



ARM Embedded Tools Reference

Software, hardware, documentation and related materials:

Copyright © 2008 Altium Limited. All Rights Reserved.

The material provided with this notice is subject to various forms of national and international intellectual property protection, including but not limited to copyright protection. You have been granted a non-exclusive license to use such material for the purposes stated in the end-user license agreement governing its use. In no event shall you reverse engineer, decompile, duplicate, distribute, create derivative works from or in any way exploit the material licensed to you except as expressly permitted by the governing agreement. Failure to abide by such restrictions may result in severe civil and criminal penalties, including but not limited to fines and imprisonment. Provided, however, that you are permitted to make one archival copy of said materials for back up purposes only, which archival copy may be accessed and used only in the event that the original copy of the materials is inoperable. Altium, Altium Designer, Board Insight, DXP, Innovation Station, LiveDesign, NanoBoard, NanoTalk, OpenBus, P-CAD, SimCode, Situs, TASKING, and Topological Autorouting and their respective logos are trademarks or registered trademarks of Altium Limited or its subsidiaries. All other registered or unregistered trademarks referenced herein are the property of their respective owners and no trademark rights to the same are claimed. v8.0 31/3/08

Table of Contents

C Language	1-1
1.1 Introduction	1-1
1.2 Data Types	1-2
1.2.1 Changing the Alignment: <code>__unaligned</code> , <code>__packed</code> and <code>__align()</code>	1-2
1.3 Placing an Object at an Absolute Address: <code>__at()</code>	1-3
1.4 Using Assembly in the C Source: <code>__asm()</code>	1-4
1.5 Pragmas to Control the Compiler	1-8
1.6 Predefined Preprocessor Macros	1-12
1.7 Switch Statement	1-13
1.8 Functions	1-14
1.8.1 Parameter Passing	1-14
1.8.2 Function Return Types	1-14
1.8.3 Inlining Functions: <code>inline</code> / <code>__noinline</code>	1-15
1.8.4 Intrinsic Functions	1-16
1.8.5 Interrupt Functions / Exception Handlers	1-18
1.8.5.1 Defining an Exception Handler: <code>__interrupt</code> keywords	1-18
1.8.5.2 Interrupt Frame: <code>__frame()</code>	1-19
1.9 Libraries	1-20
1.9.1 Printf and Scanf Routines	1-20
C++ Language	2-1
2.1 Introduction	2-1
2.2 C++ Language Extension Keywords	2-1
2.3 C++ Dialect Accepted	2-1
2.3.1 Standard Language Features Accepted	2-1
2.3.2 C++0x Language Features Accepted	2-3
2.3.3 Anachronisms Accepted	2-4
2.3.4 Extensions Accepted in Normal C++ Mode	2-4
2.4 GNU Extensions	2-5
2.5 Namespace Support	2-13
2.6 Template Instantiation	2-14
2.6.1 Automatic Instantiation	2-14
2.6.2 Instantiation Modes	2-15
2.6.3 Instantiation <code>#pragma</code> Directives	2-16
2.6.4 Implicit Inclusion	2-17
2.6.5 Exported Templates	2-18
2.6.5.1 Finding the Exported Template Definition	2-18
2.6.5.2 Secondary Translation Units	2-19
2.6.5.3 Libraries with Exported Templates	2-19
2.7 Inlining Functions	2-20
2.8 Extern Inline Functions	2-20
2.9 Pragmas to Control the C++ Compiler	2-21
2.10 Predefined Macros	2-22
2.11 Precompiled Headers	2-24
2.11.1 Automatic Precompiled Header Processing	2-24
2.11.2 Manual Precompiled Header Processing	2-26
2.11.3 Other Ways to Control Precompiled Headers	2-26
2.11.4 Performance Issues	2-26

