without this option, encodings with 0x5c as the second byte conflict with the use of the backslash as an escape character. Shift JIS in comments is supported regardless of this option.

Note that Shift JIS also includes Katakana and Hiragana.

Related information
C compiler option --language=+kanji (Allow Shift JIS Kanji in strings)
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 (.obj) 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]
-l[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 output 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 (.SECTION)
+/-symbol     e/E  List symbol definition directives
+/-generic-expansion g/G  List expansion of generic instructions
+/-generic    i/I  List generic instructions
+/-line       l/L  List C preprocessor #line directives
+/-macro      m/M  List macro definitions
+/-empty-line n/N  List empty source lines and comment 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')
+/-hll        s/S  List HLL symbolic debug informations
+/-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=DEGILMNPQRSVWXYZ
--list-format=1       -L1  All options enabled
                      Alias for --list-format=degilmnpqrsvwxyz

Default: --list-format=dEGilMnPqrsVwXyZ
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 (-I)`.

Related information

Assembler option `--list-file` (Generate list file)

Assembler option `--section-info=+list` (Display section information in list file)
Assembler 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 option `--dep-file`. The default target name is the basename of the input file, with extension `.obj`.

Example

```
asarm --dep-file --make-target=../mytarget.obj test.src
```

The assembler generates dependency lines with the default target name `../mytarget.obj` instead of `test.obj`.

Related information

Assembler option `--dep-file` (Generate dependencies in a file)
**Assembler option: --no-notes**

**Menu entry**

1. Select **Assembler » Miscellaneous**.
2. Add the option **--no-notes** to the **Additional options** field.

**Command line syntax**

`--no-notes`

**Description**

By default, the assembler generates a note section in the object file. The note section contains compiler version and invocation information, if supplied in the input file, and version and invocation information of the assembler. With this option you can suppress the generation of a note section in the output object file.

**Related information**

Section 9.5.2, *HLL Dump Output Format*
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:

```bash
asarm test.src --no-warnings=201,202
```

Related information

Assembler option --warnings-as-errors (Treat warnings as errors)
Assembler option: --old-syntax

Menu entry

1. Select Assembler » Miscellaneous.
2. Disable the option UAL syntax mode.

Command line syntax

--old-syntax

Description

In UAL syntax mode the assembler will not accept instructions which use the pre-UAL syntax and will select encodings based on the UAL syntax in case both syntaxes are the same.

With this option you can change this default behavior. The assembler will run in pre-UAL mode. The built-in function @PRE_UAL() will return true, so you can use:

```
.IF @PRE_UAL()
  ; <old code>
.ELSE
  ; <new code>
.ENDIF
```

Related information

Assembly function @PRE_UAL()
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 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:

```
--debug-info=+asm,-local
test.src
```

Specify the option file to the assembler:

```
asarm --option-file=myoptions
```

This is equivalent to the following command line:

```
asarm --debug-info=+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 .obj.

Example
To create the file relobj.obj instead of asm.obj, enter:

asarm --output=relobj.obj 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 directive .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 directive .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
1. Select Assembler » Preprocessing.
2. Enable or disable the option Use TASKING preprocessor.

Command line syntax

--preprocessor-type=type
-m type

You can set the following preprocessor types:

<table>
<thead>
<tr>
<th>Type</th>
<th>Symbol</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: --relaxed

Menu entry
1. Select Assembler » Miscellaneous.
2. Enable the option Allow 2-operand form for 3-operand instructions.

Command line syntax
--relaxed

Description
With this option you tell the assembler that a relaxed 2-operand syntax is allowed on 3-operand instructions. If the first two register operands are equal, you can replace the two registers by one.

Example
With this option enabled, instead of

```
add r1,r1,#4
```

you can write

```
add r1,#4
```

and instead of

```
add r1,r1,r2
```

you can write

```
add r1,r2
```

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:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+/-console</td>
<td>Display section summary on console</td>
</tr>
<tr>
<td>+/-list</td>
<td>List section summary in list file</td>
</tr>
</tbody>
</table>

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:

```
asarm --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[=bug,...]

Description

With this option you specify for which hardware problems the assembler should check. Please refer to Chapter 19, CPU Problem Bypasses and Checks for the numbers and descriptions. Silicon bug numbers are specified as a comma separated list. When this option is used without arguments, all silicon bugs are checked.

Example

To check for problem 602117, enter:

asarm --silicon-bug=602117 test.src

Related information

Chapter 19, CPU Problem Bypasses and Checks
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></th>
<th>g</th>
<th>Default symbol scope is global</th>
</tr>
</thead>
<tbody>
<tr>
<td>local</td>
<td>l</td>
<td>Default symbol scope is local</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 option: --thumb

Menu entry
1. Select Assembler » Miscellaneous.
2. Enable the option Assemble Thumb instructions by default.

Command line syntax
--thumb

Description
With this option you tell the assembler that the input file contains Thumb code. By default the assembler assumes that the input file contains ARM code. Specifying --thumb with --cpu=ARMv7M or with the Thumb only assembler (asarmt) is not required.

Note that the input may still contain mixed Thumb and ARM code because the .ARM, .THUMB, .CODE16 and .CODE32 directives overrule the --thumb option. Assembly code can check the setting of the --thumb option by means of the built-in assembly function @THUMB(). So, if you use @THUMB() in a .ARM part and you specified --thumb, @THUMB() still returns 1.

Related information
Assembly function @THUMB()
Assembler directives .CODE16, .CODE32, .ARM, .THUMB
Assembler option: --version (-V)

Menu entry
-

Command line syntax

--version
-v

Description

Display version information. The assembler ignores all other options or input files.

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 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)
11.5. 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 for**
   
   *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:

```
lkarm -mfkl test.obj
lkarm --map-file-format=+files,+link,+locate test.obj
```

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 uppercase and lowercase characters in symbols. By default the linker considers uppercase and lowercase 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 `.obj` 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 or enable Generate binary 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

```
--chip-output=\[basename\]:format[\:addr_size],...
-c[basename]:format[\:addr_size],...
```

You can specify the following formats:

- IHEX Intel Hex
- SREC Motorola S-records
- BIN Binary

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). For binary files the value must be 0.

Description

With this option you specify the Intel Hex or Motorola S-record output format for loading into a PROM-programmer, or you can specify the binary output format for importing in a host program.

The linker generates a file for each ROM memory defined in the LSL file, where sections are located:

```
memory memname
{  type=rom;  }
```

The name of the file is the name of the Eclipse project or, on the command line, the name of the memory device that was emitted with extension .hex, .sre or .bin. Optionally, you can specify a basename which prepends the generated file name.

The linker also always generates a task-related absolute object file in ELF/DWARF format and a memory definition file, unless you specify linker option --no-default-output.
Example

To generate Intel Hex output files for each defined memory, enter the following on the command line:

```
  lkarm --chip-output=myprog:IHEX test1.obj
```

In this case, this generates the file `myprog_memname.hex`.

Related information

Linker option `--output` (Output file)

Linker option `--no-default-output` (No default task-related output files)
Linker option: --core

Menu entry

1. Expand C/C++ Build and select Processor.
2. From the Multi-core configuration list, select a single-core or select All cores for a virtual-core project.

Command line syntax

--core=ARM-core

You can specify the following cores:

- ARM:cm0       CM0 core
- ARM:cm4       CM4 core
- ARM:cm7_n     CM7 core n, where n is 0 to the number of cores -1.
- ARM:vm7       virtual CM7 core

Default: ARM:cm7_0

Description

With this option you specify the core of the target processor for the current link task. Only one task can be assigned to a certain core.

The core is specified as ARM:cm7_n for single-core n or ARM:vm7 for virtual-core. For example, the file cyt4bf.lsl in the include.lsl directory of the vendor specific Eclipse plugin, contains a description of derivative traveo2 and the supported cores. ARM is the processor as specified in the LSL file.

Example

To link objects for core ARM:cm7_0, enter:

```
lkarm -o test.abs -CARMv7EM -D__PROC_CYT4BF8CEB_M7__
-D__DEVICE_LSL_FILE=cyt4bf.lsl -D__FLASH_SIZE=8384k -D__CORE_CM7
--core=ARM:cm7_0 --map-file test.o
-L~\carm\lib\v7EM\le\VFPv5 -lcthumb -lfpthumb -lrtthumb
```

Related information

Control program option --lsl-core (Specify LSL core)
Linker option: --cpu (-C)

Menu entry

1. Expand C/C++ Build and select Processor.
2. From the Processor selection list, make a selection by Architecture, Core or one of the manufacturers.

Command line syntax

--cpu=architecture
-C architecture

You can specify the following architectures:

- ARMv6M Link for ARMv6-M architecture
- ARMv7M Link for ARMv7-M architecture
- ARMv7EM Link for ARMv7E-M architecture

Description

With this option you specify the ARM architecture for which you create your application. The linker uses the architecture to determine which libraries must be linked and what kind of veneers to generate. The default architecture is ARMv7M and the complete list of supported architectures is: ARMv6-M, ARMv7-M or ARMv7E-M.

Related information

C compiler option --cpu (Select architecture)
Linker option: --define (-D)

Menu entry

1. Select **Linker » Script File.**

   *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 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.

Make sure you do not use a reserved keyword as a macro name, as this can lead to unexpected results.

Example

To define the stack size and start address which are used in the linker script file `arm.lsl`, enter:

```
lkarm test.obj -otest.abs --lsl-file=arm.lsl --define=__STACK=32k
   --define=__START=0x00000000
```

Related information

**Linker option --option-file** (Specify an option file)
**Linker option: --dep-file**

**Menu entry**

Eclipse uses this option in the background to create a file with extension .d (one for every input file).

**Command line syntax**

```
--dep-file[=file]
```

**Description**

With this option you tell the linker to generate dependency lines that can be used in a Makefile. The dependency information will be generated in addition to the normal output file.

By default, the information is written to the file `lkarm.d`. When you specify a filename, all dependencies will be combined in the specified file.

**Example**

```
lkarm --dep-file=test.dep test.obj
```

The linker links the file `test.obj`, which results in the output file `test.abs`, and generates dependency lines in the file `test.dep`.

**Related information**

Linker option **--make-target** (Target to use in dependencies file)
Linker option: --diag

Menu entry

1. From the Window menu, select Show View » Other » TASKING » 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:

```
lkarm --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:

```
lkarm --diag=html:all > lkerrors.html
```

**Related information**

Section 7.12, *Linker Error Messages*
Linker option: --endianness

Menu entry

1. Select Global Options.
2. Specify the Endianness: Little-endian mode or Big-endian mode.

Command line syntax

--endianness=endianness

-B
--big-endian

You can specify the following endianness:

<table>
<thead>
<tr>
<th>big</th>
<th>b</th>
<th>Big endian</th>
</tr>
</thead>
<tbody>
<tr>
<td>little</td>
<td>l</td>
<td>Little endian (default)</td>
</tr>
</tbody>
</table>

Description

By default, the linker links objects in little-endian mode. With --endianness=big you tell the linker to link the input files in big-endian mode. The endianness used must be valid for the architecture you are linking for. Depending on the endianness used, the linker links different libraries. -B is an alias for option --endianness=big.

Related information

-
**Linker option: --error-file**

**Menu entry**

-  

**Command line syntax**

```
--error-file [=file]
```

**Description**

With this option the linker redirects diagnostic messages to a file. If you do not specify a filename, the error file is `lkarm.elk`.

**Example**

To write diagnostic messages to `errors.elk` instead of `stderr`, enter:

```
lkarm --error-file=errors.elk test.obj
```

**Related information**

Section 7.12, *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 7.12, Linker Error Messages
Linker option: --extern (-e)

Menu entry
-

Command line syntax

--extern=symbol,...
-e 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:

lkarm mylib.lib

Nothing is linked and no output file will be produced, because there are no unresolved symbols when the linker searches through mylib.lib.

lkarm --extern=_START mylib.lib

In this case the linker searches for the symbol _START in the library and (if found) extracts the object that contains _START, the startup code. If this module contains new unresolved symbols, the linker looks again in mylib.lib. This process repeats until no new unresolved symbols are found.

Related information

Section 7.3, 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:

```
lkarm --first-library-first a.lib test.obj b.lib
```

If the file `test.obj` calls a function which is both present in `a.lib` and `b.lib`, normally the function in `b.lib` would be extracted. With this option the linker first tries to extract the symbol from the first library `a.lib`.

Note that routines in `b.lib` that call other routines that are present in both `a.lib` and `b.lib` are now also resolved from `a.lib`.

**Related information**

Linker option `--no-rescan` (Rescan libraries to solve unresolved externals)
Linker option: --global-map-file

Menu entry
-

Command line syntax

--global-map-file=file[:XML],...

Default: no global map file is generated

Description

With this option you tell the linker to generate a global linker map file that includes information about each of the tasks.

A global linker map file is a text or XML file that shows how the linker has mapped the sections and symbols from the various object files (.obj) to the linked object file. A locate part shows the absolute position of each section. External symbols are listed per space with their absolute address, both sorted on symbol and sorted on address.

Related information

Linker option --global-map-file-format (Format global map file)

Linker option --map-file (Generate map file for a single task)

Section 15.2, Linker Map File Format
Linker option: --global-map-file-format

Menu entry

Command line syntax

--global-map-file-format=flag,...

You can set the following flags:

+/-callgraph  c/C  Include call graph information
+/-removed    d/D  Include information on removed sections
+/-files      f/F  Include processed files information
+/-invocation i/I  Include information on invocation and tools
+/-link       k/K  Include link result information
+/-locate     l/L  Include locate result information
+/-memory     m/M  Include memory usage information
+/-nonalloc   n/N  Include information of non-alloc sections
+/-overlay    o/O  Include overlay information
+/-statics    q/Q  Include module local symbols information
+/-crossref   r/R  Include cross references information
+/-lsI        s/S  Include processor and memory information
+/-rules      u/U  Include locate rules

Use the following options for predefined sets of flags:

--global-map-file-format=0  Link information
   Alias for --global-map-file-format=cDfikLMNoQrSU
--global-map-file-format=1  Locate information
   Alias for --global-map-file-format=CDfIKMNoQRSu
--global-map-file-format=2  Most information
   Alias for --global-map-file-format=cdfiklmNoQrSu

Default: --global-map-file-format=2

Description

With this option you specify which information you want to include in the global map file.

On the command line you must use this option in combination with the option --global-map-file.
Related information

Linker option `--global-map-file` (Generate global map file)

Section 15.2, *Linker Map File Format*
Linker option: --global-type-checking

Menu entry

Command line syntax
--global-type-checking

Description
Use this option when you want the linker to check the types of variable and function references against their definitions, using DWARF 3 debug information.

This check should give the same result as the C compiler when you use MIL linking.

Related information
C compiler option --global-type-checking (Global type checking)
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:

lkarm -?
lkarm --help
lkarm

To see a detailed description of the available options, enter:

lkarm --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. Enable or disable the option Emit start address record.

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)

Section 16.2, Intel Hex Record Format
Linker option: --hex-record-size

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-record-size to the Additional options field.

Command line syntax

```
--hex-record-size=size
```

Default: 32

Description

With this option you can set the size (width) of the Intel Hex data records.

Related information

Linker option --output (Output file)

Section 16.2, Intel Hex Record Format
Linker option: --import-object

Menu entry

1. Select **Linker » Data Objects**.
   
   *The Data objects box shows the list of object files that are imported.*

2. To add a data object, click on the **Add** button in the **Data objects** box.

3. Type or select a binary file (including its path).

   Use the **Edit** and **Delete** button to change a filename or to remove a data object from the list.

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 7.6, *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:

lkarm --include-directory=c:\proj\include --lsl-file=mylsl.lsl test.obj

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.obj, test2.obj and test3.obj are incrementally linked:
1. lkarm --incremental test1.obj test2.obj --output=test.out
   test1.obj and test2.obj are linked
2. lkarm --incremental test3.obj test.out
   test3.obj and test.out are linked, task1.out is created
3. lkarm task1.out
   task1.out is located

Related information
Section 7.4, 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 TASKING support.

Related information

Linker option --warnings-as-errors (Treat warnings as errors)
Linker option: --library (-l)

Menu entry

1. Select **Linker » Libraries**.

   The Libraries 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 name 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 to use system library \textit{name}.lib, where \textit{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 \texttt{LIBARM}, unless you used the option `--ignore-default-library-path`.

Example

To search in the system library \texttt{carm.lib} (C library):

```
lkarm test.obj mylib.lib --library=carm
```

The linker links the file \texttt{test.obj} and first looks in library \texttt{mylib.lib} (in the current directory only), then in the system library \texttt{carm.lib} to resolve unresolved symbols.

Related information

- Linker option `--library-directory` (Additional search path for system libraries)
- Section 7.3, *Linking with Libraries*
Linker 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

--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\architecture\endianness.

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 LIBARM. 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 LIBARM.

3. The default directory $(PRODDIR)\libarchitecture\endianness.

Example

Suppose you call the linker as follows:

lkarm test.obj --library-directory=c:\mylibs --library=carm
First the linker looks in the directory c:\mylibs for library carm.lib (this option). If it does not find the requested libraries, it looks in the directory that is set with the environment variables LIBARM. Then the linker looks in the default directory $(PRODDIR)\libarchitecture\endianness for libraries.

Related information

Linker option --library (Link system library)

Linker option --cpu (Select architecture)

Linker option --endianness (Specify endianness)

Section 7.3.1, 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 --create=relocatable (-cl) (Stop after linking)
Linker option: --long-branch-veneers

Menu entry
1. Select Linker » Miscellaneous.
2. Enable the option Generate long-branch veneers.

Command line syntax

--long-branch-veneers[=flag]

You can specify the following flag:

    relative      r      Regular long-branch veneers

Default: r

Description

With this option you enable the linker to generate a long-branch veneer if the target of a B (ARM only, not for Thumb), BL or BLX instruction is out-of-range. The locating process of the linker may become less efficient if this option is switched on, even if no long-branch veneers are required after all. Therefore it is better to first see if out-of-range branches are in the code (unlikely) before switching on this option.

Related information

-
**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 7.8, *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 for a single task). If you do not specify a filename, the file `lkarm.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 » TASKING ARM C/C++ Project.
   
   The New C/C++ Project wizard appears.

2. Fill in the project settings in each dialog and click Next > until the ARM 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.

The LSL file can be specified in the Properties dialog:

1. Select Linker » Script File.

2. Specify a LSL file in the Linker script file (.lsl) field (default ../${ProjName}.lsl).

Command line syntax

--lsl-file=file

-dfile

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 (default.lsl). 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 7.8, Controlling the Linker with a Script
Linker 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 option --dep-file. The default target name is the basename of the input file, with extension .abs.

Example

lkarm --make-target=mytarget.abs test.obj

The linker generates dependency lines with the default target name mytarget.abs instead of test.abs.

Related information

Linker option --dep-file (Generate dependencies in a file)
**Linker option: --map-file (-M)**

**Menu entry**

1. Select **Linker » Map File**.
2. Enable the option **Generate XML map file format (.mapxml) for map file viewer**.
3. (Optional) Enable the option **Generate map file**.
4. Enable or disable the types of information to be included.

**Command line syntax**

```
--map-file [=file][::XML]
```

```
-M[=file][::XML]
```

Default (Eclipse): XML map file is generated
Default (linker): 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.

In Eclipse the XML variant of the map file (extension **.mapxml**) is used for graphical display in the map file viewer.

A linker map file is a text file that shows how the linker has mapped the sections and symbols from the various object files (**.obj**) to the linked object file. A locate part shows the absolute position of each section. External symbols are listed per space with their absolute address, both sorted on symbol and sorted on address.

**Related information**

Linker option **--map-file-format** (Format map file)

Section 15.2, **Linker Map File Format**
Linker option: --map-file-format (-m)

Menu entry

1. Select Linker » Map File.
2. Enable the option Generate XML map file format (.mapxml) for map file viewer.
3. (Optional) Enable the option Generate map file.
4. 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 c/C Include call graph information
+/-removed d/D Include information on removed sections
+/-files f/F Include processed files information
+/-invocation i/I Include information on invocation and tools
+/-link k/K Include link result information
+/-locate l/L Include locate result information
+/-memory m/M Include memory usage information
+/-nonalloc n/N Include information of non-alloc sections
+/-overlay o/O Include overlay information
+/-statics q/Q Include module local symbols information
+/-crossref r/R Include cross references information
+/-lsl s/S Include processor and memory information
+/-rules u/U Include locate rules
```

Use the following options for predefined sets of flags:

```
--map-file-format=0 -m0 Link information
  Alias for -mcDfikLMNoQrSU
--map-file-format=1 -m1 Locate information
  Alias for -mCDfIKIMNoQRSU
--map-file-format=2 -m2 Most information
  Alias for -mcdfiklmNoQrSu
```

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 for a single task)

Section 15.2, *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.

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.

Related information

-
Linker option: --new-task

Menu entry

Command line syntax

--new-task

Description

With this option the linker creates an additional task. Any options that follow only apply to the new task.

The linker processes options on the command line from left to right. To know whether a certain option belongs to a different task it uses this option. This implies that all options for a given task must be fully specified before moving on to the next.

Related information

Control program option --new-task (Link a task for LSL core)
**Linker option: --non-romable**

**Menu entry**
1. Select **Linker » Miscellaneous**.
2. Enable the option **Application is not romable**.

**Command line syntax**

```
--non-romable
```

**Description**

With this option you tell the linker that the application must not be located in ROM. The linker will locate all ROM sections, including a copy table if present, in RAM. When the application is started, the data sections are re-initialized and the BSS sections are cleared as usual.

This option is, for example, useful when you want to test the application in RAM before you put the final application in ROM. This saves you the time of flashing the application in ROM over and over again.

If you want to locate your application in RAM only, without using ROM/flash resources of the chip, for example when you run the debugger in RAM only, also specify the options **--no-rom-copy** and **--user-provided-initialization-code**.

**Related information**

- Linker option **--no-rom-copy** (Do not generate ROM copy)
- Linker option **--user-provided-initialization-code** (Own initialization code, no standard copy table)
Linker option: --no-default-output

Menu entry

Command line syntax

--no-default-output

Description

By default the linker generates an absolute object file and a memory definition file for each task. With this option you specify to the linker not to generate these files, unless explicitly specified.

Example

Invocation to create an Intel Hex output for each chip only:

lkarm -cmyprog:IHEX --no-default-output test.obj

This generates the file myprog_memname.hex. Without --no-default-output also the files task1.abs and task1.mdf are generated.

Related information

Linker option --chip-output (Generate an output file for each chip)

Control program option --no-map-file (Do not generate map file)
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 --non-romable (Application is not romable)

Linker option --user-provided-initialization-code (Own initialization code, no standard copy table)
**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:

```
lkarm --no-warnings=135,136 test.obj
```

**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
   - Compress ROM sections of copy table items

**Command line syntax**

```
--optimize=flag, ...
-o flags
```

You can set the following flags:

- `-delete-unreferenced-sections`
- `-first-fit-decreasing`
- `-copytable-compression`
- `-delete-duplicate-code`
- `-delete-duplicate-data`
- `-copytable-item-compression`

Use the following options for predefined sets of flags:

```
--optimize=0       -O0  No optimization  
                   Alias for -OCLTXYZ
--optimize=1       -O1  Default optimization  
                   Alias for -OcLtxyZ
--optimize=2       -O2  All optimizations  
                   Alias for -OcltxyZ
```
Default: \texttt{--optimize=1}

**Description**

With this option you can control the level of optimization.

Note that when you use the flag \texttt{+copytable-compression}, sections affected by the copy table are located as if they were in a clustered LSL group, if they do not have a locate restriction yet.

Note that when you use the flag \texttt{+delete-duplicate-code} and/or \texttt{+delete-duplicate-data}, different C objects or functions may have identical addresses. This means that you cannot distinguish these objects or functions with a pointer comparison as described in the ISO C standard (C99/C11 6.5.9p6). If your application relies on pointer comparisons to distinguish different objects and/or functions, disable these linker optimizations.

**Related information**

For details about each optimization see Section 7.7, *Linker Optimizations*.

Define the mutual order of sections in an LSL group in Section 17.8.2, *Creating and Locating Groups of Sections*. 
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 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:

```
--map-file=my.map          (generate a map file)
test.obj                  (input file)
--library-directory=c:\mylibs (additional search path for system libraries)
```

Specify the option file to the linker:

```
ltkarm --option-file=myoptions
```

This is equivalent to the following command line:

```
ltkarm --map-file=my.map test.obj --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.abs`.

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). Note that if you make the `addr_size` too small, the linker might give a fatal object writer error indicating an address overflow.

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_spacename` 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 `myprog.hex` of the default address space, enter:

```
lkarm test.obj --output=myprog.hex:IHEX:4
```

**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
-

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)

Menu entry
1. Select **Linker » Miscellaneous**.
2. Enable the option **Show link phases during processing**.

   *The verbose output is displayed in the Problems view and the Console view.*

Command line syntax

```bash
--verbose
-v
```

Description

With this option you put the linker in *verbose* mode. The linker prints the link phases while it processes the files. The linker prints one entry for each action it executes for a task. When you use this option twice (`-vv`) you put the linker in *extra verbose* mode. In this mode the linker also prints the filenames and it shows which objects are extracted from libraries and it shows verbose information that would normally be hidden when you use the normal verbose mode or when you run without verbose. 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.

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)
Linker option: --whole-archive

Menu entry
1. Select Linker » Miscellaneous.
2. Add the option --whole-archive to the Additional options field.

Command line syntax

--whole-archive=file

Description
This option tells the linker to directly load all object modules in a library, as if they were placed on the command line. This is different from libraries specified as input files or with the -l option, which are only used to resolve references in object files that were loaded earlier.

Example
Suppose the library myarchive.lib contains the objects my1.obj, my2.obj and my3.obj. Specifying

lkarm --whole-archive=myarchive.lib

is the same as specifying

lkarm my1.obj my2.obj my3.obj

Related information
Linker option --library (Link system library)
11.6. Control Program Options

The control program ccarm facilitates the invocation of the various components of the ARM 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 C++ compiler, C compiler, assembler or linker, it is recommended to use the control program options --pass-c++, --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:

```
ccarm -Wc-Oac test.c
ccarm --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:

```
ccarm --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: --c++

Menu entry
1. Select C/C++ Compiler » Language.
2. From the Comply to C++ standard list, select the ISO C++ standard.

Command line syntax

--c++=year

You can specify one of the following years: 03, 11 or 14.

Default: --c++=14

Description


When you also use I/O streams (option --io-streams), this option selects the proper C++ library. C++03 selects the STLport library, C++11 or later selects the libc++ library.

Related information

Control program option --exceptions (Enable exceptions)
Control program option --io-streams (C++ I/O streams)
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: --control-flow-info

Menu entry
1. Select C/C++ Compiler » Debugging.
2. Enable the option Generate control flow information.

Command line syntax
--control-flow-info

Description
Control flow information
With this option the compiler adds control flow information to the output file. The compiler generates a .debug_control_flow section which describes the basic blocks and their relations. This information can be used for code coverage analysis on optimized code.

Example
ccarm --control-flow-info test.c

Related information
Section 9.5.2, HLL Dump Output Format
Control program option --debug-info (Debug information)
Control program option: --cpu (-C)

Menu entry
1. Expand C/C++ Build and select Processor.
2. From the Processor selection list, make a selection by Architecture, Core or one of the manufacturers.

Command line syntax

```bash
--cpu=architecture | id | name
-architecture | id | name
```

You can specify the following architectures:

- **ARMv6M**: Compile/assemble for ARMv6-M architecture
- **ARMv7M**: Compile/assemble for ARMv7-M architecture
- **ARMv7EM**: Compile/assemble for ARMv7E-M architecture

Description

With this option you specify the ARM architecture for which you create your application. The architecture determines which instructions are valid and which are not. The default architecture is ARMv7M and the complete list of supported architectures is: ARMv6-M, ARMv7-M or ARMv7E-M. Instead of the architecture name you can also specify the core name, like "Cortex-M3".

If you are using a vendor specific processor definition file (see the option --processors), you can also specify a processor. The vendor processor definition file defines for each processor its full name (for example, STM32F205RB), its ID (for example, stm32f205rb), the architecture name (for example, ARMv7M), the core settings (for example, cortexm3), the on-chip flash settings, the list of silicon bugs for that processor. Each processor also defines options to supply to the linker for preprocessing the LSL file for the applicable on-chip memory definitions (for example, -D__FLASH_SIZE=128k).

The lookup sequence for names specified to this option is as follows:

1. match any of the standard architecture names (as listed above, for example ARMv7M)
2. if none matched, match with the 'id' attribute in the processor definition file (case insensitive, for example armv7_m)
3. if still none matched, match with the 'name' attribute in the processor definition file (case insensitive, for example STM32F205RB)
4. if still none matched, the control program issues a fatal error.

If you specify a full processor name (or its ID), the control program passes the option -D__PROC_id__ to C compiler and the linker. id is the 'id' attribute belonging to the processor found in one of the processor definition files, in uppercase. The control program also passes the macros defined with the
'linker_macros' property of the processor found in the processor definition file to the linker (for example, 
-D__DEVICE_LSL_FILE=stm32f2xx.lsl).

Assembly code can check the value of the option by means of the built-in function @CPU().

Example

After

ccarm --processors="vendor_support_dir\etc\processors.xml" 
  --cpu="STM32F205RB" -v -t test.c

the control program will call the tools as follows:

carm -CARMv7M -DPROC_STM32F205RB -o test.src test.c
asarmt -CARMv7M -o test.obj test.src
lkarm -o test.abs test.obj -CARMv7M -DPROC_STM32F205RB
  -DDEVICE_LSL_FILE=stm32f2xx.lsl -DFLASH_SIZE=128k -DSRAM_SIZE=64k
  --map-file -lcthumb -lfpthumb -lrtthumb

Related information

Control program option --cpu-list (Show list of processors)

C compiler option --cpu (Select architecture)

Assembler option --cpu (Select architecture)

Control program option --processors (Read additional processor definitions)

Assembly function @CPU()
Control program option: --cpu-list

Menu entry

Command line syntax

--cpu-list[=pattern]

Description

With this option the control program shows a list of supported processors as defined in the default file processors.xml and the optionally specified (see the option --processors) processors definition files. This can be useful when you want to select a processor name or id for the --cpu option.

The pattern works similar to the UNIX grep utility. You can use it to limit the output list.

Example

To show a list of all processors, enter:

ccarm --cpu-list

To show all processors that have stm32f107 in their name, enter:

ccarm --processors="vendor_support_dir\etc\processors.xml"
   --cpu-list=stm32f107

--- ~\carm\etc\processors.xml ---
  id             name           CPU       core
--- vendor_support_dir\etc\processors.xml
  id             name           CPU       core
  stm32f107rb    STM32F107RB    ARMv7M    cortexm3
  stm32f107rc    STM32F107RC    ARMv7M    cortexm3
  stm32f107vb    STM32F107VB    ARMv7M    cortexm3
  stm32f107vc    STM32F107VC    ARMv7M    cortexm3

Related information

Control program option --cpu (Select processor)

Control program option --processors (Read additional processor definitions)
Control program option: --create (-c)

Menu entry
-

Command line syntax

```
--create[=stage]
-c[stage]
```

You can specify the following stages:

<table>
<thead>
<tr>
<th>Stage</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>intermediate-c</td>
<td>c</td>
<td>Stop after C++ files are compiled to intermediate C files (.ic)</td>
</tr>
<tr>
<td>relocatable</td>
<td>l</td>
<td>Stop after the files are linked to a linker object file (.out)</td>
</tr>
<tr>
<td>mil</td>
<td>m</td>
<td>Stop after C++ files or C files are compiled to MIL (.mil)</td>
</tr>
<tr>
<td>object</td>
<td>o</td>
<td>Stop after the files are assembled to objects (.obj)</td>
</tr>
<tr>
<td>assembly</td>
<td>s</td>
<td>Stop after C++ files or C files are compiled to assembly (.src)</td>
</tr>
</tbody>
</table>

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.obj`:

```
ccarm --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, Small set 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>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>small</td>
<td>1</td>
</tr>
<tr>
<td>default</td>
<td>2</td>
</tr>
<tr>
<td>all</td>
<td>3</td>
</tr>
</tbody>
</table>

Emit small set of debug information.
Emit default symbolic debug information.
Emit full symbolic debug information.

Default: --debug-info (same as --debug-info=default)

Description

With this option you tell the control program to include debug information in the generated object file.

The control program passes the option -g suboption to the C compiler and calls the assembler with -gs.

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.

   Make sure you do not use a reserved keyword as a macro name, as this can lead to unexpected
   results.

Example

Consider the following C program with conditional code to compile a demo program and a real program:

    void main( void )
    {
    #if DEMO
        demo_func(); /* compile for the demo program */
    #else
        real_func(); /* compile for the real program */
You can now use a macro definition to set the DEMO flag:

```
ccarm --define=DEMO test.c
ccarm --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.

```
ccarm --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: --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

ccarm --dep-file=test.dep -t 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

Control program option --preprocess=+make (Generate dependencies for make)
Control program option: --diag

Menu entry

1. From the Window menu, select Show View » Other » TASKING » 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

```bash
--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.

Example

To display an explanation of message number 103, enter:

```
ccarm --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:

```
ccarm --diag=html:all > ccerrors.html
```
Related information

Section 4.7, 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: --dsp-library

Menu entry
1. Select C/C++ Compiler » Include Paths.
2. Enable the option Add CMSIS include paths.
4. Enable the option Link CMSIS DSP library.

Command line syntax
--dsp-library

Description
With this option the control program sets the C/C++ compiler macro ARM_MATH_CM0, ARM_MATH_CM0PLUS, ARM_MATH_CM3, ARM_MATH_CM4 or ARM_MATH_CM7, depending on the selected processor. These macros are required for the CMSIS arm_math.h header file to operate correctly. The control program also passes the appropriate CMSIS DSP library to the linker. When MIL linking, the MIL library variant will be used.

Example
After

carm -CCortex-M3 --dsp-library -I"cmsis\Include"
   -I"cmsis\Device\ARM\ARMCM3\Include" -v -t test.c

the control program will call the tools as follows:

+ carm -CARMv7M -D__PROC_CORTEX_M3__ -DARM_MATH_CM3 -Icmsis\Include
   -Icmsis\Device\ARM\ARMCM3\Include -o test.src test.c
+ asarmt -CARMv7M -Icmsis\Include -Icmsis\Device\ARM\ARMCM3\Include
   -o test.obj test.src
+ lkarm -o test.abs test.obj -CARMv7M -D__PROC_CORTEX_M3__
   -D__DEVICE_LSL_FILE=cm3.lsl --map-file
   -lcthm -lfpthumb -lrtc thumb -ldspthumb

Related information
Section 14.1, Using the CMSIS DSP Library
Control program option: --endianness

Menu entry
1. Select Global Options.
2. Specify the Endianness: Little-endian mode or Big-endian mode.

Command line syntax

```
--endianness=endianness
-B
--big-endian
--be32
```

You can specify the following endianness:

```
big   b   Big endian
little l   Little endian (default)
```

Description

By default, the compiler generates code for a little-endian target (least significant byte of a word at lowest byte address). With `--endianness=big` the compiler generates code for a big-endian target (most significant byte of a word at lowest byte address). `-B` is an alias for option `--endianness=big`.

Related information

-
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 diagnostic messages to a file.

The error file will be named after the output file with extension .err (for compiler) or .ers (for assembler). For the linker, the error file is lkarm.elk.

Example
To write diagnostic messages to error files instead of stderr, enter:

carm --error-file -t test.c

Related information
Control Program option --warnings-as-errors (Treat warnings as errors)
Control program option: --error-limit

Menu entry
1. Select C/C++ Compiler » 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 limit the number of error messages in one invocation to the specified number. When the limit is exceeded, the control program aborts with fatal error message F105. Warnings and informational messages are not included in the count. When 0 (zero) or a negative number is specified, the control program emits all errors. Without this option the maximum number of errors is 42. The control program also passes this option to the C++ compiler, C compiler, assembler and linker.

Related information

Section 4.7, C Compiler Error Messages
Control program option: --exceptions

Menu entry
1. Select C/C++ Compiler » Language.
2. Enable or disable the option Support for C++ exception handling.

Command line syntax

--exceptions
--no-exceptions

Default: --no-exceptions
Default in C++11 mode and later modes: --exceptions

Description

With this option you enable or disable support for exception handling in the C++ compiler.

The macro __EXCEPTIONS is defined when exception handling support is enabled.

This feature is implicitly enabled in C++11 mode and later modes.

Related information

Control program option --c++=11 (C++ standard)
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: --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:

ccarm --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-model

Menu entry

1. Select C/C++ Compiler » Floating-Point.
2. Make a selection from the Floating-point model list.
3. If you selected Custom, enable one or more individual options.

Command line syntax

--fp-model=flags

You can set the following flags:

- +/-float
- +/-rewrite
- +/-negzero
- f/F: treat 'double' as 'float'
- r/R: allow expression rewriting
- z/Z: ignore sign of -0.0

Default: --fp-model=Frz

Description

With this option you select the floating-point execution model.

With --fp-model=+float you tell the compiler to treat variables and constants of 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 automatically selects the correct libraries.

With --fp-model=+rewrite you allow the compiler to rewrite expressions by reassociating. This might result in rounding differences and possibly different exceptions. An example is to rewrite \((a\cdot c)+(b\cdot c)\) as \((a+b)\cdot c\).

With --fp-model=+negzero you allow the compiler to ignore the sign of -0.0 values. An example is to replace \((a-a)\) by zero.

Related information

Pragmas fp_negzero and fp_rewrite in Section 1.9, Pragmas to Control the Compiler.
Control program option: --fpu

Menu entry

2. Enable the option Use FPU.

Command line syntax

--fpu=fp

You can specify the following arguments:

<table>
<thead>
<tr>
<th>Alias</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPv4-sp</td>
<td>alias for VFPv4-sp</td>
</tr>
<tr>
<td>FPv5</td>
<td>alias for VFPv5</td>
</tr>
<tr>
<td>FPv5-sp</td>
<td>alias for VFPv5-sp</td>
</tr>
<tr>
<td>VFPv2</td>
<td>alias for VFPv3</td>
</tr>
<tr>
<td>VFPv3</td>
<td>Compile for VFPv3 architecture</td>
</tr>
<tr>
<td>VFPv3-sp</td>
<td>Compile for VFPv3-sp architecture</td>
</tr>
<tr>
<td>VFPv4-sp</td>
<td>Compile for VFPv4-sp architecture</td>
</tr>
<tr>
<td>VFPv5</td>
<td>Compile for VFPv5 architecture</td>
</tr>
<tr>
<td>VFPv5-sp</td>
<td>Compile for VFPv5-sp architecture</td>
</tr>
<tr>
<td>none</td>
<td>Compile for software FPU library (default)</td>
</tr>
</tbody>
</table>

Description

With this option you define the kind of FPU support with which you create your application. The v4 versions generate fused multiply-add operations instead of the regular compound operations. The v5 versions add VSEL, VMINNM/VMAXNM, several conversion and rounding instructions. The "sp" suffix denotes single precision floating-point only.

The macro __FPU_fpu__ is defined in the C source file. The fpu is converted to uppercase and the lowercase "v" and the '-' will be removed. Also when "none" is not used, the macro __FPU_VFP__ is defined in the C source file, otherwise the macro __FPU_NONE__ is defined.

Related information

__fpu function qualifier
Control program option: --global-type-checking

Menu entry
1. Select C/C++ Compiler » Diagnostics.
2. Enable the option Perform global type checking on C code.

Command line syntax
--global-type-checking

Description
The C compiler already performs type checking within each module. Use this option when you want the linker to perform type checking between modules. The control program passes this option to both the C compiler and the linker.

Related information
-
Control program option: --help (-?)

Menu entry
-

Command line syntax

`--help[,=item]`

`-?`

You can specify the following argument:

```
  options   o   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:

```
ccarm -?
ccarm --help
ccarm
```

To see a detailed description of the available options, enter:

```
ccarm --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.
4. Optionally enable the option Add CMSIS include paths.
   Use the Edit and Delete button to change a path or to remove a path from the list.

Command line syntax
--include-directory=path,...
-Ipath,...

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:

```
ccarm --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: --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,...
-Hfile,...

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
ccarm --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 --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:

- 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:

- **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

Section 2.4, Template Instantiation
Control program option: --io-streams

Menu entry

1. Select C/C++ Compiler » Language.
2. Enable the option Support for C++ I/O streams.

Command line syntax

--io-streams

Description

As I/O streams require substantial resources they are disabled by default for C++03. Use this option to enable I/O streams support in the C++03 library. So, when you use standard I/O, e.g. `cout`, or STL templates in your C++03 source, you need to specify this option.

This option adds the C++ standard library to the list of system libraries. For C++11 and later, I/O streams features are always available.

This option also enables exception handling.

Related information

Control program option --exceptions (Enable exceptions)
Control program option: --iso

Menu entry
1. Select C/C++ Compiler » Language.
2. From the Comply to C standard list, select ISO C99, ISO C11, or ISO C90.

Command line syntax
--iso={90|99|11}

Default: --iso=11

Description

Independent of the chosen ISO C standard, the control program always links libraries with C11 support.

Example
To select the ISO C99 standard on the command line:

ccarm --iso=99 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 TASKING support.

The control program passes this option to the compiler, assembler and linker.

Example

```
ccarm --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 `.obj` 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

```
ccarm --keep-temporary-files test.c
```

The control program keeps all intermediate files it generates while creating the absolute object file `test.abs`.

Related information

-
Control program option: --library (-l)

Menu entry

1. Select **Linker » Libraries**.

   *The Libraries 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 name or to remove a library from the list.

Command line syntax

```
--library=name
-l name
```

Description

With this option you tell the linker via the control program to use system library `name.lib`, 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 variable **LIBARM**, unless you used the option **--ignore-default-library-path**.