Libraries	3-1
3.1	Introduction 3-1
3.2	Library Functions 3-3
3.2.1	assert.h 3-3
3.2.2	complex.h 3-3
3.2.3	ctype.h and wctype.h 3-4
3.2.4	dbg.h 3-5
3.2.5	errno.h 3-5
3.2.6	fcntl.h 3-6
3.2.7	fenv.h 3-6
3.2.8	float.h 3-6
3.2.9	inttypes.h and stdint.h 3-6
3.2.10	io.h 3-7
3.2.11	iso646.h 3-7
3.2.12	limits.h 3-7
3.2.13	locale.h 3-7
3.2.14	malloc.h 3-8
3.2.15	math.h and tgmth.h 3-8
3.2.16	setjmp.h 3-11
3.2.17	signal.h 3-12
3.2.18	stdarg.h 3-12
3.2.19	stdbool.h 3-12
3.2.20	stddef.h 3-12
3.2.21	stdint.h 3-13
3.2.22	stdio.h and wchar.h 3-13
3.2.23	stdlib.h and wchar.h 3-18
3.2.24	string.h and wchar.h 3-20
3.2.25	time.h and wchar.h 3-22
3.2.26	unistd.h 3-23
3.2.27	wchar.h 3-24
3.2.28	wctype.h 3-25
Assembly Language	4-1
4.1	Assembly Syntax 4-1
4.2	Assembler Significant Characters 4-2
4.3	Operands of an Assembly Instruction 4-2
4.4	Symbol Names 4-2
4.4.1	Predefined Preprocessor Symbols 4-3
4.5	Registers 4-3
4.6	Assembly Expressions 4-3
4.6.1	Numeric Constants 4-4
4.6.2	Strings 4-4
4.6.3	Expression Operators 4-4
4.7	Built-in Assembly Functions 4-5
4.7.1	Overview of Built-in Assembly Functions 4-5
4.7.2	Detailed Description of Built-in Assembly Functions 4-6
4.8	Assembler Directives 4-9
4.8.1	Overview of Assembler Directives 4-9
4.8.2	Detailed Description of Assembler Directives 4-11
4.9	Macro Operations 4-46
4.9.1	Defining a Macro 4-46
4.9.2	Calling a Macro 4-46
4.9.3	Using Operators for Macro Arguments 4-47
4.9.4	Using the .FOR and .REPEAT Directives as Macros 4-49
4.9.5	Conditional Assembly 4-49
4.10	Generic Instructions 4-51
4.10.1	ARM Generic Instructions 4-51
4.10.2	ARM and Thumb-2 32-bit Generic Instructions 4-51

4.10.3	Thumb 16-bit Generic Instructions	4-53
Run-time Environment		5-1
5.1	Startup Code	5-1
5.2	Reset Handler and Vector Table	5-2
5.3	Stack and Heap	5-5
Tool Options		6-1
6.1	C Compiler Options	6-1
6.2	C++ Compiler Options	6-58
6.3	Assembler Options	6-162
6.4	Linker Options	6-196
6.5	Control Program Options	6-239
6.6	Make Utility Options	6-291
6.7	Librarian Options	6-317
List File Formats		7-1
7.1	Assembler List File Format	7-1
7.2	Linker Map File Format	7-3
Object File Formats		8-1
8.1	ELF/DWARF Object Format	8-1
8.2	Motorola S-Record Format	8-2
8.3	Intel Hex Record Format	8-5
Linker Script Language		9-1
9.1	Introduction	9-1
9.2	Structure of a Linker Script File	9-1
9.3	Syntax of the Linker Script Language	9-3
9.3.1	Preprocessing	9-3
9.3.2	Lexical Syntax	9-3
9.3.3	Identifiers	9-4
9.3.4	Expressions	9-4
9.3.5	Built-in Functions	9-5
9.3.6	LSL Definitions in the Linker Script File	9-6
9.3.7	Memory and Bus Definitions	9-6
9.3.8	Architecture Definition	9-7
9.3.9	Derivative Definition	9-9
9.3.10	Processor Definition and Board Specification	9-10
9.3.11	Section Layout Definition and Section Setup	9-10
9.4	Expression Evaluation	9-13
9.5	Semantics of the Architecture Definition	9-14
9.5.1	Defining an Architecture	9-15
9.5.2	Defining Internal Buses	9-15
9.5.3	Defining Address Spaces	9-15
9.5.4	Mappings	9-18
9.6	Semantics of the Derivative Definition	9-20
9.6.1	Defining a Derivative	9-20
9.6.2	Instantiating Core Architectures	9-20
9.6.3	Defining Internal Memory and Buses	9-21
9.7	Semantics of the Board Specification	9-22
9.7.1	Defining a Processor	9-22
9.7.2	Instantiating Derivatives	9-22
9.7.3	Defining External Memory and Buses	9-23
9.8	Semantics of the Section Setup Definition	9-24
9.8.1	Setting up a Section	9-24

9.9	Semantics of the Section Layout Definition	9-25
9.9.1	Defining a Section Layout	9-25
9.9.2	Creating and Locating Groups of Sections	9-26
9.9.3	Creating or Modifying Special Sections	9-30
9.9.4	Creating Symbols	9-32
9.9.5	Conditional Group Statements	9-33

CERT C Secure Coding Standard	10-1
--------------------------------------	-------------

10.1	Preprocessor (PRE)	10-1
10.2	Declarations and Initialization (DCL)	10-2
10.3	Expressions (EXP)	10-2
10.4	Integers (INT)	10-3
10.5	Floating Point (FLP)	10-3
10.6	Arrays (ARR)	10-3
10.7	Characters and Strings (STR)	10-4
10.8	Memory Management (MEM)	10-4
10.9	Environment (ENV)	10-4
10.10	Signals (SIG)	10-5
10.11	Miscellaneous (MSC)	10-5

MISRA-C Rules	11-1
----------------------	-------------

11.1	MISRA-C:1998	11-1
11.2	MISRA-C:2004	11-5

Index

Manual Purpose and Structure

Windows Users

The documentation explains and describes how to use the TASKING ARM toolset to program a ARM processor.

You can use the tools either with the graphical Altium Designer or from the command line in a command prompt window.

Structure

The toolset documentation consists of a user's manual (*Using the ARM Embedded Tools*), which includes a Getting Started section, and a separate reference manual (this manual).

Start by reading the *Getting Started* in Chapter 1 of the user's manual.

The other chapters in the user's manual explain how to use the compiler, assembler, linker and the various utilities.

Once you are familiar with these tools, you can use this reference manual to lookup specific options and details to make full use of the TASKING toolset.

The reference manual describes the C++ language implementation, the C language implementation and the assembly language.

Short Table of Contents

Chapter 1: C Language

The TASKING C compilers are fully compatible with ISO-C. This chapter describes the specific target 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 following language extensions are described:

- Data types
- Keywords
- Function qualifiers
- Intrinsic functions
- Pragmas
- Predefined macros

Chapter 2: C++ Language

Describes the C++ language implementation. The C++ language itself is not described.

Chapter 3: Libraries

Contains overviews of all library functions you can use in your C source. First libraries are listed per header file that contains the prototypes. These tables also show the level of implementation per function. Second, all library functions are listed and discussed into detail.

Chapter 4: Assembly Language

Describes the specific features of the assembly language as well as 'directives', which are pseudo instructions that are interpreted by the assembler.

Chapter 5: Run-time Environment

Describes the startup code used by the C compiler, the vector table, the stack layout and the heap.

Chapter 6: Tool Options

Contains a description of all tool options:

- C compiler options
- C++ compiler options
- Assembler options
- Linker options
- Control program options
- Make utility options
- Librarian options

Chapter 7: List File Formats

Contains a description of the following list file formats:

- Assembler List File Format
- Linker Map File Format

Chapter 8: Object File Formats

Contains a description of the following object file formats:

- ELF/DWARF 2 Object Format
- Motorola S-Record Format
- Intel Hex Record Format

Chapter 9: Linker Script Language

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

Chapter 10: CERT C Secure Coding Standard

Contains an overview of the supported CERT C Secure Coding Standard recommendations and rules.

Chapter 11: MISRA-C Rules

Contains a description the supported and unsupported MISRA-C code checking rules.