Example

To search in the system library `carm.lib` (C library):

```
carm test.obj mylib.lib --library=carm
```

The linker links the file `test.obj` and first looks in library `mylib.lib` (in the current directory only), then in the system library `carm.lib` 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 7.3, *Linking with Libraries*
- Chapter 14, *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

```
--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\architecture\endianness.

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 LIBARM. 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 LIBARM.
3. The default directory $(PRODDIR)\lib\architecture\endianness.

Example

Suppose you call the control program as follows:

```
ccarm test.c --library-directory=c:\mylibs --library=carm
```
First the linker looks in the directory \c:\mylibs for library carm.lib (this option). If it does not find the requested libraries, it looks in the directory that is set with the environment variables LIBARM. Then the linker looks in the default directory $(PRODDIR)\libarchitecture\endianness for libraries.

Related information

Control program option --library (Link system library)

Section 7.3.1, 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 --create=relocatable (-cl) (Stop after linking)

Linker option --link-only (Link only, no locating)
Control program option: --list-files

Menu entry

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: --lsl-core

Menu entry

1. Expand C/C++ Build and select Processor.
2. From the Multi-core configuration list, select a single-core or select All cores for a virtual-core project.

Command line syntax

--lsl-core=core

You can specify the following cores:

- **cm0**: CM0 core
- **cm4**: CM4 core
- **cm7_\(n\)**: CM7 core \(n\), where \(n\) is 0 to the number of cores -1.
- **vm7**: virtual CM7 core

Default: depends on the selected processor, for cyt4xxxxxx_m7 the default is cm7_0

Description

With this option you select the core name as specified in a linker script file, for the current link task. The control program passes the correct LSL file to the linker and specifies the correct core.

Example

If you call the control program with the following options:

```
-C cyt4bf8ceb_m7 --lsl-core=cm7_0
```

the control program calls the linker with the following options:

```
-CARMv7EM -D_PROC_CYT4BF8CEB_M7__ -D_DEVICE_LSL_FILE=cyt4bf.lsl
-D_FLASH_SIZE=8384k -D_CORE_CM7 --core=ARM:cm7_0
```

Related information

Linker option --core (Specify LSL core)
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 » TASKING ARM C/C++ Project.
   
   The New C/C++ Project wizard appears.

2. Fill in the project settings in each dialog and click Next > until the ARM 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.

The LSL file can be specified in the Properties dialog:

1. Select Linker » Script File.

2. Specify a LSL file in the Linker script file (.lsl) field (default ../${ProjName}.lsl).

Command line syntax

--lsl-file=file,...

--dfile,...

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 (default.lsl). 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 7.8, Controlling the Linker with a Script
Control program 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 .obj.

Example

ccarm --preprocess=+make --make-target=../mytarget.obj test.c

The compiler generates dependency lines with the default target name ../mytarget.obj instead of test.obj.

Related information

Control program option --preprocess=+make (Generate dependencies for make)

Control program option --dep-file (Generate dependencies in a file)
Control program option: --mil-link / --mil-split

Menu entry
1. Select C/C++ Compiler » Optimization.
2. Enable the option Build for application wide optimizations (MIL linking).
3. Select Optimize less/Build faster or Optimize more/Build slower.

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. Application wide code compaction is not possible in this case.

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.

Note that with both options some extra strict type checking is done that can cause building to fail in a way that is unforeseen and difficult to understand. For example, when you use one of these options in combination with option `--schar` you might get the following error:

```
carm E289: ["..\..\..\strlen.c" 14/1] "strlen" redeclared with a different type
carm I802: ["installation-dir\include\string.h" 44/17]
    previous declaration of "strlen"
1 errors, 0 warnings
```

This is caused by the fact that the MIL library is built without `--schar`. You can workaround this problem by rebuilding the MIL libraries.

Build for application wide optimizations (MIL linking) and Optimize less/Build faster

This option is standard MIL linking and splitting. Note that you can control the optimizations to be performed with the optimization settings.
Optimize more/Build slower

When you enable this option, the compiler's frontend does not split the MIL stream in separate modules, but feeds it directly to the compiler's backend, allowing the code compaction to be performed application wide.

Related information

Section 4.1, Compilation Process

C compiler option --mil / --mil-split
Control program option: --mixed-arm-thumb

Menu entry
2. Enable the option Use full assembler for mixed ARM and Thumb instructions.

Command line syntax
--mixed-arm-thumb

Description
With this option the control program calls the mixed ARM and Thumb assembler (asarm).

When you do not use this option, option --thumb determines which target assembler is chosen. Without --thumb: the ARM instruction set only assembler (asarma). With --thumb: the Thumb instruction set only assembler (asmarmt).

See the description of --thumb for more information.

Note that when you specify the ARMv6-M, ARMv7-M or ARMv7-EM architecture profile with --cpu, this automatically selects the Thumb-2 instruction set.

Related information
Control program option --thumb (use Thumb instruction set)
Control program option: --new-task

Menu entry
1. In the C/C++ Projects view, right-click on the name of an ARM CM0 project and select Properties.
   The Properties dialog appears.
2. In the left pane, select Project References.
3. In the right pane, select the CM4 project or CM7 or CM7_\(n\) project that must be part of the CM0 project and click Apply and Close.

Command line syntax

--new-task=core,{obj|out|lib}[,obj|out|lib]...[,linker option]...

You can specify the following cores:

- **cm4**  
  Cortex-M4 core
- **cm7_\(n\)**  
  Cortex-M7 core (\(n\) is 0 or 1).
- **vm7**  
  Virtual Cortex-M7 core (virtual-core project)

Description

With this option you can specify to link a task for a specific core. Eclipse will call this option when you reference a single-core or virtual-core project from a main CM0 project. The first argument is a core name. After the core name a comma separated task input list is specified. The input list is a list of objects, incrementally linked files (.out) or library files (.lib). You can also specify a list of linker options for this link task.

Example

To link objects foo1.obj, foo2.obj of CM7 core cm7_0 to the CM0 project, enter:

ccarm --new-task=cm7_0,foo1.obj,foo2.obj,-Mcm7_0.map myproj.c

This results in the following options being passed to the linker:

--new-task --core=ARM:cm7_0 --map-file=cm7_0.map foo1.obj foo2.obj

Related information

Linker option --new-task (Create additional task)
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 (C11) 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
ccarm --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 (carm.lib) and avoid unresolved externals:
ccarm --no-default-libraries --library=carm test.c

Related information
Control program option --library (Link system library)
Section 7.3.1, How the Linker Searches Libraries
Chapter 14, Libraries
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 (.obj) to the linked object file. A locate part shows the absolute position of each section. External symbols are listed per space with their absolute address, both sorted on symbol and sorted on address.

With this option you prevent the generation of a map file.

Related information

-
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 C compiler warnings box.
3. Enter the numbers, separated by commas or as a range, 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[-number],...]
-w[number[-number],...]

Description

With this option you can suppresses all warning messages for the various tools or specific control program 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 or a range, only the specified control program warnings are suppressed. You can specify the option --no-warnings=number multiple times.

Example

To suppress all warnings for all tools, enter:

carm 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 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:

```
--debug-info
--define=DEMO=1
test.c
```

Specify the option file to the control program:

```
carm --option-file=myoptions
```

This is equivalent to the following command line:

```
carm --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
ccarm test.c prog.c
The control program generates an ELF/DWARF object file (default) with the name test.abs.

To generate the file result.abs:
ccarm --output=result.abs 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
```

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:

```
ccarm --pass-linker=--verbose test.c
```

Related information

-
Control program option: --preprocess (-E) / --no-preprocessing-only

Menu entry
1. Select C/C++ Compiler » Preprocessing.
2. Enable the option Store preprocessor output in <file>.pre.
3. (Optional) Enable the option Keep comments in preprocessor output.
4. (Optional) Enable the option Keep #line info in preprocessor output.

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>+/-includes</td>
<td>generate a list of included source files</td>
</tr>
<tr>
<td>+/-list</td>
<td>generate a list of macro definitions</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: -ECILMP

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=+includes the compiler will generate a list of all included source files. The preprocessor output is discarded.

With --preprocess=+list the compiler will generate a list of all macro definitions. The preprocessor output is discarded.
With --preprocess=+make the compiler will generate dependency lines that can be used in a Makefile. The information is written to a file with extension .d. The preprocessor output is discarded. The default target name is the basename of the input file, with the extension .obj. 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**

carm --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.abs

**Related information**

Control program option --dep-file (Generate dependencies in a file)

Control program option --make-target (Specify target name for -Em output)
Control program option: --processors

Menu entry

1. From the Window menu, select Preferences.  
   The Preferences dialog appears.
2. Select TASKING » ARM.
3. Click the Add button to add additional processor definition files.

Command line syntax

--processors=file

Description

With this option you can specify an additional XML file with processor definitions.

The standard list of supported cores is defined in the file processors.xml. An additionally specified processor definition file defines for each processor its full name (for example, STM32F205RB), its ID (for example, stm32f205rb), the architecture name (for example, ARMv7M), the core settings (for example, cortexm3), the on-chip flash settings, the list of silicon bugs for that processor. Each processor also defines options to supply to the linker for preprocessing the LSL file for the applicable on-chip memory definitions (for example, -D__FLASH_SIZE=128k).

Vendor specific processor definition files are available within Eclipse (and can be updated by the Eclipse update feature) or can be downloaded as a vendor support archive from the TASKING website (www.tasking.com).

The control program reads the specified file after the file processors.xml in the product's etc directory. Additional XML files can override processor definitions made in XML files that are read before.

Multiple --processors options are allowed.

Eclipse generates a --processors option in the makefiles for each specified XML file.

Example

Specify an additional processor definition file (suppose processors-new.xml contains a new processor ARMNEW):

ccarm --processors=processors-new.xml --cpu=ARMNEW test.c

Related information

Control program option --cpu (Select architecture)
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
4. Optionally, when you selected function timers, enable option Exclude time spent in interrupt functions.

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-info 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: -pBcFStI

You can set the following flags:

+/block  b/B  block counters
+/callgraph  c/C  call graph
+/function  f/F  function counters
+/interrupt  i/I  exclude interrupt time
+/static  s/S  static profile generation
+/time  t/T  function timers

Default: -pBCFSTI
Default (without flags): \texttt{-pBCfSTI}

**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 13, *Profiling*.

You can obtain the following profiling data (see flags above):

**Block counters (not in combination with Call graph or Function timers)**

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.

**Function timers (not in combination with Block counters/Function counters)**

This will instrument the code to measure the time spent in a function. This includes the time spent in all sub functions (callees). Also the time spent in interrupt functions is taken into account, unless you enable option *Exclude time spent in interrupt functions* (*-pti*).

**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-info*) 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:
ccarm --profile=+block test.c

In this case the control program tells the linker to link the library pbarm.lib.

Related information

Chapter 13, Profiling
**Control program option: --schar**

**Menu entry**

1. Select **C/C++ Compiler » Language**.
2. Enable the option **Treat 'char' variables as signed**.

**Command line syntax**

--schar

**Description**

By default `char` is the same as specifying `unsigned char` as required by the ARM ABI. With this option `char` is the same as `signed char`. This option is passed to both the C++ compiler and the C compiler.

Note that this option can cause conflicts when you use it in combination with MIL linking. With MIL linking some extra strict type checking is done that can cause building to fail in a way that is unforeseen and difficult to understand. For example, when you use option **--mil-link** in combination with option **--schar** you might get the following error:

```
carm E289: ["..\..\..\strlen.c" 14/1] "strlen" redeclared with a different type
carm I802: ["installation-dir\include\string.h" 44/17]
    previous declaration of "strlen"
1 errors, 0 warnings
```

This is caused by the fact that the MIL library is built without **--schar**. You can workaround this problem by rebuilding the MIL libraries.

**Related information**

Section 1.1, *Data Types*
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[=bug,...]
```

Description

With this option you specify for which hardware problems the compiler should generate workarounds. Please refer to Chapter 19, CPU Problem Bypasses and Checks for the numbers and descriptions. Silicon bug numbers are specified as a comma separated list. When you use this option without arguments, all silicon bug workarounds are enabled.

The control program passes the option to both the compiler and the assembler.

Example

To enable workarounds for problem 602117, enter:

```
ccarm --silicon-bug=602117 test.c
```

Related information

Chapter 19, CPU Problem Bypasses and Checks

Compiler option --silicon-bug

Assembler option --silicon-bug
Control program option: --thumb

Menu entry

2. Enable the option Use Thumb instruction set.

Command line syntax

--thumb

Description

Generate code in Thumb mode or Thumb-2 mode, depending on the architecture. The Thumb instruction set is a subset of the ARM instruction set which is encoded using 16-bit instructions instead of 32-bit instructions. The Thumb-2 instruction set has 16-bit and 32-bit instructions.

Depending on this option and option --mixed-arm-thumb a target assembler is chosen. asarm is the full assembler with both ARM and Thumb instructions. asarma is the ARM instruction set only assembler. asarmt is the Thumb instruction set only assembler.

<table>
<thead>
<tr>
<th>--thumb</th>
<th>--mixed-arm-thumb</th>
<th>Assembler</th>
</tr>
</thead>
<tbody>
<tr>
<td>no</td>
<td>no</td>
<td>asarma</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>asarm</td>
</tr>
<tr>
<td>yes</td>
<td>no</td>
<td>asarmt</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>asarm --thumb</td>
</tr>
</tbody>
</table>

Note that when you specify the ARMv6-M, ARMv7-M, or ARMv7E-M architecture with --cpu, this automatically selects the Thumb-2 instruction set.

Related information

Control program option --mixed-arm-thumb (use mixed ARM and Thumb assembler)
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:

__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__:

ccarm --undefine=__TASKING__ test.c

Related information

Control program option --define (Define preprocessor macro)

Section 1.10, Predefined Preprocessor Macros
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[-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 control program 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 or a range, only the specified control program warnings are treated as an error. You can specify the option --warnings-as-errors=number multiple times.

Use one of the --pass-tool options to pass this option directly to a tool when a specific warning for that tool must be treated as an error. For example, use --pass-c=--warnings-as-errors=number to treat a specific C compiler warning as an error.

Related information

Control program option --no-warnings (Suppress some or all warnings)
Control program option --pass (Pass option to tool)
11.7. Parallel Make Utility Options

When you build a project in Eclipse, Eclipse generates a makefile and uses the make utility amk 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:

```
amk [option...] [target...] [macro=def]
```

This section describes all options for the parallel make utility.

For detailed information about the parallel make utility and using makefiles see Section 9.2, Make Utility amk.
Parallel make utility option: --always-rebuild (-a)

Command line syntax

--always-rebuild

-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

amk -a

Rebuilds all your files, regardless of whether they are out of date or not.

Related information

-
Parallel make utility option: --change-dir (-G)

Command line syntax

--change-dir=\textit{path}

\textit{-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 \textit{-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.

The macro \textit{SUBDIR} is defined with the value of \textit{path}.

Example

Suppose your makefile and other files are stored in the directory \texttt{..\myfiles}. You can call the make utility, for example, as follows:

\texttt{amk -G ..\myfiles}

Related information

-
Parallel make utility option: --diag

Command line syntax

```
--diag=[format:]\{all | msg[-msg],...\}
```

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, or you can specify a range.

**Example**

To display an explanation of message number 169, enter:

```
amk --diag=451
```

This results in the following message and explanation:

```
E451: make stopped
```

An error has occured while executing one of the commands of the target, and -k option is not specified.

To write an explanation of all errors and warnings in HTML format to file **amkerrors.html**, use redirection and enter:

```
amk --diag=html:all > amkerrors.html
```

**Related information**

-
Parallel make utility option: --dry-run (-n)

Command line syntax

--dry-run

-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

amk -n

The make utility does not perform any tasks but displays what it would do if called without the option -n.

Related information

Parallel make utility option -s (Do not print commands before execution)
Parallel make utility option: --help (-? / -h)

Command line syntax

--help [=item]

-h [item]

-?

You can specify the following arguments:

| options | 0       | 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:

amk -?
amk -h
amk --help

To see a detailed description of the available options, enter:

amk --help=options

Related information

-
Parallel make utility option: --jobs (-j) / --jobs-limit (-J)

Menu
1. From the **Project** menu, select **Properties for**

   *The Properties dialog appears.*

2. In the left pane, select **C/C++ Build**.

   *In the right pane the C/C++ Build page appears.*

3. On the Behavior tab, select **Enable parallel build**.

4. You can specify the number of parallel jobs, or you can use an optimal number of jobs. In the last case, **amk** will fork as many jobs in parallel as cores are available.

Command line syntax

```
--jobs [=number]
-j [number]
```

```
--jobs-limit [=number]
-J [number]
```

Description

When these options you can limit the number of parallel jobs. The default is 1. Zero means no limit. When you omit the `number`, **amk** uses the number of cores detected.

Option `--J` is the same as `-j`, except that the number of parallel jobs is limited by the number of cores detected.

Example

```
amk -j3
```

Limit the number of parallel jobs to 3.

Related information

-
Parallel make utility option: --keep-going (-k)

Command line syntax

--keep-going
-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

amk  -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

-
Parallel make utility option: --list-targets (-l)

Command line syntax

--list-targets
-l

Description

With this option, the make utility lists all "primary" targets that are out of date.

Example

amk -l
list of targets

Related information

-
Parallel make utility option: --makefile (-f)

Command line syntax

--makefile=my_makefile
-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

amk -f mymake

The make utility uses the file `mymake` to build your files.

Related information

-
Parallel make utility option: --no-warnings (-w)

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 751 and 756, enter:

```
amk --no-warnings=751,756
```

Related information

Parallel make utility option `--warnings-as-errors` (Treat warnings as errors)
Parallel make utility option: --silent (-s)

Command line syntax

--silent

-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

amk -s

The make utility rebuilds your files but does not print the commands it executes during the make process.

Related information

Parallel make utility option -n (Perform a dry run)
Parallel make utility option: --version (-V)

Command line syntax

--version

-v

Description

Display version information. The make utility ignores all other options or input files.

Related information

-
Parallel make utility option: --warnings-as-errors

Command line syntax

--warnings-as-errors[=number,...]

Description

If the make utility encounters an error, it stops. When you use this option without arguments, you tell the make utility to treat all warnings as errors. This means that the exit status of the make utility will be non-zero after one or more warnings. As a consequence, the make utility 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

Parallel make utility option --no-warnings (Suppress some or all warnings)
11.8. Archiver Options

The archiver and library maintainer **ararm** 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:

```
ararm 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 9.4, Archiver.

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.

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 -n -u -v</td>
</tr>
<tr>
<td>Extract an object module from the library</td>
<td>-x</td>
<td>-o -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 name</td>
<td>-a name</td>
<td></td>
</tr>
<tr>
<td>Append or move new modules before existing module name</td>
<td>-b name</td>
<td></td>
</tr>
<tr>
<td>Suppress the message that is displayed when a new library is created</td>
<td>-c</td>
<td></td>
</tr>
<tr>
<td>Create a new library from scratch</td>
<td>-n</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>Description</td>
<td>Option</td>
<td>Sub-option</td>
</tr>
<tr>
<td>-----------------------------------------------------------------</td>
<td>--------</td>
<td>------------</td>
</tr>
<tr>
<td>Miscellaneous</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Display options</td>
<td>-?</td>
<td></td>
</tr>
<tr>
<td>Display description of one or more diagnostic messages</td>
<td>--diag</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 file</td>
<td></td>
</tr>
<tr>
<td>Suppress warnings above level ( n )</td>
<td>-wn</td>
<td></td>
</tr>
</tbody>
</table>
Archiver option: --delete (-d)

Command line syntax

--delete [--verbose]
-d [-v]

Description

Delete the specified object modules from a library. With the suboption --verbose (-v) the archiver shows which files are removed.

--verbose -v Verbose: the archiver shows which files are removed.

Example

ararm --delete mylib.lib obj1.obj obj2.obj

The archiver deletes obj1.obj and obj2.obj from the library mylib.lib.

ararm -d -v mylib.lib obj1.obj obj2.obj

The archiver deletes obj1.obj and obj2.obj from the library mylib.lib and displays which files are removed.

Related information

-
Archiver option: --diag

Command line syntax

--diag=[:format:]{all | msg[-msg],...}

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. The archiver does not perform any actions. 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, or you can specify a range.

**Example**

To display an explanation of message number 102, enter:

```
ararm --diag=102
```

This results in the following message and explanation:

F102: cannot create "<file>"

The output file or a temporary file could not be created. Check if you have sufficient disk space and if you have write permissions for the specified file.

To write an explanation of all errors and warnings in HTML format to file **arerrors.html**, use redirection and enter:

```
ararm --diag=html:all > arerrors.html
```

**Related information**

-
**Archiver option: --dump (-p)**

**Command line syntax**

`--dump`  

`-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**

```
ararm --dump mylib.lib obj1.obj > file.obj
```

The archiver prints the file `obj1.obj` to standard output where it is redirected to the file `file.obj`. 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: --extract (-x)

Command line syntax

```
--extract (--modtime) [--verbose]
-x [-o] [-v]
```

Description

Extract an existing module from the library.

```
--modtime  -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.
--verbose  -v  Verbose: the archiver shows which files are extracted.
```

Example

To extract the file `obj1.obj` from the library `mylib.lib`:

```
ararm --extract mylib.lib obj1.obj
```

If you do not specify an object module, all object modules are extracted:

```
ararm -x mylib.lib
```

Related information

-
Archiver option: --help (-?)

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:

ararm -?
ararm --help
ararm

To see a detailed description of the available options, enter:

ararm --help=options

Related information

-
Archiver option: --move (-m)

Command line syntax

--move [-a posname] [-b posname]
-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.

--after=posname  -a posname  Move the specified object module(s) after the existing module.
--before=posname  -b posname  Move the specified object module(s) before the existing module.

Example

Suppose the library mylib.lib contains the following objects (see option --print):

obj1.obj
obj2.obj
obj3.obj

To move obj1.obj to the end of mylib.lib:

ararm --move mylib.lib obj1.obj

To move obj3.obj just before obj2.obj:

ararm -m -b obj3.obj mylib.lib obj2.obj

The library mylib.lib after these two invocations now looks like:

obj3.obj
obj2.obj
obj1.obj

Related information

Archiver option --print (-t) (Print library contents)
Archiver option: --option-file (-f)

Command line syntax

```
--option-file=file
-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 `--option-file (-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 '"' 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:

```
-x mylib.lib obj1.obj
-w5
```
Specify the option file to the archiver:

```
ararm --option-file=myoptions
```

This is equivalent to the following command line:

```
ararm -x mylib.lib obj1.obj -w5
```

**Related information**

-
Archiver option: --print (-t)

Command line syntax

--print [--symbols=0|1]
-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.

--symbols=0 -s0 Displays per object the name of the object itself and all symbols in the object.
--symbols=1 -s1 Displays the symbols of all object files in the library in the form library_name:object_name:symbol_name

Example

ararm --print mylib.lib

The archiver prints a list of all object modules in the library mylib.lib:

ararm -t -s0 mylib.lib

The archiver prints per object all symbols in the library. For example:

cstart.obj
  symbols:
    Reset_Handler

Related information

-
Archiver option: --replace (-r)

Command line syntax

```
--replace [--after=posname] [--before=posname]
[--create] [--new] [--newer-only] [--verbose]

-r [-a posname] [-b posname][--create] [-n] [-u] [-v]
```

Description

You can use the option --replace (-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 --replace (-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 after/before a specified place instead.

```
--after=posname -a posname Insert the specified object module(s) after the existing module posname.
--before=posname -b posname Insert the specified object module(s) before the existing module posname.
--create -c Suppress the message that is displayed when a new library is created.
--new -n Create a new library from scratch. If the library already exists, it is overwritten.
--newer-only -u Insert the specified object module only if it is newer than the module in the library.
--verbose -v Verbose: the archiver shows which files are replaced.
```

The suboptions -a or -b have no effect when an object is added to the library.

Example

Suppose the library mylib.lib contains the following object (see option --print):

```
obj1.obj
```

To add obj2.obj to the end of mylib.lib:

```
ararm --replace mylib.lib obj2.obj
```
To insert `obj3.obj` just before `obj2.obj`:

```
ararm -r -b obj2.obj mylib.lib obj3.obj
```

The library `mylib.lib` after these two invocations now looks like:

```
obj1.obj
obj3.obj
obj2.obj
```

**Creating a new library**

To *create a new library file*, add an object file and specify a library that does not yet exist:

```
ararm --replace newlib.lib obj1.obj
```

The archiver creates the library `newlib.lib` and adds the object `obj1.obj` 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 `--new (-n)`:

```
ararm -r -n mylib.lib obj1.obj
```

The archiver overwrites the library `mylib.lib` and adds the object `obj1.obj` to it. The new library `mylib.lib` only contains `obj1.obj`.

**Related information**

Archiver option `--print (-t)` (Print library contents)
Archiver option: --version (-V)

Command line syntax

--version

-v

Description

Display version information. The archiver ignores all other options or input files.

Related information

-
Archiver option: --warning (-w)

Command line syntax

--warning=level
-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 suppress warnings above level 5:

ararm --extract --warning=5 mylib.lib obj1.obj

Related information

-
11.9. HLL Object Dumper Options

The high level language (HLL) dumper hldumparm is a program to dump information about an absolute object file (.abs).

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:

```
hldumparm -FdhMsy test.abs
hldumparm --dump-format=+dump,+hllsymbols,-modules,+sections,+symbols test.abs
```

When you do not specify an option, a default value may become active.
HLL object dumper option: --blank-out (-b)

Command line syntax

--blank-out [=flag]

-b[flag]

You can specify the following format flags:

+/-labels I/L Black out hexadecimal address and labels.

Default: --blank-out=L

Description

With this option you can blank out addresses and optionally labels in all dump phases. Instead of the addresses and labels crosses (X's) are shown.

The +labels sub-option blanks out hexadecimal addresses and labels. With the -labels sub-option only hexadecimal addresses are blanked out. This is the default.

This option is useful when you want to compare the output, but want to ignore the addresses and labels.

Example

hldumparm -F2 hello.abs

----------- Section dump -----------

.section .text, at(0x000001a0)

000001a0 02 48        main:            ldr     r0, [pc, #0x8]
000001a2 01 68                         ldr     r1, [r0, #0x0]
000001a4 02 48                         ldr     r0, [pc, #0x8]
000001a6 00 f0 d5 b8                   b       printf
000001aa c0 46                         .dh 0184
000001ac 84 01                         .dh 0004
000001ae 04 00                         .dh 0194
000001b0 94 01                         .dh 0000
000001b2 00 00                         .endsec

hldumparm -F2 --blank-out hello.abs

----------- Section dump -----------

.section .text, at(0x000001a0)

XXXXXXXX 02 48        main:            ldr     r0, [pc, #0x8]
XXXXXXXX 01 68                         ldr     r1, [r0, #0x0]
XXXXXXXX 02 48                         ldr     r0, [pc, #0x8]
XXXXXXXX 00 f0 d5 b8                   b       printf
```assembly
mov r8, r8
.dh 0184
.dh 0004
.dh 0194
.dh 0000
.endsec
```

```assembly
hldumparm -F2 --blank-out=+labels hello.abs
```

```assembly
------------ Section dump ------------

.section .text, at(0x000001a0)
```

```assembly
ldr r0, [pc, #0x8]
ldr r1, [r0, #0x0]
ldr r0, [pc, #0x8]
b XXXXXXXXXX
mov r8, r8
.dh 0184
.dh 0004
.dh 0194
.dh 0000
.endsec
```

**Related information**

-
HLL object dumper option: --class (-c)

Command line syntax

--class [=class]

-c [class]

You can specify one of the following classes:

- all: Dump contents of all sections.
- code: Dump contents of code sections.
- data: Dump contents of data sections.

Default: --class=all

Description

With this option you can restrict the output to code or data only. This option affects all parts of the output, except the module list. The effect is listed in the following table.

<table>
<thead>
<tr>
<th>Output part</th>
<th>Effect of --class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module list</td>
<td>Not restricted</td>
</tr>
<tr>
<td>Section list</td>
<td>Only lists sections of the specified class</td>
</tr>
<tr>
<td>Section dump</td>
<td>Only dumps the contents of the sections of the specified class</td>
</tr>
<tr>
<td>HLL symbol table</td>
<td>Only lists symbols of the specified class</td>
</tr>
<tr>
<td>Assembly level symbol table</td>
<td>Only lists symbols defined in sections of the specified class</td>
</tr>
<tr>
<td>Note sections</td>
<td>Not restricted</td>
</tr>
</tbody>
</table>

By default all sections are included.

Related information

Section 9.5.2, HLL Dump Output Format
HLL object dumper option: --copy-table

Command line syntax

--copy-table

Description

With this option the HLL object dumper attempts to translate the specified code address to the destination address of a copy table copy command during disassembly.

Related information

-
HLL object dumper option: --diag

Command line syntax

```
--diag=[format:]{all | msg[-msg],...}
```

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. The HLL object dumper does not process any files. 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, or you can specify a range.

**Example**

To display an explanation of message number 101, enter:

```
hldumparm --diag=101
```

This results in the following message and explanation:

```
F101: cannot create "<file>"
```

The output file or a temporary file could not be created. Check if you have sufficient disk space and if you have write permissions for the specified file.

To write an explanation of all errors and warnings in HTML format to file **hldumperrors.html**, use redirection and enter:

```
hldumparm --diag=html:all > hldumperrors.html
```

**Related information**

-
**HLL object dumper option: --disassembly-intermix (-i)**

**Command line syntax**

```
--disassembly-intermix[=flag]
-i[flag]
```

You can specify the following format flags:

- `+/-single-line` \( s/S \) Force the insert to be limited to the first preceding source line.

Default: `--disassembly-intermix=S`

**Description**

With this option the disassembly is intermixed with HLL source code. The source is searched for as described with option `--source-lookup-path`

The `+single-line` sub-option forces the insert to be limited to the first preceding source line. With the `-single-line` sub-option all source lines that belong to the address are prefixed. For example comments are thus also visible. This is the default.

**Example**

```
hldumparm --disassembly-intermix --source-lookup-path=c:\mylib\src hello.abs
```

**Related information**

HLL object dumper option `--source-lookup-path`
HLL object dumper option: --disassembly-without-encoding (-r)

Command line syntax

```
--disassembly-without-encoding
-r
```

Description

With this option the address and encoding are not part of the disassembly of a code section. This is useful when you only want the disassembly part.

Example

```
hldumparm -F2 hello.abs
```

--------- Section dump ---------

```
000001a0 02 48     main:
000001a2 01 68
000001a4 02 48
000001a6 00 f0 d5 b8
000001aa 04 00
000001ac 00 00
000001b0 94 01
000001b2 00 00

.section .text, at(0x000001a0)
```

```
lr     r0, [pc, #0x8]
ldr     r1, [r0, #0x0]
ldr     r0, [pc, #0x8]
b       printf
mov     r8, r8
    .dh 0184
    .dh 0004
    .dh 0194
    .dh 0000
.endsec
```

```
hldumparm -F2 --disassembly-without-encoding hello.abs
```

--------- Section dump ---------

```
000001a0 02 48     main:
000001a2 01 68
000001a4 02 48
000001a6 00 f0 d5 b8
000001aa 04 00
000001ac 00 00
000001b0 94 01
000001b2 00 00

.section .text, at(0x000001a0)
```

```
lr     r0, [pc, #0x8]
lr     r1, [r0, #0x0]
lr     r0, [pc, #0x8]
b       printf
mov     r8, r8
    .dh 0184
    .dh 0004
    .dh 0194
    .dh 0000
.endsec
```

Related information
HLL object dumper option: --dump-format (-F)

Command line syntax

--dump-format[=flag, ...]

-F [flag]...

You can specify the following format flags:

+/-dump d/D Dump the contents of the sections in the object file. Code sections can be disassembled, data sections are dumped.

+/-debug-control-flow f/F Dump the debug control flow section.

+/-hllsymbols h/H List the high level language symbols, with address, size and type.

+/-modules m/M Print a list of modules found in object file.

+/-note n/N Dump all ELF .note sections.

+/-sections s/S Print a list of sections with start address, length and type.

+/-symbols y/Y List the low level symbols, with address and length (if known).

0 Alias for DFHMNSY (nothing)

1 Alias for DFhMNSY (only HLL symbols)

2 Alias for dFHMNSY (only section contents)

3 Alias for dfhmnsy (everything)

Default: --dump-format=dFhmnsy

Description

With this option you can control which parts of the dump output you want to see. By default, all parts are dumped, except for part 7.

1. Module list
2. Section list
3. Section dump (disassembly)
4. HLL symbol table
5. Assembly level symbol table
6. Note sections
7. Debug control flow section

You can limit the number of sections that will be dumped with the options --sections and --section-types.
Related information
Section 9.5.2, *HLL Dump Output Format*
HLL object dumper option: --expand-symbols (-e)

Command line syntax

--expand-symbols [=flag],...
-e [flag]...

You can specify one of the following flags:

+/-basic-types b/B  Expand arrays with basic C types.
+/-fullpath f/F  Include the full path to the field level.
+/-gap-info g/G  Insert gap markers where data is not consecutive.
+/-nesting-indicator n/N  Print nesting bars.

Default (no flags): --expand-symbols=BFGN

Description

With this option you specify that all struct, union and array symbols are expanded with their fields in the HLL symbol dump.

With --expand-symbols=+basic-types, HLL struct and union symbols are listed including all fields. Array members are expanded in one array member per line regardless of the HLL type. For the fields the types and names are indented with 2 spaces.

With --expand-symbols=+fullpath, all fields of structs and unions and all members of non-basic type arrays are expanded and prefixed with their parent's names.

With --expand-symbols=+gap-info, unused memory in complex data types (structures and unions) between data objects and between code objects is shown as {gap} parts. This option is useful to optimize data memory usage. This option only works if debug information is available in the ELF file.

With --expand-symbols=+nesting-indicator, vertical bars (|) are shown to make it easier to see the expanded structs, unions and arrays.

Example

hldumparm -F1 hello.abs

--------- HLL symbol table --------

00040000 24 struct   _dbg_request [dbg.c]
00040018 80 static char  stdin_buf[80] [_iob.c]

hldumparm -e -F1 hello.abs

--------- HLL symbol table --------

00040000 24 struct   _dbg_request [dbg.c]
hldumparm -eb -F1 hello.abs

---------- HLL symbol table ----------

hldumparm -ef -F1 hello.abs

---------- HLL symbol table ----------
hldumparm -eg -F1 hello.abs

--------- HLL symbol table ---------

<table>
<thead>
<tr>
<th>Address</th>
<th>Size</th>
<th>Type</th>
<th>Name</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>00040000</td>
<td>24</td>
<td>struct</td>
<td>_dbg_request</td>
<td>dbg.c</td>
</tr>
<tr>
<td>00040000</td>
<td>4</td>
<td>int</td>
<td>_errno</td>
<td></td>
</tr>
<tr>
<td>00040004</td>
<td>4</td>
<td>enum</td>
<td>nr</td>
<td></td>
</tr>
<tr>
<td>00040008</td>
<td>16</td>
<td>union</td>
<td>u</td>
<td></td>
</tr>
<tr>
<td>00040008</td>
<td>4</td>
<td>struct</td>
<td>exit</td>
<td></td>
</tr>
<tr>
<td>00040008</td>
<td>4</td>
<td>int</td>
<td>status</td>
<td></td>
</tr>
<tr>
<td>0004000c</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00040008</td>
<td>8</td>
<td>struct</td>
<td>open</td>
<td></td>
</tr>
<tr>
<td>00040008</td>
<td>4</td>
<td>const char</td>
<td>* pathname</td>
<td></td>
</tr>
<tr>
<td>0004000c</td>
<td>2</td>
<td>unsigned short int</td>
<td>flags</td>
<td></td>
</tr>
<tr>
<td>0004000e</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00040010</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00040018</td>
<td>80</td>
<td>static char</td>
<td>stdin_buf[80]</td>
<td>_iob.c</td>
</tr>
</tbody>
</table>

hldumparm -en -F1 hello.abs

--------- HLL symbol table ---------

<table>
<thead>
<tr>
<th>Address</th>
<th>Size</th>
<th>Type</th>
<th>Name</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>00040000</td>
<td>24</td>
<td>struct</td>
<td>_dbg_request</td>
<td>dbg.c</td>
</tr>
<tr>
<td>00040000</td>
<td>4</td>
<td>int</td>
<td>_errno</td>
<td></td>
</tr>
<tr>
<td>00040004</td>
<td>4</td>
<td>enum</td>
<td>nr</td>
<td></td>
</tr>
<tr>
<td>00040008</td>
<td>16</td>
<td>union</td>
<td>u</td>
<td></td>
</tr>
<tr>
<td>00040008</td>
<td>4</td>
<td>struct</td>
<td>exit</td>
<td></td>
</tr>
<tr>
<td>00040008</td>
<td>4</td>
<td>int</td>
<td>status</td>
<td></td>
</tr>
<tr>
<td>0004000c</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00040008</td>
<td>8</td>
<td>struct</td>
<td>open</td>
<td></td>
</tr>
<tr>
<td>00040008</td>
<td>4</td>
<td>const char</td>
<td>* pathname</td>
<td></td>
</tr>
<tr>
<td>0004000c</td>
<td>2</td>
<td>unsigned short int</td>
<td>flags</td>
<td></td>
</tr>
<tr>
<td>0004000e</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00040010</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00040018</td>
<td>80</td>
<td>static char</td>
<td>stdin_buf[80]</td>
<td>_iob.c</td>
</tr>
</tbody>
</table>

**Related information**

Section 9.5.2, *HLL Dump Output Format*
**HLL object dumper option: --help (-?)**

**Command line syntax**

--help

-?

**Description**

Displays an overview of all command line options.

**Example**

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

```bash
hldumparm -?
hldumparm --help
hldumparm
```

**Related information**

-
HLL object dumper option: --hex (-x)

Command line syntax

--hex

-x

Description

With this option you can control the way data sections and code sections are dumped. By default, the contents of data sections are represented by directives. A new directive will be generated for each symbol. ELF labels in the section are used to determine the start of a directive. ROM sections are represented with .db, .dh, .dw, .dd kind of directives, depending on the size of the data. RAM sections are represented with .ds directives, with a size operand depending on the data size. This can be either the size specified in the ELF symbol, or the size up to the next label. Code sections are dumped as disassembly.

With option --hex, no directives will be generated for ROM data sections and no disassembly dump will be done for code sections. Instead ROM data sections and code sections are dumped as hexadecimal code with ASCII translation. RAM sections will be represented with only a start address and a size indicator.

Example

hldumparm -F2 --section=.rodata hello.abs

---------- Section dump ----------

.section .data, '.rodata', at(0x0000018c)
.db 77,6f,72,6c,64,00 ; world.
.endsec

.section .data, '.rodata', at(0x00000194)
.db 48,65,6c,6c,6f,20,25,73,21,0a,00 ; Hello %s!..
.endsec

hldumparm -F2 --section=.rodata --hex hello.abs

---------- Section dump ----------

section 5 (.rodata):
000018c 77 6f 72 6c 64 00 world.

section 7 (.rodata):
0000194 48 65 6c 6c 6f 20 25 73 21 0a 00 Hello %s!..

Related information

Section 9.5.2, HLL Dump Output Format
HLL object dumper 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 HLL object dumper.

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:

--symbols=hll
--class=code
hello.abs
Specify the option file to the HLL object dumper:

```
hldumparm --option-file=myoptions
```

This is equivalent to the following command line:

```
hldumparm --symbols=hll --class=code hello.abs
```

**Related information**
HLL object dumper option: --output (-o)

Command line syntax

--output=file

-o file

Description

By default, the HLL object dumper dumps the output on stdout. With this option you specify to dump the information in the specified file.

The default output format is text, but you can specify another output format with option --output-type.

Example

hldumparm --output=dump.txt hello.abs

The HLL object dumper dumps the output in file dump.txt.

Related information

HLL object dumper option --output-type
HLL object dumper option: --output-type (-T)

Command line syntax

```
--output-type [=type]
-T [type]
```

You can specify one of the following types:

- **text**: t  Output human readable text.
- **xml**: x  Output XML.

Default: `--output-type=text`

Description

With this option you can specify whether the output is formatted as plain text or as XML.

Related information

HLL object dumper option `--output`
HLL object dumper option: --print-mangled-symbols (-P)

Command line syntax

--print-mangled-symbols

-P

Description

The C++ compiler can generate unreadable symbol names. These symbols cannot easily be related to your C++ source file anymore. Therefore the HLL dumper by default demangles C++ function names and variable names in the HLL symbol table. With this option you can override this default setting and print the mangled names instead.

Example

hldumparm hellocpp.abs

---------- HLL symbol table ----------

Address    Size HLL Type             Name
00001254     20 void                 __register_finalization_routine()

hldumparm --print-mangled-symbols hellocpp.abs

---------- HLL symbol table ----------

Address    Size HLL Type             Name
00001254     20 void                 __Z31__register_finalization_routinev()
HLL object dumper option: --sections (-s)

Command line syntax

--sections=\texttt{name},... 
-s\texttt{name},...

Description

With this option you can restrict the output to the specified sections only. This option affects the following parts of the output:

<table>
<thead>
<tr>
<th>Output part</th>
<th>Effect of --sections</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module list</td>
<td>Not restricted</td>
</tr>
<tr>
<td>Section list</td>
<td>Only lists the specified sections</td>
</tr>
<tr>
<td>Section dump</td>
<td>Only dumps the contents of the specified sections</td>
</tr>
<tr>
<td>HLL symbol table</td>
<td>Not restricted</td>
</tr>
<tr>
<td>Assembly level symbol table</td>
<td>Only lists symbols defined in the specified sections</td>
</tr>
<tr>
<td>Note sections</td>
<td>Not restricted</td>
</tr>
</tbody>
</table>

By default all sections are included.

Related information

Section 9.5.2, HLL Dump Output Format
HLL object dumper option: --source-lookup-path (-L)

Command line syntax

--source-lookup-path=\path
-L\path

Description

With this option you can specify an additional path where your source files are located. If you want to specify multiple paths, use the option --source-lookup-path for each separate path.

The order in which the HLL object dumper will search for source files when intermixed disassembly is used, is:

1. The path obtained from the HLL debug information.
2. The path that is specified with the option --source-lookup-path. If multiple paths are specified, the paths will be searched for in the order in which they are given on the command line.

Example

Suppose you call the HLL object dumper as follows:

hldumparm --disassembly-intermix --source-lookup-path=c:\mylib\src hello.abs

First the HLL object dumper looks in the directory found in the HLL debug information of file hello.abs for the location of the source file(s). If it does not find the file(s), it looks in the directory c:\mylib\src.

Related information

HLL object dumper option --disassembly-intermix
HLL object dumper option: --symbols (-S)

Command line syntax

--symbols [=type]
-S[type]

You can specify one of the following types:

<table>
<thead>
<tr>
<th>Type</th>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>asm</td>
<td>a</td>
<td>Display assembly symbols in code dump.</td>
</tr>
<tr>
<td>hll</td>
<td>h</td>
<td>Display HLL symbols in code dump.</td>
</tr>
<tr>
<td>none</td>
<td>n</td>
<td>Display plain addresses in code dump.</td>
</tr>
</tbody>
</table>

Default: --symbols=asm

Description

With this option you can control symbolic information in the disassembly and data dump. For data sections this only applies to symbols used as labels at the data addresses. Data within the data sections will never be replaced with symbols.

Only symbols that are available in the ELF or DWARF information are used. If you build an application without HLL debug information the --symbols=hll option will result in the same output as with --symbols=none. The same applies to the --symbols=asm option when all symbols are stripped from the ELF file.

Example

hldumparm -F2 hello.abs

---------- Section dump ----------

000001fc fe e7       _Exit:           b       _Exit
.section .text.libc, at(0x000001fc)
.endsec

hldumparm --symbols=none -F2 hello.abs

---------- Section dump ----------

000001fc fe e7      
.section .text.libc, at(0x000001fc)
.b       0x1fc
.endsec

Related information

Section 9.5.2, HLL Dump Output Format
HLL object dumper option: --version (-V)

Command line syntax

--version
-v

Description

Display version information. The HLL object dumper ignores all other options or input files.

Related information

-
HLL object dumper option: --xml-base-filename (-X)

Command line syntax

```
--xml-base-filename
-X
```

Description

With this option the `<File name>` field in the XML output only contains the filename of the object file. By default, any path name, if present, is printed as well.

Example

```
hldumparm --output-type=xml --output=hello.xml ../hello.abs
```

The field `<File name="../hello.abs">` is used in `hello.xml`.

```
hldumparm --output-type=xml --output=hello.xml -X ../hello.abs
```

The field `<File name="hello.abs">` is used in `hello.xml`. The path is stripped from the filename.

Related information

HLL object dumper option --output-type
11.10. Expire Cache Utility Options

With the utility expirearm you can limit the size of the cache (C compiler option --cache) by removing all files older than a few days or by removing older files until the total size of the cache is smaller than a specified size. See also Section 12.5, Compiler Cache.

The invocation syntax is:

```
expirearm [option]... cache-directory
```

The compiler cache is present in the directory carmcache under the specified cache-directory.

This section describes all options for the expire cache utility.

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.
Expire cache utility option: --access (-a)

Command line syntax

--access

-a

Description

Use the last access time instead of the last modification time to determine which files to delete.

Example

expirearm --access --days=7 "installation-dir\mproject\.cache"

Related information
Expire cache utility option: --days (-d)

Menu entry
1. Select C/C++ Compiler » Optimization » Compilation Speed.
2. Enable the option Cache generated code to improve the compilation speed.
3. In the Directory for cached files field, enter the name for the location of the cache.
   By default this is the .cache directory under your project directory.
4. Specify the Maximum days files will live in the cache.

Command line syntax
--days=n
-dn

Description
Remove all files older than n days from the cache.

Example
To remove all files older than seven days, enter:
expirearm --days=7 "installation-dir\mproject\.cache"

Related information
**Expire cache utility option: --diag**

**Command line syntax**

```
--diag=[format:]\{all | msg[-msg],...\}
```

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, or you can specify a range.

With this option the expire cache utility does not remove any files.

**Example**

To display an explanation of message number 204, enter:

```
expirearm --diag=204
```

This results in the following message and explanation:

```
E204: failed to remove "<file>" <<cause>>
```

The removal of the indicated file failed. The `<cause>` provides more details of the problem.

To write an explanation of all errors and warnings in HTML format to file `expirearm_errors.html`, use redirection and enter:

```
expirearm --diag=html:all > expirearm_errors.html
```

**Related information**

-
Expire cache utility option: --dry-run (-n)

Command line syntax

--dry-run

-n

Description

With this option you put the expire utility in verbose mode. The utility shows which files would be deleted, without actually removing them.