Conventions Used in this Manual

Notation for syntax

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

bold	Type this part of the syntax literally.
<i>italics</i>	Substitute the italic word by an instance. For example: <i>filename</i> Type the name of a file in place of the word <i>filename</i> .
{ }	Encloses a list from which you must choose an item.
[]	Encloses items that are optional. For example carm [-?] Both carm and carm -? are valid commands.
	Separates items in a list. Read it as OR.
...	You can repeat the preceding item zero or more times.

Example

carm [*option*]... *filename*

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

```
carm test.c
carm -g test.c
carm -g -s test.c
```

Not valid is:

```
carm -g
```

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

Icons

The following illustrations are used in this manual:



Note: notes give you extra information.



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



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



Command line: type your input on the command line.



Reference: follow this reference to find related topics.

Related Publications

C Standards

- ISO/IEC 9899:1999(E), Programming languages – C [ISO/IEC]
More information on the standards can be found at <http://www.ansi.org>
- DSP-C, An Extension to ISO/IEC 9899:1999(E),
Programming languages – C [TASKING, TK0071-14]

C++ Standards

- The C++ Programming Language (second edition)
by Bjarne Strastrup (1991, Addison Wesley)
- ISO/IEC 14882:1998 C++ standard [ANSI]
More information on the standards can be found at <http://www.ansi.org>
- The Annotated C++ Reference Manual
by Margaret A. Ellis and Bjarne Strastrup (1990, Addison Wesley)

CERT C Secure Coding Standard

- The CERT C Secure Coding Standard
by Robert C. Seacord (October 2008, Addison Wesley)
- The CERT C Secure Coding Standard web site <http://www.securecoding.cert.org/>
For general information about CERT secure coding, see <http://www.cert.org/secure-coding>

MISRA-C

- Guidelines for the Use of the C Language in Vehicle Based Software [MIRA limited, 1998]
See also <http://www.misra.org.uk>
- MISRA-C:2004: Guidelines for the use of the C Language in critical systems [MIRA limited, 2004]
See also <http://www.misra-c.com>

TASKING Tools

- Using the ARM Embedded Tools
[Altium, GU0116]

ARM

- ARM Architecture Reference Manual – ARM DDI 0100I
[2005, ARM Limited]
- ARM v7-M Architecture Application Level Reference Manual – ARM DDI 0405A-01
[2006, ARM Limited]



1 C Language

Summary

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.

1.1 Introduction

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

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

- 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
- attribute to specify absolute addresses
- 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.

1.2 Data Types

The TASKING C compiler for the ARM architecture (**carm**) supports the following data types:

Type	C Type	Size (bit)	Align (bit)	Limits
Boolean	_Bool	8	8	0 or 1
Character	char signed char	8	8	$-2^7 \dots 2^7 - 1$
	unsigned char	8	8	$0 \dots 2^8 - 1$
Integral	short signed short	16	16	$-2^{15} \dots 2^{15} - 1$
	unsigned short	16	16	$0 \dots 2^{16} - 1$
	enum	32	32	$-2^{31} \dots 2^{31} - 1$
	int signed int long signed long	32	32	$-2^{31} \dots 2^{31} - 1$
	unsigned int unsigned long	32	32	$0 \dots 2^{32} - 1$
	long long signed long long	64	64	$-2^{63} \dots 2^{63} - 1$
	unsigned long long	64	64	$0 \dots 2^{64} - 1$
Pointer	pointer to function or data	32	32	$0 \dots 2^{32} - 1$
Floating-Point	float (23-bit mantissa)	32	32	$-3.402\text{E}+38 \dots -1.175\text{E}-38$ $1.175\text{E}-38 \dots 3.402\text{E}+38$
	double long double (52-bit mantissa)	64	64	$-1.798\text{E}+308 \dots -2.225\text{E}-308$ $2.225\text{E}-308 \dots 1.798\text{E}+308$
	_Imaginary float	32	32	$-3.402\text{E}+38\text{i} \dots -1.175\text{E}-38\text{i}$ $1.175\text{E}-38\text{i} \dots 3.402\text{E}+38\text{i}$
	_Imaginary double _Imaginary long double	64	64	$-1.798\text{E}+308\text{i} \dots -2.225\text{E}-308\text{i}$ $2.225\text{E}-308\text{i} \dots 1.798\text{E}+308\text{i}$
	_Complex float	64	32	real part + imaginary part
	_Complex double _Complex long double	128	64	real part + imaginary part

Table 1-1: Data Types for the ARM

1.2.1 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 bit for bit-fields or one byte for other objects or 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.