Related information

Expire cache utility option --verbose (Verbose output)
Expire cache utility option: --help (−?)

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:

expirearm −?
expirearm --help
expirearm

To see a detailed description of the available options, enter:

expirearm --help=options

Related information

-
Expire cache utility option: --megabytes (-m)

**Menu entry**

1. Select **C/C++ Compiler » Optimization » Compilation Speed**.
2. Enable the option **Cache generated code to improve the compilation speed**.
3. In the **Directory for cached files** field, enter the name for the location of the cache.
   
   *By default this is the .cache directory under your project directory.*
4. Enable the option **Clear cache upon project clean**.
   
   *Each time you use Project » Clean... the cache is cleared.*

**Command line syntax**

```
--megabytes=m

-m m
```

**Description**

Reduce the size of the cache to $m$ MBytes by removing files from the cache, starting with the oldest file. With a size of 0 (zero) you clear the entire cache.

**Example**

To reduce the compiler cache size to 4 MB, enter:

```
expirearm --megabytes=4 "installation-dir\mproject\.cache"
```

Older files are removed until the total size of the cache is smaller than 4 MB.

To clear the compiler cache, enter:

```
expirearm --megabytes=0 "installation-dir\mproject\.cache"
```

**Related information**

-
Expire cache utility option: --totals (-t)

Command line syntax

--totals
-t

Description

Show the total size of the cache and the number of directories and files. This option is implicit when invoked without the --days and --megabytes options.

Example

expirearm -t "installation-dir\mproject\.cache"

installation-dir\mproject\.cache\carmcache:
1 MB, 3 directories, 3 files

Related information
Expire cache utility option: --verbose (-v)

Command line syntax

--verbose

-v

Description

With this option you put the expire cache utility in verbose mode. The utility shows which files are being deleted.

Example

expirearm -v --megabytes=0 "installation-dir\mproject\.cache"

2014-06-24 12:36:15 installation-dir\mproject\.cache\carmcache\cstart\30aa7935
2014-06-24 12:36:17 installation-dir\mproject\.cache\carmcache\myproject\6f0a3ba4

Related information

-
Expire cache utility option: --version (-V)

Command line syntax

--version

-v

Description

Display version information and exit. The expire cache utility ignores all other options.

Related information

-
Chapter 12. Influencing the Build Time

In general many settings have influence on the build time of a project. Any change in the tool settings of your project source will have more or less impact on the build time. The following sections describe several issues that can have significant influence on the build time.

12.1. MIL Linking

With MIL linking (see Section 4.5.1, *Generic Optimizations (frontend)*) it is possible to let the compiler apply optimizations application wide. This can yield significant optimization improvements, but the build times can also be significantly longer. MIL linking itself can require significant time, but also the changed build process implies longer build times. The MIL linking settings in Eclipse are:

- **Build for application wide optimizations (MIL linking)**

  This enables MIL linking. The build process changes: the C files are translated to intermediate code (MIL files) and the generated MIL files of the whole project are linked together by the C compiler. The next step depends on the setting of the option below.

- **Application wide optimization mode: Optimize more/Build slower**

  When this option is enabled, the compiler runs the code generator immediately on the completely linked MIL stream, which represents the entire application. This way the code generator can perform several optimizations, such as "code compaction", at application scope. But this also requires significantly more memory and requires more time to generate code. Besides that, it is no longer possible to do incremental builds. With each build the full MIL linking phase and code generation has to be done, even with the smallest change that would in a normal build (not MIL linking) require only a single module to be translated.

- **Application wide optimization mode: Optimize less/Build faster**

  When this option is disabled, the compiler splits the MIL stream after MIL linking in separate modules. This allows the code generation to be performed for the modified modules only, and will therefore be faster than with the other option enabled. Although the MIL stream is split in separate modules after MIL linking, it still may happen that modifying a single C source file results in multiple MIL files to be compiled. This is a natural result of global optimizations, where the code generated for multiple modules was affected by the change.

In general, if you do not need code compaction, for example because you are optimizing fully for speed, it is recommended to choose *Optimize less/Build faster*.

12.2. Optimization Options

In general any optimization may require more work to be done by the compiler. But this does not mean that disabling all optimizations (level 0) gives the fastest compilation time. Disabling optimizations may result in more code being generated, resulting in more work for other parts of the compiler, like for example the register allocator.
12.3. Automatic Inlining

Automatic inlining is an optimization which can result in significant longer build time. The overall functions will get bigger, often making it possible to do more optimizations. But also often resulting in more registers to be in use in a function, giving the register allocation a tougher job.

12.4. Code Compaction

When you disable the code compaction optimization, the build times may be shorter. Certainly when MIL linking is used where the full application is passed as a single MIL stream to the code generation. Code compaction is however an optimization which can make a huge difference when optimizing for code size. When size matters it makes no sense to disable this option. When you choose to optimize for speed (--tradeoff=0) the code compaction is automatically disabled.

12.5. Compiler Cache

The C compiler has support for caching intermediate results to avoid full compilations. When the source code after preprocessing and relevant compiler options and the compiler version are the same as in a previous invocation, the previous result is copied to the output file. The cache only works when there is a single C input file and a single output file (no --mil-split).

To enable caching from Eclipse:

1. From the Project menu, select Properties for

   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 » Compilation Speed.

4. Enable the option Cache generated code to improve the compilation speed.

5. In the Directory for cached files field, enter the name for the location of the cache.

   By default this is the .cache directory under your project directory.

6. Specify the Maximum days files will live in the cache.

7. (Optional) Enable the option Clear cache upon project clean.

   Each time you use Project » Clean... the cache is cleared.

Eclipse calls the C compiler with option --cache. The cache directory may be shared, for instance by placing it on a network drive. The compiler creates a directory carmcache in the specified directory.

When a result from the cache is used, the C compiler generates a comment line in the assembly source file to notify that. In that case be aware of the following:
In case source merging is enabled an older version of the source is still shown. As long as a source change has no effect on the preprocessed code, the cached version of the output file is used.

Some options, like `--define`, `--include-directory` and `--output` are not part of the hash used for the cache. As long as a change in these options has no influence on the preprocessed code, the cached version of the output is used. This means that the options listed as comments in the generated assembly file might not match the options actually used.

With every compilation of a file that results in a cache miss, a new file is stored in the cache. Old files are not removed from the cache automatically because that would slow down the compiler too much. To keep the cache size reasonable specify a maximum number of days the files will live in the cache. Eclipse uses the utility `expirearm` for this. It is recommended to run this utility frequently, for example with each time the project is linked. For more information on this utility see Section 9.6, *Expire Cache Utility*.

### 12.6. Header Files

Many applications include all header files in each module, often by including them all within a single include file. Processing header files takes time. It is a good programming practice to only include the header files that are really required in a module, because:

- it is clear what interfaces are used by a module
- an incremental build after modifying a header file results in less modules required to be rebuild
- it reduces compile time

### 12.7. Parallel Build

The make utility `amk`, which is used by Eclipse, has a feature to build jobs in parallel. This means that multiple modules can be compiled in parallel. With today's multi-core processors this means that each core can be fully utilized. In practice even on single core machines the compile time decreases when using parallel jobs. On multi-core machines the build time even improves further when specifying more parallel jobs than the number of cores.

In Eclipse you can control the parallel build behavior:

1. From the **Project** menu, select **Properties for**

   *The Properties dialog appears.*

2. In the left pane, select **C/C++ Build**.

   *In the right pane the C/C++ Build page appears.*

3. On the Behavior tab, select **Enable parallel build**.

4. You can specify the number of parallel jobs, or you can use an optimal number of jobs. In the last case, **amk** will fork as many jobs in parallel as cores are available.
12.8. Number of Sections

The linker speed depends on the number of sections in the object files. The more sections, the longer the locating will take. You can decrease the link time by creating output sections in the LSL file. For example:

Use compiler option \texttt{--rename-sections=.text={name}}

\begin{verbatim}
section_layout ::linear
{
    group (ordered)
    {
        section "code_output1" ( size = 64k, attributes = x, fill=0xFF,
        overflow = "code_output2")
        {
            select "*__cocofun*";
        }
    }
}
\end{verbatim}
Chapter 13. 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.

13.1. 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.

13.1.1. Methods of Profiling

There are several methods of profiling: recording by an instruction set simulator, 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 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.

13.2. 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 dynamic profiling 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.

When you use dynamic profiling on a target board, function `clock()` must be added to the application to get profiling timer results.

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++ » TASKING ARM 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.*

**13.2.1. 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 for**  
   *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/Function 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-info**) does not affect profiling, execution time or code size.*

**Block counters** (not in combination with Call graph or Function timers)

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/Function 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.

### 13.2.1.1. 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.

### 13.2.1.2. 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 7.3, **Linking with Libraries** for an overview of the (profiling) libraries.

### 13.2.2. 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.

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 **Debug** menu, select **Debug Configurations**...

   The **Debug Configurations** dialog appears.

2. Select **TASKING C/C++ Debugger** and click the **New launch configuration** button to add a new configuration.

3. In the **Name** field enter the name of the configuration. By default, this is the name of the project, but you can give your configuration any name you want to distinguish it from the project name. For example enter **profiling.simulator** to identify the simulator debug configuration.

4. On the **Target** tab, select the **ARM Simulator**.
5. Click the **Apply** button.

6. 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.

7. Click **Yes** to open the TASKING Debug perspective.

   The TASKING Debug perspective opens while the application has stopped before it enters `main()`

8. From the main toolbar, 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. From the **Project** menu, select **Properties for**

   The Properties dialog appears.

2. In the left pane, expand **C/C++ Build** and select **Stack/Heap**.

   In the right pane the Stack/Heap property page appears.

3. Enter larger values for **heap** and click **OK**.

   The project LSL file is updated automatically according to the new settings.
Presumable incorrect call graph

The call graph is based on the compiled source code. Due to compiler optimizations the call graph may therefor 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.

13.2.3. 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 Open 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:

<table>
<thead>
<tr>
<th>Module</th>
<th>The C source module in which the function resides.</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Line</td>
<td>The line number of the function definition in the C source module.</td>
</tr>
<tr>
<td>Function</td>
<td>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 (▼) and select Show Block Counts.</td>
</tr>
<tr>
<td>Total Time</td>
<td>The total amount of time in seconds that was spent in this function and all of its sub-functions.</td>
</tr>
<tr>
<td>Self Time</td>
<td>The amount of time in seconds that was spent in the function itself. This excludes the time spent in the sub-functions. So, self time = function's total time - total times of the called functions.</td>
</tr>
<tr>
<td>% in Function</td>
<td>This is the relative amount of time spent in this function, calculated as a percentage of the total application time. These should add up to 100%. The total application time is determined by taking the total time of the call graph. This is usually main. Example:</td>
</tr>
<tr>
<td></td>
<td>Total time of main: 0.002000</td>
</tr>
<tr>
<td></td>
<td>Self time of function foo: 0.000100</td>
</tr>
<tr>
<td></td>
<td>% in Function = (0.000100 / 0.002000) * 100 = 5%</td>
</tr>
<tr>
<td>Calls</td>
<td>Number of times the function has been executed.</td>
</tr>
<tr>
<td>#Callers</td>
<td>Number of functions by which the function was called.</td>
</tr>
<tr>
<td>#Callees</td>
<td>Number of functions that was actually called from this function.</td>
</tr>
<tr>
<td>Contribution %</td>
<td>In the caller table: shows for which part (in percent) the caller contributes to the time spent in the focus function.</td>
</tr>
<tr>
<td></td>
<td>In the callee table: shows how much time the focus function has spent relatively in each of its callees.</td>
</tr>
<tr>
<td>Calls %</td>
<td>In the caller table: shows how often each callee was called as a percentage of all calls from the focus function.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
</tbody>
</table>

Common 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>Select Profiling File(s)</td>
<td>Opens a dialog where you can specify profiling files for display.</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 Profiling Type group box, select Dynamic Profiling.
3. In the Profiling Files group box, disable the option Use default.
4. Click the Add... button, select the `.prf` files you want to load and click Open to confirm your choice.
5. Make sure the correct symbol file is selected, in this example `profiling.abs`.
6. Click OK to finish.

13.3. 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 13.2, Profiling using Code Instrumentation (Dynamic Profiling).

13.3.1. 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 for
   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 (panse).

### 13.3.2. 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 Open Perspective dialog appears.

2. Select the TASKING Profiler perspective and click OK.

   The TASKING Profiler perspective opens. This perspective is explained in Section 13.2.3, 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 Profiling Type group box, select Static Profiling.

3. 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).

4. Click OK to finish.
Chapter 14. 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 C11) and some functions of the floating-point library.

A number of standard operations within C are too complex to generate inline code for (too much code). These operations are implemented as run-time library functions to save code.

Section 14.2, Library Functions, gives an overview of all library functions you can use, grouped per header file. A number of functions declared in wchar.h are parallel to functions in other header files. These are discussed together.

Section 14.3, C Library Reentrancy, gives an overview of which functions are reentrant and which are not.

C library / floating-point library / run-time library

The following libraries are included in the ARM toolset. Both Eclipse and the control program ccarm automatically select the appropriate libraries depending on the specified options.

<table>
<thead>
<tr>
<th>Libraries</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>carm[s].lib</td>
<td>C libraries for ARM and Thumb instructions respectively</td>
</tr>
<tr>
<td>cthumb[s].lib</td>
<td>Optional letter:</td>
</tr>
<tr>
<td></td>
<td>s = single precision floating-point (control program option --fp-model=+float)</td>
</tr>
<tr>
<td>fmapc.lib</td>
<td>Floating-point run-time libraries for ARM and Thumb (contains floating-point</td>
</tr>
<tr>
<td>fpthumb.lib</td>
<td>functions needed by the C compiler)</td>
</tr>
<tr>
<td>rtarm.lib</td>
<td>Run-time library for ARM and Thumb (contains other run-time functions</td>
</tr>
<tr>
<td>rtthumb.lib</td>
<td>needed by the C compiler)</td>
</tr>
<tr>
<td>pbarm.lib / pbthumb.lib</td>
<td>Profiling libraries for ARM and Thumb</td>
</tr>
<tr>
<td>pcarm.lib / pcthumb.lib</td>
<td>pb = block/function counter</td>
</tr>
<tr>
<td>pctarm.lib / pctthumb.lib</td>
<td>pc = call graph</td>
</tr>
<tr>
<td>pdarm.lib / pdthumb.lib</td>
<td>pct = call graph and timing</td>
</tr>
<tr>
<td>ptarm.lib / ptthumb.lib</td>
<td>pd = dummy</td>
</tr>
<tr>
<td></td>
<td>pt = function timing</td>
</tr>
<tr>
<td>dspthumb[s].lib</td>
<td>CMSIS DSP libraries</td>
</tr>
<tr>
<td></td>
<td>Optional letter:</td>
</tr>
<tr>
<td></td>
<td>s = single precision floating-point</td>
</tr>
</tbody>
</table>

For the C libraries, profiling libraries and CMIS DPS libraries, also MIL library variants are present (files with extension .ma).

Sources for the libraries are present in the directories lib\src, lib\src.* in the form of an executable. If you run the executable it will extract the sources in the corresponding directory.
C++ library

The TASKING C++ compiler supports the STLport C++ libraries and the libc++ libraries. STLport and libc++ are multi-platform ISO C++ Standard Library implementations. They are free, open-source products, which are delivered with the TASKING C++ compiler. The STLport library supports standard templates and I/O streams as defined by the C++03 standard. The libc++ library supports standard templates, STL features and I/O streams as defined by the C++11 standard and later.

The STLport libraries have been built with C++ compiler option --c++=03. The libc++ libraries have been built with C++ compiler option --c++=14.

I/O streams are disabled by default for C++03. So, when you use standard I/O streams, e.g. cout, or STL templates in your C++03 source make sure to specify C++ compiler option --io-streams.

The include files for the STLport C++ libraries are present in directory include.stl relative to the product installation directory.

The include files for the libc++ libraries are present in directory include.cxx 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.


You can find more information on the libc++ library on the following site: http://libcxx.llvm.org/

The following C++ libraries are delivered with the product:

<table>
<thead>
<tr>
<th>Libraries</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cparm*[s][x].lib</td>
<td>C++ libraries for ARM and Thumb</td>
</tr>
<tr>
<td>cpthumb*[s][x].lib</td>
<td>Optional letter: s = single precision floating-point</td>
</tr>
<tr>
<td></td>
<td>x = exception handling</td>
</tr>
<tr>
<td>stlarmx.lib</td>
<td>STLport C++ libraries (exception handling variants only)</td>
</tr>
<tr>
<td>stltthumbx.lib</td>
<td></td>
</tr>
<tr>
<td>cxxarm*[s].x.lib</td>
<td>libc++ libraries (exception handling variants only)</td>
</tr>
<tr>
<td>cxxthumxb*[s].x.lib</td>
<td>Optional letter: s = single precision floating-point</td>
</tr>
</tbody>
</table>

To rebuild a library

1. Change to the library source directory, for example installdir\lib\src.

2. Run the program unpack-*.exe with Administrator rights. This will extract the library sources.
3. Change to the directory of the library set used by your project. For example, v7M\le\cthumb\lib. The lib subdirectory is for the normal library, the ma subdirectory is for the MIL archive.

4. Run the makefile by executing installdir\bin\amk.exe without arguments.

5. Copy the generated library, in this example cthumb.lib, to the corresponding directory installdir\lib\v7M\le.

Meaning of the source subdirectories:

- [v6M] libraries for ARMv6-M architectures
- [v7EM] libraries for ARMv7E-M architectures
- [v7M] libraries for ARMv7-M architectures
- [le] little-endian library variant
- [be] big-endian library variant
14.1. Using the CMSIS DSP Library

Part of the CMSIS standard is a DSP library. The CMSIS DSP library is included in the TASKING product's cmsis folder. The library is also available as a pre-built library file in the lib folders for v6M, v7M and v7EM. The libraries are dspthumb.lib (double precision floating point) and dspthumbs.lib (single precision floating point). MIL libraries are present also: dspthumb.ma and dspthumbs.ma.

To use the CMSIS DSP library in your Eclipse project

1. From the Project menu, select Properties for

   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 » Include Paths .

4. Enable the option Add CMSIS include paths.

5. On the Tool Settings tab, select Linker » Libraries.

6. Enable the option Link CMSIS DSP library.

   This passes the option --dsp-library of the control program (ccarm) for compilation of C/C++ files and for linking. With this option the control program sets the compiler macro ARM_MATH_CM0, ARM_MATH_CM0PLUS, ARM_MATH_CM3 or ARM_MATH_CM4 or ARM_MATH_CM7, depending on the selected processor. These macros are required for the CMSIS arm_math.h header file to operate correctly. With --dsp-library the control program also selects the appropriate library.

14.2. Library Functions

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

Wide character functionality is supported in the C library.

14.2.1. assert.h

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

For C11 or later only, the following macro is defined:

#define static_assert _Static_assert
14.2.2. complex.h

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

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

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

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

This implementation uses the obvious implementation for complex multiplication; and a more sophisticated implementation for division and absolute value calculations which handles underflow, overflow and infinities with more care. The ISO 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**

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

### 14.2.3. 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><strong>ctype.h</strong></th>
<th><strong>wctype.h</strong></th>
<th><strong>Description</strong></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;)</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>isxdigit</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><code>tolower</code></td>
<td><code>towlower</code></td>
<td>Returns (c) converted to a lowercase character if it is an uppercase character, otherwise (c) is returned.</td>
</tr>
<tr>
<td><code>toupper</code></td>
<td><code>towupper</code></td>
<td>Returns (c) converted to an uppercase character if it is a lowercase character, otherwise (c) is returned.</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td><code>_tolower</code></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>
<td></td>
</tr>
<tr>
<td><code>_toupper</code></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>
<td></td>
</tr>
<tr>
<td><code>isascii</code></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>
<td></td>
</tr>
<tr>
<td><code>toascii</code></td>
<td>Converts c to an ASCII value (strip highest bit). This function is not defined in ISO C99.</td>
<td></td>
</tr>
</tbody>
</table>

### 14.2.4. 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 C.

- `_dbg_trap` Low level function to trap debug events
- `_argcv(const char *buf, size_t size)` Low level function for command line argument passing

### 14.2.5. errno.h

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EPERM</td>
<td>Operation not permitted</td>
</tr>
<tr>
<td>ENOENT</td>
<td>No such file or directory</td>
</tr>
<tr>
<td>EINTR</td>
<td>Interrupted system call</td>
</tr>
<tr>
<td>EIO</td>
<td>I/O error</td>
</tr>
<tr>
<td>EBADF</td>
<td>Bad file number</td>
</tr>
<tr>
<td>EAGAIN</td>
<td>No more processes</td>
</tr>
<tr>
<td>ENOMEM</td>
<td>Not enough core</td>
</tr>
<tr>
<td>EACCES</td>
<td>Permission denied</td>
</tr>
<tr>
<td>EFAULT</td>
<td>Bad address</td>
</tr>
<tr>
<td>EEXIST</td>
<td>File exists</td>
</tr>
<tr>
<td>ENOTDIR</td>
<td>Not a directory</td>
</tr>
<tr>
<td>EISDIR</td>
<td>Is a directory</td>
</tr>
<tr>
<td>EINVAL</td>
<td>Invalid argument</td>
</tr>
<tr>
<td>ENFILE</td>
<td>File table overflow</td>
</tr>
<tr>
<td>EMFILE</td>
<td>Too many open files</td>
</tr>
<tr>
<td>ETXTBSY</td>
<td>Text file busy</td>
</tr>
<tr>
<td>ENOSPC</td>
<td>No space left on device</td>
</tr>
<tr>
<td>EPIPE</td>
<td>Broken pipe</td>
</tr>
<tr>
<td>EROFS</td>
<td>Read-only file system</td>
</tr>
<tr>
<td>EROFS</td>
<td>Illegal seek</td>
</tr>
</tbody>
</table>

The following error codes are defined as macros in `errno.h`:
The header file `except.h` contains the ARM specific software floating-point exception handling interface definition. This header file is not defined in ISO C.

```
_fp_install_trap_handler( void (*)( _fp_exception_info_t * ) exception )
```

Installs a floating-point trap handler.

```
_fp_get_exception_mask( void )
```

Returns the exception mask.

```
_fp_set_exception_mask( int )
```

Sets the exception mask. A value of 0xFF traps all floating-point exceptions.

```
_fp_get_exception_status( void )
```

Returns the exception status.

```
_fp_set_exception_status( int )name)
```

Sets the exception status.

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

- **EFINVOP** Invalid operation, for instance 0.0/0.0, 0.0*INF or on a NaN
- **EFDIVZ** Division by zero, for instance 12.5/0.0 (not 0.0/0.0)
- **EFOVFL** Overflow, when the result of an operation is too large
- **EFUNFL** Underflow, when the result of an operation is too small
EFINEXCT   Inexact
EFALL      Combination of all of the above enlisted exceptions

INF means infinite which is the largest absolute floating-point number.
NAN means Not A Number, this is a special notation for an undefined floating-point number.

14.2.7. 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 C.

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

14.2.8. fenv.h

Contains mechanisms to control the floating-point environment.

fegetenv  Stores the current floating-point environment.
feholdexcept    Saves the current floating-point environment and installs an environment that ignores all floating-point exceptions.

fesetenv  Stores the current setting of the floating-point status flags.
feclearexcept  Clears the current exception status flags corresponding to the flags specified in the argument.
fegetexceptflag  Stores the current setting of the floating-point status flags.
feraiseexcept   Raises the exceptions represented in the argument. As a result, other exceptions may be raised as well.

fesetexceptflag  Sets the current floating-point status flags.

fetestexcept   Returns the bitwise-OR of the exception macros corresponding to the exception flags which are currently set and are specified in the argument.

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.

fesetround   Sets the current rounding directions.

For each supported rounding mode, a macro is defined. The following rounding mode macro is defined:

FE_TONEAREST
14.2.9. float.h

The header file float.h defines the characteristics of the real floating-point types float, _Float16, double and long double.

float.h used to contain prototypes for the functions copysign(f), isinf(f), isfinite(f), isnan(f) and scalbf(f). These functions have accordingly to the ISO C99 standard been moved to the header file math.h. See also Section 14.2.16, 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.
- isnff(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).
- isnan(double d) Test the variable d on being NaN (Not a Number, IEEE-754).
- scalbf(float f, int p) Returns f * 2^p for integral values without computing 2^N.
- scalb(double d, int p) Returns d * 2^p for integral values without computing 2^N. (See also scalbn in Section 14.2.16, math.h and tgmath.h)

14.2.10. 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.

- imaxabs(intmax_t j) Returns the absolute value of j
- imaxdiv(intmax_t numer, intmax_t denom) Computes numer/denom and numer % denom. The result is stored in the quot and rem components of the imaxdiv_t structure type.
- strtoimax(const char * restrict nptr, char ** restrict endptr, int base) Convert string to maximum sized integer. (Compare strtoll)
- strtoumax(const char * restrict nptr, char ** restrict endptr, int base) Convert string to maximum sized unsigned integer. (Compare strtoull)
The header file `io.h` contains prototypes for low level I/O functions. This header file is not defined in ISO C.

- `_close(fd)`
  Used by the functions `close` and `fclose`. *(FSS implementation)*
- `_lseek(fd, offset, whence)`
  Used by all file positioning functions: `fgetpos`, `fseek`, `fsetpos`, `ftell`, `rewind`. *(FSS implementation)*
- `_open(fd, flags)`
  Used by the functions `fopen` and `freopen`. *(FSS implementation)*
- `_read(fd, *buff, cnt)`
  Reads a sequence of characters from a file. *(FSS implementation)*
- `_unlink(*name)`
  Used by the function `remove`. *(FSS implementation)*
- `_write(fd, *buffer, cnt)`
  Writes a sequence of characters to a file. *(FSS implementation)*

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

```c
#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  ^=
```

The header file `limits.h` contains the sizes of integral types, defined as macros.

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

```c
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>Code</th>
<th>Category</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>LC_ALL</td>
<td>0</td>
<td>LC_NUMERIC</td>
<td>3</td>
</tr>
<tr>
<td>LC_COLLATE</td>
<td>1</td>
<td>LC_TIME</td>
<td>4</td>
</tr>
<tr>
<td>LC_CTYPE</td>
<td>2</td>
<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.

14.2.15. 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 14.2.26, `stdlib.h` and `wchar.h`.

`malloc(size)`

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

`aligned_alloc(alignment, size)`

(C11 only) Allocates space for an object whose alignment is specified by `alignment` and 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`, while preserving its contents. If the new size is smaller than the old size, some contents at the end of the old region will be discarded. If the new size is larger than the old size, all of the old contents are preserved and any bytes in the new object beyond the size of the old object will have indeterminate values.

14.2.16. math.h and tgmath.h

The header file `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 `double`, `float` and `long double`. They are respectively named `function`, `functionf`, `functionl`. All `long` type functions, though
declared in `math.h`, are implemented as the `double` type variant which nearly always meets the requirement in embedded applications.

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

### Trigonometric and hyperbolic functions

<table>
<thead>
<tr>
<th><code>math.h</code></th>
<th><code>tgmath.h</code></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sin</td>
<td>sinf</td>
<td><code>sin</code></td>
</tr>
<tr>
<td>cos</td>
<td>cosf</td>
<td><code>cos</code></td>
</tr>
<tr>
<td>tan</td>
<td>tanf</td>
<td><code>tan</code></td>
</tr>
<tr>
<td>asin</td>
<td>asinf</td>
<td><code>asin</code></td>
</tr>
<tr>
<td>acos</td>
<td>acosf</td>
<td><code>acos</code></td>
</tr>
<tr>
<td>atan</td>
<td>atanf</td>
<td><code>atan</code></td>
</tr>
<tr>
<td>atan2</td>
<td>atan2f</td>
<td><code>atan2</code></td>
</tr>
<tr>
<td>sinh</td>
<td>sinhf</td>
<td><code>sinh</code></td>
</tr>
<tr>
<td>cosh</td>
<td>coshf</td>
<td><code>cosh</code></td>
</tr>
<tr>
<td>tanh</td>
<td>tanhf</td>
<td><code>tanh</code></td>
</tr>
<tr>
<td>asinh</td>
<td>asinhf</td>
<td><code>asinh</code></td>
</tr>
<tr>
<td>acosh</td>
<td>acoshf</td>
<td><code>acosh</code></td>
</tr>
<tr>
<td>atanh</td>
<td>atanhf</td>
<td><code>atanh</code></td>
</tr>
</tbody>
</table>

### Exponential and logarithmic functions

All of these functions are new in ISO C99, except for `exp`, `log` and `log10`.

<table>
<thead>
<tr>
<th><code>math.h</code></th>
<th><code>tgmath.h</code></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp</td>
<td>expf</td>
<td><code>exp</code></td>
</tr>
<tr>
<td>exp2</td>
<td>exp2f</td>
<td><code>exp2</code></td>
</tr>
<tr>
<td>expm1</td>
<td>expmf</td>
<td><code>expm1</code></td>
</tr>
<tr>
<td>log</td>
<td>logf</td>
<td><code>log</code></td>
</tr>
<tr>
<td>log10</td>
<td>log10f</td>
<td><code>log10</code></td>
</tr>
<tr>
<td>log1p</td>
<td>log1pf</td>
<td><code>log1p</code></td>
</tr>
<tr>
<td>log2</td>
<td>log2f</td>
<td><code>log2</code></td>
</tr>
<tr>
<td>ilogb</td>
<td>ilogbf</td>
<td><code>ilogb</code></td>
</tr>
<tr>
<td>logb</td>
<td>logbf</td>
<td><code>logb</code></td>
</tr>
</tbody>
</table>
frexp, ldexp, modf, scalbn, scalbln

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

**frexp, ldexp, modf, scalbn, scalbln**

Splits a float $x$ into fraction $f$ and exponent $n$, so that:

$0.0 \leq f < 1.0$ and $f \cdot 2^n = x$. Returns $f$, stores $n$.

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

Splits a float $x$ into fraction $f$ and integer $n$, so that:

$|f| < 1.0$ and $f + n = x$. Returns $f$, stores $n$.

Computes the result of $x \cdot \text{FLT\_RADIX}^n$, efficiently, not normally by computing $\text{FLT\_RADIX}^n$ explicitly.

Same as scalbn but with argument $n$ as long int.

**Rounding functions**

<table>
<thead>
<tr>
<th>math.h</th>
<th>tgmath.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ceil</td>
<td>ceilf</td>
<td>ceil</td>
</tr>
<tr>
<td>floor</td>
<td>floorf</td>
<td>floor</td>
</tr>
<tr>
<td>rint</td>
<td>rintf</td>
<td>rintl</td>
</tr>
<tr>
<td>lrint</td>
<td>lrintf</td>
<td>lrintl</td>
</tr>
<tr>
<td>llrint</td>
<td>llrintf</td>
<td>llrintl</td>
</tr>
<tr>
<td>nearbyint</td>
<td>nearbyintf</td>
<td>nearbyintl</td>
</tr>
<tr>
<td>round</td>
<td>roundf</td>
<td>roundl</td>
</tr>
<tr>
<td>lround</td>
<td>lroundf</td>
<td>lroundl</td>
</tr>
<tr>
<td>llround</td>
<td>llroundf</td>
<td>llroundl</td>
</tr>
<tr>
<td>trunc</td>
<td>truncf</td>
<td>truncl</td>
</tr>
</tbody>
</table>

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

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

Returns the rounded integer value as an int according to the current rounding direction. See fenv.h.

Returns the rounded integer value as a long int according to the current rounding direction. See fenv.h.

Returns the rounded integer value as a long long int according to the current rounding direction. See fenv.h.

Returns the rounded integer value as a floating-point according to the current rounding direction. See fenv.h.

Returns the nearest integer value of $x$ as int.

Returns the nearest integer value of $x$ as long int.

Returns the nearest integer value of $x$ as long long int.

Returns the truncated integer value $x$.

**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>

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

Returns the remainder $r$ of $x \cdot n$. $n$ is chosen as $\text{trunc}(x/y)$. $r$ may not have the same sign as $x$.

Same as remainder. In addition, the argument $\ast\ast\text{quo}$ is given a specific value (see ISO).
### Power and absolute-value functions

<table>
<thead>
<tr>
<th>math.h</th>
<th>tgamath.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cbtr</td>
<td>cbrtf</td>
<td>cbttl cbtr</td>
</tr>
<tr>
<td>fabs</td>
<td>fabsf</td>
<td>fabsl fabs</td>
</tr>
<tr>
<td>fma</td>
<td>fmaf</td>
<td>fmal fma</td>
</tr>
<tr>
<td>hypot</td>
<td>hypotf</td>
<td>hypotl hypot</td>
</tr>
<tr>
<td>pow</td>
<td>powf</td>
<td>powl power</td>
</tr>
<tr>
<td>sqrt</td>
<td>sqrtf</td>
<td>sqrtl sqrt</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>math.h</th>
<th>tgamath.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>copysign</td>
<td>copysignf</td>
<td>copysignl copysign</td>
</tr>
<tr>
<td>nan</td>
<td>nanf</td>
<td>nanl –</td>
</tr>
<tr>
<td>nextafter</td>
<td>nextafterf</td>
<td>nextafterl nextafter</td>
</tr>
<tr>
<td>nexttoward</td>
<td>nexttowardf</td>
<td>nexttowardl nexttoward</td>
</tr>
</tbody>
</table>

### Positive difference, maximum, minimum

<table>
<thead>
<tr>
<th>math.h</th>
<th>tgamath.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fdim</td>
<td>fdimf</td>
<td>fdiml fdim</td>
</tr>
<tr>
<td>fmax</td>
<td>fmaxf</td>
<td>fmaxl fmax</td>
</tr>
<tr>
<td>fmin</td>
<td>fminf</td>
<td>fminl fmin</td>
</tr>
</tbody>
</table>

### Error and gamma

<table>
<thead>
<tr>
<th>math.h</th>
<th>tgamath.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>erf</td>
<td>erff</td>
<td>erfl erf</td>
</tr>
<tr>
<td>erfc</td>
<td>erfcf</td>
<td>erfcl erc</td>
</tr>
<tr>
<td>lgamma</td>
<td>lgammaf</td>
<td>lgamml lgamma</td>
</tr>
<tr>
<td>tgamma</td>
<td>tgammaf</td>
<td>tgamml tgamma</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 `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>isgreatequal</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) \lor (x) &gt; (y) )</td>
</tr>
<tr>
<td>isunordered</td>
<td>-</td>
<td>Returns 1 if its arguments are unordered, 0 otherwise.</td>
</tr>
</tbody>
</table>

**Classification macros**

The next are implemented as macros. These macros are type generic and 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>fpclassify</td>
<td>-</td>
<td>Returns the class of its argument: FP_INFINITE, FP_NAN, FP_NORMAL, FP_SUBNORMAL or FP_ZERO</td>
</tr>
<tr>
<td>isfinite</td>
<td>-</td>
<td>Returns a nonzero value if and only if its argument has a finite value</td>
</tr>
<tr>
<td>isnf</td>
<td>-</td>
<td>Returns a nonzero value if and only if its argument has an infinite value</td>
</tr>
<tr>
<td>isnan</td>
<td>-</td>
<td>Returns a nonzero value if and only if its argument has NaN value.</td>
</tr>
<tr>
<td>isnormal</td>
<td>-</td>
<td>Returns a nonzero value if an only if its argument has a normal value.</td>
</tr>
<tr>
<td>signbit</td>
<td>-</td>
<td>Returns a nonzero value if and only if its argument value is negative.</td>
</tr>
</tbody>
</table>

**14.2.17. 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().
```
14.2.18. 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>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGINT</td>
<td>Receipt of an interactive attention signal</td>
</tr>
<tr>
<td>SIGILL</td>
<td>Detection of an invalid function message</td>
</tr>
<tr>
<td>SIGFPE</td>
<td>An erroneous arithmetic operation (for example, zero divide, overflow)</td>
</tr>
<tr>
<td>SIGSEGV</td>
<td>An invalid access to storage</td>
</tr>
<tr>
<td>SIGTERM</td>
<td>A termination request sent to the program</td>
</tr>
<tr>
<td>SIGABRT</td>
<td>Abnormal termination, such as is initiated by the abort function</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIG_DFL</td>
<td>Default behavior is used</td>
</tr>
<tr>
<td>SIG_IGN</td>
<td>The signal is ignored</td>
</tr>
</tbody>
</table>

The function returns the previous value of `signalfunction` for the specific signal, or SIG_ERR if an error occurs.

14.2.19. stdalign.h

This C11 header file contains the following macro definitions about alignment:

```c
#define alignas __Alignas
#define __alignas_is_defined 1
#define alignof __Alignof
#define __alignof_is_defined 1
```

14.2.20. stdarg.h

The facilities in this header file gives you a portable way to access variable arguments lists, such as needed for `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.
```
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.

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.

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.

14.2.21. stdatomic.h

This C11 header file contains macro definitions and type definitions and functions for performing atomic operations. The functions are implemented as macro definitions that map on intrinsic functions (see Section 1.12.4.1, Atomic Intrinsic Functions).

The following atomic lock-free macros are defined:

#define ATOMIC_BOOL_LOCK_FREE 2
#define ATOMIC_CHAR_LOCK_FREE 2
#define ATOMIC_CHAR16_T_LOCK_FREE 2
#define ATOMIC_CHAR32_T_LOCK_FREE 2
#define ATOMIC_WCHAR_T_LOCK_FREE 2
#define ATOMIC_SHORT_LOCK_FREE 2
#define ATOMIC_INT_LOCK_FREE __ATOMIC_INT_LOCK_FREE__
#define ATOMIC_LONG_LOCK_FREE __ATOMIC_LONG_LOCK_FREE__
#define ATOMIC_LLONG_LOCK_FREE 1
#define ATOMIC_POINTER_LOCK_FREE 2

The following structure type represents a lock-free, primitive atomic flag:

typedef struct
{
   __Alignas(4) _Atomic(uint32_t) flag;
} atomic_flag;

Macro which may be used to used to initialize an atomic_flag to the clear state:

#define ATOMIC_FLAG_INIT { 0 }

Initialization

Macro for initializing an atomic object:

#define ATOMIC_VAR_INIT(value) (value)
<table>
<thead>
<tr>
<th>Macro</th>
<th>Intrinsic function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic_init(obj, value)</td>
<td>__c11_atomic_init(obj, value)</td>
<td>Initializes the atomic object pointed to by obj to the value value.</td>
</tr>
</tbody>
</table>

**Order and consistency**

The memory order can be one of:

```c
typedef enum {
    memory_order_relaxed,
    memory_order_consume,
    memory_order_acquire,
    memory_order_release,
    memory_order_acq_rel,
    memory_order_seq_cst,
    memory_order;
} memory_order;
```

The following macro terminates a dependency chain; the argument does not carry a dependency to the return value.

```c
#define kill_dependency(value) (value)
```

**Fences**

<table>
<thead>
<tr>
<th>Macro</th>
<th>Intrinsic function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic_thread_fence(order)</td>
<td>__c11_atomic_thread_fence(order)</td>
<td>Depending on the memory order, this operation has either no effects, is an acquire fence, a release fence, both acquire and release fence, or sequentially consistent acquire and release fence.</td>
</tr>
<tr>
<td>atomic_signal_fence(order)</td>
<td>__c11_atomic_signal_fence(order)</td>
<td>Equivalent to atomic_thread_fence, except that the resulting ordering constraints are established only between a thread and a signal handler executed in the same thread.</td>
</tr>
</tbody>
</table>
Lock-free property

<table>
<thead>
<tr>
<th>Macro</th>
<th>Intrinsic function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic_is_lock_free(obj)</td>
<td>__c11_atomic_is_lock_free(obj)</td>
<td>Indicates whether or not the object pointed to by <em>obj</em> is lock-free.</td>
</tr>
</tbody>
</table>

Atomic integer types

For each type an atomic type name is declared as a type that has the same representation and alignment requirements as the corresponding direct type. See the header file `stdatomic.h` for the complete list. For example:

```c
typedef _Atomic _Bool                    atomic_bool;
typedef _Atomic char                    atomic_char;
typedef _Atomic short                   atomic_short;
typedef _Atomic int                     atomic_int;
typedef _Atomic long                    atomic_long;
typedef _Atomic long long               atomic_llong;
```

Operations on atomic types

<table>
<thead>
<tr>
<th>Macro</th>
<th>Intrinsic function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic_store(obj, desired)</td>
<td>__c11_atomic_store(obj, desired, memory_order_seq_cst)</td>
<td>Atomically replaces the value pointed to by <em>obj</em> with the value of <em>desired</em>. Memory is affected according to the value of <em>order</em>.</td>
</tr>
<tr>
<td>atomic_store_explicit(obj, desired, order)</td>
<td>__c11_atomic_store(obj, desired, order)</td>
<td>Atomically replaces the value pointed to by <em>obj</em> with the value of <em>desired</em>. Memory is affected according to the value of <em>order</em>.</td>
</tr>
<tr>
<td>atomic_load(obj)</td>
<td>__c11_atomic_load(obj, memory_order_seq_cst)</td>
<td>Atomically returns the value pointed to by <em>obj</em>.</td>
</tr>
<tr>
<td>atomic_load_explicit(obj, order)</td>
<td>__c11_atomic_load(obj, order)</td>
<td>Atomically returns the value pointed to by <em>obj</em>.</td>
</tr>
<tr>
<td>atomic_exchange(obj, desired)</td>
<td>__c11_atomic_exchange(obj, desired, memory_order_seq_cst)</td>
<td>Atomically replaces the value pointed to by <em>obj</em> with the value of <em>desired</em>. Memory is affected according to the value of <em>order</em>. These operations are atomic read-modify-write operations.</td>
</tr>
<tr>
<td>atomic_exchange_explicit(obj, desired, order)</td>
<td>__c11_atomic_exchange(obj, desired, order)</td>
<td>Atomically replaces the value pointed to by <em>obj</em> with the value of <em>desired</em>. Memory is affected according to the value of <em>order</em>. These operations are atomic read-modify-write operations.</td>
</tr>
</tbody>
</table>
**Intrinsic function**

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic_compare_exchange_strong(obj, expected, desired)</td>
<td>Atomically compares the value pointed to by <em>obj</em> for equality with that in <em>expected</em>. If the comparison is true, it replaces the value pointed to by <em>obj</em> with <em>desired</em>, and memory is affected according to the value of <em>success</em>. If the comparison is false, it updates the value in <em>expected</em> with the value pointed to by <em>obj</em>, and memory is affected according to the value of <em>failure</em>. These operations are atomic read-modify-write operations.</td>
</tr>
<tr>
<td>atomic_compare_exchange_strong_explicit(obj, expected, desired, success, failure)</td>
<td></td>
</tr>
<tr>
<td>atomic_compare_exchange_weak(obj, expected, desired)</td>
<td>Atomically replaces the value pointed to by <em>obj</em> with the result of the computation applied to the value pointed to by <em>obj</em> and the given <em>operand</em>. Memory is affected according to the value of <em>order</em>. The <em>key</em> values are: add (+, addition) sub (-, subtraction) or (</td>
</tr>
<tr>
<td>atomic_compare_exchange_weak_explicit(obj, expected, desired, success, failure)</td>
<td></td>
</tr>
<tr>
<td>atomic_fetch_key(obj, operand)</td>
<td></td>
</tr>
<tr>
<td>atomic_fetch_key_explicit(obj, operand, order)</td>
<td></td>
</tr>
</tbody>
</table>

**Atomic flag type and operations**

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>atomic_flag_test_and_set(obj)</td>
<td>Atomically sets the value pointed to by <em>obj</em> to true. Memory is affected according to the value of <em>order</em>.</td>
</tr>
<tr>
<td>atomic_flag_test_and_set_explicit(obj, order)</td>
<td></td>
</tr>
<tr>
<td>atomic_flag_clear(obj)</td>
<td>Atomically sets the value pointed to by <em>obj</em> to false. Memory is affected according to the value of <em>order</em>.</td>
</tr>
<tr>
<td>atomic_flag_clear_explicit(obj, order)</td>
<td></td>
</tr>
</tbody>
</table>
14.2.22. 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.

```c
#define bool _Bool
#define true 1
#define false 0
#define __bool_true_false_are_defined 1
```

14.2.23. stddef.h

This header file defines the types for common use:

- `ptrdiff_t` Signed integer type of the result of subtracting two pointers.
- `size_t` Unsigned integral type of the result of the `sizeof` operator.
- `max_align_t` (C11 only) Object type whose alignment is as great as the largest alignment of any scalar type.
- `wchar_t` Integer type to represent character codes in large character sets.

Besides these types, the following macros are defined:

```c
#define NULL (void *) 0
#define offsetof(_type, _member)  
```

14.2.24. stdint.h

See Section 14.2.10, inttypes.h and stdint.h

14.2.25. 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
The header file also defines type fpos_t as an unsigned long.

### Macros

<table>
<thead>
<tr>
<th>Header</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>Expands to the null pointer constant (void *) 0 for C or 0 (zero) for C++.</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) NOTE: WEOF need not to be a negative number as long as its value does not correspond to a member of the wide character set. (Defined in 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 setvbuf function.</td>
</tr>
<tr>
<td>_IOLBF</td>
<td>Expand to an integer expression, suitable for use as argument to the setvbuf function.</td>
</tr>
<tr>
<td>_IONBF</td>
<td>Expand to an integer expression, suitable for use as argument to the setvbuf function.</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 the fseek function.</td>
</tr>
<tr>
<td>SEEK_END</td>
<td>Expand to an integer expression, suitable for use as the third argument to the fseek function.</td>
</tr>
<tr>
<td>SEEK_SET</td>
<td>Expand to an integer expression, suitable for use as the third argument to the fseek function.</td>
</tr>
<tr>
<td>stderr</td>
<td>Expressions of type &quot;pointer to FILE&quot; that point to the FILE objects associated with standard error, input and output streams.</td>
</tr>
<tr>
<td>stdin</td>
<td></td>
</tr>
<tr>
<td>stdout</td>
<td></td>
</tr>
</tbody>
</table>

### File access

<table>
<thead>
<tr>
<th>Header</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>&quot;r&quot;</td>
<td>read; open text file for reading</td>
</tr>
<tr>
<td>&quot;w&quot;</td>
<td>write; create text file for writing; if the file already exists, its contents is discarded</td>
</tr>
<tr>
<td>&quot;a&quot;</td>
<td>append; open existing text file or create new text file for writing at end of file</td>
</tr>
<tr>
<td>&quot;r+&quot;</td>
<td>open text file for update; reading and writing</td>
</tr>
<tr>
<td>&quot;w+&quot;</td>
<td>create text file for update; previous contents if any is discarded</td>
</tr>
<tr>
<td>&quot;a+&quot;</td>
<td>append; open or create text file for update, writes at end of file</td>
</tr>
</tbody>
</table>

*(FSS implementation)*
stdio.h

fclose(name)  Flushed the data stream and closes the specified file that was previously
opened with fopen. (FSS implementation)

fflush(name)  If stream is an output stream, any buffered but unwritten data is written.
Else, the effect is undefined. (FSS implementation)

treopen(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', 'l', '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, 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>%</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 d, i, n, o, u and x may be preceded by 'h' if the argument is a pointer to short rather than int, or by 'hh' if the argument is a pointer to char, or by 'l' (letter ell) if the argument is a pointer to long or by 'll' for a pointer to long long, 'j' for a pointer to intmax_t or uintmax_t, 'z' for a pointer to size_t or 't' for a pointer to ptrdiff_t. The conversion characters e, f, and g may be preceded by 'l' if the argument is a pointer to double rather than float, and by 'L' for a pointer to a long double.

A conversion specifier. '*', maximum field width and length modifier are optional, the conversion character is not. The conversion character must be one of the following, if a character following '*' is not in the list, the behavior is undefined.

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

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

scanf conversion characters
The C library functions `printf()`, `fprintf()`, `vfprintf()`, `vsprintf()`, ... call one single function, `_doprint()`, that deals with the format string and arguments. The same applies to all `scanf` type functions, which call the function `_doscan()`, and also for the `wprintf` and `wscanf` type functions which call `_dowprint()` and `_dowscan()` respectively. The C library contains three versions of these functions.
routines: int, long and long long versions. If you use floating-point the formatter function for floating-point _doflt() or _dowflt() is called. Depending on the formatting arguments you use, the correct routine is used from the library. Of course the larger the version of the routine the larger your produced code will be.

Note that when you call any of the printf/scanf routines indirectly, the arguments are not known and always the long long version with floating-point support is used from the library.

Example:

```c
#include <stdio.h>

long L;

void main(void)
{
    printf( "This is a long: %ld\n", L );
}
```

The linker extracts the long version without floating-point support from the library.

See also the description of #pragma weak in Section 1.9, Pragmas to Control the Compiler.

### Character input/output

<table>
<thead>
<tr>
<th>stdio.h</th>
<th>wchar.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fgetc(stream)</td>
<td>fgetwc(stream)</td>
<td>Reads one character from stream. Returns the read character, or EOF/WEOF on error. (FSS implementation)</td>
</tr>
<tr>
<td>getc(stream)</td>
<td>getwc(stream)</td>
<td>Same as fgetc/fgetwc except that is implemented as a macro. (FSS implementation) NOTE: Currently #defined as getchar()/getwchar() because FILE I/O is not supported. Returns the read character, or EOF/WEOF on error.</td>
</tr>
<tr>
<td>getchar(stdin)</td>
<td>getwchar(stdin)</td>
<td>Reads one character from the stdin stream. Returns the character read or EOF/WEOF on error. Implemented as macro. (FSS implementation)</td>
</tr>
<tr>
<td>fgets(*s, n, stream)</td>
<td>fgetws(*s, n, stream)</td>
<td>Reads at most the next n-1 characters from the stream into array s until a newline is found. Returns s or NULL or EOF/WEOF on error. (FSS implementation)</td>
</tr>
<tr>
<td>gets(*s)</td>
<td>-</td>
<td>(C90/C99 only) Reads characters from the stdin stream into array s until end-of-file is encountered or a newline is found. The newline is replaced by a NULL character. Returns s or NULL on EOF. (FSS implementation)</td>
</tr>
</tbody>
</table>
**ExecutionContext**

**Configuration**

<table>
<thead>
<tr>
<th>stdio.h</th>
<th>wchar.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ungetc(c, stream)</td>
<td>ungetwc(c, stream)</td>
<td>Pushes character c back onto the input stream. Returns EOF/WEOF on error.</td>
</tr>
<tr>
<td>fputc(c, stream)</td>
<td>fputwc(c, stream)</td>
<td>Put character c onto the given stream. Returns EOF/WEOF on error. (<em>FSS implementation</em>)</td>
</tr>
<tr>
<td>putc(c, stream)</td>
<td>putwc(c, stream)</td>
<td>Same as fputc/fputwc except that is implemented as a macro. (<em>FSS implementation</em>)</td>
</tr>
<tr>
<td>putchar(c, stdout)</td>
<td>putwchar(c, stdout)</td>
<td>Put character c onto the stdout stream. Returns EOF/WEOF on error. Implemented as macro. (<em>FSS implementation</em>)</td>
</tr>
<tr>
<td>fputs(*s, stream)</td>
<td>fputs(*s, stream)</td>
<td>Writes string s to the given stream. Returns EOF/WEOF on error. (<em>FSS implementation</em>)</td>
</tr>
<tr>
<td>puts(*s)</td>
<td>-</td>
<td>Writes string s to the stdout stream. Returns EOF/WEOF on error. (<em>FSS implementation</em>)</td>
</tr>
</tbody>
</table>

**Direct input/output**

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

- ftell(stream) | Returns the current file position for stream, or -1L on error. (*FSS implementation*) |
- rewind(stream) | Sets the file position indicator for the stream to the beginning of the file. This function is equivalent to:
  (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>Function</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 14.2.5, errno.h )</td>
</tr>
</tbody>
</table>

**14.2.26. 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 14.2.14, `locale.h`).

**Numeric conversions**

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

```c
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.

```c
float wcstof(*s,**endp)
double wcstod(*s,**endp)
long double wcstold(*s,**endp)
```

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.

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

**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.

aligned_alloc(alignment, size)  (C11 only) Allocates space for an object whose alignment is specified by alignment and 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, while preserving its contents. If the new size is smaller than the old size, some contents at the end of the old region will be discarded. If the new size is larger than the old size, all of the old contents are preserved and any bytes in the new object beyond the size of the old object will have indeterminate values.

Environment communication

abort()  Causes abnormal program termination. If the signal SIGABRT is caught, the signal handler may take over control. (See Section 14.2.18, signal.h).

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

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

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

at_quick_exit(*func)  (C11 only) Registers the function pointed to by func to be called (without arguments) when quick_exit is called. Returns zero if the registration succeeds, nonzero if it fails.

quick_exit(status)  (C11 only) Causes normal program termination. Calls all functions registered by the at_quick_exit function, in the reverse order of their registration, and then calls _Exit.

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

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

bsearch(*key, *base, n, size, *cmp)

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

qsort(*base, n, size, *cmp)

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

Integer arithmetic

int abs(j)
long labs(j)
long long llabs(j)

div_t div(x,y)
ldiv_t ldiv(x,y)
lldiv_t lldiv(x,y)

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.

Multibyte/wide character and string conversions

mblen(*s,n)

Determines the number of bytes in the multibyte character pointed to by \( s \). At most \( n \) characters will be examined. (See also mbrlen in Section 14.2.33, wchar.h).

mbtowc(*pwc,*s,n)

Converts the multibyte character in \( s \) to a wide character code and stores it in \( pwc \). At most \( n \) characters will be examined.

wctomb(*s,wc)

Converts the wide character \( wc \) into a multibyte representation and stores it in the string pointed to by \( s \). At most MB_CUR_MAX characters are stored.

mbstowcs(*pwcs,*s,n)

Converts a sequence of multibyte 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 14.2.33, wchar.h).

wcstombs(*s,*pwcs,n)

Converts a sequence of wide characters in the array pointed to by \( pwcs \) into multibyte characters and stores at most \( n \) multibyte characters into the string pointed to by \( s \). (See also wcsrtowmb in Section 14.2.33, wchar.h).

14.2.27. stdnoreturn.h

This C11 header file contains the following macro definition:

#define noreturn _Noreturn
14.2.28. 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>memcpy(*s1,*s2,n)</td>
<td>wmemcpys1,*s2,n)</td>
<td>Copies n characters from *s2 into *s1 and returns *s1. If *s1 and *s2 overlap the result is undefined.</td>
</tr>
<tr>
<td>memmove(*s1,*s2,n)</td>
<td>wmemmove(*s1,*s2,n)</td>
<td>Same as memcpy, but overlapping strings are handled correctly. Returns *s1.</td>
</tr>
<tr>
<td>strcpy(*s1,*s2)</td>
<td>wcscpy(*s1,*s2)</td>
<td>Copies *s2 into *s1 and returns *s1. If *s1 and *s2 overlap the result is undefined.</td>
</tr>
<tr>
<td>strncpy(*s1,*s2,n)</td>
<td>wcsncpy(*s1,*s2,n)</td>
<td>Copies not more than n characters from *s2 into *s1 and returns *s1. If *s1 and *s2 overlap the result is undefined.</td>
</tr>
<tr>
<td>strcat(*s1,*s2)</td>
<td>wcscat(*s1,*s2)</td>
<td>Appends a copy of *s2 to *s1 and returns *s1. If *s1 and *s2 overlap the result is undefined.</td>
</tr>
<tr>
<td>strncat(*s1,*s2,n)</td>
<td>wcsncat(*s1,*s2,n)</td>
<td>Appends not more than n characters from *s2 to *s1 and returns *s1. If *s1 and *s2 overlap the result is undefined.</td>
</tr>
</tbody>
</table>

Comparison functions

<table>
<thead>
<tr>
<th>string.h</th>
<th>wchar.h</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>memcmp(*s1,*s2,n)</td>
<td>wmemcmp(*s1,*s2,n)</td>
<td>Compares the first n characters of *s1 to the first n characters of *s2. Returns &lt; 0 if *s1 &lt; *s2, 0 if *s1 = = *s2, or &gt; 0 if *s1 &gt; *s2.</td>
</tr>
<tr>
<td>strcmp(*s1,*s2)</td>
<td>wcsncmp(*s1,*s2)</td>
<td>Compares string *s1 to *s2. Returns &lt; 0 if *s1 &lt; *s2, 0 if *s1 = = *s2, or &gt; 0 if *s1 &gt; *s2.</td>
</tr>
<tr>
<td>strncmp(*s1,*s2,n)</td>
<td>wcsncmp(*s1,*s2,n)</td>
<td>Compares the first n characters of *s1 to the first n characters of *s2. Returns &lt; 0 if *s1 &lt; *s2, 0 if *s1 = = *s2, or &gt; 0 if *s1 &gt; *s2.</td>
</tr>
<tr>
<td>strcoll(*s1,*s2)</td>
<td>wcsncmp(*s1,*s2)</td>
<td>Performs a local-specific comparison between string *s1 and string *s2 according to the LC_COLLATE category of the current locale. Returns &lt; 0 if *s1 &lt; *s2, 0 if *s1 = = *s2, or &gt; 0 if *s1 &gt; *s2. (See Section 14.2.14, locale.h)</td>
</tr>
<tr>
<td>strxfrm(*s1,*s2,n)</td>
<td>wcsxfrm(*s1,*s2,n)</td>
<td>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.</td>
</tr>
</tbody>
</table>
### 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 strspn/wcsspn 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>wcstok(*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 14.2.5, 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>

### 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 defines 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

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.

clock

Returns the difference \( t_1 - t_0 \) in seconds.
diff_time(t1,t0)

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.

mktime(tm *tp)

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

time(*timer)

Time conversion

Converts the broken-down time in the structure pointed to by \( tp \) into a string in the form Tue Oct 22 16:15:14 2019\n\0. Returns a pointer to this string.

asctime(tm *tp)

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

cctime(*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.

gmtime(*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.

localtime(*timer)

Formatted time

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

strft ime(*s, smax, *fmt, tm *tp) wcsft ime(*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 strftime is locale-specific using the LC_TIME category (see Section 14.2.14, locale.h).

You can use the next conversion specifiers:
Libraries

%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)
%s  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
%V  week number of the year (00-53), week 1 has the first Sunday
%W  ISO 8601 week number (01-53) in the week-based year
%w  weekday (0-6, Sunday is 0)
%W  week number of the year (00-53), week 1 has the first Monday
%x  local date representation
%X  local time representation
%y  year without century (00-99)
%Y  year with century
%z  ISO 8601 offset of time zone from UTC, or nothing
%Z  time zone name, if any
14.2.30. typeinfo.h

The header file `typeinfo.h` contains a number of internal macros that are used by other header files to obtain information about the integer types. This header file is not defined in ISO C.

14.2.31. uchar.h

The C11 header file `uchar.h` declares types and functions for manipulating Unicode characters.

This header file declares the types:

- `char16_t`: Unsigned integer type used for 16-bit characters.
- `char32_t`: Unsigned integer type used for 32-bit characters.
- `size_t`: Unsigned integer type of the result of the `sizeof` operator.
- `wchar_t`: Integer type to represent character codes in large character sets.

The functions perform conversions between multibyte characters and Unicode 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 Unicode characters:

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

- `mbrtoc16(*pc16,*s,n,*ps)`: Converts a multibyte character *s to a 16-bit character *pc16 according to conversion state ps.
- `c16rtomb(*s,c16,*ps)`: Converts a 16-bit character c16 to a multibyte character according to conversion state ps and stores the multibyte character in *s.
- `mbrtoc32(*pc32,*s,n,*ps)`: Converts a multibyte character *s to a 32-bit character *pc32 according to conversion state ps.
- `c32rtomb(*s,c32,*ps)`: Converts a 32-bit character c32 to a multibyte character according to conversion state ps and stores the multibyte character in *s.

14.2.32. unistd.h

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)

cwd(*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)

14.2.33. wchar.h

Many functions in wchar.h represent the wide character variant of other functions so these are discussed together. (See Section 14.2.25, stdio.h and wchar.h, Section 14.2.26, stdlib.h and wchar.h, Section 14.2.28, string.h and wchar.h and Section 14.2.29, time.h and wchar.h).

The remaining functions are described below. They perform conversions between multibyte 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

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, multibyte characters are 1 byte long (MB_CUR_MAX and MB_LEN_MAX are defined as 1) and this will never occur.

mbsinit(*ps)

Determines whether the object pointed to by *ps, is an initial conversion state. Returns a non-zero value if so.

mbsrtowcs(*pwcs,**src,n,**ps)

Restartable version of mbstowcs. See Section 14.2.26, 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 14.2.26, stdlib.h and wchar.h. The initial conversion state is specified by *ps. The input wide string is specified indirectly by *src.

mbtowc(*s,ps,n)

Converts a multibyte character *s to a wide character according to conversion state *ps. See also mbtowc in Section 14.2.26, stdlib.h and wchar.h.

wctomb(*s,wc,ps)

Converts a wide character *wc to a multibyte character according to conversion state *ps and stores the multibyte character in *s.

btowc(c)

Returns the multibyte character corresponding to the wide character *c. Returns EOF on error.

wctob(c)

Returns the multibyte character corresponding to the wide character *c. The returned multibyte 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.

14.2.34. wctype.h

Most functions in wctype.h represent the wide character variant of functions declared in ctype.h and are discussed in Section 14.2.3, 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 14.2.14, 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.
wcctype(*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 14.2.14, 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.

### 14.3. C Library Reentrancy

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

The explanation of the cause why a function is not reentrant sometimes refers to a footnote because the explanation is too lengthy for the table.
<table>
<thead>
<tr>
<th>Function</th>
<th>Not reentrant because</th>
</tr>
</thead>
<tbody>
<tr>
<td>_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>_getflft</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>Uses global File System Simulation buffer, _dbg_request</td>
</tr>
<tr>
<td>access</td>
<td>Sets errno.</td>
</tr>
<tr>
<td>acos acosf acosl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>acosh acoshf acoshl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>aligned_alloc</td>
<td>See malloc (5).</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>at_quick_exit defines static array with function pointers to execute when quick_exit is called.</td>
</tr>
<tr>
<td>atan2 atan2f atan2l</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>atanh atanhf atanhl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>atexit</td>
<td>atexit defines static array with function pointers to execute at exit of program.</td>
</tr>
<tr>
<td>atof</td>
<td>-</td>
</tr>
<tr>
<td>atoi</td>
<td>-</td>
</tr>
<tr>
<td>atol</td>
<td>-</td>
</tr>
<tr>
<td>bsearch</td>
<td>-</td>
</tr>
<tr>
<td>btowc</td>
<td>-</td>
</tr>
<tr>
<td>c16rtomb</td>
<td>Sets errno. Uses static internal_state variable.</td>
</tr>
<tr>
<td>c32rtomb</td>
<td>Sets errno. Uses static internal_state variable.</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 cacosh cfacoshl</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>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>csinh csinhf csinhl</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>-</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>Calls fclose. See (1)</td>
</tr>
<tr>
<td>cleanup</td>
<td>-</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</td>
<td>-</td>
</tr>
<tr>
<td>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 csqrtl</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>tanh ctanhf ctanhl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>ctime</td>
<td>Calls asctime</td>
</tr>
<tr>
<td>difftime</td>
<td>-</td>
</tr>
<tr>
<td>div ldiv lldiv</td>
<td>-</td>
</tr>
<tr>
<td>erf erfl erff</td>
<td>-</td>
</tr>
<tr>
<td>Function</td>
<td>Not reentrant because</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>erf, erfc, erfcl</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>exit</td>
<td>Sets errno.</td>
</tr>
<tr>
<td>exp, expf, expl</td>
<td>Sets errno.</td>
</tr>
<tr>
<td>exp2, exp2f, exp2l</td>
<td>Sets errno.</td>
</tr>
<tr>
<td>expm1, expm1f, expm1l</td>
<td>Sets errno via calls to other functions.</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>-</td>
</tr>
<tr>
<td>fclearexcept</td>
<td>Writes PSW bits.</td>
</tr>
<tr>
<td>fegetenv</td>
<td>- (reads PSW bits)</td>
</tr>
<tr>
<td>fegetexceptflag</td>
<td>- (reads PSW bits via calls to other functions)</td>
</tr>
<tr>
<td>fegetround</td>
<td>- (reads PSW bits)</td>
</tr>
<tr>
<td>feholdexcept</td>
<td>Reads/writes PSW bits via calls to other functions.</td>
</tr>
<tr>
<td>feof</td>
<td>Uses values in iob[ ]. See (1).</td>
</tr>
<tr>
<td>feraiseexcept</td>
<td>Writes PSW bits.</td>
</tr>
<tr>
<td>ferror</td>
<td>Uses values in iob[ ]. See (1).</td>
</tr>
<tr>
<td>fesetenv</td>
<td>Writes PSW bits.</td>
</tr>
<tr>
<td>fesetexceptflag</td>
<td>Writes PSW bits via calls to other functions.</td>
</tr>
<tr>
<td>fesetround</td>
<td>Writes PSW bits.</td>
</tr>
<tr>
<td>fetestexcept</td>
<td>- (reads PSW bits)</td>
</tr>
<tr>
<td>feupdateenv</td>
<td>Writes PSW bits via calls to other functions.</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>-</td>
</tr>
<tr>
<td>fmax, fmaxf, fmaxl</td>
<td>-</td>
</tr>
<tr>
<td>fmin, fminf, fminl</td>
<td>-</td>
</tr>
<tr>
<td>fmod, fmodf, fmodl</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, fwprintf</td>
<td>Uses iob[ ]. See (1).</td>
</tr>
<tr>
<td>fputc, fputwc</td>
<td>Uses iob[ ]. See (1).</td>
</tr>
<tr>
<td>Function</td>
<td>Not reentrant because</td>
</tr>
<tr>
<td>-----------------</td>
<td>---------------------------------------------------------------------------------------</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, frexpfl, 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>getchar, getwchar</td>
<td></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>gmtime defines static structure</td>
</tr>
<tr>
<td>hypot, hypotfl, hypotl</td>
<td>Sets errno via calls to other functions.</td>
</tr>
<tr>
<td>ilogb, ilogbf, ilogbl</td>
<td></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>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>Function</td>
<td>Not reentrant because</td>
</tr>
<tr>
<td>---------------------------</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 iswxdigit</td>
<td></td>
</tr>
<tr>
<td>ldexp ldexpf ldexpfl</td>
<td>Sets errno. See (2).</td>
</tr>
<tr>
<td>lgamma lgammaf lgammal</td>
<td>Sets errno.</td>
</tr>
<tr>
<td>llrint llrintf llrintl</td>
<td></td>
</tr>
<tr>
<td>llround llroundf llroundl</td>
<td>Sets errno.</td>
</tr>
<tr>
<td>localeconv</td>
<td>N.A.; skeleton function</td>
</tr>
<tr>
<td>localtime</td>
<td></td>
</tr>
<tr>
<td>log logf logl</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>Sets errno.</td>
</tr>
<tr>
<td>log2 log2f log2l</td>
<td>Sets errno.</td>
</tr>
<tr>
<td>logb logbf logbl</td>
<td>Sets errno.</td>
</tr>
<tr>
<td>longjmp</td>
<td></td>
</tr>
<tr>
<td>lrint lrintf lrintl</td>
<td></td>
</tr>
<tr>
<td>lround lroundf lroundl</td>
<td>Sets errno.</td>
</tr>
<tr>
<td>lseek</td>
<td>Calls _lseek</td>
</tr>
<tr>
<td>Function</td>
<td>Not reentrant because</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------------------------------------</td>
</tr>
<tr>
<td>lstat</td>
<td><em>(Not implemented)</em></td>
</tr>
<tr>
<td>malloc</td>
<td>Needs kernel support. See (5).</td>
</tr>
<tr>
<td>mblen</td>
<td>N.A., skeleton function</td>
</tr>
<tr>
<td>mbrlen</td>
<td>Sets errno.</td>
</tr>
<tr>
<td>mbrtoc16</td>
<td>Sets errno. Uses static internal_state variable.</td>
</tr>
<tr>
<td>mbrtoc32</td>
<td>Sets errno. Uses static internal_state variable.</td>
</tr>
<tr>
<td>mbtowc</td>
<td>Sets errno. Uses static internal_state variable.</td>
</tr>
<tr>
<td>mbsinit</td>
<td>-</td>
</tr>
<tr>
<td>mbsrtowcs</td>
<td>Sets errno.</td>
</tr>
<tr>
<td>mbstowcs</td>
<td>N.A., skeleton function</td>
</tr>
<tr>
<td>mbtowc</td>
<td>N.A., skeleton function</td>
</tr>
<tr>
<td>memchr</td>
<td>-</td>
</tr>
<tr>
<td>wmemchr</td>
<td>-</td>
</tr>
<tr>
<td>memcmp</td>
<td>-</td>
</tr>
<tr>
<td>wmemcmp</td>
<td>-</td>
</tr>
<tr>
<td>memcpy</td>
<td>-</td>
</tr>
<tr>
<td>wmemcpy</td>
<td>-</td>
</tr>
<tr>
<td>memmove</td>
<td>-</td>
</tr>
<tr>
<td>wmemmove</td>
<td>-</td>
</tr>
<tr>
<td>memset</td>
<td>-</td>
</tr>
<tr>
<td>wmemset</td>
<td>-</td>
</tr>
<tr>
<td>mktime</td>
<td>-</td>
</tr>
<tr>
<td>modf</td>
<td>-</td>
</tr>
<tr>
<td>modff</td>
<td>-</td>
</tr>
<tr>
<td>modfl</td>
<td>-</td>
</tr>
<tr>
<td>nan</td>
<td>-</td>
</tr>
<tr>
<td>nanf</td>
<td>-</td>
</tr>
<tr>
<td>nanl</td>
<td>-</td>
</tr>
<tr>
<td>nearbyint</td>
<td>-</td>
</tr>
<tr>
<td>nearbyintf</td>
<td>-</td>
</tr>
<tr>
<td>nearbyintl</td>
<td>-</td>
</tr>
<tr>
<td>nextafter</td>
<td>-</td>
</tr>
<tr>
<td>nextafterf</td>
<td>-</td>
</tr>
<tr>
<td>nextafterl</td>
<td>-</td>
</tr>
<tr>
<td>nexttoward</td>
<td>-</td>
</tr>
<tr>
<td>nexttowardf</td>
<td>-</td>
</tr>
<tr>
<td>nexttowardl</td>
<td>-</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</td>
<td>Sets errno. See (2)</td>
</tr>
<tr>
<td>powf</td>
<td>Sets errno. See (2)</td>
</tr>
<tr>
<td>powl</td>
<td>Sets errno. See (2)</td>
</tr>
<tr>
<td>printf</td>
<td>Uses iob[]. See (1)</td>
</tr>
<tr>
<td>wprintf</td>
<td>Uses iob[]. See (1)</td>
</tr>
<tr>
<td>putc</td>
<td>Uses iob[]. See (1)</td>
</tr>
<tr>
<td>putwc</td>
<td>Uses iob[]. See (1)</td>
</tr>
<tr>
<td>putchar</td>
<td>Uses iob[]. See (1)</td>
</tr>
<tr>
<td>putwchar</td>
<td>Uses iob[]. See (1)</td>
</tr>
<tr>
<td>puts</td>
<td>-</td>
</tr>
<tr>
<td>qsort</td>
<td>Calls _Exit.</td>
</tr>
<tr>
<td>raise</td>
<td>Updates the signal handler table</td>
</tr>
</tbody>
</table>
### Not reentrant because

<table>
<thead>
<tr>
<th>Function</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<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 <code>_read</code></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, <code>_dbg_request</code></td>
</tr>
<tr>
<td>remquo remquof remquol</td>
<td></td>
</tr>
<tr>
<td>rename</td>
<td>Uses global File System Simulation buffer, <code>_dbg_request</code></td>
</tr>
<tr>
<td>rewind</td>
<td>Eventually calls <code>_lseek</code></td>
</tr>
<tr>
<td>rint rintf rintl</td>
<td></td>
</tr>
<tr>
<td>round roundf roundl</td>
<td></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 <code>_doscan</code>. 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 sinhl</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, <code>_dbg_request</code></td>
</tr>
<tr>
<td>strcat wcscat</td>
<td></td>
</tr>
<tr>
<td>strchr wcscchr</td>
<td></td>
</tr>
<tr>
<td>strcmp wscmp</td>
<td></td>
</tr>
<tr>
<td>strcoll wcscoll</td>
<td></td>
</tr>
<tr>
<td>strcpy wcscpy</td>
<td></td>
</tr>
<tr>
<td>strcspn wcscspn</td>
<td></td>
</tr>
<tr>
<td>strerror</td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Not reentrant because</td>
</tr>
<tr>
<td>----------------------</td>
<td>---------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>strftime wcsftime</td>
<td>-</td>
</tr>
<tr>
<td>strlen wcslen</td>
<td>-</td>
</tr>
<tr>
<td>strncat wcsncat</td>
<td>-</td>
</tr>
<tr>
<td>strcmp wcsncmp</td>
<td>-</td>
</tr>
<tr>
<td>strncpy wcsncpy</td>
<td>-</td>
</tr>
<tr>
<td>strpbrk wcspbrk</td>
<td>-</td>
</tr>
<tr>
<td>strrchr wcsrchr</td>
<td>-</td>
</tr>
<tr>
<td>strspn wcsspn</td>
<td>-</td>
</tr>
<tr>
<td>strstr wcsstr</td>
<td>-</td>
</tr>
<tr>
<td>strtod wcstod</td>
<td>-</td>
</tr>
<tr>
<td>strtof wcstof</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 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 tgammal</td>
<td>Sets errno.</td>
</tr>
<tr>
<td>time</td>
<td>Uses static variable which defines initial start time</td>
</tr>
<tr>
<td>tmpfile</td>
<td>Uses iob[]. See (1).</td>
</tr>
<tr>
<td>tmpnam</td>
<td>Uses local buffer to build filename. Function can be adapted to use user buffer. This makes the function non 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>towlower</td>
<td>-</td>
</tr>
<tr>
<td>towupper</td>
<td>-</td>
</tr>
<tr>
<td>trunc truncf truncl</td>
<td>-</td>
</tr>
</tbody>
</table>
Several functions in the C library are not reentrant due to the following reasons:

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

The following description discusses these items in more detail. The numbers at the beginning 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 \texttt{iob[]} array. The functions which use elements of this array access these elements via pointers (\texttt{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 \texttt{iob[]} array. Currently, the \texttt{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 \texttt{iob[]}, it is apparent that the \texttt{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 \texttt{iob[]} array to use. Thus the library is suitable for interfacing to that kernel only.

Another approach for the \texttt{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 \texttt{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 \texttt{iob[]} can use the standard file handles \texttt{stdin}, \texttt{stdout} and \texttt{stderr} (which are the first three entries in \texttt{iob[]}). Thus all I/O for these three file handles should be located in one module.

(2) \texttt{errno} declaration

Several functions in the C library set the global variable \texttt{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 \texttt{errno} already have a return type (this is the reason for using \texttt{errno}) it is not possible to check successful completion via the return type.

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

\texttt{errno} can be set to \texttt{ERR\_FORMAT} by the print and scan functions in the C library if you specify illegal format strings.

\texttt{errno} will never be set to \texttt{ERR\_NOLONG} or \texttt{ERR\_NOPOINT} since the C library supports long and pointer conversion routines for input and output.

\texttt{errno} can be set to \texttt{ERANGE} by the following functions: \texttt{exp()}, \texttt{strtol()}, \texttt{strtoul()} and \texttt{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 \texttt{errno} is set to \texttt{ERANGE}.

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

\texttt{errno} can be set to \texttt{ERR\_POS} by the file positioning functions \texttt{ftell()}, \texttt{fsetpos()} and \texttt{fgetpos()}.

(3) \texttt{ungetc}

Currently the \texttt{ungetc} buffer is static. For each file entry in the \texttt{iob[]} structure array, there is one character available in the buffer to \texttt{unget} a character.

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

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

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

(5) malloc

Malloc uses a heap space which is assigned at locate time. Thus this implementation is not reentrant. Making a reentrant malloc requires some sort of system call to obtain free memory space on a per process basis. This is not easy to solve within the current context of the library. This requires 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 15. List File Formats

This chapter describes the format of the assembler list file and the linker map file.

15.1. 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 6.5, 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 ARM vx.yrz - ARM assembler Build yymmddqq
Title Page 1

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 directive .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 directives .LIST/.NOLIST,.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
  |
  |
  0000 00B5 2 2 15 push {lr}
  0002 0348 1 3 16 ldr r0,.L3
  0004 0168 1 4 17 ldr r1,[r0,#0]
  0006 0348 1 5 18 ldr r0,.L3+4
  0008 rrFrrrDr 3 8 19 bl printf
  000C 0020 1 9 20 movs r0,#0
  000E 00BD 2 11 21 pop {pc}
  |
  |
  0000 38 .ds 2
  | RESERVED
  0001
**15.2. 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 (.obj) 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 7.11, *Generating a Map File*.

With the linker option `--map-file-format` you can specify which parts of the map file you want to see. To specify the same for the global map file, use `linker option --global-map-file-format`. Both options have the same defaults and accept the same arguments.

In Eclipse the linker map file (*project*.mapxml) 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 page displays a part of the map file. You can use the drop-down list or the Outline view to navigate through the different tables and you can use the following buttons.

<table>
<thead>
<tr>
<th>Icon</th>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>⇠</td>
<td>Back</td>
<td>Goes back one page in the history list.</td>
</tr>
<tr>
<td>⇢</td>
<td>Forward</td>
<td>Goes forward one page in the history list.</td>
</tr>
<tr>
<td>![ ]</td>
<td>Next Table</td>
<td>Shows the next table from the drop-down list.</td>
</tr>
<tr>
<td>![ ]</td>
<td>Previous Table</td>
<td>Shows the previous table from the drop-down list.</td>
</tr>
</tbody>
</table>

When you right-click in the view, a popup menu appears (for example, to reset the layout of a table). 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 system memory and user memory as defined in the linker script file (*.1sl).
| **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). |
| **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: ARM:ARM: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. |
| **Recursive** | Indicates if recursion is detected yes or no. |
| **Entry Points** | A list of entry points used for estimation of the stack usage. |

**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 (.obj) 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.

The following example is a part of a call graph in the textual version of the map file (.map):

Reset_Handler [0,104]
  | +-- main [4,104]
  |     | +-- printf [20,100]
  |     |     | +-- _doprint [16,80]
  |     |     |     | +-- _io_putchar [4,64]
  |     |     |     |     | +-- fputc [4,60]
  |     |     |     |     |     | +-- _flsbuf [0,56]
  |     |     |     |     |     |     | +-- _dofls [20,56]
  |     |     |     |     |     |     |     | +-- _flsbuf.c:.cocofun_1 [0,0]
  |     |     |     |     |     |     |     |     | +-- _fflush *
  |     |     |     |     |     |     |     |     |     | +-- _host_write *
  |     |     |     |     |     |     |     |     |     | +-- exit [8,68]

- A * after a function name indicates that the call tree starting with this function is shown separately, with a * in front of the function name.
• A * in front of a function name indicates that the function is not considered a "root" in the call graph since it is called by one or more other functions.

• An additional R (not shown in this example) indicates this function is part of a recursive call chain. If both a leaf and the root of a tree are marked this way, all nodes in between are in a recursive chain.

• An '__INDIRECT__' entry (not shown in this example) indicates an indirect function call. It is not an actual function. Each function listed as a caller of the '__INDIRECT__' placeholder symbol places a call through a function pointer. Each function listed as a callee of the '__INDIRECT__' placeholder symbol has its address taken (and used).

• [ ] after a function contains information about the stack usage. The first field is the amount of stack used by the function and the second field is the amount of stack used by the function including its callees.

In the graphical version of the map file, you can expand or collapse a single node. Use the / buttons to expand/collapse all nodes in the call graph. Hover the mouse over a function (root, callee or node) to see information about the stack usage.

<table>
<thead>
<tr>
<th>Icon</th>
<th>Meaning</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>⚡ Root</td>
<td>This function is the top of the call graph. If there are interrupt handlers, there can be several roots.</td>
<td></td>
</tr>
<tr>
<td>🔍 Callee</td>
<td>This function is referenced by several No leaf functions. Right-click on the function and select Expand all References to see all functions that reference this function. Select Back to Caller to return to the calling function.</td>
<td></td>
</tr>
<tr>
<td>⚪ Node</td>
<td>A normal node (function) in the call graph.</td>
<td></td>
</tr>
<tr>
<td>➡ Caller</td>
<td>This function calls a function which is listed separately in the call graph. Right-click on the function and select Go to Callee to see the callee. Hover the mouse over the function to see a popup with all callees.</td>
<td></td>
</tr>
</tbody>
</table>

**Overlay**

This part is empty for the ARM.

**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 memory chip and group and sorted on space address. In Eclipse, right-click in the table and select Configure Columns to specify which columns you want to see. If you hover the mouse over a section, you get a popup with information about the section. If you select a range of sections, in the Fast View bar (at the bottom) you will see information about the selected range, such as the total size, how many sections are selected and how many gaps are present.

| # | The line number and default sort order. |
| Section | The name and id of the section. The number between '(' ')' uniquely identifies the section. Sections within square brackets [ ] will be copied during initialization from ROM to the corresponding section name in RAM. |
**List File Formats**

<table>
<thead>
<tr>
<th>Sect. size (hex)</th>
<th>The size of the section in minimum addressable units (hexadecimal or decimal).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sect. size (dec)</td>
<td></td>
</tr>
<tr>
<td>Group</td>
<td>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.</td>
</tr>
<tr>
<td>Start address</td>
<td>The first address of the section in the address space.</td>
</tr>
<tr>
<td>End address</td>
<td>The last address of the section in the address space.</td>
</tr>
<tr>
<td>Symbols in sect.</td>
<td>The names of the external symbols that are referenced in the section. See Locate Result: Symbols below.</td>
</tr>
<tr>
<td>Defined in</td>
<td>The names of the input modules the section is defined in. See Link Result: [in] File above.</td>
</tr>
<tr>
<td>Referenced in</td>
<td>The names of the modules that contain a reference to the section. See Cross References above.</td>
</tr>
<tr>
<td>Address space</td>
<td>The name of the address space.</td>
</tr>
<tr>
<td>Chip name</td>
<td>The names of the memory chips as defined in the linker script file (*.lsl) in the memory definitions.</td>
</tr>
<tr>
<td>Chip addr</td>
<td>The absolute offset of the section from the start of a memory chip.</td>
</tr>
<tr>
<td>Locate type:properties</td>
<td>The locate rule type and properties. See Locate Rules below.</td>
</tr>
</tbody>
</table>

The following buttons are available in this part of the map file.

<table>
<thead>
<tr>
<th>Icon</th>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Configure Section Filter</td>
<td>Opens the Configure Section Filter dialog. Here you can select which sections you want to see in the map file and how.</td>
</tr>
<tr>
<td></td>
<td>Enable Highlighting</td>
<td>All sections that are marked with &quot;Highlight&quot; in the Configure Section Filter dialog will be highlighted in the table.</td>
</tr>
<tr>
<td></td>
<td>Enable Collapsing</td>
<td>All sections that are marked with &quot;Collapse&quot; in the Configure Section Filter dialog will appear collapsed in the table.</td>
</tr>
<tr>
<td></td>
<td>Only Show Matching Lines</td>
<td>All lines that are not part of the selection in the Configure Section Filter dialog will be hidden.</td>
</tr>
<tr>
<td></td>
<td>Show Gaps</td>
<td>Also shows the gaps in the map file. Click the button again to hide the gaps.</td>
</tr>
</tbody>
</table>

**Configure Section Filter Dialog**

In this dialog you can filter which sections you want to see in the map file and how. Click Add to add a new filter. Explanation of the columns and fields:

- **Highlight**: Marks the section as a candidate for highlighting. Turn on Enable Highlighting to see the effect.
- **Color**: The highlight color.
**Collapse**
Marks the section as a candidate for collapsing. Turn on **Enable Collapsing** to see the effect.

**Section name**
A filter to select a section or group of sections. Wildcards are allowed. Wildcards follow the rules of regular expressions. To get help on which wildcards are supported, press **Ctrl-space**. Click an item in the list for help, double-click to add the wildcard.

**Start address**
The first address of the section in the address space for this filter.

**End address**
The last address of the section in the address space for this filter.

**Address space**
The name of the address space.

**Chip name**
The name of the memory chip as defined in the linker script file (*.lsl) in the memory definitions.

**Hide gaps smaller than**
If gaps are shown in the map file, here you can limit the number of gaps you want to see.

The meaning of the check boxes is the same as the corresponding buttons available in this part of the map file.

**Locate Result: Symbols**
This part of the map file lists all external symbols per address space name.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Address</strong></td>
<td>The absolute address of the symbol in the address space.</td>
</tr>
<tr>
<td><strong>Name</strong></td>
<td>The name of the symbol.</td>
</tr>
<tr>
<td><strong>Space</strong></td>
<td>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: <strong>ARM:ARM:linear</strong>.</td>
</tr>
</tbody>
</table>

**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.

You can expand or collapse a part of the information.

**Locate Rules**
This part of the map file shows the rules the linker uses to locate sections.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Address space</strong></td>
<td>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.</td>
</tr>
<tr>
<td>Type</td>
<td>The rule type:</td>
</tr>
<tr>
<td>-----------------</td>
<td>----------------</td>
</tr>
<tr>
<td>ordered/contiguous/clustered/unrestricted</td>
<td>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.</td>
</tr>
<tr>
<td>absolute</td>
<td>The section must be located at the address shown in the Properties column.</td>
</tr>
<tr>
<td>ranged</td>
<td>The section must be located anywhere in the address ranges shown in the Properties column; end addresses are not included in the range.</td>
</tr>
<tr>
<td>page</td>
<td>The sections must be located in some address range with a size not larger than shown in the Properties column; the first number is the page size, the second part is the address range restriction within the page.</td>
</tr>
<tr>
<td>ranged page</td>
<td>Both the ranged and the paged restriction apply. In the Properties column the range restriction is listed first, followed by the paged restriction between parenthesis.</td>
</tr>
<tr>
<td>ballooned</td>
<td>After locating all sections, the largest remaining gap in the space is used completely for the stack and/or heap.</td>
</tr>
</tbody>
</table>

| Properties      | The contents depends on the Type column. |
| Prio            | The locate priority of the rule. A higher priority value gives a rule precedence over a rule with a lower priority, but only if the two rules have the same type and the same properties. The relative order of rules of different types or different properties is not affected by this priority value. You can set the priority with the priority group attribute in LSL |

| Sections        | The sections to which the rule applies; |
| restrictions between sections are shown in this column: |

<table>
<thead>
<tr>
<th></th>
<th>ordered</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>contiguous</td>
</tr>
<tr>
<td></td>
<td>clustered</td>
</tr>
</tbody>
</table>

For contiguous sections, the linker uses the section order as shown here. Clustered sections can be located in any relative order.
## Removed Sections

This part of the map file shows the sections which are removed from the output file as a result of the optimization option to delete unreferenced sections and or duplicate code or constant data (linker option --optimize=cxy).

<table>
<thead>
<tr>
<th>Section</th>
<th>The name of the section which has been removed.</th>
</tr>
</thead>
<tbody>
<tr>
<td>File</td>
<td>The name of the input object file where the section is removed from.</td>
</tr>
<tr>
<td>Library</td>
<td>The name of the library where the object file is part of.</td>
</tr>
<tr>
<td>Symbol</td>
<td>The symbols that were present in the section.</td>
</tr>
<tr>
<td>Reason</td>
<td>The reason why the section has been removed. This can be because the section is unreferenced or duplicated.</td>
</tr>
</tbody>
</table>
Chapter 16. Object File Formats

This chapter describes the format of several object files.

16.1. ELF/DWARF Object Format

The TASKING VX-toolset for ARM by default produces objects in the ELF/DWARF 3 format.

For a complete description of the ELF format, please refer to the Tool Interface Standard (TIS).

For a complete description of the DWARF format, please refer to the DWARF Debugging Information Format Version 3. See http://dwarfstd.org/

16.2. Intel Hex Record Format

Intel Hex records describe the hexadecimal object file format for 8-bit, 16-bit and 32-bit microprocessors. The hexadecimal object file is an ASCII representation of an absolute binary object file. There are six different types of records:

- Data Record (8-, 16, or 32-bit formats)
- End of File Record (8-, 16, or 32-bit formats)
- Extended Segment Address Record (16, or 32-bit formats)
- Start Segment Address Record (16, or 32-bit formats)
- Extended Linear Address Record (32-bit format only)
- Start Linear Address Record (32-bit format only)

To generate an Intel Hex output file:

1. From the Project menu, select Properties for

   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 » Output Format.

4. Enable the option Generate Intel Hex format file.

5. (Optional) Specify the Size of addresses (in bytes) for Intel Hex records.

6. (Optional) Enable or disable the option Emit start address record.

By default the linker generates records in the 32-bit format (4-byte addresses).
General Record Format

In the output file, the record format is:

<table>
<thead>
<tr>
<th>:</th>
<th>length</th>
<th>offset</th>
<th>type</th>
<th>content</th>
<th>checksum</th>
</tr>
</thead>
</table>

where:

: is the record header.

*length* is the record length which specifies the number of bytes of the *content* field. This value occupies one byte (two hexadecimal digits). The linker outputs records of 255 bytes (32 hexadecimal digits) or less; that is, *length* is never greater than 0xFF.

*offset* is the starting load offset specifying an absolute address in memory where the data is to be located when loaded by a tool. This field is two bytes long. This field is only used for Data Records. In other records this field is coded as four ASCII zero characters (‘0000’).

*type* is the record type. This value occupies one byte (two hexadecimal digits). The record types are:

<table>
<thead>
<tr>
<th>Byte Type</th>
<th>Record Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>Data</td>
</tr>
<tr>
<td>01</td>
<td>End of file</td>
</tr>
<tr>
<td>02</td>
<td>Extended segment address (not used)</td>
</tr>
<tr>
<td>03</td>
<td>Start segment address (not used)</td>
</tr>
<tr>
<td>04</td>
<td>Extended linear address (32-bit)</td>
</tr>
<tr>
<td>05</td>
<td>Start linear address (32-bit)</td>
</tr>
</tbody>
</table>

*content* is the information contained in the record. This depends on the record type.

*checksum* is the record checksum. The linker computes the checksum by first adding the binary representation of the previous bytes (from length to content). The linker then computes the result of sum modulo 256 and subtracts the remainder from 256 (two’s complement). Therefore, the sum of all bytes following the header is zero.
Extended Linear Address Record

The Extended Linear Address Record specifies the two most significant bytes (bits 16-31) of the absolute address of the first data byte in a subsequent Data Record:

```
: 02 0000 04 upper_address  checksum
```

The 32-bit absolute address of a byte in a Data Record is calculated as:

```
( address + offset + index ) modulo 4G
```

where:

- `address` is the base address, where the two most significant bytes are the `upper_address` and the two least significant bytes are zero.
- `offset` is the 16-bit offset from the Data Record.
- `index` is the index of the data byte within the Data Record (0 for the first byte).

Example:

```
:02000040FFFFB
```

Data Record

The Data Record specifies the actual program code and data.

```
: length offset 00 data checksum
```

The `length` byte specifies the number of `data` bytes. The linker has an option (`--hex-record-size`) that controls the length of the output buffer for generating Data records. The default buffer length is 32 bytes.

The `offset` is the 16-bit starting load offset. Together with the address specified in the Extended Address Record it specifies an absolute address in memory where the data is to be located when loaded by a tool.

Example:

```
:0F002000023222754E00754F04AF4FAE4E22C3
```

Object File Formats
Start Linear Address Record

The Start Linear Address Record contains the 32-bit program execution start address.

|   | 04 | 0000 | 05 | address | checksum |

With linker option `--hex-format=S` you can prevent the linker from emitting this record.

Example:

```
:0400000500FF0003F5
  |   |   |    | checksum
  |   |   |    | address
  |   |    | type
  |    offset
  | length
```

End of File Record

The hexadecimal file always ends with the following end-of-file record:

```
:00000001FF
  |   |    | checksum
  |   | type
  | offset
  | length
```

16.3. Motorola S-Record Format

To generate a Motorola S-record output file:

1. From the **Project** menu, select **Properties for**

   *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 » Output Format**.

4. Enable the option **Generate S-records file**.

5. (Optional) Specify the **Size of addresses (in bytes) for Motorola S records**.

By default, the linker produces output in Motorola S-record format with three types of S-records (4-byte addresses): S0, S3 and S7. Depending on the size of addresses you can force other types of S-records. They have the following layout:
**S0 - record**

<table>
<thead>
<tr>
<th>S0</th>
<th>length</th>
<th>0000</th>
<th>comment</th>
<th>checksum</th>
</tr>
</thead>
</table>

A linker generated S-record file starts with an S0 record with the following contents:

```
S0
A linker generated S-record file starts with an S0 record with the following contents:

\[ lkarm \]
S0800006C6B61726DE0
```

The S0 record is a comment record and does not contain relevant information for program execution.

where:

- **S0** is a comment record and does not contain relevant information for program execution.
- **length** represents the number of bytes in the record, not including the record type and length byte. This value occupies one byte (two hexadecimal digits).
- **comment** contains the name of the linker.
- **checksum** is the record checksum. The linker computes the checksum by first adding the binary representation of the bytes following the record type (starting with the length byte) to just before the checksum. Then the one's complement is calculated of this sum. The least significant byte of the result is the checksum. The sum of all bytes following the record type is 0xFF.

**S1 / S2 / S3 - record**

This record is the program code and data record for 2-byte, 3-byte or 4-byte addresses respectively.

<table>
<thead>
<tr>
<th>S1</th>
<th>length</th>
<th>address</th>
<th>code bytes</th>
<th>checksum</th>
</tr>
</thead>
<tbody>
<tr>
<td>S2</td>
<td>length</td>
<td>address</td>
<td>code bytes</td>
<td>checksum</td>
</tr>
<tr>
<td>S3</td>
<td>length</td>
<td>address</td>
<td>code bytes</td>
<td>checksum</td>
</tr>
</tbody>
</table>

where:

- **S1** is the program code and data record for 2-byte addresses.
- **S2** is the program code and data record for 3-byte addresses.
- **S3** is the program code and data record for 4-byte addresses (this is the default).
- **length** represents the number of bytes in the record, not including the record type and length byte. This value occupies one byte (two hexadecimal digits).
- **address** contains the code or data address.
- **code bytes** contains the actual program code and data.
- **checksum** is the record checksum. The checksum calculation is identical to S0.
S7 / S8 / S9 - record

This record is the termination record for 4-byte, 3-byte or 2-byte addresses respectively.

<table>
<thead>
<tr>
<th>S7</th>
<th>length</th>
<th>address</th>
<th>checksum</th>
</tr>
</thead>
<tbody>
<tr>
<td>S8</td>
<td>length</td>
<td>address</td>
<td>checksum</td>
</tr>
<tr>
<td>S9</td>
<td>length</td>
<td>address</td>
<td>checksum</td>
</tr>
</tbody>
</table>

where:

S7 is the termination record for 4-byte addresses (this is the default). S7 is the corresponding termination record for S3 records.

S8 is the termination record for 3-byte addresses. S8 is the corresponding termination record for S2 records.

S9 is the termination record for 2-byte addresses. S9 is the corresponding termination record for S1 records.

length represents the number of bytes in the record, not including the record type and length byte. This value occupies one byte (two hexadecimal digits).

address contains the program start address.

checksum is the record checksum. The checksum calculation is identical to S0.

Example:

S70500000000FA

| | | checksum |
| | address |
| | length |
Chapter 17. 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.

17.1. 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 vector table.

This specification is normally written by TASKING. TASKING 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 17.4, 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.

When you build an ASIC or use a derivative that is not (yet) supported by the TASKING tools, you may have to write a derivative definition.
When you want to use multiple cores of the same type, you must instantiate the cores in a derivative definition, since the linker automatically instantiates only a single core for an unused architecture.

See Section 17.5, Semantics of the Derivative Definition for a detailed description of LSL in the derivative definition.

The processor definition

The processor definition describes an instance of a derivative. Typically the processor definition instantiates one derivative only (single-core processor). A processor that contains multiple cores having the same (homogeneous) or different (heterogeneous) architecture can also be described by instantiating multiple derivatives of the same or different types in separate processor definitions.

See Section 17.6, Semantics of the Board Specification for a detailed description of LSL in the processor definition.

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 17.6.3, 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 17.8, *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
}
```

**17.2. 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.

**17.2.1. Preprocessing**

When the linker loads an LSL file, the linker first processes it with a C-style preprocessor. As such, it strips C and C++ comments. Lines starting with the `#` character are taken as commands for the preprocessor. You can use the standard ISO C99 preprocessor directives, including:

```
#include "file"
#include <file>
```

Preprocess and include file `file` at this point in the LSL file.

For example:
#include "arch.lsl"

Preprocess and include the file arch.lsl at this point in the LSL file.

#if condition
#else
#endif

If the condition evaluates to a non-zero value, copy the following lines, up to an #else or #endif command, skip lines between #else and #endif, if present. If the condition evaluates to zero, skip the lines up to the #else command, or #endif if no #else is present, and copy the lines between the #else and #endif commands.

#ifdef identifier
#else
#endif

Same as #if, but with defined(identifier) as condition.

#error text

Causes a fatal error the given message (optional).

17.2.2. 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 '//'.

### 17.2.3. Identifiers and Tags

```
arch_name ::= IDENTIFIER
bus_name ::= IDENTIFIER
core_name ::= IDENTIFIER
derivative_name ::= IDENTIFIER
file_name ::= DQSTRING
group_name ::= IDENTIFIER
heap_name ::= section_name
map_name ::= IDENTIFIER
mem_name ::= IDENTIFIER
proc_name ::= IDENTIFIER
section_name ::= DQSTRING
space_name ::= IDENTIFIER
stack_name ::= section_name
symbol_name ::= DQSTRING
tag_attr ::= (tag<tag>^0)
tag ::= tag = DQSTRING
```

A tag is an arbitrary text that can be added to a statement.

### 17.2.4. Expressions

The expressions and operators in this section work the same as in ISO C.

```
number ::= OCT_NUM
          | DEC_NUM
          | HEX_NUM

expr ::= number
       | symbol_name
       | unary_op expr
       | expr binary_op expr
       | expr ? expr : expr
       | ( expr )
       | function_call

unary_op ::= !   // logical NOT
           | ~    // bitwise complement
           | -    // negative value
```
binary_op ::= ^ // exclusive OR
    | * // multiplication
    | / // division
    | % // modulus
    | + // addition
    | - // subtraction
    | >> // right shift
    | << // left shift
    | == // equal to
    | != // not equal to
    | > // greater than
    | < // less than
    | >= // greater than or equal to
    | <= // less than or equal to
    | & // bitwise AND
    | | // bitwise OR
    | && // logical AND
    | || // logical OR

17.2.5. Built-in Functions

function_call ::= absolute ( expr )
    | addressof ( addr_id )
    | checksum ( checksum_algo, expr, expr )
    | exists ( section_name )
    | max ( expr, expr )
    | min ( expr, expr )
    | sizeof ( size_id )

addr_id ::= sect : section_name
    | group : group_name
    | mem : mem_name

checksum_algo ::= crc32w

size_id ::= sect : section_name
    | group : group_name
    | mem : mem_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.

• The checksum() function can only be used in a struct statement.

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 offset of `addr_id`, which is a named section, group, or memory in the address space of the section layout. If the referenced object is a group or memory, it must be defined in the LSL file. To get the offset of the section with the name `asect`:

```c
addressof( sect: "asect")
```

This function only works in assignments and `struct` statements.

**checksum()**

```c
int checksum( checksum_algo, expr, expr )
```

Returns the computed checksum over a contiguous address range. The first argument specifies how the checksum must be computed (see below), the second argument is an expression that represents the start address of the range, while the third argument represents the end address (exclusive). The value of the end address expression must be strictly larger than the value of the start address (i.e. the size of the checksum address range must be at least one MAU). Each address in the range must point to a valid memory location. Memory locations in the address range that are not occupied by a section are filled with zeros.

The only checksum algorithm (`checksum_algo`) currently supported is `crc32w`. This algorithm computes the checksum using a Cyclic Redundancy Check with the "CRC-32" polynomial 0xEDB88320. The input range is processed per 4-byte word. Those 4 bytes are passed to the checksum algorithm in reverse order if the target architecture is little-endian. For big-endian targets, this checksum algorithm is equal to a regular byte-wise CRC-32 implementation. Both the start address and end address values must be aligned on 4 MAUs. The behavior of this checksum algorithm is undefined when used in an address space that has a MAU size not equal to 8.

```c
checksum( crc32w,
    addressof( mem:foo ),
    addressof( mem:foo ) + sizeof( mem:foo ) )
```

This function only works in `struct` statements.

**exists()**

```c
int exists( section_name )
```
The function returns 1 if the section \texttt{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 \texttt{mysection} exists in one of the object files that is specified to the linker:

\texttt{exists( "mysection" )}

\textbf{max()}

\texttt{int max( expr, expr )}

Returns the value of the expression that has the largest value. To get the highest value of two symbols:

\texttt{max( "sym1" , "sym2" )}

\textbf{min()}

\texttt{int min( expr, expr )}

Returns the value of the expression that has the smallest value. To get the lowest value of two symbols:

\texttt{min( "sym1" , "sym2" )}

\textbf{sizeof()}

\texttt{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":

\texttt{sizeof( sect: "asection" )}

The \texttt{group} and \texttt{sect} arguments only work in assignments and \texttt{struct} statements. The \texttt{mem} argument can be used anywhere in section layouts. If the referenced object is a group or memory, it must be defined in the LSL file.

\textbf{17.2.6. LSL Definitions in the Linker Script File}

\texttt{description ::= <definition>^1}

\texttt{definition ::= architecture\_definition}

\texttt{| derivative\_definition}

\texttt{| board\_spec}

\texttt{| section\_definition}

\texttt{| section\_setup}

\textbullet{} \texttt{At least one architecture\_definition must be present in the LSL file.}
17.2.7. Memory and Bus Definitions

\[\text{mem_def} \ ::= \ \text{memory} \ \text{mem_name} <\text{tag_attr}>^{0|1} \{ \ <\text{mem_descr} \ ;\}^{>0} \]

- A `mem_def` defines a memory with the `mem_name` as a unique name.

\[\text{mem_descr} \ ::= \ \text{type} = <\text{reserved}>^{0|1} \ \text{mem_type} \]
\[\quad \text{mau} = \text{expr} \]
\[\quad \text{size} = \text{expr} \]
\[\quad \text{speed} = \text{number} \]
\[\quad \text{priority} = \text{number} \]
\[\quad \text{exec_priority} = \text{number} \]
\[\quad \text{fill} = <\text{fill_values}>^{0|1} \]
\[\quad \text{mapping} \]

- 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 `priority` (or `speed`) statement (if absent, the default value is 1).
- A `mem_def` contains zero or one `exec_priority` statement.
- A `mem_def` contains zero or one `fill` statement.
- A `mem_def` contains at least one `mapping`

\[\text{mem_type} \ ::= \ \text{rom} \quad \text{ram} \quad \text{nvram} \quad \text{blockram} \]

\[\text{fill_values} \ ::= \ \text{expr} \]
\[\quad [\text{expr}, \text{expr}]^{>0} \]

\[\text{bus_def} \ ::= \ \text{bus} \ \text{bus_name} \{ \ <\text{bus_descr} \ ;\}^{>0} \]

- A `bus_def` statement defines a bus with the given `bus_name` as a unique name within a core architecture.

\[\text{bus_descr} \ ::= \ \text{mau} = \text{expr} \]
\[\quad \text{width} = \text{expr} \quad \text{mapping} \]
\[\quad \text{of data bits} \]
\[\quad \text{legal destination} \]
\[\quad \text{'bus' only} \]

- 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 \texttt{dest = bus}: ).

\[
\text{mapping} ::= \text{map} <\text{map}\_\text{name}>^{0|1} ( \text{map}\_\text{descr} <, \text{map}\_\text{descr}>^{>0} )
\]

\[
\text{map}\_\text{descr} ::= \text{dest} = \text{destination} \\
\text{dest}\_\text{dbits} = \text{range} \\
\text{dest}\_\text{offset} = \text{expr} \\
\text{size} = \text{expr} \\
\text{src}\_\text{dbits} = \text{range} \\
\text{src}\_\text{offset} = \text{expr} \\
\text{reserved} \\
\text{priority} = \text{number} \\
\text{exec}\_\text{priority} = \text{number} \\
\text{tag}
\]

A \text{map}\_\text{descr} requires at least the \text{size} and \text{dest} statements.

A \text{map}\_\text{descr} contains zero or one \text{priority} statement (if absent, the default value is 0).

A \text{map}\_\text{descr} contains zero or one \text{exec}\_\text{priority} statement.

Each \text{map}\_\text{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 \text{src}\_\text{dbits} or \text{dest}\_\text{dbits} statement is not present, its value defaults to the \text{width} value if the source/destination is a bus, and to the \text{mau} size otherwise.

The \text{reserved} statement is allowed only in mappings defined for a memory.

\[
\text{destination} ::= \text{space} : \text{space}\_\text{name} \\
\text{bus} : <\text{proc}\_\text{name} | \text{core}\_\text{name} >^{0|1} \text{bus}\_\text{name}
\]

A \text{space}\_\text{name} refers to a defined address space.

A \text{proc}\_\text{name} refers to a defined processor.

A \text{core}\_\text{name} refers to a defined core.

A \text{bus}\_\text{name} refers to a defined bus.

The following mappings are allowed (source to destination)

- \text{space} => \text{space}
- \text{space} => \text{bus}
- \text{bus} => \text{bus}
- memory => bus

\[ \text{range} := \text{expr} \ldots \text{expr} \]

- With address ranges, the end address is not part of the range.

### 17.2.8. Architecture Definition

\[
\text{architecture_definition} := \text{architecture } \text{arch_name} \\
\quad \langle ( \text{parameter_list} ) \rangle^{0|1} \\
\quad \langle \text{extends } \text{arch_name} \\
\quad \quad \langle ( \text{argument_list} ) \rangle^{0|1} >^{0|1} \rangle^{0|1} \\
\quad \{ \langle \text{arch_spec} \rangle^{=0} \}^{=0} \}
\]

- 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.

\[
\text{parameter_list} := \text{parameter } <, \text{ parameter}>^{=0}
\]

\[
\text{parameter} := \text{IDENTIFIER } = \text{expr}^{0|1}
\]

\[
\text{argument_list} := \text{expr } <, \text{ expr}>^{=0}
\]

\[
\text{arch_spec} := \text{bus_def} \\
\quad | \text{space_def} \\
\quad | \text{endianness_def}
\]

\[
\text{space_def} := \text{space } \text{space_name} <\text{tag_attr}>^{0|1} \{ <\text{space_descr};>^{=0} \}
\]

- A space_def defines an address space with the given space_name as a unique name within an architecture.

\[
\text{space_descr} := \text{space_property} ; \\
\quad | \text{section_definition} //\text{no space ref} \\
\quad | \text{vector_table_statement} \\
\quad | \text{reserved_range}
\]

\[
\text{space_property} := \text{id} = \text{number} //\text{as used in object} \\
\quad | \text{mau} = \text{expr} \\
\quad | \text{align} = \text{expr} \\
\quad | \text{page_size} = \text{expr} <[\text{range}] <[\text{range}]>^{=0}>^{0|1} \\
\quad | \text{page} \\
\quad | \text{direction} = \text{direction} \\
\quad | \text{stack_def}
\]
A `space_def` contains exactly one `id` and one `mau` statement.

A `space_def` contains at most one `align` statement.

A `space_def` contains at most one `page_size` statement.

A `space_def` contains at least one `mapping`.

```
heap_def ::= heap heap_name ( heap_descr <, heap_descr >>=0 )
```

A `heap_def` defines a heap with the `heap_name` as a unique name.

```
stack_def ::= stack stack_name ( stack_descr <, stack_descr >>=0 )
```

A `stack_def` defines a stack with the `stack_name` as a unique name.

```
stack_descr ::= min_size = expr
| grows = direction
| align = expr
| fixed
| threads = expr
| entry_points = entry_point_list
| attributes
| tag
```

```
entry_point_list ::= symbol_name
| [ symbol_name <, symbol_name >>=0 ]
```

The `min_size` statement must be present.

The `min_size` value must be 1 or greater.

You can specify at most one `align` statement, one `grows` statement and one `threads` statement.

Each stack definition can have one or more `entry_points` statements for stack estimation. The `symbol_name` corresponds to the caller name in the `.CALLS` directive as generated by the compiler.

```
heap_def ::= heap heap_name ( heap_descr <, heap_descr >>=0 )
```

A `heap_def` defines a heap with the `heap_name` as a unique name.

```
heap_descr ::= min_size = expr
| grows = direction
| align = expr
| fixed
| attributes
| tag
```

The `min_size` statement must be present.
• The **min_size** value must be 1 or greater.

• You can specify at most one **align** statement and one **grows** statement.

\[
direction ::= \text{low-to-high} \\
| \text{high-to-low}
\]

• If you do not specify the **grows** statement, the stack and heap grow **low-to-high**.

\[
copy_table_def ::= \text{copytable} <( \text{copy_table_descr} <, \text{copy_table_descr} >=0 )>=0|1
\]

• A **space_def** contains at most one **copytable** statement.

• Exactly one copy table must be defined in one of the spaces.

\[
copy_table_descr ::= \text{align} = \text{expr} \\
| \text{copy_unit} = \text{expr} \\
| \text{dest} <\text{space_name}> = \text{space_name} \\
| \text{page} \\
| \text{tag}
\]

• The **copy_unit** is defined by the size in MAUs in which the startup code moves data.

• The **dest** statement is only required when the startup code initializes memory used by another processor that has no access to ROM.

• A **space_name** refers to a defined address space.

\[
start_addr ::= \text{start_address} ( \text{start_addr_descr} <, \text{start_addr_descr} >=0 )
\]

\[
start_addr_descr ::= \text{run_addr} = \text{expr} \\
| \text{symbol} = \text{symbol_name}
\]

• A **symbol_name** refers to the section that contains the startup code.

\[
vector_table_statement ::= \text{vector_table} \text{section_name} \langle \text{vecttab_spec} <, \text{vecttab_spec} >=0 \rangle \langle \text{vector_def} >=0 \rangle
\]

\[
vecttab_spec ::= \text{vector_size} = \text{expr} \\
| \text{size} = \text{expr} \\
| \text{id_symbol_prefix} = \text{symbol_name} \\
| \text{run_addr} = \text{addr_absolute} \\
| \text{template} = \text{section_name} \\
| \text{template_symbol} = \text{symbol_name} \\
| \text{vector_prefix} = \text{section_name} \\
| \text{fill} = \text{vector_value} \\
| \text{no_inline} \\
| \text{copy} \\
| \text{tag}
\]
vector_def ::= vector ( vector_spec <, vector_spec>^0 ) ;

vector_spec ::= id = vector_id_spec
               | fill = vector_value
               | optional
               | tag

vector_id_spec ::= number
                | [ range ] <, [ range ]>^0

vector_value ::= symbol_name
               | [ number <, number]^0 ]
               | loop <[ expr ]>^0|1

reserved_range ::= reserved <tag_attr>^0|1 expr .. expr ;

- The end address is not part of the range.

endianness_def ::= endianness { <endianness_type;>^1 }
endianness_type ::= big
                  | little

17.2.9. Derivative Definition

derivative_definition ::= derivative derivative_name
                         (( parameter_list )>^0|1
                         <extends derivative_name (( argument_list )>^0|1
                         , derivative_name (( argument_list )>^0|1>^0
                         , derivative_name (( argument_list )>^0|1>^0
                         { <derivative_spec>^0

- A derivative_definition defines a derivative with the given derivative_name as a unique name.

derivative_spec ::= core_def
                   | bus_def
                   | mem_def
                   | section_definition // no processor name
                   | section_setup

core_def ::= core core_name { <core_descr ;>^0 }

- A core_def defines a core with the given core_name as a unique name.
- At least one core_def must be present in a derivative_definition.

core_descr ::= architecture = arch_name
              (( argument_list )>^0|1
              | endianness = ( endianness_type
                            , endianness_type>^0 )
• An arch_name refers to a defined core architecture.

• Exactly one architecture statement must be present in a core_def.

17.2.10. Processor Definition and Board Specification

```
import core_name
space_id_offset = number
```

• 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.

17.2.11. Section Setup

```
section_setup ::= section_setup space_ref <tag_attr>^0|^1
{ <section_setup_item>^0|^1 }

section_setup_item ::= vector_table_statement
| reserved_range
| stack_def ;
| heap_def ;
| copy_table_def ;
| start_address ;
| modify linktime_modification

linktime_modification ::= input ( input_modifier <,input_modifier>^0 )
{ <select_section_statement ; >^0 }

input_modifier ::= space = subtable_space_ref
| attributes = < <+|-> attribute>^0
| copy
```

• An input_modifier contains at most one space statement.
• An input Modifier contains at most one attributes statement.

### 17.2.12. Section Layout Definition

```plaintext
section_definition ::= section_layout <space_ref>^0|1
  <( space_layout_properties )>^0|1
  { <section_statement>^>=0 }
```

• A section definition inside a space definition does not have a space_ref.

• All global section definitions have a space_ref.

```plaintext
space_ref ::= <proc_name>^0|1 : <core_name>^0|1 :
  space_name < | space_name^>=0
```

• 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.

```plaintext
space_layout_properties ::= space_layout_property <, space_layout_property >=0
```

```plaintext
space_layout_property ::= locate_direction
  | tag
```

```plaintext
locate_direction ::= direction = direction
```

```plaintext
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.

```plaintext
section_statement ::= simple_section_statement ;
  | aggregate_section_statement
```

```plaintext
simple_section_statement ::= assignment
  | select_section_statement
  | special_section_statement
  | memcpy_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
       |  tag

• +attribute means: select all sections that have this attribute.

• -attribute means: select all sections that do not have this attribute.

special_section_statement
 ::=  heap  heap_name <stack_heap_mods>^0|1
       |  stack  stack_name <stack_heap_mods>^0|1
       |  copytable
       |  reserved  section_name <reserved_specs>^0|1

• Special sections cannot be selected in load-time groups.

stack_heap_mods  ::=  (  stack_heap_mod  <,  stack_heap_mod>^>0 )

stack_heap_mod  ::=  size  =  expr
                  |  tag

reserved_specs  ::=  (  reserved_spec  <,  reserved_spec>^>0 )

reserved_spec  ::=  attributes
                  |  fill_spec
                  |  size  =  expr
                  |  alloc_allowed  =  absolute  |  ranged

• 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).

memcpy_statement
 ::=  memcpy  section_name
       (  memcpy_spec  <,  memcpy_spec>^0|1 )

memcpy_spec  ::=  memory  =  memory_reference
                |  fill_spec
• A `memcopy` statement must contain exactly one `memory` statement.

• A `memcopy` statement can contain at most one `fill_spec`.

```
fll_spec ::= fill = fill_values
fill_values ::= expr
  | [ expr <, expr]>=0 ]
```

```
aggregate_section_statement ::= { <section_statement>=0 } 
  | group_descr 
  | if_statement 
  | section_creation_statement 
  | struct_statement
```

```
group_descr ::= group <group_name>=0|1 <( group_specs )>=0|1
  section_statement
```

• For every group with a name, the linker defines a label.

• No two groups for address spaces of a core 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 
  | tag
```

• The `allow-cross-references` property is only allowed for `overlay` groups.

• The `copy` and `nocopy` properties cannot be applied both to the same group.

• 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
```
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.

• The end address is not part of the range.

memory_reference ::= mem : <proc_name >0|1 mem_name </ map_name>0|1

• A proc_name refers to a defined processor.

• A mem_name refers to a defined memory.

• A map_name refers to a defined memory mapping.

if_statement ::= if ( expr ) section_statement
<else section_statement>=0|1

section_creation_statement ::= section section_name ( section_specs )
{ <section_statement2>=0 }

b  // blanked (cleared) sections
p  // protected sections

Linker Script Language (LSL)
**17.3. 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. Symbol references are only allowed in symbol assignments and `struct` statements.
17.4. Semantics of the Architecture Definition

Keywords in the architecture definition

architecture
  extends
endianness          big  little
bus
  mau
  width
  map
space
  id
  mau
  align
  page_size
  page
direction        low_to_high  high_to_low
stack
  min_size
  grows         low_to_high  high_to_low
  align
  fixed
  threads
  entry_points
  attributes    b
heap
  min_size
  grows         low_to_high  high_to_low
  align
  fixed
  attributes    b
copytable
  align
  copy_unit
dest
  page
vector_table
  vector_size
size
  id_symbol_prefix
run_addr
template
template_symbol
vector_prefix
fill
no_inline
copy
vector
17.4.1. 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:

```LSL
architecture name
{
    definitions
}
```

If you are defining multiple core architectures that show great resemblance, you can define the common features in a parent core architecture and extend this with a child core architecture that contains specific features. The child inherits all features of the parent. With the keyword `extends` you create a child core architecture:

```LSL
architecture name_child_arch extends name_parent_arch
{
    definitions
}
```

A core architecture can have any number of parameters. These are identifiers which get values assigned on instantiation or extension of the architecture. You can use them in any expression within the core architecture. Parameters can have default values, which are used when the core architecture is instantiated with less arguments than there are parameters defined for it. When you extend a core architecture you can pass arguments to the parent architecture. Arguments are expressions that set the value of the parameters of the sub-architecture.

```LSL
architecture name_child_arch (parm1, parm2=1)
    extends name_parent_arch (arguments)
{
    definitions
}
```
17.4.2. 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 and must be non-zero.

- 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 17.4.4, *Mappings*.

```
bus  bus_name
{
    mau  =  8;
    width  =  8;
    map  ( map_description );
}
```

17.4.3. 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 and must be non-zero.

- 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 17.8.2, *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 17.4.4, *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 17.8.3, *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.

A stack may have an **attributes** property with value **b**. Such a stack must be cleared at program startup. No other attributes are allowed.

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.

For each stack, a stack size estimation may be computed (and listed in a map file) from a call graph. Each root node of the call graph is treated as a separate thread that can run independently from the other threads. Root nodes can be specified using the **entry_points** keyword. The estimated stack usage for a root node is the highest sum of stack usage values along a path to a leaf node. The total estimated stack usage of a link task is the sum of the calculated stack usage of such independent call graphs. If only a limited number of these threads can make use of a specific stack at a time, you can specify this by assigning a number to the **threads** keyword on that stack's definition. When **threads** is set to *n*, only the *n* highest stack usage numbers of root nodes are summed. A **threads** argument equal to zero or negative is ignored.

A stack definition may have one or more **entry_points** statements that specify the code that uses that stack - all functions that are reachable (by calls) from the entry points are considered to be using the stack at run-time. Each symbol name specified as an entry point must match a node in the call graph, which must not have more than one caller. As a result of the entry point specification, the specific call (edge) is removed from the call graph (so the symbol becomes a root). A symbol may be declared as entry point for multiple stacks.

In the LSL file the value of the **threads** argument can be determined by the macro **__STACK_THREADS**. By default the macro is set to 2. Defining this macro adds a **threads** keyword to **stack**.

In the LSL file the value of the **entry_points** argument can be determined by the macro **__STACK_ENTRY_POINTS**. By default the macro is set to "__START".

```c
#ifndef __STACK_THREADS
#   if defined(__CPU_ARMV7M__) || defined(__CPU_ARMV7EM__) || defined(__CPU_ARMV6M__)
#       define __STACK_THREADS             2
```

832
# endif
#endif
#define __STACK_ENTRY_POINTS  "__START"

stack "stack"
{
    #ifdef __STACK_FIXED
        fixed,
    #endif
    #ifdef __STACK_THREADS
        threads = __STACK_THREADS,
    #endif
    entry_points = __STACK_ENTRY_POINTS,
    align = 4,
    min_size = __STACK,
    grows = high_to_low
};

- 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 17.8.3, 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 17.8, 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 17.8.2, 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.

```
vector_table "vector_table" (vector_size=m, size=n, run_addr=x, ...)
```

See the following example of a vector table definition:

```
vector_table "vector_table" (vector_size = 4, size = 16, run_addr=0,
    template=".text.handler_address",
    template_symbol="_lc_vector_handler",
    vector_prefix="_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", optional);
    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.

When the keyword **optional** is set on a vector specification with a symbol value and the symbol is 
not found, no error is reported. A default fill value is used if the symbol was not found. With other values 
the attribute has no effect.

**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 17.8.3, *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.

```plaintext
space space_name
{
    id = 1;
    mau = 8;
    align = 8;
}```
17.4.4. 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).

A mapping can optionally have a name which can be referenced in an address assignment.

If you define a memory and the memory mapping must not be used by default when locating sections in address spaces, you can specify the reserved argument. This marks all address space areas that the mapping points to as reserved. If a section has an absolute or address range restriction, the reservation...
is lifted and the section may be located at these locations. This feature is only useful when more than one mapping is available for a range of memory addresses, otherwise the `memory` keyword with the same name would be used.

For example:

```lsl
memory xrom
{
  mau = 8;
  size = 1M;
  type = rom;
  map cached (dest=bus:mycore:local_bus, dest_offset=0x80000000,
               size=1M);
  map uncached (dest=bus:mycore:local_bus, dest_offset=0xa0000000,
                 size=1M, reserved);
}
```

**Mapping priority**

If you define a memory you can set a locate priority on a mapping with the keywords `priority` and `exec_priority`. The values of these priorities are relative which means they add to the priority of memories. Whereas a priority set on the memory applies to all address space areas reachable through any mapping of the memory, a priority set on a mapping only applies to address space areas reachable through the mapping. The memory mapping with the highest priority is considered first when locating. To set only a priority for non-executable (data) sections, add a `priority` keyword with the desired value and an `exec_priority` set to zero. To set only a priority for executable (code) sections, simply set an `exec_priority` keyword to the desired value.

The default for a mapping `priority` is zero, while the default for `exec_priority` is the same as the specified `priority`. If you specify a value for `priority` in LSL it must be greater than zero. A value for `exec_priority` must be greater or equal to zero.

For more information about priority values see the description of the `memory priority` keyword.

```lsl
memory myram
{
  mau = 8;
  size = 112k;
  type = ram;
  map (dest=bus:mycore:local_bus, dest_offset=0xd0000000,
       size=112k, priority=8, exec_priority=0);
  map (dest=bus:mycore:local_bus, dest_offset=0x70000000,
       size=112k);
}
```

**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 KiB 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.

architecture mycore
{
    bus i_bus
    {
        mau = 4;
    }

    space i_space
    {
        map (dest=bus:i_bus, size=256);
    }
}

bus e_bus
{
    mau = 16;
    width = 16;
    map (dest = bus:mycore:i_bus, src_dbits = 0..7, dest_dbits = 0..7 )
}

It is not possible to map an internal bus to an external bus.
17.5. Semantics of the Derivative Definition

Keywords in the derivative definition

derivative
   extends
core
   architecture
   import
   space_id_offset
bus
   mau
   width
   map
memory
   type             reserved rom  ram  nvram  blockram
   mau
   size
   speed
   priority
   exec_priority
   fill
   map
section_layout
section_setup

map
   dest        bus  space
   dest_dbits
   dest_offset
   size
   src_dbits
   src_offset
   priority
   exec_priority
   reserved

17.5.1. 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 one or more parent derivatives 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:

```c
derivative name_child_deriv extends name_parent_derivs
{
    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.

```c
derivative name_child_deriv (parm1, parm2=1)
    extends name_parent_deriv (arguments)
{
    definitions
}
```

### 17.5.2. 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:

  ```c
  core mycore_1
  {
      architecture = mycorearch;
  }
  
  core mycore_2
  {
      architecture = mycorearch;
  }
  ```

  If the architecture definition has parameters you must specify the arguments that correspond with the parameters. For example `mycorearch1` expects two parameters which are used in the architecture definition:

  ```c
  core mycore
  {
      architecture = mycorearch1 (1,2);
  }
  ```

- With the keyword `import` you can combine multiple cores with the same architecture into a single link task. The imported cores share a single symbol namespace.
• The address spaces in each imported core must have a unique ID in the link task. With the keyword `space_id_offset` you specify for each imported core that the space IDs of the imported core start at a specific offset.

```plaintext
core mycore_1
{
    architecture = mycorearch;
    space_id_offset = 100; // add 100 to all space IDs in
                           // the architecture definition
}
core mycore_2
{
    architecture = mycorearch;
    space_id_offset = 200; // add 200 to all space IDs in
                           // the architecture definition
}
core mycore
{
    architecture = mycorearch;
    import mycore_1; // add all address spaces of mycore_1 for linking
    import mycore_2; // add all address spaces of mycore_2 for linking
}
```

### 17.5.3. Defining Internal Memory and Buses

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. 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 17.6.3, 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
  - `blockram`: writing is possible both at load-time and run-time. Changes are applied in RAM, so after a full device reset the data in a blockram reverts to the original state.

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 17.8.2, Creating and Locating Groups of Sections).

- The `mau` field specifies the MAU size (Minimum Addressable Unit) of the memory. This field is required and must be non-zero.
• The size field specifies the size in MAU of the memory. This field is required.

• The priority field specifies a locate priority for a memory. The speed field has the same meaning but is considered deprecated. By default, a memory has its priority set to 1. The memories with the highest priority are considered first when trying to locate a rule. Subsequently, the next highest priority memories are added if the rule was not located successfully, and so on until the lowest priority that is available is reached or the rule is located. The lowest priority value is zero. Sections with an ordered and/or contiguous restriction are not affected by the locate priority. If such sections also have a page restriction, the locate priority is still used to select a page.

• If an exec_priority is specified for a memory, the regular priority (either specified or its default value) does not apply to locate rules with only executable sections. Instead, the supplied value applies for such rules. Additionally, the exec_priority value is used for any executable unrestricted sections, even if they appear in an unrestricted rule together with non-executable sections.

• The map field specifies how this memory maps onto an (internal) bus. The mapping can have a name. Mappings are described in Section 17.4.4, Mappings.

• 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.

```plaintext
memory mem_name
{
    type = rom;
    mau = 8;
    fill = 0xaa;
    size = 64k;
    priority = 2;
    map map_name ( map_description );
}
```

With the bus keyword you define a bus in a derivative definition. Buses are described in Section 17.4.2, Defining Internal Buses.

### 17.6. Semantics of the Board Specification

#### Keywords in the board specification

processor
derivative
bus
mau
width
map
memory
type reserved rom ram nvram blockram
mau
size
17.6.1. 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:

```
processor proc_name
{
    processor definition
}
```

17.6.2. 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:

```
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);
}

17.6.3. 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;
    fill = 0xaa;
    size = 64k;
    priority = 2;
    map map_name ( map_description );
}

For a description of the keywords, see Section 17.5.3, 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 17.4.2, 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.

17.7. Semantics of the Section Setup Definition

Keywords in the section setup definition

```plaintext
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
    copytable
        align
        copy_unit
        dest
        page
    vector_table
        vector_size
        size
        id_symbol_prefix
        run_addr
        template
        template_symbol
        vector_prefix
        fill
        no_inline
        copy
        vector
        id
        fill  loop
        optional
    reserved
    start_address
        run_addr
        symbol
    modify_input
    space
```
17.7.1. Setting up a Section

With the keyword `section_setup` you can define stacks, heaps, copy tables, vector tables, start address and/or reserved address ranges outside their address space definition. In addition you can configure input section modifications.

```plaintext
section_setup :::my_space
{
  vector_table_statements
  reserved_address_range
  stack_definition
  heap_definition
  copy_table_definition
  start_address
  input_section_modifications
}
```

See the subsections Stacks and heaps, Copy tables, Start address, Vector table and Reserved address ranges in Section 17.4.3, Defining Address Spaces for details on the keywords `stack`, `heap`, `copytable`, `vector_table` and `reserved`.

Input section modifications

Before sections are located and before selections defined in `section_layout` are performed, you can still modify a few section properties. These are:

- change the address space of a section
- add (+w) or remove (−w) the writable attribute
- add (+p) or remove (−p) the protected attribute
- add (+u) or remove (−u) the uncached attribute

You cannot set the protected attribute on linker created sections like reserved sections and output sections.

Sections are selected the same way as in groups in a `section_layout`. Instead of `attributes=+w` you can use the `copy` keyword.

```plaintext
section_setup :::A
{
  modify input (space=:::B, attributes=+w)
  {
    select "mysection";
  }
}
```

Note that the new address space must be used to select a modified section in a `section_layout`. To locate the section `mysection` in the example somewhere, it must be selected in a `section_layout`
for space : $\texttt{::B}$. If the link result is output to a file, for example by only linking or incremental linking, the modified properties are exported. So, when the resulting file is used in another invocation of the linker, the section can appear in a different address space.

17.8. 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 p
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 ranged
copytable
memcopy
  memory
fill
section
size
blocksize
attributes    r w x
fill
overflow
```
17.8.1. 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
}
```

Clone sections

Clone sections are placed in special address spaces that are characterized by a set of regular address spaces. Those spaces each have different “local” memory across some common address range. The clone space is created in the “main” core of the task, not in a core that is imported. For a section layout, the clone space is specified by replacing the simple space name by a list of full names for the “real” spaces. These names are separated by vertical bars (`|`), and the colons in the full names are replaced by underscores. The processor name (including the following underscore) can be omitted from the space names in the list if it is equal to the clone space's processor, and the order of spaces in the list is not important. The next example shows a definition for a multi-core ARM with clone sections in address space linear:

```plaintext
section_layout ARM:vm7:cm7_0_linear|cm7_1_linear
{
    section statements
}
```

Locate direction

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 )
{
```
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.

17.8.2. 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.

```linker_script
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 17.8.3, 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'

```linker_script
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.

When you use wildcards, the linker skips sections with an absolute address from the selection process, for example, a start section already having an absolute start address.

Note that when you select sections with an exact name (no wildcards), all sections with that name are automatically protected against unreferenced section removal. With a selection using wildcards, matching sections are selected, but matching sections that are unreferenced may be removed.

- 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)
  - p protected sections

To select all read-only sections:

```c

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:
group refgrp (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**

```linker_script_language

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 sections in a 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 sections in a 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 according to the align value. The alignment of a section is first determined by its own initial alignment and the defined alignment for the address space. Alignments are never decreased, if multiple alignments apply to a section, the largest one is used.

   • 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. You cannot apply both `copy` and `nocopy` to the same statement.

2. **Define the mutual order of sections in an LSL 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.
Note that when you use the linker optimization option `--optimize=+copytable-compression`, unrestricted sections affected by the copy table are located as if they were in a clustered LSL group. This option is enabled by default.

- 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 in a group can be restricted either to a single absolute address, or to a number of address ranges (not including the end address). With an expression you can specify that the group should be located at the absolute address specified by the expression:

  group (ordered, run_addr = 0xa00f0000)

  A group with an absolute address must be ordered, the first section in the group is located at the specified absolute address.

  You can use the `[offset]` variant to locate the group at the given absolute offset in memory:

  group (ordered, run_addr = mem:A[0x1000])

  A group with an absolute address must be ordered, the first section in the group is located at the specified absolute offset in memory.

A range can be an absolute space address range, written as `[expr .. expr]`, a complete memory device, written as `mem:mem_name`, or a memory address range, `mem:mem_name[expr .. expr]`

  group (run_addr = mem:my_dram)

  You can use the `|` to specify an address range of more than one physical memory device:

  group (run_addr = mem:A | mem:B)
When used in top-level section layouts, a memory name refers to a board-level memory. You can select on-chip memory with `mem:proc_name:mem_name`. If the memory has multiple parallel mappings towards the current address space, you can select a specific named mapping in the memory by appending `/map_name` to the memory specifier. The linker then maps memory offsets only through that mapping, so the address(es) where the sections in the group are located are determined by that memory mapping.

```plaintext
group (run_addr = mem:CPU1:A/cached)
```

- 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.

- If an ordered group or sequential group has an absolute address and contains sections that have separate page restrictions (not defined in LSL), all those sections are located in a single page. In other cases, for example when an unrestricted group has an address range assigned to it, the paged sections may be located in different pages.

For overlays, the linker reserves memory at the run-time start address as large as the largest element in the overlay group.

- The `page` keyword tells the linker to place the group in one page. Instead of specifying a run-time address, you can specify a page and optional a page number. Page numbers start from zero. If you omit the page number, the linker chooses a page.

  The `page` keyword refers to pages in the address space as defined in the architecture definition.

- 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 17.4.3, *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`.

```plaintext
group (priority=2)
{
  select "importantcode1";
  select "importantcode2";
}
```

### 17.8.3. 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`.

```plaintext
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 17.4.3, *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`.

```plaintext
```
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. The same applies for reserved sections with `alloc_allowed=ranged` set. Sections restricted to a fixed address range can also 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, nocopy and load_addr properties and the load_addr property cannot have an address specified.

The fill field contains a bit pattern that the linker writes to all unused space in the output section. When all input sections have initialized code or data you must specify a fill pattern. If you do not specify a fill pattern, all input sections must be scratch sections (not cleared and not initialized), or BSS sections. The fill pattern is aligned at the start of the output section.

In the following example, the sections myinput1 and myinput2 are assumed to have initialized data, so the fill keyword is needed on the output section.

As with a reserved section you can use the attributes field to set the access type of the output section.

group ( ... )
{
    section "myoutput" ( size = 4k, attributes = r,
        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.

In the following example, the sections .data.tsk1.* and .data.tsk2.* do not contain initialized data, so the fill keyword should not be used on the output section.

group ( ... )
{
    section "tsk1_data" (size=4k, attributes=rw,
overflow = "overflow_data")} 
} 
section "tsk2_data" (size=4k, attributes=rw, 
overflow = "overflow_data")} 
} 
section "overflow_data" (size=4k, attributes=rw) 

With the keyword **blocksize**, the size of the output section will adapt to the size of its content. For example:

```c
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 KiB, if the content is 11 KiB, the section will be 12 KiB, 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.

When the **copy** property is set on an enclosing group, a ROM copy is created for the output section and the output section itself is made writable causing it to be located in RAM by default. For this to work, the output section and its input sections must be read-only and the output section must have a **fill** property.

A copy table can also be inserted into an output section, but only if two additional conditions are met:

- The copy table is the last section added to the output section.
- There must be sufficient room in the output section to accommodate the additional size of the copy table.

A copy table will likely increase in size after being added to the output section, so if you would add sections after the copy table selection, this would overwrite part of the copy table. The linker will emit an error message if either of the conditions is not met.
section "myoutput_tbl" { size = 4k, attributes = r, fill = 0
{
    select "myinput";
    select "table"; // select the copy table

}

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.

Memory copy sections

• If a memory (usually RAM) needs to be initialized by a different core than the one(s) that will use it, a copy of the contents of the memory can be placed in a section using a memcpy statement in a section_layout. All data (including code) present in the specified memory is then placed in a new section with the provided name and appropriate attributes. Unused areas in the memory are filled in the section using the supplied fill pattern or with zeros if no fill pattern is specified. If the memory contains a memory copy section the result is undefined. The actual initialization of the memory at run-time needs to be done separately, this LSL feature only directs the linker to make the data located in the memory available for initialization. Note that a memory of type ram cannot hold initialized data, use type blockram instead.

Structures

• A struct statement in a section_layout creates a section and fills it with numbers that each occupy one or more MAUs. The new section must be named by providing a double-quoted string after the struct keyword. Each element has the form expr: number; where the expression provides the value to insert in the section and the number determines the number of MAUs occupied by the expression value. Elements are placed in the section in the order in which they appear in the struct body without any gaps between them. Multi-MAU elements are split into MAUs according to the endianness of the target. A struct section is read-only and it cannot be copied to RAM at startup (using the copy group attribute). No default alignment is set.

For example,

struct "mystruct"
{
    0x1234 : 2;
    addressof( mem:foo ) : 4;
    addressof( mem:foo ) + sizeof( mem:foo ) : 4;
    checksum( crc32w,
        addressof( mem:foo ),

}
17.8.4. 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.

```
section_layout
{
    "_lc_bs" := "_lc_ub_stack";
    // when the symbol _lc_bs occurs as an undefined reference
    // in an object file, the linker allocates space for the stack
}
```

17.8.5. 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).

```
group ( ... )
{
    if ( exists( "mysection" ) )
        select "mysection";
    else
        reserved "myreserved" ( size=2k );
}
```
Chapter 18. 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 installation directory or a vendor specific Eclipse plugin (install-dir\eclipse\plugins\com.tasking.arm.vendors.vendor_version\etc) 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.

18.1. 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 vendor specific etc directory, 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 Import 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 and use the XML Schema file dtc.xsd, also present in the vendor specific etc directory.

Inspect the DTC XML schema file dtc.xsd for a description of the allowed elements and the available attributes. 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 configurations (configuration element) and one or more communication methods (communicationMethod element). The Import 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: STMicroelectronics STM3210B-Eval board
   lsl
   configuration: Default
   lsl
If, in the Import Board Configuration wizard in Eclipse, you set the board configuration to "J-Link-ARM", 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` "J-Link over USB" element
- the lsl part under the `configuration"J-Link-ARM" in the communicationMethod "J-Link over USB" 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.

```c
#define __DTC_IGNORE
```

### 18.2. 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>name</td>
<td>A descriptive name of the processor derivative.</td>
<td></td>
</tr>
<tr>
<td>cpu</td>
<td>Defines the CPU name, as for example supplied with the <code>option --cpu</code> 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>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 communication method.</td>
<td>rs232, tcpip, can, none</td>
</tr>
</tbody>
</table>
| debugInstrument     | The debug instrument DLL/Shared library file to be used for this communication method. Do not supply a path or a filename suffix. | | }
| gdiMethod           | This is the method used for communication. Allowed values: rs232, tcpip, can, none | | }
| def                 | Defines a set of elements as a macro. The macro can be expanded using the ref element. | Isl, resource, initialize, ref, configuration, flashMonitor |
| id                  | The macro name. | | }
| resource            | Defines a resource definition that can be used by Eclipse, the debugger or by the debug instrument. | | }
| id                  | The identifier name used by the debugger or debug instrument to retrieve the value. | | }
| value               | The value assigned to the resource. | | }
| ref                 | 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. | | }
| id                  | The name of the referenced macro. | | }
| configuration       | Defines a configuration. | Isl, flashMonitor, processor |
| name                | The descriptive name of the configuration. | | }
| initialize          | 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. | | }
| resourceId          | The name of the resource to be used. | | }
| name                | The name of the register to be initialized. | | }
| value               | 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. | | }

---

**Debug Target Configuration Files**

---

863
### 18.3. Special Resource Identifiers

The following resource IDs are available in the TASKING VX-toolset for ARM:

**J-Link debug instrument (DI): dijlinkarm**

<table>
<thead>
<tr>
<th>Resource Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arm.luminary.clockrate</td>
<td>Clock rate in Hz. Necessary for the initialization of register USECRL.</td>
</tr>
</tbody>
</table>
### ST-Link debug instrument (DI): distlink

<table>
<thead>
<tr>
<th>Resource Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>distlink.protocol</td>
<td>The interface for using ST-Link. Can be JTAG or SWD.</td>
</tr>
</tbody>
</table>
Chapter 19. CPU Problem Bypasses and Checks

ARM publishes errata sheets for reporting both CPU core functional problems and deviations from the electrical and timing specifications.

For some of these functional problems in the CPU core itself, the TASKING VX-toolset for ARM compiler provides workarounds. In fact these are software workarounds for hardware problems.

This chapter lists a summary of functional problems which can be bypassed by the compiler toolset. Please refer to the ARM errata sheets for the CPU core 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 for**
   
   *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 bug numbers you can provide to the compiler and assembler option **--silicon-bug**. WA means a workaround by the compiler, WL means a workaround by the linker, CK means a check by the assembler.

<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
<th>Workaround</th>
<th>Check</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>602117</td>
<td>LDRD with base in list may result in incorrect base register when interrupted or faulted</td>
<td>WA</td>
<td>CK</td>
<td>Cortex-M3 / Cortex-M3 with ETM</td>
</tr>
<tr>
<td>752419</td>
<td>LDR SP, mem may result in incorrect SP when interrupted</td>
<td>WA</td>
<td>CK</td>
<td>Cortex-M3 / Cortex-M3 with ETM / Cortex-M4 / Cortex-M4 with FPU</td>
</tr>
<tr>
<td>760170</td>
<td>-submit-</td>
<td>WL</td>
<td></td>
<td>XMC4000 family</td>
</tr>
<tr>
<td></td>
<td>The use of BX LR for return from interrupt may trigger a prefetch problem</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
602117 -- LDRD with base in list may result in incorrect base register when interrupted or faulted

ARM reference

602117

Command line option

--silicon-bug=602117

Description

In the Cortex-M3 or Cortex-M3 with ETM the LDRD instruction with the base register in the list of the form LDRD Ra, Rb, [Ra, #imm] may not complete after the load of the first destination register due to an interrupt before the completion of the second load or due to the second load getting a bus fault or an MPU fault. When you use the C compiler option --silicon-bug=602117 the compiler will replace the LDRD instruction by an ADD instruction and two LDR instructions which will produce exactly the same functionality:

ADD R12, Ra, #imm
LDR Ra, [R12, #0]
LDR Rb, [R12, #4]

When you use the assembler option --silicon-bug=602117, the assembler issues a warning when the 602117 problem is present.

Related information

ARM Core Cortex-M3 - Errata Notice
752419 / 752770 -- LDR SP, mem may result in incorrect SP when interrupted

ARM reference

752419 and 752770

Command line option

--silicon-bug=752419
--silicon-bug=752770

Description

In the Cortex-M3, Cortex-M3 with ETM Cortex-M4 and Cortex-M4 with FPU interrupted loads to SP can cause erroneous behavior. In the compiler the problem only occurs when the __setsp() intrinsic is used, or when VLAs are used (via the __free() intrinsic). When you use the C compiler option --silicon-bug=752419 the compiler generates alternate code in these instances and enough optimizations are blocked for these specific bits of code so that the suspected instructions should not be admitted.

When you use the assembler option --silicon-bug=752419, the assembler issues a warning when the 752419 problem is present. It is recommended to enable the assembler check for this silicon problem as well to be sure the suspected instructions will never be generated.

Option --silicon-bug=752770 is an alias for --silicon-bug=752419.

Related information

ARM Core Cortex-M3 - Errata Notice
ARM Core Cortex-M4 - Errata Notice
pmc_cm_001 -- The use of BX LR for return from interrupt may trigger a prefetch problem

Infineon reference
PMU_CM.001

Command line option
--silicon-bug=pmc_cm_001

Description
In the XMC4000 family a branch from a non-cacheable to a cacheable address space instruction may corrupt the program execution. When you use the control program option --silicon-bug=pmc_cm_001 the control program passes the option -DSILICON_BUG_PMC_CM_001 to the linker. To bypass this CPU functional problem, the preprocessor define SILICON_BUG_PMC_CM_001 is used in the xmc4*.lsl linker script files. The vector table is defined in such a way that the interrupt functions are called via a veneer which returns by using PUSH LR, POP PC instead of BX LR.

Related information
Infineon XMC4000 Documents - Errata Sheet
Chapter 20. CERT C Secure Coding Standard

The CERT C Secure Coding Standard provides rules and recommendations for secure coding in the C programming language. The goal of these rules and recommendations is to eliminate insecure coding practices and undefined behaviors that can lead to exploitable vulnerabilities. The application of the secure coding standard will lead to higher-quality systems that are robust and more resistant to attack.

This chapter contains an overview of the CERT C Secure Coding Standard recommendations and rules that are supported by the TASKING toolset.

For details see the CERT C Secure Coding Standard web site. For general information about CERT secure coding, see www.cert.org/secure-coding.

Identifiers

Each rule and recommendation is given a unique identifier. These identifiers consist of three parts:

- a three-letter mnemonic representing the section of the standard
- a two-digit numeric value in the range of 00-99
- the letter "C" indicates that this is a C language guideline

The three-letter mnemonic is used to group similar coding practices and to indicate to which category a coding practice belongs.

The numeric value is used to give each coding practice a unique identifier. Numeric values in the range of 00-29 are reserved for recommendations, while values in the range of 30-99 are reserved for rules.

C compiler invocation

With the C compiler option --cert you can enable one or more checks for the CERT C Secure Coding Standard recommendations/rules. With --diag=cert you can see a list of the available checks, or you can use a three-letter mnemonic to list only the checks in a particular category. For example, --diag=pre lists all supported checks in the preprocessor category.

20.1. Preprocessor (PRE)

PRE01-C Use parentheses within macros around parameter names

Parenthesize all parameter names in macro definitions to avoid precedence problems.
Macro replacement lists should be parenthesized

Macro replacement lists should be parenthesized to protect any lower-precedence operators from the surrounding expression. The example below is syntactically correct, although the \(!=\) operator was omitted. Enclosing the constant \(-1\) in parenthesis will prevent the incorrect interpretation and force a compiler error:

```c
#define EOF -1  // should be (-1)
int getchar(void);
void f(void)
{
    if (getchar() EOF) // != operator omitted
    {
        /* ... */
    }
}
```

Wrap multi-statement macros in a do-while loop

When multiple statements are used in a macro, enclose them in a \texttt{do-while} statement, so the macro can appear safely inside \texttt{if} clauses or other places that expect a single statement or a statement block. Braces alone will not work in all situations, as the macro expansion is typically followed by a semicolon.

Do not conclude a single statement macro definition with a semicolon

Macro definitions consisting of a single statement should not conclude with a semicolon. If required, the semicolon should be included following the macro expansion. Inadvertently inserting a semicolon can change the control flow of the program.

20.2. Declarations and Initialization (DCL)

Declare objects with appropriate storage durations

The lifetime of an automatic object ends when the function returns, which means that a pointer to the object becomes invalid.

Declare identifiers before using them

The ISO C90 standard allows implicit typing of variables and functions. Because implicit declarations lead to less stringent type checking, they can often introduce unexpected and erroneous behavior or even security vulnerabilities. The ISO C99 standard requires type identifiers and forbids implicit function declarations. For backwards compatibility reasons, the TASKING C compiler assumes an implicit declaration and continues translation after issuing a warning message (W505 or W535).
**DCL32-C** Guarantee that mutually visible identifiers are unique

The compiler encountered two or more identifiers that are identical in the first 31 characters. The ISO C99 standard allows a compiler to ignore characters past the first 31 in an identifier. Two distinct identifiers that are identical in the first 31 characters may lead to problems when the code is ported to a different compiler.

**DCL35-C** Do not invoke a function using a type that does not match the function definition

This warning is generated when a function pointer is set to refer to a function of an incompatible type. Calling this function through the function pointer will result in undefined behavior. Example:

```c
void my_function(int a);
int main(void)
{
  int (*new_function)(int a) = my_function;
  return (*new_function)(10); /* the behavior is undefined */
}
```

### 20.3. Expressions (EXP)

**EXP01-C** Do not take the size of a pointer to determine the size of the pointed-to type

The size of the object(s) allocated by malloc(), calloc() or realloc() should be a multiple of the size of the base type of the result pointer. Therefore, the sizeof expression should be applied to this base type, and not to the pointer type.

**EXP12-C** Do not ignore values returned by functions

The compiler gives this warning when the result of a function call is ignored at some place, although it is not ignored for other calls to this function. This warning will not be issued when the function result is ignored for all calls, or when the result is explicitly ignored with a (void) cast.

**EXP30-C** Do not depend on order of evaluation between sequence points

Between two sequence points, an object should only be modified once. Otherwise the behavior is undefined.

**EXP32-C** Do not access a volatile object through a non-volatile reference

If an attempt is made to refer to an object defined with a volatile-qualified type through use of an lvalue with non-volatile-qualified type, the behavior is undefined.

**EXP33-C** Do not reference uninitialized memory

Uninitialized automatic variables default to whichever value is currently stored on the stack or in the register allocated for the variable. Consequently, uninitialized memory can cause a program to behave in an unpredictable or unplanned manner and may provide an avenue for attack.
EXP34-C  Ensure a null pointer is not dereferenced

Attempting to dereference a null pointer results in undefined behavior, typically abnormal program termination.

EXP37-C  Call functions with the arguments intended by the API

When a function is properly declared with function prototype information, an incorrect call will be flagged by the compiler. When there is no prototype information available at the call, the compiler cannot check the number of arguments and the types of the arguments. This message is issued to warn about this situation.

EXP38-C  Do not call offsetof() on bit-field members or invalid types

The behavior of the offsetof() macro is undefined when the member designator parameter designates a bit-field.

20.4. Integers (INT)

INT30-C  Ensure that unsigned integer operations do not wrap

A constant with an unsigned integer type is truncated, resulting in a wrap-around.

INT34-C  Do not shift a negative number of bits or more bits than exist in the operand

The shift count of the shift operation may be negative or greater than or equal to the size of the left operand. According to the C standard, the behavior of such a shift operation is undefined. Make sure the shift count is in range by adding appropriate range checks.

INT35-C  Evaluate integer expressions in a larger size before comparing or assigning to that size

If an integer expression is compared to, or assigned to a larger integer size, that integer expression should be evaluated in that larger size by explicitly casting one of the operands.

20.5. Floating Point (FLP)

FLP30-C  Do not use floating point variables as loop counters

To avoid problems with limited precision and rounding, floating point variables should not be used as loop counters.

FLP35-C  Take granularity into account when comparing floating point values

Floating point arithmetic in C is inexact, so floating point values should not be tested for exact equality or inequality.

FLP36-C  Beware of precision loss when converting integral types to floating point

Conversion from integral types to floating point types without sufficient precision can lead to loss of precision.
20.6. Arrays (ARR)

ARR01-C Do not apply the sizeof operator to a pointer when taking the size of an array

A function parameter declared as an array, is converted to a pointer by the compiler. Therefore, the sizeof operator applied to this parameter yields the size of a pointer, and not the size of an array.

ARR34-C Ensure that array types in expressions are compatible

Using two or more incompatible arrays in an expression results in undefined behavior.

ARR35-C Do not allow loops to iterate beyond the end of an array

Reading or writing of data outside the bounds of an array may lead to incorrect program behavior or execution of arbitrary code.

20.7. Characters and Strings (STR)

STR30-C Do not attempt to modify string literals

Writing to a string literal has undefined behavior, as identical strings may be shared and/or allocated in read-only memory.

STR33-C Size wide character strings correctly

Wide character strings may be improperly sized when they are mistaken for narrow strings or for multi-byte character strings.

STR34-C Cast characters to unsigned types before converting to larger integer sizes

A signed character is sign-extended to a larger signed integer value. Use an explicit cast, or cast the value to an unsigned type first, to avoid unexpected sign-extension.

STR36-C Do not specify the bound of a character array initialized with a string literal

The compiler issues this warning when the character buffer initialized by a string literal does not provide enough room for the terminating null character.

20.8. Memory Management (MEM)

MEM00-C Allocate and free memory in the same module, at the same level of abstraction

The compiler issues this warning when the result of the call to malloc(), calloc() or realloc() is discarded, and therefore not free()d, resulting in a memory leak.

MEM08-C Use realloc() only to resize dynamically allocated arrays

Only use realloc() to resize an array. Do not use it to transform an object to an object of a different type.
MEM30-C Do not access freed memory

When memory is freed, its contents may remain intact and accessible because it is at the memory manager’s discretion when to reallocate or recycle the freed chunk. The data at the freed location may appear valid. However, this can change unexpectedly, leading to unintended program behavior. As a result, it is necessary to guarantee that memory is not written to or read from once it is freed.

MEM31-C Free dynamically allocated memory exactly once

Freeing memory multiple times has similar consequences to accessing memory after it is freed. The underlying data structures that manage the heap can become corrupted. To eliminate double-free vulnerabilities, it is necessary to guarantee that dynamic memory is freed exactly once.

MEM32-C Detect and handle memory allocation errors

The result of realloc() is assigned to the original pointer, without checking for failure. As a result, the original block of memory is lost when realloc() fails.

MEM33-C Use the correct syntax for flexible array members

Use the ISO C99 syntax for flexible array members instead of an array member of size 1.

MEM34-C Only free memory allocated dynamically

Freeing memory that is not allocated dynamically can lead to corruption of the heap data structures.

MEM35-C Allocate sufficient memory for an object

The compiler issues this warning when the size of the object(s) allocated by malloc(), calloc() or realloc() is smaller than the size of an object pointed to by the result pointer. This may be caused by a sizeof expression with the wrong type or with a pointer type instead of the object type.

20.9. Environment (ENV)

ENV32-C All atexit handlers must return normally

The compiler issues this warning when an atexit() handler is calling a function that does not return. No atexit() registered handler should terminate in any way other than by returning.

20.10. Signals (SIG)

SIG30-C Call only asynchronous-safe functions within signal handlers

SIG32-C Do not call longjmp() from inside a signal handler

Invoking the longjmp() function from within a signal handler can lead to undefined behavior if it results in the invocation of any non-asynchronous-safe functions, likely compromising the integrity of the program.
20.11. Miscellaneous (MSC)

MSC32-C  Ensure your random number generator is properly seeded

Ensure that the random number generator is properly seeded by calling srand().
Chapter 21. MISRA C Rules

This chapter contains an overview of the supported and unsupported MISRA C rules.

21.1. MISRA C:1998

This section lists all supported and unsupported MISRA C:1998 rules.

See also Section 4.6.2, 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. (A) Other languages should only be used with an interface standard.
3. (A) Inline assembly is only allowed in dedicated C functions.
4. (A) Provision should be made for appropriate run-time checking.
5. (R) Only use characters and escape sequences defined by ISO C.
6. (R) Character values shall be restricted to a subset of ISO 106460-1.
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.
typedef names shall not be reused.

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.

Octal constants (other than zero) shall not be used.

All object and function identifiers shall be declared before use.

Identifiers shall not hide identifiers in an outer scope.

Declarations should be at function scope where possible.

All declarations at file scope should be static where possible.

Identifiers shall not have both internal and external linkage.

Identifiers with external linkage shall have exactly one definition.

Multiple declarations for objects or functions shall be compatible.

External objects should not be declared in more than one file.

The register storage class specifier should not be used.

The use of a tag shall agree with its declaration.

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.

Braces shall be used in the initialization of arrays and structures.

Only the first, or all enumeration constants may be initialized.

The right hand operand of && or || shall not contain side effects.

The operands of a logical && or || shall be primary expressions.

Assignment operators shall not be used in Boolean expressions.

Logical operators should not be confused with bitwise operators.

Bitwise operations shall not be performed on signed integers.

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.

The unary minus shall not be applied to an unsigned expression.

sizeof should not be used on expressions with side effects.

The implementation of integer division should be documented.

The comma operator shall only be used in a for condition.

Don't use implicit conversions which may result in information loss.

Redundant explicit casts should not be used.

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.

48. (A) Mixed arithmetic should use explicit casting.

49. (A) Tests of a (non-Boolean) value against 0 should be made explicit.

50. (R) F.P. variables shall not be tested for exact equality or inequality.

51. (A) Constant unsigned integer expressions should not wrap-around.

52. (R) There shall be no unreachable code.

53. (R) All non-null statements shall have a side-effect.

54. (R) A null statement shall only occur on a line by itself.

55. (A) Labels should not be used.

56. (R) The goto statement shall not be used.

57. (R) The continue statement shall not be used.

58. (R) The break statement shall not be used (except in a switch).

59. (R) An if or loop body shall always be enclosed in braces.

60. (A) All if, else if constructs should contain a final else.

61. (R) Every non-empty case clause shall be terminated with a break.

62. (R) All switch statements should contain a final default case.

63. (A) A switch expression should not represent a Boolean case.

64. (R) Every switch shall have at least one case.

65. (R) Floating-point variables shall not be used as loop counters.

66. (A) A for should only contain expressions concerning loop control. A violation is reported when the loop initialization or loop update expression modifies an object that is not referenced in the loop test.

67. (A) Iterator variables should not be modified in a for loop.

68. (R) Functions shall always be declared at file scope.

69. (R) Functions with variable number of arguments shall not be used.

70. (R) Functions shall not call themselves, either directly or indirectly. A violation will be reported for direct or indirect recursive function calls in the source file being checked. Recursion via functions in other source files, or recursion via function pointers is not detected.

71. (R) Function prototypes shall be visible at the definition and call.

72. (R) The function prototype of the declaration shall match the definition.

73. (R) Identifiers shall be given for all prototype parameters or for none.

74. (R) Parameter identifiers shall be identical for declaration/definition.

75. (R) 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.

const should be used for reference parameters not modified.

A function should have a single point of exit.

Every exit point shall have a return of the declared return type.

For void functions, return shall not have an expression.

Function calls with no parameters should have empty parentheses.

If a function returns error information, it should be tested. A violation is reported when the return value of a function is ignored.

#include shall only be preceded by other directives or comments.

Non-standard characters shall not occur in #include directives.

#include shall be followed by either <filename> or "filename".

Plain macros shall only be used for constants/qualifiers/specifiers.

Macros shall not be #define'd and #undef'd within a block.

#undef should not be used.

A function should be used in preference to a function-like macro.

A function-like macro shall not be used without all arguments.

Macro arguments shall not contain pre-preprocessing directives. A violation is reported when the first token of an actual macro argument is '#'.

Macro definitions/parameters should be enclosed in parentheses.

Don't use undefined identifiers in pre-processing directives.

A macro definition shall contain at most one # or ## operator.

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.

defined shall only be used in one of the two standard forms.

Pointer arithmetic should not be used.

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.

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.

Non-constant pointers to functions shall not be used.

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.
   x means that the rule is not supported by the TASKING C compiler. (R) is a required rule, (A) is an advisory
   rule.
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.

21.2. MISRA C:2004

This section lists all supported and unsupported MISRA C:2004 rules.

See also Section 4.6.2, 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.
Environment


1.2 (R) No reliance shall be placed on undefined or unspecified behavior.

1.3 (R) 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.

1.4 (R) The compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers.

1.5 (A) 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.
An identifier with external linkage shall have exactly one external definition. (R)8.9

All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required. (R)8.10

The static storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage. (R)8.11

When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization. (R)8.12

**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. (R)9.1

Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures. (R)9.2

In an enumerator list, the "=” construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized. (R)9.3

**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 of the same signedness that is no wider than the underlying type of the expression.

The value of a complex expression of floating type may only be cast to a type that is no wider than the underlying type of the expression.

If the bitwise operators ~ and << are applied to an operand of underlying type unsigned char or 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 unsigned type. (R)10.6
Pointer type conversions

11.1 (R) Conversions shall not be performed between a pointer to a function and any type other than an integral type.

11.2 (R) 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.

11.3 (A) 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 `const` or `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 `sizeof` operator shall not be used on expressions that contain side effects.

12.4 (R) The right-hand operand of a logical `&&` or `||` operator shall not contain side effects.

12.5 (R) The operands of a logical `&&` or `||` shall be `primary-expressions`.

12.6 (A) The operands of logical operators (`&&`, `||` and `!`) should be effectively Boolean. Expressions that are effectively Boolean should not be used as operands to operators other than (`&&`, `||` and `!`).

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 (`++`) and decrement (`--`) 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 for statement shall not contain any objects of floating type.

13.5 (R) 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.
16.2 (R) 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.
16.3 (R) Identifiers shall be given for all of the parameters in a function prototype declaration.
16.4 (R) The identifiers used in the declaration and definition of a function shall be identical.
16.5 (R) Functions with no parameters shall be declared with parameter type void.
16.6 (R) The number of arguments passed to a function shall match the number of parameters.
16.7 (A) 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.
16.8 (R) All exit paths from a function with non-void return type shall have an explicit return statement with an expression.
16.9 (R) A function identifier shall only be used with either a preceding & or with a parenthesized parameter list, which may be empty.
16.10 (R) 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

x 17.1 (R) Pointer arithmetic shall only be applied to pointers that address an array or array element.
x 17.2 (R) Pointer subtraction shall only be applied to pointers that address elements of the same array.
17.3 (R) >, >=, <, <= 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.
17.4 (R) Array indexing shall be the only allowed form of pointer arithmetic.
17.5 (A) 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.
17.6 (R) 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

18.1 (R) All structure or union types shall be complete at the end of a translation unit.
18.2 (R) An object shall not be assigned to an overlapping object.
x 18.3 (R) An area of memory shall not be reused for unrelated purposes.
18.4 (R) 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

x 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.

21.3. MISRA C:2012

This section lists all supported and unsupported MISRA C:2012 rules.

See also Section 4.6.2, 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. (M) is a mandatory rule, (R) is a required rule, (A) is an advisory rule.

A standard C environment

1.1 (R) The program shall contain no violations of the standard C syntax and constraints, and shall not exceed the implementation’s translation limits.
1.2 (A) Language extensions should not be used.
1.3 (R) There shall be no occurrence of undefined or critical unspecified behavior.

Unused code

2.1 (R) A project shall not contain unreachable code.
2.2 (R) There shall be no dead code.
2.3 (A) A project should not contain unused type declarations.
A project should not contain unused tag declarations. (A) 2.4
A project should not contain unused macro declarations. (A) 2.5
A function should not contain unused label declarations. (A) 2.6
There should be no unused parameters in functions. (A) 2.7

Comments

3.1 (R) The character sequences /* and // shall not be used within a comment.
3.2 (R) Line-splicing shall not be used in // comments.

Character sets and lexical conventions

4.1 (R) Octal and hexadecimal escape sequences shall be terminated.
4.2 (A) Trigraphs should not be used.

Identifiers

5.1 (R) External identifiers shall be distinct.
5.2 (R) Identifiers declared in the same scope and name space shall be distinct.
5.3 (R) An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.
5.4 (R) Macro identifiers shall be distinct.
5.5 (R) Identifiers shall be distinct from macro names.
5.6 (R) A typedef name shall be a unique identifier.
5.7 (R) A tag name shall be a unique identifier.
5.8 (R) Identifiers that define objects or functions with external linkage shall be unique.
5.9 (A) Identifiers that define objects or functions with internal linkage should be unique.

Types

6.1 (R) Bit-fields shall only be declared with an appropriate type.
6.2 (R) Single-bit named bit-fields shall not be of a signed type.

_literals and constants

7.1 (R) Octal constants shall not be used.
7.2 (R) A "u" or "U" suffix shall be applied to all integer constants that are represented in an unsigned type.
7.3 (R) The lowercase character "l" shall not be used in a literal suffix trivial.
7.4 (R) A string literal shall not be assigned to an object unless the object's type is "pointer to const-qualified char".
Declarations and definitions

8.1 (R) Types shall be explicitly specified.
8.2 (R) Function types shall be in prototype form with named parameters.
8.3 (R) All declarations of an object or function shall use the same names and type qualifiers.
8.4 (R) A compatible declaration shall be visible when an object or function with external linkage is defined.
8.5 (R) An external object or function shall be declared once in one and only one file.
8.6 (R) An identifier with external linkage shall have exactly one external definition.
8.7 (A) Functions and objects should not be defined with external linkage if they are referenced in only one translation unit.
8.8 (R) The \textit{static} storage class specifier shall be used in all declarations of objects and functions that have internal linkage.
8.9 (A) An object should be defined at block scope if its identifier only appears in a single function.
8.10 (R) An inline function shall be declared with the \textit{static} storage class.
8.11 (A) When an array with external linkage is declared, its size should be explicitly specified.
8.12 (R) Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique.
8.13 (A) A pointer should point to a \textit{const}-qualified type whenever possible.
8.14 (R) The \textit{restrict} type qualifier shall not be used.

Initialization

9.1 (M) The value of an object with automatic storage duration shall not be read before it has been set.
9.2 (R) The initializer for an aggregate or union shall be enclosed in braces.
9.3 (R) Arrays shall not be partially initialized.
9.4 (R) An element of an object shall not be initialized more than once.
9.5 (R) Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly.

The essential type model

10.1 (R) Operands shall not be of an inappropriate essential type.
10.2 (R) Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations.
10.3 (R) The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category.
10.4 (R) Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category.
10.5 (A) The value of an expression should not be cast to an inappropriate essential type.
10.6 (R) The value of a composite expression shall not be assigned to an object with wider essential type.
10.7 (R) If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type.
10.8 (R) The value of a composite expression shall not be cast to a different essential type category or a wider essential type.

**Pointer type conversions**

11.1 (R) Conversions shall not be performed between a pointer to a function and any other type.
11.2 (R) Conversions shall not be performed between a pointer to an incomplete type and any other type.
11.3 (R) A cast shall not be performed between a pointer to object type and a pointer to a different object type.
11.4 (A) A conversion should not be performed between a pointer to object and an integer type.
11.5 (A) A conversion should not be performed from pointer to `void` into pointer to object.
11.6 (R) A cast shall not be performed between pointer to `void` and an arithmetic type.
11.7 (R) A cast shall not be performed between pointer to object and a non-integer arithmetic type.
11.8 (R) A cast shall not remove any `const` or `volatile` qualification from the type pointed to by a pointer.
11.9 (R) The macro `NULL` shall be the only permitted form of integer null pointer constant.

**Expressions**

12.1 (A) The precedence of operators within expressions should be made explicit.
12.2 (R) The right hand operand of a shift operator shall lie in the range zero to one less than the width in bits of the essential type of the left hand operand.
12.3 (A) The comma operator should not be used.
12.4 (A) Evaluation of constant expressions should not lead to unsigned integer wrap-around.
12.5 (M) The `sizeof` operator shall not have an operand which is a function parameter declared as "array of type".

**Side effects**

13.1 (R) Initializer lists shall not contain persistent side effects.
13.2 (R) The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders.
13.3 (A) A full expression containing an increment (++) or decrement (--) operator should have no other potential side effects other than that caused by the increment or decrement operator.

13.4 (A) The result of an assignment operator should not be used.

13.5 (R) The right hand operand of a logical && or || operator shall not contain persistent side effects.

13.6 (M) The operand of the sizeof operator shall not contain any expression which has potential side effects.

**Control statement expressions**

14.1 (R) A loop counter shall not have essentially floating type.

14.2 (R) A for loop shall be well-formed.

14.3 (R) Controlling expressions shall not be invariant.

14.4 (R) The controlling expression of an if statement and the controlling expression of an iteration-statement shall have essentially Boolean type.

**Control flow**

15.1 (A) The goto statement should not be used.

15.2 (R) The goto statement shall jump to a label declared later in the same function.

15.3 (R) Any label referenced by a goto statement shall be declared in the same block, or in any block enclosing the goto statement.

15.4 (A) There should be no more than one break or goto statement used to terminate any iteration statement.

15.5 (A) A function should have a single point of exit at the end.

15.6 (R) The body of an iteration-statement or a selection-statement shall be a compound-statement.

15.7 (R) All if ... else if constructs shall be terminated with an else statement.

**Switch statements**

16.1 (R) All switch statements shall be well-formed.

16.2 (R) A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.

16.3 (R) An unconditional break statement shall terminate every switch-clause.

16.4 (R) Every switch statement shall have a default label.

16.5 (R) A default label shall appear as either the first or the last switch label of a switch statement.

16.6 (R) Every switch statement shall have at least two switch-clauses.

16.7 (R) A switch-expression shall not have essentially Boolean type.
Functions

17.1 (R) The features of `<stdarg.h>` shall not be used.
17.2 (R) Functions shall not call themselves, either directly or indirectly.
17.3 (M) A function shall not be declared implicitly.
17.4 (M) All exit paths from a function with non-void return type shall have an explicit `return` statement with an expression.
17.5 (A) The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements.
17.6 (M) The declaration of an array parameter shall not contain the `static` keyword between the `[ ]`.
17.7 (R) The value returned by a function having non-void return type shall be used.
17.8 (A) A function parameter should not be modified.

Pointers and arrays

18.1 (R) A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand.
18.2 (R) Subtraction between pointers shall only be applied to pointers that address elements of the same array.
18.3 (R) The relational operators `>`, `>=`, `<` and `<=` shall not be applied to objects of pointer type except where they point into the same object.
18.4 (A) The `+`, `−`, `+=` and `−=` operators should not be applied to an expression of pointer type.
18.5 (A) Declarations should contain no more than two levels of pointer nesting.
18.6 (R) The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist.
18.7 (R) Flexible array members shall not be declared.
18.8 (R) Variable-length array types shall not be used.

Overlapping storage

19.1 (M) An object shall not be assigned or copied to an overlapping object.
19.2 (A) The `union` keyword should not be used.

Preprocessing directives

20.1 (A) `#include` directives should only be preceded by preprocessor directives or comments.
20.2 (R) The ``, " or `\` characters and the `/*` or `//` character sequences shall not occur in a header file name.
20.3 (R) The `#include` directive shall be followed by either a `<filename>` or "filename" sequence.
A macro shall not be defined with the same name as a keyword.

#undef should not be used.

Tokens that look like a preprocessing directive shall not occur within a macro argument.

Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses.

The controlling expression of a #if or #elif preprocessing directive shall evaluate to 0 or 1.

All identifiers used in the controlling expression of #if or #elif preprocessing directives shall be #define'd before evaluation.

The # and ## preprocessor operators should not be used.

A macro parameter immediately following a # operator shall not immediately be followed by a ## operator.

A macro parameter used as an operand to the # or ## operators, which is itself subject to further macro replacement, shall only be used as an operand to these operators.

A line whose first token is # shall be a valid preprocessing directive.

All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if, #ifdef or #ifndef directive to which they are related.

**Standard libraries**

#define and #undef shall not be used on a reserved identifier or reserved macro name.

A reserved identifier or macro name shall not be declared.

The memory allocation and deallocation functions of <stdlib.h> shall not be used.

The standard header file <setjmp.h> shall not be used.

The standard header file <signal.h> shall not be used.

The Standard Library input/output functions shall not be used.

The atof, atoi, atol and atoll functions of <stdlib.h> shall not be used.

The library functions abort, exit and system of <stdlib.h> shall not be used.

The library functions bsearch and qsort of <stdlib.h> shall not be used.

The Standard Library time and date functions shall not be used.

The standard header file <tgmath.h> shall not be used.

The exception handling features of <fenv.h> should not be used.

Any value passed to a function in <ctype.h> shall be representable as an unsigned char or be the value EOF.

The Standard Library function memcmp shall not be used to compare null terminated strings.

The pointer arguments to the Standard Library functions memcpy, memmove and memcmp shall be pointers to qualified or unqualified versions of compatible types.
21.16 (R) The pointer arguments to the Standard Library function `memcmp` shall point to either a pointer type, an essentially signed type, an essentially unsigned type, an essentially Boolean type or an essentially enum type.

21.17 (M) Use of the string handling functions from `<string.h>` shall not result in accesses beyond the bounds of the objects referenced by their pointer parameters.

21.18 (M) The `size_t` argument passed to any function in `<string.h>` shall have an appropriate value.

21.19 (M) The pointers returned by the Standard Library functions `localeconv`, `getenv`, `setlocale` or `strerror` shall only be used as if they have pointer to const-qualified type.

21.20 (M) The pointer returned by the Standard Library functions `asctime`, `ctime`, `gmtime`, `localtime`, `localeconv`, `getenv`, `setlocale` or `strerror` shall not be used following a subsequent call to the same function.

### Resources

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>22.1 (R)</td>
<td>All resources obtained dynamically by means of Standard Library functions shall be explicitly released.</td>
</tr>
<tr>
<td>22.2 (M)</td>
<td>A block of memory shall only be freed if it was allocated by means of a Standard Library function.</td>
</tr>
<tr>
<td>22.3 (R)</td>
<td>The same file shall not be open for read and write access at the same time on different streams.</td>
</tr>
<tr>
<td>22.4 (M)</td>
<td>There shall be no attempt to write to a stream which has been opened as read-only.</td>
</tr>
<tr>
<td>22.5 (M)</td>
<td>A pointer to a FILE object shall not be dereferenced.</td>
</tr>
<tr>
<td>22.6 (M)</td>
<td>The value of a pointer to a FILE shall not be used after the associated stream has been closed.</td>
</tr>
<tr>
<td>22.7 (R)</td>
<td>The macro <code>EOF</code> shall only be compared with the unmodified return value from any Standard Library function capable of returning <code>EOF</code>.</td>
</tr>
<tr>
<td>22.8 (R)</td>
<td>The value of <code>errno</code> shall be set to zero prior to a call to an errno-setting-function.</td>
</tr>
<tr>
<td>22.9 (R)</td>
<td>The value of <code>errno</code> shall be tested against zero after calling an errno-setting-function.</td>
</tr>
<tr>
<td>22.10 (R)</td>
<td>The value of <code>errno</code> shall only be tested when the last function to be called was an errno-setting-function.</td>
</tr>
</tbody>
</table>
Chapter 22. C Implementation-defined Behavior

The TASKING C compiler for ARM fully supports the ISO C standard, but some parts of the ISO C standard are implementation-defined. This chapter describes how the implementation-defined areas and the locale-specific areas of the C language are implemented in the TASKING C compiler for ISO C99 and ISO C11. Below are some remarks on the other behaviors as mentioned in the standard.

Unspecified behavior

Unspecified behavior is the use of an unspecified value, or other behavior where the ISO-C standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance. Some of the unspecified behaviors are relevant to users of the TASKING C compiler for ARM. Some unspecified behaviors are specified in the Application Binary Interface (ABI). The silicon vendor is responsible for the ABI.

Undefined behavior

Undefined behavior is behavior, upon use of a non-portable or erroneous program construct or of erroneous data, for which the ISO C standard imposes no requirements.

Some undefined behaviors may trigger a compiler error or warning. TASKING does not provide any guarantees about whether or not the compiler issues an error or warning. It is important to know whether your software contains undefined behaviors since this will make the source non-portable between compiler vendors and between other processors.

The MISRA C and CERT coding guides do not refer to undefined behaviors explicitly.

22.1. C99 Implementation-defined Behavior

Implementation-defined behavior is unspecified behavior where each implementation documents how the choice is made.

The following sections describe the implementation-defined characteristics. The section numbers listed in parenthesis refer to the corresponding sections in the ISO C99 standard. The order in this chapter is the same as used in Annex J.3 of the ISO/IEC 9899:1999 (E) standard.

22.1.1. Translation

• How a diagnostic is identified (3.10, 5.1.1.3).

The C compiler diagnostics are explained in Section 4.7, C Compiler Error Messages.

• Whether each nonempty sequence of white-space characters other than new-line is retained or replaced by one space character in translation phase 3 (5.1.1.2).

White-space is retained.
22.1.2. Environment

- The mapping between physical source file multibyte characters and the source character set in translation phase 1 (5.1.1.2).

Use of variable length encoded characters in the source file in comments and string literals is permitted. A one-on-one mapping is done without interpretation of multibyte characters.

- The name and type of the function called at program startup in a freestanding environment (5.1.2.1).

The function called at program startup (in cstart.c) is called `main`. The prototype for `main` in cstart.c is:

```c
extern int main(int argc, char *argv[]);
```

- The effect of program termination in a freestanding environment (5.1.2.1).

Execution is halted if the program is executed under control of a debugger, otherwise the program will loop forever in function `_Exit()`.

- An alternative manner in which the `main` function may be defined (5.1.2.2.1).

You can change the definition of `main` by altering file cstart.c.

- The values given to the strings pointed to by the `argv` argument to `main` (5.1.2.2.1).

The strings get their values from the arguments given in file cstart.c. The program arguments are treated case sensitive.

- What constitutes an interactive device (5.1.2.3).

The streams `stdin`, `stdout`, and `stderr` are treated as interactive devices. The debugger uses these streams with File System Simulation (FSS) windows to interact.

- The set of signals, their semantics, and their default handling (7.14).

The signals are described in Section 14.2.18, `signal.h`.

- Signal values other than `SIGFPE`, `SIGILL`, and `SIGSEGV` that correspond to a computational exception (7.14.1.1).

There are no other values that correspond to a computational exception. All signal values are described in Section 14.2.18, `signal.h`.

- Signals for which the equivalent of `signal(sig, SIG_IGN);` is executed at program startup (7.14.1.1).

By default the implementation does not ignore any signals at program startup.
• The set of environment names and the method for altering the environment list used by the `getenv` function (7.20.4.5).

There are no implementation-defined environment names that are used by the `getenv` function. A skeleton is provided for the `getenv` function in the C library, because the embedded environment has no operating system. The `getenv` function calls the name as a `void` function.

• The manner of execution of the string by the `system` function (7.20.4.6).

A skeleton is provided for the `system` function in the C library, because the embedded environment has no operating system. The `system` function calls the string as a `void` function.

### 22.1.3. Identifiers

• Which additional multibyte characters may appear in identifiers and their correspondence to universal character names (6.4.2).

No additional multibyte characters are supported in an identifier.

• The number of significant initial characters in an identifier (5.2.4.1, 6.4.2).

All characters in an identifier are significant.

### 22.1.4. Characters

• The number of bits in a byte (3.6).

There are eight bits in a byte.

• The values of the members of the execution character set (5.2.1).

Only 8-bit characters are supported. The values of the execution character set are the same as that of the source character set. The same representation value is used for each member in the characters sets except for the escape sequences.

• The unique value of the member of the execution character set produced for each of the standard alphabetic escape sequences (5.2.2).

The following table contains an overview of the escape sequences and their byte value in the execution character set.

<table>
<thead>
<tr>
<th>Escape sequence</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>\a</td>
<td>7</td>
</tr>
<tr>
<td>\b</td>
<td>8</td>
</tr>
<tr>
<td>\f</td>
<td>12</td>
</tr>
<tr>
<td>\n</td>
<td>10</td>
</tr>
<tr>
<td>\r</td>
<td>13</td>
</tr>
<tr>
<td>\t</td>
<td>9</td>
</tr>
</tbody>
</table>
• The value of a `char` object into which has been stored any character other than a member of the basic execution character set (6.2.5).

Any 8-bit value can be stored in a `char` object.

• Which of signed `char` or unsigned `char` has the same range, representation, and behavior as "plain" `char` (6.2.5, 6.3.1.1).

By default "plain" `char` is the same as specifying unsigned `char`. With C compiler option `--schar` you can change the default to signed `char`.

• The mapping of members of the source character set (in character constants and string literals) to members of the execution character set (6.4.4.4, 5.1.1.2).

The mapping is one-to-one. The values of the execution character set are the same as that of the source character set. The same representation value is used for each member in the characters sets except for the escape sequences.

• The value of an integer character constant containing more than one character or containing a character or escape sequence that does not map to a single-byte execution character (6.4.4.4).

For any character constant containing more than one character, a warning is issued and the value is truncated to type `char`.

• The value of a wide character constant containing more than one multibyte character, or containing a multibyte character or escape sequence not represented in the extended execution character set (6.4.4.4).

A wide character constant can contain at most two multibyte characters. Its value is the concatenation of the multibyte characters represented in a signed short int.

• The current locale used to convert a wide character constant consisting of a single multibyte character that maps to a member of the extended execution character set into a corresponding wide character code (6.4.4.4).

By default, the "C" locale is used.

• The current locale used to convert a wide string literal into corresponding wide character codes (6.4.5).

By default, the "C" locale is used.

• The value of a string literal containing a multibyte character or escape sequence not represented in the execution character set (6.4.5).

All source characters can be represented in the execution character set.
22.1.5. Integers

• Any extended integer types that exist in the implementation (6.2.5).

All types are described in Section 1.1, Data Types.

• Whether signed integer types are represented using sign and magnitude, two’s complement, or one’s complement, and whether the extraordinary value is a trap representation or an ordinary value (6.2.6.2).

Signed integer types are represented in two's complement. The most significant bit is the sign bit. 1 is negative, 0 is positive.

• The rank of any extended integer type relative to another extended integer type with the same precision (6.3.1.1).

All types are described in Section 1.1, Data Types.

• The result of, or the signal raised by, converting an integer to a signed integer type when the value cannot be represented in an object of that type (6.3.1.3).

At compile time, when converting integer types and a value does not fit in a type, the compiler issues a warning and the value is truncated. At run-time no warning or signal is given and the value is truncated.

• The results of some bitwise operations on signed integers (6.5).

The result of E1>>E2 is E1 right shifted E2 bit positions. If E1 has a signed type and a negative value, the shift behavior is implemented as an arithmetic shift. The empty position in the most significant bit is filled with a copy of the original most significant bit.

22.1.6. Floating-Point

• The accuracy of the floating-point operations and of the library functions in \texttt{<math.h> and <complex.h>} that return floating-point results (5.2.4.2.2).

A float has 24 significant mantissa bits and 8 exponent bits. A double or long double has 53 significant mantissa bits and 11 exponent bits. This is conform IEEE-754 for single precision and double precision floating-point. Internally the compiler uses a mantissa of 80 bits. The results of floating-point operations are rounded to the nearest IEEE-754 format.

The accuracy of \texttt{sqrt()} and other \texttt{math.h} functions is defined unknown.

• The rounding behaviors characterized by non-standard values of \texttt{FLT_ROUNDS} (5.2.4.2.2).

No non-standard values are used.

• The evaluation methods characterized by non-standard negative values of \texttt{FLT_EVAL_METHOD} (5.2.4.2.2).

\texttt{FLT_EVAL_METHOD} is defined as 0. No non-standard values are used.
• The direction of rounding when an integer is converted to a floating-point number that cannot exactly represent the original value (6.3.1.4).

The round to nearest method is used. FLT_ROUNDS is ignored.

• The direction of rounding when a floating-point number is converted to a narrower floating-point number (6.3.1.5).

The round to nearest method is used. FLT_ROUNDS is ignored.

• How the nearest representable value or the larger or smaller representable value immediately adjacent to the nearest representable value is chosen for certain floating constants (6.4.4.2).

The round to nearest method is used. FLT_ROUNDS is ignored.

• Whether and how floating expressions are contracted when not disallowed by the FP_CONTRACT pragma (6.5).

Pragma FP_CONTRACT is ignored.

• The default state for the FENV_ACCESS pragma (7.6.1).

The default state of pragma FENV_ACCESS is "off". This pragma is ignored.

• Additional floating-point exceptions, rounding modes, environments, and classifications, and their macro names (7.6, 7.12).

No additional floating-point exceptions, rounding modes, environments, and classifications are defined.

• The default state for the FP_CONTRACT pragma (7.12.2).

The default state of pragma FP_CONTRACT is "off". This pragma is ignored.

• Whether the "inexact" floating-point exception can be raised when the rounded result actually does equal the mathematical result in an IEC 60559 conformant implementation (F.9).

The "inexact" floating-point exception is not supported.

• Whether the "underflow" (and "inexact") floating-point exception can be raised when a result is tiny but not inexact in an IEC 60559 conformant implementation (F.9).

Subnormals are not supported, so this situation cannot arise.

22.1.7. Arrays and Pointers

• The result of converting a pointer to an integer or vice versa (6.3.2.3).

All non-pointer conversions to and from a 32-bit pointer are implemented as a conversion to or from a 32-bit integer type.

• The size of the result of subtracting two pointers to elements of the same array (6.5.6).

The size of ptrdiff_t is 32 bits. The difference in address location is expressed in bytes.
22.1.8. Hints

• The extent to which suggestions made by using the register storage-class specifier are effective (6.7.1).

The compiler does not make assumptions based on the register storage-class specifier. So, basically this keyword is ignored, except that you cannot take the address of a register variable. The compiler issues an error in that case.

• The extent to which suggestions made by using the inline function specifier are effective (6.7.4).

With the inline keyword you force the compiler to inline the specified function, regardless of the optimization strategy of the compiler itself. For more information see Section 1.12.2, Inlining Functions: inline.

22.1.9. Structures, Unions, Enumerations, and Bit-fields

• Whether a "plain" int bit-field is treated as a signed int bit-field or as an unsigned int bit-field (6.7.2, 6.7.2.1).

An int bit-field is always treated as signed int.

• Allowable bit-field types other than _Bool, signed int, and unsigned int (6.7.2.1).

All integer types as specified in Section 1.1, Data Types are allowable bit-field types.

• Whether a bit-field can straddle a storage-unit boundary (6.7.2.1).

A bit-field cannot straddle a storage-unit boundary. If insufficient space remains, the bit-field is put into the next unit.

• The order of allocation of bit-fields within a unit (6.7.2.1).

Allocation starts at the least significant bit up to the most significant bit. If the following bit-field fits within the same unit, it is allocated starting at the next available bit.

• The alignment of non-bit-field members of structures (6.7.2.1). This should present no problem unless binary data written by one implementation is read by another.

The alignment of non-bit-field members of structures is the same as the alignment for data types as specified in Section 1.1, Data Types.

• The integer type compatible with each enumerated type (6.7.2.2).

The compiler always uses 32-bit integers for enumeration.
22.1.10. Qualifiers

• What constitutes an access to an object that has volatile-qualified type (6.7.3).

Any reference to an object with volatile type results in an access. The order in which volatile objects are accessed is defined by the order expressed in the source code. References to non-volatile objects are scheduled in arbitrary order, within the constraints given by dependencies.

If the compiler option --language=volatile (-Av) is set, all references to non-volatile objects result in an access before the access to a volatile object that occurs subsequently in the source file takes place. The volatile access acts as a memory barrier.

22.1.11. Preprocessing Directives

• How sequences in both forms of header names are mapped to headers or external source file names (6.4.7).

Sequences in header names are mapped to file names as is. The backslash "\" is not interpreted as an escape sequence. The backslash "\" (Windows) or forward slash "/" (Windows and Linux) is interpreted as a standard directory separator.

• Whether the value of a character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set (6.10.1).

A character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set.

• Whether the value of a single-character character constant in a constant expression that controls conditional inclusion may have a negative value (6.10.1).

A single-character character constant in a constant expression that controls conditional inclusion may have a negative value.

• The places that are searched for an included < > delimited header, and how the places are specified or the header is identified (6.10.2).

How the compiler searches for include files is explained in Section 4.3, How the Compiler Searches Include Files.

• How the named source file is searched for in an included " " delimited header (6.10.2).

How the compiler searches for include files is explained in Section 4.3, How the Compiler Searches Include Files.

• The method by which preprocessing tokens (possibly resulting from macro expansion) in a #include directive are combined into a header name (6.10.2).

Preprocessing tokens in a #include directive are combined the same way as outside a #include directive.
• The nesting limit for include processing (6.10.2).

There is no nesting limit for include processing.

• Whether the # operator inserts a \ character before the \ character that begins a universal character name in a character constant or string literal (6.10.3.2).

The # operator inserts a \ character before every \ character in a character constant or string literal.

• The behavior on each recognized non-STDC #pragma directive (6.10.6).

All non-STDC pragmas are described in Section 1.9, Pragmas to Control the Compiler.

• The definitions for __DATE__ and __TIME__ when respectively, the date and time of translation are not available (6.10.8).

The date and time of translation are always available, macros __DATE__ and __TIME__ are always defined.

22.1.12. Library Functions

• Any library facilities available to a freestanding program, other than the minimal set required by clause 4 (5.1.2.1).

All library functions are described in Chapter 14, Libraries. 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. If the application runs under control of the debugger and FSS is used, then the low-level behavior is equal to the host system's file access behavior.

In the implementation in the C library, the basic sequences of a multibyte character consist of single bytes (MB_LEN_MAX is set to 1). If you want full multibyte support, you need to change the C library. See the notes in the header files stdio.h and wchar.h for more information.

• The format of the diagnostic printed by the assert macro (7.2.1.1).

The assert() function is implemented as a macro in assert.h. The output is:

Assertion failed: (expression) file filename, line linenum when the parameter evaluates to zero.

• The representation of the floating-point status flags stored by the fegetexceptflag function (7.6.2.2).

Exception flags are supported and are defined in header file fenv.h.

#define FE_DIVBYZERO   (1 << 1)
#define FE_INEXACT     (1 << 4)
#define FE_INVALID     (1 << 0)
- Whether the feraiseexcept function raises the "inexact" floating-point exception in addition to the "overflow" or "underflow" floating-point exception (7.6.2.3).

  Function feraiseexcept does not raise FE_INEXACT if underflow or overflow is raised.

- Strings other than "C" and " " that may be passed as the second argument to the setlocale function (7.11.1.1).

  No other strings are predefined. A NULL pointer as the second argument returns the "C" locale. Any other string than "C" or " " can be passed as the second argument to the setlocale function and results in NULL.

- The types defined for float_t and double_t when the value of the FLT_EVAL_METHOD macro is less than 0 or greater than 2 (7.12).

  The FLT_EVAL_METHOD macro can only have the values 0, 1 or 2.

- Domain errors for the mathematical functions, other than those required by this International Standard (7.12.1).

  No other domain errors exist, other than those required by the standard.

- The values returned by the mathematical functions on domain errors (7.12.1).

  On domain errors (errno is set to EDOM), the mathematical functions return a value as specified in the following table.

<table>
<thead>
<tr>
<th>Math function</th>
<th>Return value on EDOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>acos(</td>
<td>x</td>
</tr>
<tr>
<td>asin(</td>
<td>x</td>
</tr>
<tr>
<td>log(x &lt; 0.0)</td>
<td>-HUGE_VAL</td>
</tr>
<tr>
<td>pow(x &lt;= 0.0)</td>
<td>0.0</td>
</tr>
<tr>
<td>sqrt(x &lt; 0.0)</td>
<td>NaN</td>
</tr>
</tbody>
</table>

- The values returned by the mathematical functions on underflow range errors, whether errno is set to the value of the macro ERANGE when the integer expression math_errhandling & MATH_ERRNO is nonzero, and whether the "underflow" floating-point exception is raised when the integer expression math_errhandling & MATH_ERREXCEPT is nonzero. (7.12.1).

  On underflow range errors, the mathematical functions return 0.0. math_errhandling is set to MATH_ERRNO, errno is not set to ERANGE on underflow range errors.
C Implementation-defined Behavior

- Whether a domain error occurs or zero is returned when an fmod function has a second argument of zero (7.12.10.1).

  Zero (0.0) is returned when an fmod function has a second argument of zero.

- The base-2 logarithm of the modulus used by the remquo functions in reducing the quotient (7.12.10.3).

  The remquo function calculates at least 8 bits of the quotient.

- Whether the equivalent of signal(sig, SIG_DFL); is executed prior to the call of a signal handler, and, if not, the blocking of signals that is performed (7.14.1.1).

  The equivalent of signal(sig, SIG_DFL); is executed prior to the call of a signal handler.

- The null pointer constant to which the macro NULL expands (7.17).

  Macro NULL is defined as (void *) 0.

- Whether the last line of a text stream requires a terminating new-line character (7.19.2).

  Both a new-line character (\n) and end-of-file (EOF) are recognized as the termination character of a line.

- Whether space characters that are written out to a text stream immediately before a new-line character appear when read in (7.19.2).

  Space characters written to a stream immediately before a new-line character are preserved.

- The number of null characters that may be appended to data written to a binary stream (7.19.2).

  I/O related functions are implemented using file system simulation (FSS). This system can be used by the debugger to simulate an I/O environment. If the application runs under control of the debugger and FSS is used, then the low-level behavior is equal to the host system's file access behavior. The library does not append any null characters. It depends on the open() function on the host environment what happens. You can write your own _open() function if necessary.

- Whether the file position indicator of an append-mode stream is initially positioned at the beginning or end of the file (7.19.3).

  I/O related functions are implemented using file system simulation (FSS). This system can be used by the debugger to simulate an I/O environment. If the application runs under control of the debugger and FSS is used, then the low-level behavior is equal to the host system's file access behavior. Where the file position indicator of an append-mode stream is initially positioned depends on how the low-level file routines are implemented in your application.

- Whether a write on a text stream causes the associated file to be truncated beyond that point (7.19.3).

  I/O related functions are implemented using file system simulation (FSS). This system can be used by the debugger to simulate an I/O environment. If the application runs under control of the debugger and FSS is used, then the low-level behavior is equal to the host system's file access behavior. Whether a write on a text stream causes the associated file to be truncated beyond that point depends on how the low-level file routines are implemented in your application.
• The characteristics of file buffering (7.19.3).
  Files can be unbuffered, fully buffered or line buffered. What actually happens depends on how the
  low-level file routines are implemented in your application.

• Whether a zero-length file actually exists (7.19.3).
  This depends on how the low-level file routines are implemented in your application.

• The rules for composing valid file names (7.19.3).
  This depends on how the low-level file routines are implemented in your application.

• Whether the same file can be simultaneously open multiple times (7.19.3).
  This depends on how the low-level file routines are implemented in your application.

• The nature and choice of encodings used for multibyte characters in files (7.19.3).
  Use of variable length encoded characters in files in comments and string literals is permitted.

• The effect of the remove function on an open file (7.19.4.1).
  This depends on how the low-level file routines are implemented in your application.

• The effect if a file with the new name exists prior to a call to the rename function (7.19.4.2).
  This depends on how the low-level file routines are implemented in your application.

• Whether an open temporary file is removed upon abnormal program termination (7.19.4.3).
  This depends on how the low-level file routines are implemented in your application.

• Which changes of mode are permitted (if any), and under what circumstances (7.19.5.4).
  The freopen() function first calls fclose() and then calls _fopen() with the new mode.

• The style used to print an infinity or NaN, and the meaning of any n-char or n-wchar sequence printed
  for a NaN (7.19.6.1, 7.24.2.1).
  The style used to print an infinity or NaN is inf and nan respectively (INF or NAN for the F conversion
  specifier). n-char or n-wchar sequences are not used for nan.

• The output for %p conversion in the fprintf or fwprintf function (7.19.6.1, 7.24.2.1).
  The argument is treated as having type void *. The value will be printed as a hexadecimal value,
  similar to %x.

• The interpretation of a – character that is neither the first nor the last character, nor the second where
  a ^ character is the first, in the scanlist for %[ conversion in the fscanf or fwscanf function (7.19.6.2,
  7.24.2.1).
  A – character is treated as a normal character.
• The set of sequences matched by a %p conversion and the interpretation of the corresponding input item in the fscanf or fswscanf function (7.19.6.2, 7.24.2.2).

The format of %p matches the format of %x. The input for %p is a hexadecimal value, which is converted to a value with type void *.

• The value to which the macro errno is set by the fgetpos, fsetpos, or ftell functions on failure (7.19.9.1, 7.19.9.3, 7.19.9.4).

If errno is set to a value depends on how the low-level file routines are implemented in your application.

• The meaning of any n-char or n-wchar sequence in a string representing a NaN that is converted by the strtod, strtof, strtold, wcstod, wcstof, or wcstold function (7.20.1.3, 7.24.4.1.1).

An n-char or n-wchar sequence in a string representing a NaN is ignored.

• Whether or not the strtod, strtof, strtold, wcstod, wcstof, or wcstold function sets errno to ERANGE when underflow occurs (7.20.1.3, 7.24.4.1.1).

errno is set to ERANGE when underflow occurs and the value returned is 0.0.

• Whether the calloc, malloc, and realloc functions return a null pointer or a pointer to an allocated object when the size requested is zero (7.20.3).

NULL is returned when a size of zero is requested.

• Whether open streams with unwritten buffered data are flushed, open streams are closed, or temporary files are removed when the abort or _Exit function is called (7.20.4.1, 7.20.4.4).

When the abort() or _Exit() function is called, open streams with unwritten buffered data are not flushed, open streams are not closed, and temporary files are not removed.

• The termination status returned to the host environment by the abort, exit, or _Exit function (7.20.4.1, 7.20.4.3, 7.20.4.4).

exit() and _Exit() use the input value as termination status. abort() calls _Exit() with EXIT_FAILURE.

• The value returned by the system function when its argument is not a null pointer (7.20.4.6).

A skeleton is provided for the system() function in the C library, because the embedded environment has no operating system. The system() function returns the value 0.

• The local time zone and Daylight Saving Time (7.23.1).

The default time zone is UTC. Daylight Saving Time is not available (tm_isdst=-1).

• The range and precision of times representable in clock_t and time_t (7.23).

clock_t is defined as unsigned long long, time_t is defined as unsigned long. The resolution of the clock is defined by CLOCK_PER_SEC, which defaults to 300000000 (300MHz) and is defined in time.h.
• The era for the clock function (7.23.2.1).

The clock function returns the current processor time.

• The replacement string for the \%Z specifier to the strftime, and wcsftime functions in the "C" locale (7.23.3.5, 7.24.5.1).

\%Z is replaced by the time zone name, by default UTC.

• Whether or when the trigonometric, hyperbolic, base-e exponential, base-e logarithmic, error, and log gamma functions raise the "inexact" floating-point exception in an IEC 60559 conformant implementation (F.9).

The "inexact" floating-point exception is not supported.

• Whether the functions in <math.h> honor the rounding direction mode in an IEC 60559 conformant implementation (F.9).

The round to nearest method is used. FLT_ROUNDS is defined as 1.

22.1.13. Architecture

• The values or expressions assigned to the macros specified in the headers <float.h>, <limits.h>, and <stdint.h> (5.2.4.2, 7.18.2, 7.18.3).

Macros in <float.h>:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLT_RADIX</td>
<td>2</td>
</tr>
<tr>
<td>FLT_EVAL_METHOD</td>
<td>0</td>
</tr>
<tr>
<td>FLT_ROUNDS</td>
<td>1 (round to nearest)</td>
</tr>
<tr>
<td>FLT_MANT_DIG</td>
<td>24</td>
</tr>
<tr>
<td>FLT_DIG</td>
<td>6</td>
</tr>
<tr>
<td>FLT_EPSILON</td>
<td>1.19209290E-07F</td>
</tr>
<tr>
<td>FLT_MIN</td>
<td>1.17549435E-38F</td>
</tr>
<tr>
<td>FLT_MIN_EXP</td>
<td>-125</td>
</tr>
<tr>
<td>FLT_MIN_10_EXP</td>
<td>-37</td>
</tr>
<tr>
<td>FLT_MAX</td>
<td>3.40282347E+38F</td>
</tr>
<tr>
<td>FLT_MAX_EXP</td>
<td>+128</td>
</tr>
<tr>
<td>FLT_MAX_10_EXP</td>
<td>+38</td>
</tr>
<tr>
<td>[L]DBL_MANT_DIG</td>
<td>53</td>
</tr>
<tr>
<td>[L]DBL_DIG</td>
<td>15</td>
</tr>
<tr>
<td>[L]DBL_EPSILON</td>
<td>2.2204460492503131E-16</td>
</tr>
<tr>
<td>[L]DBL_MIN</td>
<td>2.2250738585072014E-308</td>
</tr>
</tbody>
</table>
### Macros in `<float.h>`:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>[L]DBL_MIN_EXP</td>
<td>-1021</td>
</tr>
<tr>
<td>[L]DBL_MIN_10_EXP</td>
<td>-307</td>
</tr>
<tr>
<td>[L]DBL_MAX</td>
<td>1.7976931348623157E+308</td>
</tr>
<tr>
<td>[L]DBL_MAX_EXP</td>
<td>+1024</td>
</tr>
<tr>
<td>[L]DBL_MAX_10_EXP</td>
<td>+308</td>
</tr>
<tr>
<td>DECIMAL_DIG</td>
<td>17 (for double FP), 9 (for single FP)</td>
</tr>
<tr>
<td>FLT16_MANT_DIG</td>
<td>11</td>
</tr>
<tr>
<td>FLT16_DIG</td>
<td>3</td>
</tr>
<tr>
<td>FLT16_EPSILON</td>
<td>9.765625E-4F</td>
</tr>
<tr>
<td>FLT16_MIN</td>
<td>6.103515625E-05F</td>
</tr>
<tr>
<td>FLT16_MIN_EXP</td>
<td>-13</td>
</tr>
<tr>
<td>FLT16_MIN_10_EXP</td>
<td>-4</td>
</tr>
<tr>
<td>FLT16_MAX</td>
<td>65504.0F</td>
</tr>
<tr>
<td>FLT16_MAX_EXP</td>
<td>+16</td>
</tr>
<tr>
<td>FLT16_MAX_10_EXP</td>
<td>+4</td>
</tr>
<tr>
<td>FLT16_HAS_SUBNORM</td>
<td>1</td>
</tr>
<tr>
<td>FLT16_TRUE_MIN</td>
<td>_FLT16_MIN</td>
</tr>
<tr>
<td>FLT16_DECIMAL_DIG</td>
<td>5</td>
</tr>
</tbody>
</table>

### Macros in `<limits.h>`:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR_BIT</td>
<td>8</td>
</tr>
<tr>
<td>SCHAR_MIN</td>
<td>-SCHAR_MAX-1</td>
</tr>
<tr>
<td>SCHAR_MAX</td>
<td>0x7f</td>
</tr>
<tr>
<td>UCHAR_MAX</td>
<td>0xffU</td>
</tr>
<tr>
<td>CHAR_MIN</td>
<td>UCHAR_MIN or SCHAR_MIN if option --schar is set</td>
</tr>
<tr>
<td>CHAR_MAX</td>
<td>UCHAR_MAX or SCHAR_MAX if option --schar is set</td>
</tr>
<tr>
<td>MB_LEN_MAX</td>
<td>1</td>
</tr>
<tr>
<td>SHRT_MIN</td>
<td>-SHRT_MAX-1</td>
</tr>
<tr>
<td>SHRT_MAX</td>
<td>0x7fff</td>
</tr>
<tr>
<td>USHRT_MAX</td>
<td>0xffffU</td>
</tr>
<tr>
<td>INT_MIN</td>
<td>-INT_MAX-1</td>
</tr>
<tr>
<td>INT_MAX</td>
<td>0x7fffffff</td>
</tr>
<tr>
<td>UINT_MAX</td>
<td>0xffffffffU</td>
</tr>
<tr>
<td>LONG_MIN</td>
<td>-LONG_MAX-1</td>
</tr>
</tbody>
</table>
The limit macros in `<limits.h>` for exact-width, minimum-width and fastest-width integer types have the same ranges as `char, short, int, long` and `long long`. Furthermore the following macros are defined:

<table>
<thead>
<tr>
<th>Macro &lt;limits.h&gt;</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>LONG_MAX</td>
<td>0x7fffffffL</td>
</tr>
<tr>
<td>ULONG_MAX</td>
<td>0xffffffffUL</td>
</tr>
<tr>
<td>LLONG_MIN</td>
<td>-LLONG_MAX-1</td>
</tr>
<tr>
<td>LLONG_MAX</td>
<td>0x7fffffffffffffffLL</td>
</tr>
<tr>
<td>ULLONG_MAX</td>
<td>0xffffffffffffffffULL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Macro &lt;stdint.h&gt;</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTPTR_MIN</td>
<td>INT32_MIN</td>
</tr>
<tr>
<td>INTPTR_MAX</td>
<td>INT32_MAX</td>
</tr>
<tr>
<td>UINTPTR_MAX</td>
<td>UINT32_MAX</td>
</tr>
<tr>
<td>INTMAX_MIN</td>
<td>INT64_MIN</td>
</tr>
<tr>
<td>INTMAX_MAX</td>
<td>INT64_MAX</td>
</tr>
<tr>
<td>UINTMAX_MAX</td>
<td>UINT64_MAX</td>
</tr>
<tr>
<td>PTRDIFF_MIN</td>
<td>__PTRDIFF_MIN</td>
</tr>
<tr>
<td>PTRDIFF_MAX</td>
<td>__PTRDIFF_MAX</td>
</tr>
<tr>
<td>SIG_ATOMIC_MIN</td>
<td>INT32_MIN</td>
</tr>
<tr>
<td>SIG_ATOMIC_MAX</td>
<td>INT32_MAX</td>
</tr>
<tr>
<td>SIZE_MAX</td>
<td>__SIZE_MAX</td>
</tr>
<tr>
<td>WCHAR_MIN</td>
<td>__WCHAR_MIN</td>
</tr>
<tr>
<td>WCHAR_MAX</td>
<td>__WCHAR_MAX</td>
</tr>
<tr>
<td>WINT_MIN</td>
<td>0</td>
</tr>
<tr>
<td>WINT_MAX</td>
<td>UINT32_MAX</td>
</tr>
</tbody>
</table>

- The number, order, and encoding of bytes in any object (when not explicitly specified in this International Standard) (6.2.6.1).

All types are described in Section 1.1, Data Types.

- The value of the result of the `sizeof` operator (6.5.3.4).

The value of the size of the data types is described in Section 1.1, Data Types. Divide the size by 8 to get bytes, because the table lists the size of the data types in bits.
22.2. C99 Locale-specific Behavior

Locale-specific behavior is behavior that depends on local conventions of nationality, culture, and language that each implementation documents.

The following items describe the locale-specific characteristics, as indicated in Annex J.4 of the ISO/IEC 9899:1999 (E) standard.

• Additional members of the source and execution character sets beyond the basic character set (5.2.1).

    The compiler accepts all one-byte characters in the host's default character set. Use of variable length encoded characters in the source file in comments and string literals is permitted.

    In the implementation in the C library, the basic sequences of a multibyte character consist of single bytes (MB_LEN_MAX is set to 1). If you want full multibyte support, you need to change the C library. See the notes in the header files stdio.h and wchar.h for more information.

• The presence, meaning, and representation of additional multibyte characters in the execution character set beyond the basic character set (5.2.1.2).

    Use of variable length encoded characters in the source file in comments and string literals is permitted.

• The shift states used for the encoding of multibyte characters (5.2.1.2).

    A multibyte character must be a single byte when in the initial shift state.

• The direction of writing of successive printing characters (5.2.2).

    The direction of writing depends on the application and the display device.

• The decimal-point character (7.1.1).

    The default decimal-point character is a '.'.

• The set of printing characters (7.4, 7.25.2).

    The set of printing characters are the characters for which the isprint() function returns true. Printing characters are characters in the range 32 (space) to 126.

• The set of control characters (7.4, 7.25.2).

    The set of control characters are the characters for which the iscntrl() function returns true. Control characters are characters in the range 0 to 31 and 127.

• The sets of characters tested for by the isalpha, isblank, islower, ispunct, isspace, isupper, iswalpha, iswblank, iswlower, iswpunct, iswspace, or iswupper functions (7.4.1.2, 7.4.1.3, 7.4.1.7, 7.4.1.9, 7.4.1.10, 7.4.1.11, 7.25.2.1.2, 7.25.2.1.3, 7.25.2.1.7, 7.25.2.1.9, 7.25.2.1.10, 7.25.2.1.11).

    The characters tested for are specified in the following table.

<table>
<thead>
<tr>
<th>Function</th>
<th>Characters tested</th>
</tr>
</thead>
<tbody>
<tr>
<td>isalpha</td>
<td>a-z, A-Z</td>
</tr>
</tbody>
</table>
The native environment (7.11.1.1).

The native environment is the same as the "C" locale.

Additional subject sequences accepted by the numeric conversion functions (7.20.1, 7.24.4.1).

No additional subject sequences are accepted.

The collation sequence of the execution character set (7.21.4.3, 7.24.4.4.2).

Only the "C" locale is supported. The \texttt{strcoll()} function is the same as the \texttt{strcmp()} function. The \texttt{wcscoll()} function is the same as the \texttt{wcscmp()} function.

The contents of the error message strings set up by the \texttt{strerror} function (7.21.6.2).

The error message strings returned by \texttt{strerror()} depend on the argument. Typically, the values for the argument come from \texttt{errno.h}. For a list of messages see Section 14.2.5, \texttt{errno.h}).

The formats for time and date (7.23.3.5, 7.24.5.1).

English names for months and days are used.

\texttt{%c} is replaced by the following date and time representation: \texttt{%a \%b \%e \%H:\%M:\%S \%Y}

\texttt{%x} is replaced by the following date representation: \texttt{%m/%d/%y}

\texttt{%X} is replaced by the following time representation: \texttt{%H:\%M:\%S}

Character mappings that are supported by the \texttt{towctrans} function (7.25.1).

The character mappings supported by the \texttt{towctrans()} function are defined in \texttt{wctype.h::to_lower} and \texttt{to_upper}.

Character classifications that are supported by the \texttt{iswctype} function (7.25.1).

The character classifications supported by the \texttt{iswctype()} function are defined in \texttt{wctype.h::alnum, _alpha, _cntrl, _digit, _graph, _lower, _print, _punct, _space, _upper, _xdigit and _blank}. 
22.3. C11 Implementation-defined Behavior

Implementation-defined behavior is unspecified behavior where each implementation documents how the choice is made.

The following sections describe the implementation-defined characteristics. The section numbers listed in parenthesis refer to the corresponding sections in the ISO C11 standard. The order in this chapter is the same as used in Annex J.3 of the ISO/IEC 9899:2011 (E) standard.

22.3.1. Translation

• How a diagnostic is identified (3.10, 5.1.1.3).

The C compiler diagnostics are explained in Section 4.7, C Compiler Error Messages.

• Whether each nonempty sequence of white-space characters other than new-line is retained or replaced by one space character in translation phase 3 (5.1.1.2).

White-space is retained.

22.3.2. Environment

• The mapping between physical source file multibyte characters and the source character set in translation phase 1 (5.1.1.2).

Use of variable length encoded characters in the source file in comments and string literals is permitted. A one-on-one mapping is done without interpretation of multibyte characters.

• The name and type of the function called at program startup in a freestanding environment (5.1.2.1).

The function called at program startup (in cstart.c) is called main. The prototype for main in cstart.c is:

```c
extern int main( int argc, char *argv[] );
```

• The effect of program termination in a freestanding environment (5.1.2.1).

Execution is halted if the program is executed under control of a debugger, otherwise the program will loop forever in function _Exit().

• An alternative manner in which the main function may be defined (5.1.2.2.1).

You can change the definition of main by altering file cstart.c.

• The values given to the strings pointed to by the argv argument to main (5.1.2.2.1).

The strings get their values from the arguments given in file cstart.c. The program arguments are treated case sensitive.
What constitutes an interactive device (5.1.2.3).

The streams stdin, stdout and stderr are treated as interactive devices. The debugger uses these streams with File System Simulation (FSS) windows to interact.

Whether a program can have more than one thread of execution in a freestanding environment (5.1.2.4).

There is only a single thread of execution.

The set of signals, their semantics, and their default handling (7.14).

The signals are described in Section 14.2.18, signal.h.

Signal values other than SIGFPE, SIGILL, and SIGSEGV that correspond to a computational exception (7.14.1.1).

There are no other values that correspond to a computational exception. All signal values are described in Section 14.2.18, signal.h.

Signals for which the equivalent of signal(sig, SIG_IGN); is executed at program startup (7.14.1.1).

By default the implementation does not ignore any signals at program startup.

The set of environment names and the method for altering the environment list used by the getenv function (7.22.4.5).

There are no implementation-defined environment names that are used by the getenv function. A skeleton is provided for the getenv function in the C library, because the embedded environment has no operating system. The getenv function calls the name as a void function.

The manner of execution of the string by the system function (7.22.4.6).

A skeleton is provided for the system() function in the C library, because the embedded environment has no operating system. The system() function calls the string as a void function.

22.3.3. Identifiers

Which additional multibyte characters may appear in identifiers and their correspondence to universal character names (6.4.2).

No additional multibyte characters are supported in an identifier.

The number of significant initial characters in an identifier (5.2.4.1, 6.4.2).

All characters in an identifier are significant.

22.3.4. Characters

The number of bits in a byte (3.6).

There are eight bits in a byte.
The values of the members of the execution character set (5.2.1).

Only 8-bit characters are supported. The values of the execution character set are the same as that of the source character set. The same representation value is used for each member in the characters sets except for the escape sequences.

The unique value of the member of the execution character set produced for each of the standard alphabetic escape sequences (5.2.2).

The following table contains an overview of the escape sequences and their byte value in the execution character set.

<table>
<thead>
<tr>
<th>Escape sequence</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>\a</td>
<td>7</td>
</tr>
<tr>
<td>\b</td>
<td>8</td>
</tr>
<tr>
<td>\f</td>
<td>12</td>
</tr>
<tr>
<td>\n</td>
<td>10</td>
</tr>
<tr>
<td>\r</td>
<td>13</td>
</tr>
<tr>
<td>\t</td>
<td>9</td>
</tr>
<tr>
<td>\v</td>
<td>11</td>
</tr>
</tbody>
</table>

The value of a char object into which has been stored any character other than a member of the basic execution character set (6.2.5).

Any 8-bit value can be stored in a char object.

Which of signed char or unsigned char has the same range, representation, and behavior as "plain" char (6.2.5, 6.3.1.1).

By default "plain" char is the same as specifying unsigned char. With C compiler option --schar you can change the default to signed char.

The mapping of members of the source character set (in character constants and string literals) to members of the execution character set (6.4.4.4, 5.1.1.2).

The mapping is one-to-one. The values of the execution character set are the same as that of the source character set. The same representation value is used for each member in the characters sets except for the escape sequences.

The value of an integer character constant containing more than one character or containing a character or escape sequence that does not map to a single-byte execution character (6.4.4.4).

For any character constant containing more than one character, a warning is issued and the value is truncated to type char.

The value of a wide character constant containing more than one multibyte character or a single multibyte character that maps to multiple members of the extended execution character set, or containing a
multibyte character or escape sequence not represented in the extended execution character set (6.4.4.4).

A wide character constant can contain at most two multibyte characters. Its value is the concatenation of the multibyte characters represented in a signed short int.

• The current locale used to convert a wide character constant consisting of a single multibyte character that maps to a member of the extended execution character set into a corresponding wide character code (6.4.4.4).

By default, the "C" locale is used.

• Whether differently-prefixed wide string literal tokens can be concatenated and, if so, the treatment of the resulting multibyte character sequence (6.4.5).

Differently-prefixed wide string literals can be concatenated. The encoding prefix of the first literal determines the treatment of all literals.

• The current locale used to convert a wide string literal into corresponding wide character codes (6.4.5).

By default, the "C" locale is used.

• The value of a string literal containing a multibyte character or escape sequence not represented in the execution character set (6.4.5).

All source characters can be represented in the execution character set.

• The encoding of any of wchar_t, char16_t, and char32_t where the corresponding standard encoding macro (__STDC_ISO_10646__, __STDC_UTF_16__, or __STDC_UTF_32__) is not defined (6.10.8.2).

The typedefs are implemented with the following types:

<table>
<thead>
<tr>
<th>Typedef</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>wchar_t</td>
<td>unsigned short</td>
</tr>
<tr>
<td>char16_t</td>
<td>unsigned short</td>
</tr>
<tr>
<td>char32_t</td>
<td>unsigned int</td>
</tr>
</tbody>
</table>

### 22.3.5. Integers

• Any extended integer types that exist in the implementation (6.2.5).

All types are described in Section 1.1, Data Types.

• Whether signed integer types are represented using sign and magnitude, two’s complement, or ones’ complement, and whether the extraordinary value is a trap representation or an ordinary value (6.2.6.2).

Signed integer types are represented in two’s complement. The most significant bit is the sign bit. 1 is negative, 0 is positive.
• The rank of any extended integer type relative to another extended integer type with the same precision (6.3.1.1).

All types are described in Section 1.1, Data Types.

• The result of, or the signal raised by, converting an integer to a signed integer type when the value cannot be represented in an object of that type (6.3.1.3).

At compile time, when converting integer types and a value does not fit in a type, the compiler issues a warning and the value is truncated. At run-time no warning or signal is given and the value is truncated.

• The results of some bitwise operations on signed integers (6.5).

The result of E1>>E2 is E1 right shifted E2 bit positions. If E1 has a signed type and a negative value, the shift behavior is implemented as an arithmetic shift. The empty position in the most significant bit is filled with a copy of the original most significant bit.

22.3.6. Floating-Point

• The accuracy of the floating-point operations and of the library functions in <math.h> and <complex.h> that return floating-point results (5.2.4.2.2).

A float has 24 significant mantissa bits and 8 exponent bits. A double or long double has 53 significant mantissa bits and 11 exponent bits. This is conform IEEE-754 for single precision and double precision floating-point. Internally the compiler uses a mantissa of 80 bits. The results of floating-point operations are rounded to the nearest IEEE-754 format.

The accuracy of sqrt() and other math.h functions is defined unknown.

• The accuracy of the conversions between floating-point internal representations and string representations performed by the library functions in <stdio.h>, <stdlib.h>, and <wchar.h> (5.2.4.2.2).

The accuracy of the conversions is unknown.

• The rounding behaviors characterized by non-standard values of FLT_ROUNDS (5.2.4.2.2).

No non-standard values are used.

• The evaluation methods characterized by non-standard negative values of FLT_EVAL_METHOD (5.2.4.2.2).

FLT_EVAL_METHOD is defined as 0. No non-standard values are used.

• The direction of rounding when an integer is converted to a floating-point number that cannot exactly represent the original value (6.3.1.4).

The round to nearest method is used. FLT_ROUNDS is ignored.
• The direction of rounding when a floating-point number is converted to a narrower floating-point number (6.3.1.5).

The round to nearest method is used. FLT_ROUNDS is ignored.

• How the nearest representable value or the larger or smaller representable value immediately adjacent to the nearest representable value is chosen for certain floating constants (6.4.4.2).

The round to nearest method is used. FLT_ROUNDS is ignored.

• Whether and how floating expressions are contracted when not disallowed by the FP_CONTRACT pragma (6.5).

Pragma FP_CONTRACT is ignored.

• The default state for the FENV_ACCESS pragma (7.6.1).

The default state of pragma FENV_ACCESS is "off". This pragma is ignored.

• Additional floating-point exceptions, rounding modes, environments, and classifications, and their macro names (7.6, 7.12).

No additional floating-point exceptions, rounding modes, environments, and classifications are defined.

• The default state for the FP_CONTRACT pragma (7.12.2).

The default state of pragma FP_CONTRACT is "off". This pragma is ignored.

### 22.3.7. Arrays and Pointers

• The result of converting a pointer to an integer or vice versa (6.3.2.3).

All non-pointer conversions to and from a 32-bit pointer are implemented as a conversion to or from a 32-bit integer type.

• The size of the result of subtracting two pointers to elements of the same array (6.5.6).

The size of `ptrdiff_t` is 32 bits. The difference in address location is expressed in bytes.

### 22.3.8. Hints

• The extent to which suggestions made by using the register storage-class specifier are effective (6.7.1).

The compiler does not make assumptions based on the register storage-class specifier. So, basically this keyword is ignored, except that you cannot take the address of a register variable. The compiler issues an error in that case.
The extent to which suggestions made by using the inline function specifier are effective (6.7.4).

With the inline keyword you force the compiler to inline the specified function, regardless of the optimization strategy of the compiler itself. For more information see Section 1.12.2, Inlining Functions: inline.

22.3.9. Structures, Unions, Enumerations, and Bit-fields

• Whether a "plain" int bit-field is treated as a signed int bit-field or as an unsigned int bit-field (6.7.2, 6.7.2.1).

An int bit-field is always treated as signed int.

• Allowable bit-field types other than _Bool, signed int, and unsigned int (6.7.2.1).

All integer types as specified in Section 1.1, Data Types are allowable bit-field types.

• Whether atomic types are permitted for bit-fields (6.7.2.1).

Atomic types are not permitted for bit-fields.

• Whether a bit-field can straddle a storage-unit boundary (6.7.2.1).

A bit-field cannot straddle a storage-unit boundary. If insufficient space remains, the bit-field is put into the next unit.

• The order of allocation of bit-fields within a unit (6.7.2.1).

Allocation starts at the least significant bit up to the most significant bit. If the following bit-field fits within the same unit, it is allocated starting at the next available bit.

• The alignment of non-bit-field members of structures (6.7.2.1). This should present no problem unless binary data written by one implementation is read by another.

The alignment of non-bit-field members of structures is the same as the alignment for data types as specified in Section 1.1, Data Types.

• The integer type compatible with each enumerated type (6.7.2.2).

The compiler always uses 32-bit integers for enumeration.

22.3.10. Qualifiers

• What constitutes an access to an object that has volatile-qualified type (6.7.3).

Any reference to an object with volatile type results in an access. The order in which volatile objects are accessed is defined by the order expressed in the source code. References to non-volatile objects are scheduled in arbitrary order, within the constraints given by dependencies.

If the compiler option --language=+volatile (-Av) is set, all references to non-volatile objects result in an access before the access to a volatile object that occurs subsequently in the source file takes place. The volatile access acts as a memory barrier.
22.3.11. Preprocessing Directives

• The locations within #pragma directives where header name preprocessing tokens are recognized (6.4, 6.4.7).

Within a #pragma directive, header name preprocessing tokens are not recognized.

• How sequences in both forms of header names are mapped to headers or external source file names (6.4.7).

Sequences in header names are mapped to file names as is. The backslash "\" is not interpreted as an escape sequence. The backslash "\" (Windows) or forward slash "/" (Windows and Linux) is interpreted as a standard directory separator.

• Whether the value of a character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set (6.10.1).

A character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set.

• Whether the value of a single-character character constant in a constant expression that controls conditional inclusion may have a negative value (6.10.1).

A single-character character constant in a constant expression that controls conditional inclusion may have a negative value.

• The places that are searched for an included < > delimited header, and how the places are specified or the header is identified (6.10.2).

How the compiler searches for include files is explained in Section 4.3, How the Compiler Searches Include Files.

• How the named source file is searched for in an included " " delimited header (6.10.2).

How the compiler searches for include files is explained in Section 4.3, How the Compiler Searches Include Files.

• The method by which preprocessing tokens (possibly resulting from macro expansion) in a #include directive are combined into a header name (6.10.2).

Preprocessing tokens in a #include directive are combined the same way as outside a #include directive.

• The nesting limit for #include processing (6.10.2).

There is no nesting limit for #include processing.

• Whether the # operator inserts a \ character before the \ character that begins a universal character name in a character constant or string literal (6.10.3.2).

The # operator inserts a \ character before every \ character in a character constant or string literal.
• The behavior on each recognized non-STDC #pragma directive (6.10.6).

All non-STDC pragmas are described in Section 1.9, Pragmas to Control the Compiler.

• The definitions for __DATE__ and __TIME__ when respectively, the date and time of translation are not available (6.10.8.1).

The date and time of translation are always available, macros __DATE__ and __TIME__ are always defined.

### 22.3.12. Library Functions

• Any library facilities available to a freestanding program, other than the minimal set required by clause 4 (5.1.2.1).

All library functions are described in Chapter 14, Libraries. 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. If the application runs under control of the debugger and FSS is used, then the low-level behavior is equal to the host system's file access behavior.

In the implementation in the C library, the basic sequences of a multibyte character consist of single bytes (MB_LEN_MAX is set to 1). If you want full multibyte support, you need to change the C library. See the notes in the header files stdio.h and wchar.h for more information.

• The format of the diagnostic printed by the assert macro (7.2.1.1).

The assert() function is implemented as a macro in assert.h. The output is:

Assertion failed: (expression) file filename, line linenumber

when the parameter evaluates to zero.

• The representation of the floating-point status flags stored by the fetroundflag function (7.6.2.2).

Exception flags are supported and are defined in header file fenv.h.

```c
#define FE_DIVBYZERO  (1 << 1)
#define FE_INEXACT    (1 << 4)
#define FE_INVALID    (1 << 0)
#define FE_OVERFLOW   (1 << 2)
#define FE_UNDERFLOW  (1 << 3)
```

• Whether the feraiseexcept function raises the "inexact" floating-point exception in addition to the "overflow" or "underflow" floating-point exception (7.6.2.3).

Function feraiseexcept does not raise FE_INEXACT if underflow or overflow is raised.
• Strings other than "C" and " " that may be passed as the second argument to the setlocale function (7.11.1.1).

No other strings are predefined. A NULL pointer as the second argument returns the "C" locale. Any other string than "C" or " " can be passed as the second argument to the setlocale function and results in NULL.

• The types defined for float_t and double_t when the value of the FLT_EVAL_METHOD macro is less than 0 (7.12).

The FLT_EVAL_METHOD macro can only have the values 0, 1 or 2.

• Domain errors for the mathematical functions, other than those required by this International Standard (7.12.1).

No other domain errors exist, other than those required by the standard.

• The values returned by the mathematical functions on domain errors or pole errors (7.12.1).

On domain errors (errno is set to EDOM), the mathematical functions return a value as specified in the following table.

<table>
<thead>
<tr>
<th>Math function</th>
<th>Return value on EDOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>acos(</td>
<td>x</td>
</tr>
<tr>
<td>asin(</td>
<td>x</td>
</tr>
<tr>
<td>log(x &lt; 0.0)</td>
<td>-HUGE_VAL</td>
</tr>
<tr>
<td>pow(x &lt;= 0.0)</td>
<td>0.0</td>
</tr>
<tr>
<td>sqrt(x &lt; 0.0)</td>
<td>NaN</td>
</tr>
</tbody>
</table>

• The values returned by the mathematical functions on underflow range errors, whether errno is set to the value of the macro ERANGE when the integer expression math_errhandling & MATH_ERRNO is nonzero, and whether the "underflow" floating-point exception is raised when the integer expression math_errhandling & MATH_ERREXCEPT is nonzero. (7.12.1).

On underflow range errors, the mathematical functions return 0.0. math_errhandling is set to MATH_ERRNO. errno is not set to ERANGE on underflow range errors.

• Whether a domain error occurs or zero is returned when an fmod function has a second argument of zero (7.12.10.1).

Zero (0.0) is returned when an fmod function has a second argument of zero.

• Whether a domain error occurs or zero is returned when a remainder function has a second argument of zero (7.12.10.2).

A domain error occurs.
• The base-2 logarithm of the modulus used by the remquo functions in reducing the quotient (7.12.10.3).

The remquo function calculates at least 8 bits of the quotient.

• Whether a domain error occurs or zero is returned when a remquo function has a second argument of zero (7.12.10.3).

A domain error occurs.

• Whether the equivalent of signal(sig, SIG_DFL); is executed prior to the call of a signal handler, and, if not, the blocking of signals that is performed (7.14.1.1).

The equivalent of signal(sig, SIG_DFL); is executed prior to the call of a signal handler.

• The null pointer constant to which the macro NULL expands (7.19).

Macro NULL is defined as (void *) 0.

• Whether the last line of a text stream requires a terminating new-line character (7.21.2).

Both a new-line character (\n) and end-of-file (EOF) are recognized as the termination character of a line.

• Whether space characters that are written out to a text stream immediately before a new-line character appear when read in (7.21.2).

Space characters written to a stream immediately before a new-line character are preserved.

• The number of null characters that may be appended to data written to a binary stream (7.21.2).

I/O related functions are implemented using file system simulation (FSS). This system can be used by the debugger to simulate an I/O environment. If the application runs under control of the debugger and FSS is used, then the low-level behavior is equal to the host system's file access behavior. The library does not append any null characters. It depends on the open() function on the host environment what happens. You can write your own _open() function if necessary.

• Whether the file position indicator of an append-mode stream is initially positioned at the beginning or end of the file (7.21.3).

I/O related functions are implemented using file system simulation (FSS). This system can be used by the debugger to simulate an I/O environment. If the application runs under control of the debugger and FSS is used, then the low-level behavior is equal to the host system's file access behavior. Where the file position indicator of an append-mode stream is initially positioned depends on how the low-level file routines are implemented in your application.

• Whether a write on a text stream causes the associated file to be truncated beyond that point (7.21.3).

I/O related functions are implemented using file system simulation (FSS). This system can be used by the debugger to simulate an I/O environment. If the application runs under control of the debugger and FSS is used, then the low-level behavior is equal to the host system's file access behavior. Whether a write on a text stream causes the associated file to be truncated beyond that point depends on how the low-level file routines are implemented in your application.
• The characteristics of file buffering (7.21.3).

Files can be unbuffered, fully buffered or line buffered. What actually happens depends on how the low-level file routines are implemented in your application.

• Whether a zero-length file actually exists (7.21.3).

This depends on how the low-level file routines are implemented in your application.

• The rules for composing valid file names (7.21.3).

This depends on how the low-level file routines are implemented in your application.

• Whether the same file can be simultaneously open multiple times (7.21.3).

This depends on how the low-level file routines are implemented in your application.

• The nature and choice of encodings used for multibyte characters in files (7.21.3).

Use of variable length encoded characters in files in comments and string literals is permitted.

• The effect of the `remove` function on an open file (7.21.4.1).

This depends on how the low-level file routines are implemented in your application.

• The effect if a file with the new name exists prior to a call to the `rename` function (7.21.4.2).

This depends on how the low-level file routines are implemented in your application.

• Whether an open temporary file is removed upon abnormal program termination (7.21.4.3).

This depends on how the low-level file routines are implemented in your application.

• Which changes of mode are permitted (if any), and under what circumstances (7.21.5.4).

The `freopen()` function first calls `fclose()` and then calls `_fopen()` with the new mode.

• The style used to print an infinity or NaN, and the meaning of any n-char or n-wchar sequence printed for a NaN (7.21.6.1, 7.29.2.1).

The style used to print an infinity or NaN is `inf` and `nan` respectively (`INF` or `NAN` for the `F` conversion specifier). n-char or n-wchar sequences are not used for `nan`.

• The output for `%p` conversion in the `fprintf` or `fwprintf` function (7.21.6.1, 7.29.2.1).

The argument is treated as having type `void *`. The value will be printed as a hexadecimal value, similar to `%x`.

• The interpretation of a `–` character that is neither the first nor the last character, nor the second where a `^` character is the first, in the scanlist for `%[` conversion in the `fscanf` or `fwscanf` function (7.21.6.2, 7.29.2.1).

A `–` character is treated as a normal character.
• The set of sequences matched by a %p conversion and the interpretation of the corresponding input item in the fscanf or fwscanf function (7.21.6.2, 7.29.2.2).

The format of %p matches the format of %x. The input for %p is a hexadecimal value, which is converted to a value with type void *.

• The value to which the macro errno is set by the fgetpos, fsetpos, or ftell functions on failure (7.21.9.1, 7.21.9.3, 7.21.9.4).

If errno is set to a value depends on how the low-level file routines are implemented in your application.

• The meaning of any n-char or n-wchar sequence in a string representing a NaN that is converted by the strtod, strtof, strtold, wcstod, wcstof, or wcstold function (7.22.1.3, 7.29.4.1.1).

An n-char or n-wchar sequence in a string representing a NaN is ignored.

• Whether or not the strtod, strtof, strtold, wcstod, wcstof, or wcstold function sets errno to ERANGE when underflow occurs (7.22.1.3, 7.29.4.1.1).

errno is set to ERANGE when underflow occurs and the value returned is 0.0.

• Whether the calloc, malloc, and realloc functions return a null pointer or a pointer to an allocated object when the size requested is zero (7.22.3).

NULL is returned when a size of zero is requested.

• Whether open streams with unwritten buffered data are flushed, open streams are closed, or temporary files are removed when the abort or _Exit function is called (7.22.4.1, 7.22.4.4).

When the abort() or _Exit() function is called, open streams with unwritten buffered data are not flushed, open streams are not closed, and temporary files are not removed.

• The termination status returned to the host environment by the abort, exit, _Exit, or quick_exit function (7.22.4.1, 7.22.4.4, 7.22.4.5, 7.22.4.7).

abort() calls _Exit() with EXIT_FAILURE.

exit() and quick_exit call _Exit() with their input value.

_exit() returns the input value to the host environment.

• The value returned by the system function when its argument is not a null pointer (7.22.4.6).

A skeleton is provided for the system() function in the C library, because the embedded environment has no operating system. The system() function returns the value 0.

• The range and precision of times representable in clock_t and time_t (7.27).

clock_t is defined as unsigned long long, time_t is defined as unsigned long. The resolution of the clock is defined by CLOCK_PER_SEC, which defaults to 300000000 (300MHz) and is defined in time.h.
• The local time zone and Daylight Saving Time (7.27.1).

The default time zone is UTC. Daylight Saving Time is not available (tm_isdst=-1).

• The era for the clock function (7.27.2.1).

The clock function returns the current processor time.

• The TIME_UTC epoch (7.27.2.5).

The timespec_get() function is based on the clock() function. Therefore, the epoch is the starting time of the clock() function.

• The replacement string for the %Z specifier to the strftime, and wcsftime functions in the "C" locale (7.27.3.5, 7.29.5.1).

%Z is replaced by the time zone name, by default UTC.

• Whether the functions in <math.h> honor the rounding direction mode in an IEC 60559 conformant implementation, unless explicitly specified otherwise (F.10).

The round to nearest method is used. FLT_ROUNDS is defined as 1.

22.3.13. Architecture

• The values or expressions assigned to the macros specified in the headers <float.h>, <limits.h>, and <stdint.h> (5.2.4.2, 7.20.2, 7.20.3).

Macros in <float.h>:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLT_RADIX</td>
<td>2</td>
</tr>
<tr>
<td>FLT_ROUNDS</td>
<td>1 (round to nearest)</td>
</tr>
<tr>
<td>FLT_EVAL_METHOD</td>
<td>0</td>
</tr>
<tr>
<td>FLT_MANT_DIG</td>
<td>24</td>
</tr>
<tr>
<td>FLT_DIG</td>
<td>6</td>
</tr>
<tr>
<td>FLT_EPSILON</td>
<td>1.19209290E-07F</td>
</tr>
<tr>
<td>FLT_MIN</td>
<td>1.17549435E-38F</td>
</tr>
<tr>
<td>FLT_MIN_EXP</td>
<td>-125</td>
</tr>
<tr>
<td>FLT_MIN_10_EXP</td>
<td>-37</td>
</tr>
<tr>
<td>FLT_MAX</td>
<td>3.40282347E+38F</td>
</tr>
<tr>
<td>FLT_MAX_EXP</td>
<td>+128</td>
</tr>
<tr>
<td>FLT_MAX_10_EXP</td>
<td>+38</td>
</tr>
<tr>
<td>FLT_HAS_SUBNORM</td>
<td>0</td>
</tr>
<tr>
<td>FLT_TRUE_MIN</td>
<td>1.17549435E-38F</td>
</tr>
</tbody>
</table>
### Value Macros

**Macros in `<float.h>`:**

<table>
<thead>
<tr>
<th>Macro</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLT_DECIMAL_DIG</td>
<td>9</td>
</tr>
<tr>
<td>[L]DBL_MANT_DIG</td>
<td>53</td>
</tr>
<tr>
<td>[L]DBL_DIG</td>
<td>15</td>
</tr>
<tr>
<td>[L]DBL_EPSILON</td>
<td>2.2204460492503131E-16</td>
</tr>
<tr>
<td>[L]DBL_MIN</td>
<td>2.2250738585072014E-308</td>
</tr>
<tr>
<td>[L]DBL_MIN_EXP</td>
<td>-1021</td>
</tr>
<tr>
<td>[L]DBL_MIN_10_EXP</td>
<td>-307</td>
</tr>
<tr>
<td>[L]DBL_MAX</td>
<td>1.7976931348623157E+308</td>
</tr>
<tr>
<td>[L]DBL_MAX_EXP</td>
<td>+1024</td>
</tr>
<tr>
<td>[L]DBL_MAX_10_EXP</td>
<td>+308</td>
</tr>
<tr>
<td>[L]DBL_HAS_SUBNORM</td>
<td>0</td>
</tr>
<tr>
<td>[L]DBL_TRUE_MIN</td>
<td>2.2250738585072014E-308</td>
</tr>
<tr>
<td>[L]DBL_DECIMAL_DIG</td>
<td>17</td>
</tr>
<tr>
<td>DECIMAL_DIG</td>
<td>17 (for double FP), 9 (for single FP)</td>
</tr>
<tr>
<td>FLT16_MANT_DIG</td>
<td>11</td>
</tr>
<tr>
<td>FLT16_DIG</td>
<td>3</td>
</tr>
<tr>
<td>FLT16_EPSILON</td>
<td>9.765625E-4F</td>
</tr>
<tr>
<td>FLT16_MIN</td>
<td>6.103515625E-05F</td>
</tr>
<tr>
<td>FLT16_MIN_EXP</td>
<td>-13</td>
</tr>
<tr>
<td>FLT16_MIN_10_EXP</td>
<td>-4</td>
</tr>
<tr>
<td>FLT16_MAX</td>
<td>65504.0F</td>
</tr>
<tr>
<td>FLT16_MAX_EXP</td>
<td>+16</td>
</tr>
<tr>
<td>FLT16_MAX_10_EXP</td>
<td>+4</td>
</tr>
<tr>
<td>FLT16_HAS_SUBNORM</td>
<td>1</td>
</tr>
<tr>
<td>FLT16_TRUE_MIN</td>
<td>_FLT16_MIN</td>
</tr>
<tr>
<td>FLT16_DECIMAL_DIG</td>
<td>5</td>
</tr>
</tbody>
</table>

### Macros in `<limits.h>`:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR_BIT</td>
<td>8</td>
</tr>
<tr>
<td>SCHAR_MIN</td>
<td>-SCHAR_MAX-1</td>
</tr>
<tr>
<td>SCHAR_MAX</td>
<td>0x7f</td>
</tr>
<tr>
<td>UCHAR_MAX</td>
<td>0xffU</td>
</tr>
<tr>
<td>CHAR_MIN</td>
<td>UCHAR_MIN or SCHAR_MIN if option --schar is set</td>
</tr>
<tr>
<td>CHAR_MAX</td>
<td>UCHAR_MAX or SCHAR_MAX if option --schar is set</td>
</tr>
</tbody>
</table>
### The limit macros in `<stdint.h>`

The limit macros in `<stdint.h>` for exact-width, minimum-width and fastest-width integer types have the same ranges as `char`, `short`, `int`, `long` and `long long`. Furthermore the following macros are defined:

<table>
<thead>
<tr>
<th>Macro <code>&lt;stdint.h&gt;</code></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT_MIN</td>
<td>-INT_MAX - 1</td>
</tr>
<tr>
<td>INT_MAX</td>
<td>0x7fffffff</td>
</tr>
<tr>
<td>UINT_MAX</td>
<td>0xffffffffU</td>
</tr>
<tr>
<td>LONG_MIN</td>
<td>-LONG_MAX - 1</td>
</tr>
<tr>
<td>LONG_MAX</td>
<td>0x7fffffffL</td>
</tr>
<tr>
<td>ULONG_MAX</td>
<td>0xffffffffUL</td>
</tr>
<tr>
<td>LONGLONG_MIN</td>
<td>-LONGLONG_MAX - 1</td>
</tr>
<tr>
<td>LONGLONG_MAX</td>
<td>0x7fffffffffffffffL</td>
</tr>
<tr>
<td>ULLONGLONG_MAX</td>
<td>0xffffffffffffffffULL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Macro <code>&lt;limits.h&gt;</code></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MB_LEN_MAX</td>
<td>1</td>
</tr>
<tr>
<td>SHRT_MIN</td>
<td>-SHRT_MAX - 1</td>
</tr>
<tr>
<td>SHRT_MAX</td>
<td>0x7fff</td>
</tr>
<tr>
<td>USHRT_MAX</td>
<td>0xffffU</td>
</tr>
<tr>
<td>INT_MIN</td>
<td>-INT_MAX - 1</td>
</tr>
<tr>
<td>INT_MAX</td>
<td>0x7fffffff</td>
</tr>
<tr>
<td>UINT_MAX</td>
<td>0xffffffffU</td>
</tr>
<tr>
<td>LONG_MIN</td>
<td>-LONG_MAX - 1</td>
</tr>
<tr>
<td>LONG_MAX</td>
<td>0x7fffffffL</td>
</tr>
<tr>
<td>ULONG_MAX</td>
<td>0xffffffffUL</td>
</tr>
<tr>
<td>LONGLONG_MIN</td>
<td>-LONGLONG_MAX - 1</td>
</tr>
<tr>
<td>LONGLONG_MAX</td>
<td>0x7fffffffffffffffL</td>
</tr>
<tr>
<td>ULLONGLONG_MAX</td>
<td>0xffffffffffffffffULL</td>
</tr>
</tbody>
</table>
• The result of attempting to indirectly access an object with automatic or thread storage duration from a thread other than the one with which it is associated (6.2.4).

Threads are not supported (__STDC_NO_THREADS__ = 1).

• The number, order, and encoding of bytes in any object (when not explicitly specified in this International Standard) (6.2.6.1).

All types are described in Section 1.1, Data Types.

• Whether any extended alignments are supported and the contexts in which they are supported (6.2.8).

For automatic objects, an extended alignment of 8 is supported. For statically allocated objects, extended alignments greater than or equal to 8 are supported.

• Valid alignment values other than those returned by an _Alignof expression for fundamental types, if any (6.2.8).

Any nonnegative integral power of two can be used as additional alignment value.

• The value of the result of the sizeof and _Alignof operators (6.5.3.4).

The values of the size and alignment of the data types are described in Section 1.1, Data Types. Divide the numbers by 8 to get bytes.

22.4. C11 Locale-specific Behavior

Locale-specific behavior is behavior that depends on local conventions of nationality, culture, and language that each implementation documents.

The following items describe the locale-specific characteristics, as indicated in Annex J.4 of the ISO/IEC 9899:2011 (E) standard.

• Additional members of the source and execution character sets beyond the basic character set (5.2.1).

The compiler accepts all one-byte characters in the host's default character set. Use of variable length encoded characters in the source file in comments and string literals is permitted.

In the implementation in the C library, the basic sequences of a multibyte character consist of single bytes (MB_LEN_MAX is set to 1). If you want full multibyte support, you need to change the C library. See the notes in the header files stdio.h and wchar.h for more information.

• The presence, meaning, and representation of additional multibyte characters in the execution character set beyond the basic character set (5.2.1.2).

Use of variable length encoded characters in the source file in comments and string literals is permitted.

• The shift states used for the encoding of multibyte characters (5.2.1.2).

A multibyte character must be a single byte when in the initial shift state.
- The direction of writing of successive printing characters (5.2.2).
  The direction of writing depends on the application and the display device.
- The decimal-point character (7.1.1).
  The default decimal-point character is a '.'.
- The set of printing characters (7.4, 7.30.2).
  The set of printing characters are the characters for which the `isprint()` function returns true. Printing characters are characters in the range 32 (space) to 126.
- The set of control characters (7.4, 7.30.2).
  The set of control characters are the characters for which the `iscntrl()` function returns true. Control characters are characters in the range 0 to 31 and 127.
- The sets of characters tested for by the `isalpha, isblank, islower, ispunct, isspace, isupper, iswalpha, iswblank, iswlower, iswpunct, iswspace, or iswupper` functions (7.4.1.2, 7.4.1.3, 7.4.1.7, 7.4.1.9, 7.4.1.10, 7.4.1.11, 7.30.2.1.2, 7.30.2.1.3, 7.30.2.1.7, 7.30.2.1.9, 7.30.2.1.10, 7.30.2.1.11).
  The characters tested for are specified in the following table.

<table>
<thead>
<tr>
<th>Function</th>
<th>Characters tested</th>
</tr>
</thead>
<tbody>
<tr>
<td>isalpha</td>
<td>a-z, A-Z</td>
</tr>
<tr>
<td>isblank</td>
<td>' ' (space), '\t' (tab)</td>
</tr>
<tr>
<td>islower</td>
<td>a-z</td>
</tr>
<tr>
<td>ispunct</td>
<td>!, &quot;, #, $, %, &amp;,, (, )*, +, -. , /=; ;&lt;, =, &gt;, @, [] ^, _, `{,</td>
</tr>
<tr>
<td>isspace</td>
<td>' ' (space), '\t', '\n', '\v', '\f', '\r'</td>
</tr>
<tr>
<td>isupper</td>
<td>A-Z</td>
</tr>
</tbody>
</table>

- The native environment (7.11.1.1).
  The native environment is the same as the "C" locale.
- Additional subject sequences accepted by the numeric conversion functions (7.22.1, 7.29.4.1).
  No additional subject sequences are accepted.
- The collation sequence of the execution character set (7.24.4.3, 7.29.4.4.2).
  Only the "C" locale is supported. The `strcoll()` function is the same as the `strcmp()` function. The `wcscoll()` function is the same as the `wcscmp()` function.
- The contents of the error message strings set up by the `strerror` function (7.24.6.2).
  The error message strings returned by `strerror()` depend on the argument. Typically, the values for the argument come from `errno.h`. For a list of messages see Section 14.2.5, `errno.h`).
• The formats for time and date (7.27.3.5, 7.29.5.1).

  English names for months and days are used.

  `%c` is replaced by the following date and time representation: `%a %b %e %H:%M:%S %Y`%n

  `%x` is replaced by the following date representation: `%m/%d/%Y`%n

  `%X` is replaced by the following time representation: `%H:%M:%S`%n

• Character mappings that are supported by the `towctrans` function (7.30.1).

  The character mappings supported by the `towctrans()` function are defined in `wctype.h`: `_to_lower` and `_to_upper`.

• Character classifications that are supported by the `iswctype` function (7.30.1).

  The character classifications supported by the `iswctype()` function are defined in `wctype.h`: `_alnum`, `_alpha`, `_cntrl`, `_digit`, `_graph`, `_lower`, `_print`, `_punct`, `_space`, `_upper`, `_xdigit` and `_blank`. 
Chapter 23. C++ Implementation-defined Behavior

The TASKING C++ compiler for ARM® fully supports the ISO C++14 standard, but some parts of the ISO C++14 standard are implementation-defined. This chapter describes how the implementation-defined areas and the implementation quantities of the C++ language are implemented in the TASKING C++ compiler for ISO C++14.

23.1. C++14 Implementation-defined Behavior

Implementation-defined behavior is unspecified behavior where each implementation documents how the choice is made.

The section numbers listed refer to the corresponding sections in the ISO/IEC 14882:2014 standard.

1.3.6 diagnostic message

The C++ compiler diagnostics are explained in Section 5.3, C++ Compiler Error Messages.

1.3.10 implementation-defined behavior

This chapter describes the C++ implementation-defined behavior.

1.3.11 implementation limits

See Section 23.2, C++14 Implementation Quantities.

1.4 Implementation compliance

Freestanding implementation, LLVM libc++ standard library.

1.7 The C++ memory model

There are 8 bits in a byte.

1.9 Program execution

The streams stdin, stdout and stderr are treated as interactive devices.

1.10 Multi-threaded executions and data races

There is only a single thread of execution.
2.2 Phases of translation

1. Use of variable length encoded characters in the source file in comments and string literals is permitted. A one-on-one mapping is done without interpretation of multibyte characters.

2. Sources of translation units containing definitions of templates that are being instantiated are required to be available.

2.3 Character sets

Only 8-bit characters are supported. The values of the execution character set are the same as that of the source character set. The same representation value is used for each member in the characters sets except for the escape sequences.

2.9 Header names

How the C++ compiler searches for include files is explained in Section 5.2, How the C++ Compiler Searches Include Files.

2.11 Identifiers

No multibyte character or any other non standard character may appear in identifiers.

2.14.3 Character literals

Multibyte characters are supported in character literals if option C++ compiler option --multibyte-chars is set, otherwise no multibyte character is allowed in character literals.

2.14.4 Floating literals

The round to nearest representable value is used if the scaled floating-point value is not representable.

2.14.5 String literals

Duplicated string literals are merged.

3.6.1 Main function

The function called at program startup (in cstart.c) is called main. The prototype for main in cstart.c is:

extern int main( int argc, char *argv[] );

3.6.2 Initialization of non-local variables

Dynamic initialization of a non-local variable with static storage duration is done before the first statement of main.
3.7.4.2 Deallocation functions

Any use of an invalid pointer value has undefined behavior.

3.7.4.3 Safely-derived pointers

Relaxed pointer safety is used.

3.9 Types

See the data types in Section 1.1, Data Types.

3.9.1 Fundamental types

By default char is the same as specifying unsigned char. With C++ compiler option --schar (-s) a plain char is the same as signed char.

A float has an exponent of 8 bits and a significand of 24 bits. A double or long double has an exponent of 11 bits and a significand of 53 bits. This is conform IEEE-754 for single precision and double precision floating-point. Internally the compiler uses a significand of 80 bits. The results of floating-point operations are rounded to the nearest IEEE-754 format.

3.9.2 Compound types

Pointers are 4 bytes.

3.11 Alignment

See the data types in Section 1.1, Data Types.

4.1 Lvalue-to-rvalue conversion

Any use of an invalid pointer value has undefined behavior.

4.7 Integral conversions

At compile time, when converting integer types and a value does not fit in a type, the compiler issues a warning and the value is truncated. At run-time no warning or signal is given and the value is truncated.

4.8 Floating point conversions

The round to nearest method is used. FLT_ROUNDS is ignored.

4.9 Floating-integral conversions

The round to nearest method is used. FLT_ROUNDS is ignored.
4.13 Integer conversion rank

See the data types in Section 1.1, Data Types.

5.2.2 Function call

Passing a potentially-evaluated argument of class type having a non-trivial copy constructor, a non-trivial move constructor, or a non-trivial destructor, with no corresponding parameter is supported. A warning (W1300) is issued in any of these cases. The argument is passed as if it was trivially copyable, i.e. no copy/move constructors and destructors are invoked to pass the argument.

5.2.8 Type identification

TypeId expression results are only of type std::type_info. No derived name classes are provided.

5.2.10 Reinterpret cast

Mapping remains the same. Mapping of pointers to addresses and vice versa uses int or larger type.

Converting a function pointer to an object pointer or vice versa is supported.

5.3.3 Sizeof

See the data types in Section 1.1, Data Types.

5.3.4 New

See the data types in Section 1.1, Data Types.

5.7 Additive operators

ptrdiff_t is defined as the smaller of int or long that can hold a pointer value.

5.8 Shift operators

The result of E1>>E2 is E1 right shifted E2 bit positions. If E1 has a signed type and a negative value, the shift behavior is implemented as an arithmetic shift. The empty position in the most significant bit is filled with a copy of the original most significant bit.

7 Declarations

See Section 1.8, Attributes.

7.1.6.1 The cv-qualifiers

Any reference to an object with volatile type results in an access. The order in which volatile objects are accessed is defined by the order expressed in the source code. References to non-volatile objects are scheduled in arbitrary order, within the constraints given by dependencies.
7.1.6.2 Simple type specifiers

By default char is the same as specifying unsigned char. With C++ compiler option --schar (-s) a plain char is the same as signed char.

int bit-fields are signed by default.

7.2 Enumeration declarations

The C++ compiler always treats enum types as int.

7.4 The asm declaration

See Section 1.7, Using Assembly in the C Source: __asm(). Both asm() and __asm() entities are supported.

7.5 Linkage specifications

Routines declared with extern "C" will not have name mangling performed on them. The default mode, extern "C++", performs name mangling. No other language linkages are supported.

8.4 Function definitions

The reserved identifier __func__ expands to the current function name (as well as macro __FUNCTION__). An error is issued if it is used outside of a function.

9.6 Bit-fields

Allocation of a bit-field starts at the least significant bit up to the most significant bit. If the following bit-field fits within the same unit, it is allocated starting at the next available bit.

A bit-field cannot straddle a storage-unit boundary. If insufficient space remains, the bit-field is put into the next unit.

int bit-fields are signed by default.

14 Templates

Only C and C++ linkages are supported.

14.7.1 Implicit instantiation

Recursive instantiations are limited to 64 by default.

15.5.1 The terminate() function

The stack is unwound before terminate() is called.
15.5.2 The unexpected() function

When `unexpected()` exits by throwing an exception the exception must not violate the exception specification that caused `unexpected()` to be called in the first place. If it does violate that exception specification, `std::bad_exception` is thrown provided it is permitted by the violated exception specification. If it is not permitted, `terminate()` is called.

16.1 Conditional inclusion

A character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set. A single-character character constant in a constant expression that controls conditional inclusion may have a negative value.

16.2 Source file inclusion

Sequences in header names are mapped to file names as is. The backslash "\" is not interpreted as an escape sequence. The backslash "\" (Windows) or forward slash "/" (Windows and UNIX) is interpreted as a standard directory separator. How the C++ compiler searches for include files is explained in Section 5.2, How the C++ Compiler Searches Include Files.

16.6 Pragma directive

The C++ compiler supports the pragmas as explained in Section 2.7, Pragmas to Control the C++ Compiler.

16.8 Predefined macro names

The C++ compiler supports the predefined macros as explained in Section 2.8, Predefined Macros.

17.6.5.8 Reentrancy

Any function with `__attribute__((const))` is reentrant.

17.6.5.12 Restrictions on exception handling

See the C++14 library documentation at releases.llvm.org.

17.6.5.14 Value of error codes

See the C++14 library documentation at releases.llvm.org.

18.2 Types

`NULL` has a value of 0.

`ptrdiff_t` is defined as the smaller of `int` or `long` that can hold a pointer value.

`size_t` is defined as `unsigned int`. 
18.3.1 In general

See header files `<limits>`, `<climits>` and `<cfloat>`.

18.3 Start and termination

`exit()` returns the value of argument status; `EXIT_SUCCESS` has a value of 0; `EXIT_FAILURE` has a value of 1.

18.6.2.1 Class bad_alloc

`bad_alloc::what()` returns an empty string.

18.6.2.2 Class bad_array_length

`bad_array_length::what()` is not implemented.

18.6.2.3 Class bad_array_new_length

`bad_array_new_length::what()` is not implemented.

18.7.1 Class type_info

`typeid(T).name()` returns the mangled name of type T.

18.7.2 Class bad_cast

`bad_cast::what()` returns an empty string.

18.7.3 Class bad_typeid

`bad_typeid::what()` returns an empty string.

18.8.1 Class exception

`exception::what()` returns an empty string.

18.8.2 Class bad_exception

`bad_exception::what()` returns an empty string.

18.10 Other runtime support

Any non POF (plain old functions) signal handler yields undefined behavior.

20.8.4 Pointer safety

See the C++14 library documentation at releases.llvm.org.
20.8.9.1 allocator members

See the data types in Section 1.1, Data Types.

20.8.11 Temporary buffers

See the data types in Section 1.1, Data Types.

20.9.2.2.1 shared_ptr

Throws bad_alloc.

20.10.9.1.4 Placeholders

There are 10 placeholders. See header file <functional> for details.

20.11.7.6 Other transformations

See the data types in Section 1.1, Data Types.

20.13.7.1 Class system_clock

Values are truncated to the required precision.

21.2.2 traits typedefs

See the C++14 library documentation at releases.llvm.org.

21.2.3.2 struct char_traits<char16_t>

See the C++14 library documentation at releases.llvm.org.

21.2.3.3 struct char_traits<char32_t>

See the C++14 library documentation at releases.llvm.org.

21.2.3.4 struct char_traits<wchar_t>

See the C++14 library documentation at releases.llvm.org.

21.4 Class template basic_string

See the C++14 library documentation at releases.llvm.org.

22.3.1 Class locale

See the C++14 library documentation at releases.llvm.org.
22.3.1.1.1 Type locale::category
See the C++14 library documentation at releases.llvm.org.

22.3.1.2 locale constructors and destructor
See the C++14 library documentation at releases.llvm.org.

22.3.1.3 locale members
See the C++14 library documentation at releases.llvm.org.

22.3.1.5 locale static members
See the C++14 library documentation at releases.llvm.org.

22.4.1.3 ctype specializations

table_size = 256

22.4.1.4 Class template codecvt
See the C++14 library documentation at releases.llvm.org.

22.4.5.1.2 time_get virtual functions
See the C++14 library documentation at releases.llvm.org.

22.4.5.3.1 time_put members
See the C++14 library documentation at releases.llvm.org.

22.4.5.3.2 time_put virtual functions
See the C++14 library documentation at releases.llvm.org.

22.4.7.1.2 messages virtual functions
See the C++14 library documentation at releases.llvm.org.

23.3.2.1 Class template array overview
See the C++14 library documentation at releases.llvm.org.

23.3.3.1 Class template deque overview
See the C++14 library documentation at releases.llvm.org.
23.3.4.1 Class template dynarray overview
See the C++14 library documentation at releases.llvm.org.

23.3.5.1 Class template forward_list overview
See the C++14 library documentation at releases.llvm.org.

23.3.6.1 Class template list overview
See the C++14 library documentation at releases.llvm.org.

23.3.7.1 Class template vector overview
See the C++14 library documentation at releases.llvm.org.

23.3.8 Class vector<bool>
See the C++14 library documentation at releases.llvm.org.

23.4.4.1 Class template map overview
See the C++14 library documentation at releases.llvm.org.

23.4.5.1 Class template multimap overview
See the C++14 library documentation at releases.llvm.org.

23.4.6.1 Class template set overview
See the C++14 library documentation at releases.llvm.org.

23.4.7.1 Class template multiset overview
See the C++14 library documentation at releases.llvm.org.

23.5.4.1 Class template unordered_map overview
See the C++14 library documentation at releases.llvm.org.

23.5.4.2 unordered_map constructors
See the C++14 library documentation at releases.llvm.org.

23.5.5.1 Class template unordered_multimap overview
See the C++14 library documentation at releases.llvm.org.
23.5.5.2 unordered_multimap constructors
See the C++14 library documentation at releases.llvm.org.

23.5.6.1 Class template unordered_set overview
See the C++14 library documentation at releases.llvm.org.

23.5.6.2 unordered_set constructors
See the C++14 library documentation at releases.llvm.org.

23.5.7.1 Class template unordered_multiset overview
See the C++14 library documentation at releases.llvm.org.

23.5.7.2 unordered_multiset constructors
See the C++14 library documentation at releases.llvm.org.

25.3.12 Random shuffle
See the C++14 library documentation at releases.llvm.org.

26.4.8 complex transcendentals
\[ \text{pow}(0, 0) = (\text{NaN}, \text{NaN}) \]

26.5.5 Engines and engine adaptors with predefined parameters
See the C++14 library documentation at releases.llvm.org.

26.5.6 Class random_device
See the C++14 library documentation at releases.llvm.org.

26.5.8.1 In general
See the C++14 library documentation at releases.llvm.org.

26.8 C library
See the C++14 library documentation at releases.llvm.org.

27.2.2 Positioning type limitations
See the C++14 library documentation at releases.llvm.org.
27.5 Iostreams base classes
See the C++14 library documentation at releases.llvm.org.

27.5.3.4 ios_base static members
See the C++14 library documentation at releases.llvm.org.

27.5.5.4 basic_ios flags functions
See the C++14 library documentation at releases.llvm.org.

27.7.2.1.3 Class basic_istream::sentry
See the C++14 library documentation at releases.llvm.org.

27.8.2.1 basic_stringbuf constructors
See the C++14 library documentation at releases.llvm.org.

27.8.2.4 Overridden virtual functions
See the C++14 library documentation at releases.llvm.org.

27.9.1.2 basic_filebuf constructors
See the C++14 library documentation at releases.llvm.org.

27.9.1.5 Overridden virtual functions
See the C++14 library documentation at releases.llvm.org.

28.5.1 Bitmask type syntax_option_type
See the C++14 library documentation at releases.llvm.org.

28.5.3 Implementation-defined error_type
Values of enum error_type are present in header <regex>.

28.7 Class template regex_traits
See the C++14 library documentation at releases.llvm.org.

28.8 Class template basic_regex
See the C++14 library documentation at releases.llvm.org.
28.10 Class template match_results

See the C++14 library documentation at releases.llvm.org.

30.2.3 Native handles

native_handle_type is defined either as pthread_t or as pthread_mutex_t. Both pthread_t and pthread_mutex_t are defined as int.

30.3.1 Class thread

native_handle_type is defined as pthread_t, which is defined as int.

30.4.1.2.1 Class mutex

native_handle_type is not used in this class.

30.4.1.2.2 Class recursive_mutex

native_handle_type is defined as pthread_mutex_t, which is defined as int.

30.4.1.3.1 Class timed_mutex

native_handle_type is not used in this class.

30.4.1.3.2 Class recursive_timed_mutex

native_handle_type is not used in this class.

30.5.1 Class condition_variable

native_handle_type is not used in this class.

30.6.1 Overview

Values of enum class future_errc:

future_already_retrieved = 1
promise_already_satisfied = 2
no_state = 3
broken_promise = 4

Values of enum class launch:

async = 1
defered = 2
any = async | deferred
A.2 Lexical conventions

No multibyte character or any other non standard character may appear in identifiers.

C.1.10 Clause 16: preprocessing directives

Macro __STDC__ always expands to 0.

C.4.2.4 Macro NULL

Macro NULL has a value of 0.

D.6 Old iostreams members

See the C++14 library documentation at releases.llvm.org.

23.2. C++14 Implementation Quantities

The following items describe the actual limits of the TASKING C++ compiler as indicated in Annex B, Implementation quantities of the ISO/IEC 14882:2014 standard. In most cases the C++ compiler is limited only by available memory on the user's PC. This is marked with "(no limit)".

<table>
<thead>
<tr>
<th>Item</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nesting levels of compound statements, iteration control structures,</td>
<td>(no limit)</td>
</tr>
<tr>
<td>and selection control structures</td>
<td></td>
</tr>
<tr>
<td>Nesting levels of conditional inclusion</td>
<td>(no limit)</td>
</tr>
<tr>
<td>Pointer, array, and function declarators (in any combination)</td>
<td>(no limit)</td>
</tr>
<tr>
<td>modifying a class, arithmetic, or incomplete type in a declaration</td>
<td></td>
</tr>
<tr>
<td>Nesting levels of parenthesized expressions within a full-expression</td>
<td>(no limit)</td>
</tr>
<tr>
<td>Number of characters in an internal identifier or macro name</td>
<td>9999999</td>
</tr>
<tr>
<td>Number of characters in an external identifier</td>
<td>9999999</td>
</tr>
<tr>
<td>External identifiers in one translation unit</td>
<td>(no limit)</td>
</tr>
<tr>
<td>Identifiers with block scope declared in one block</td>
<td>(no limit)</td>
</tr>
<tr>
<td>Macro identifiers simultaneously defined in one translation unit</td>
<td>(no limit)</td>
</tr>
<tr>
<td>Parameters in one function definition</td>
<td>(no limit)</td>
</tr>
<tr>
<td>Arguments in one function call</td>
<td>(no limit)</td>
</tr>
<tr>
<td>Parameters in one macro definition</td>
<td>(no limit)</td>
</tr>
<tr>
<td>Arguments in one macro invocation</td>
<td>(no limit)</td>
</tr>
<tr>
<td>Item</td>
<td>Limit</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td>Characters in one logical source line</td>
<td>(no limit)</td>
</tr>
<tr>
<td>Characters in a string literal (after concatenation)</td>
<td>4294967295</td>
</tr>
<tr>
<td>Size of an object</td>
<td>4294967295</td>
</tr>
<tr>
<td>Nesting levels for <code>#include</code> files</td>
<td>(no limit, however recursive includes of the same file are limited to 10 levels)</td>
</tr>
<tr>
<td>Case labels for a <code>switch</code> statement (excluding those for any nested <code>switch</code> statements)</td>
<td>(no limit)</td>
</tr>
<tr>
<td>Data members in a single class</td>
<td>(no limit)</td>
</tr>
<tr>
<td>Enumeration constants in a single enumeration</td>
<td>(no limit)</td>
</tr>
<tr>
<td>Levels of nested class definitions in a single member-specification</td>
<td>(no limit)</td>
</tr>
<tr>
<td>Functions registered by <code>atexit()</code></td>
<td>32</td>
</tr>
<tr>
<td>Functions registered by <code>at_quick_exit()</code></td>
<td>32</td>
</tr>
<tr>
<td>Direct and indirect base classes</td>
<td>(no limit)</td>
</tr>
<tr>
<td>Direct base classes for a single class</td>
<td>(no limit)</td>
</tr>
<tr>
<td>Members declared in a single class</td>
<td>(no limit)</td>
</tr>
<tr>
<td>Final overriding virtual functions in a class, accessible or not</td>
<td>(no limit)</td>
</tr>
<tr>
<td>Direct and indirect virtual bases of a class</td>
<td>(no limit)</td>
</tr>
<tr>
<td>Static members of a class</td>
<td>(no limit)</td>
</tr>
<tr>
<td>Friend declarations in a class</td>
<td>(no limit)</td>
</tr>
<tr>
<td>Access control declarations in a class</td>
<td>(no limit)</td>
</tr>
<tr>
<td>Member initializers in a constructor definition</td>
<td>(no limit)</td>
</tr>
<tr>
<td>Scope qualifications of one identifier</td>
<td>(no limit)</td>
</tr>
<tr>
<td>Nested external specifications</td>
<td>(no limit)</td>
</tr>
<tr>
<td>Recursive <code>constexpr</code> function invocations</td>
<td>255</td>
</tr>
<tr>
<td>Full-expressions evaluated within a core constant expression</td>
<td>(no limit)</td>
</tr>
<tr>
<td>Template arguments in a template declaration</td>
<td>65535</td>
</tr>
<tr>
<td>Recursively nested template instantiations, including substitution during template argument deduction (14.8.2)</td>
<td>64 (this is the default, but you can change it with C++ compiler option <code>--pending-instantiations</code>)</td>
</tr>
<tr>
<td>Handlers per <code>try</code> block</td>
<td>(no limit)</td>
</tr>
<tr>
<td>Throw specifications on a single function declaration</td>
<td>(no limit)</td>
</tr>
<tr>
<td>Number of placeholders (20.10.9.1.4)</td>
<td>10</td>
</tr>
</tbody>
</table>