Example:

```
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:

```
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:

```
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 Placing an Object at an Absolute Address: `__at()`

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

Examples

```
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`.

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

The variable `i` is placed at address `0x1000` and is initialized at 1.

```
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.

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

1.4 Using Assembly in the C Source: `__asm()`

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

Furthermore, assembly blocks are not interpreted by the compiler: they are regarded as a black box. So, it is your responsibility to make sure that the assembly block is syntactically correct.

General syntax of the `__asm` keyword

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

<i>instruction_template</i>	Assembly instructions that may contain parameters from the input list or output list in the form: <i>%parm_nr</i>
<i>%parm_nr[.regnum]</i>	Parameter number in the range 0 .. 9. With the optional <i>.regnum</i> you can access an individual register from a register pair.
<i>output_param_list</i>	[[" <i>=[&]constraint_char</i> "(<i>C_expression</i>)],...]
<i>input_param_list</i>	[[" <i>constraint_char</i> "(<i>C_expression</i>)],...]
<i>&</i>	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.
<i>constraint_char</i>	Constraint character: the type of register to be used for the <i>C_expression</i> .
<i>C_expression</i>	Any C expression. For output parameters it must be an <i>lvalue</i> , that is, something that is legal to have on the left side of an assignment.
<i>register_save_list</i>	[[" <i>register_name</i> "],...]
<i>register_name:q</i>	Name of the register you want to reserve.
<pre>char a, b; int result;</pre>	

```
void main(void)  
{  
    a = 3;  
    b = 4;  
    __asm( "ADD %0,%1,%2" : "=r"(result): "r"(a), "r"(b) );  
}
```

`%0` corresponds with the first C variable, `%1` with the second and so on.

Generated assembly code:

```
main: .type func
      ldr    r1, .L2
      mov    r0, #3
      strb   r0, [r1, #0]
      mov    r2, #4
      strb   r2, [r1, #1]
      ADD    r0, r0, r2
      str    r0, [r1, #4]
      bx     lr
      .size  main, $-main
      .align 4
.L2:
      .dw    a
```

Specifying registers for C variables

With a *constraint character* you specify the register type for a parameter. In the example above, the `r` is used to force the use of registers (Rn) for the parameters `a` and `b`.

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

Constraint character	Type	Operand	Remark
R	general purpose register (64 bits)	r0 .. r11	Thumb mode r0 .. r7 Based on the specified register, a register pair is formed (64-bit). For example r0r1.
r	general purpose register	r0 .. r11, lr	Thumb mode r0 .. r7
i	immediate value	#value	
l	label	<i>label</i>	
m	memory label	<i>variable</i>	stack or memory operand, a fixed address
<i>number</i>	other operand	same as % <i>number</i>	Input constraint only. The <i>number</i> must refer to an output parameter. Indicates that % <i>number</i> and <i>number</i> are the same register. Use % <i>number</i> .0 and % <i>number</i> .1 to indicate the first and second half of a register pair when used in combination with R.

Table 1-2: Available input/output operand constraints for the ARM

Loops and conditional jumps

The compiler does not detect loops that are coded 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 contained in the same statement.

Example 1: no input or output

A simple example without input or output parameters. You can use any instruction or label. Note that you can use standard C escape sequences.

```
__asm( "nop\n\t"
      "nop" );
```

Generated code:

```
nop
nop
```

Example 2: using output parameters

Assign the result of inline assembly to a variable. 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. Finally, the compiler generates code to assign the result to the output variable.

```
char out;

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

Generated assembly code:

```
mov     r0,#0xff
ldr     r1,.L2
strb    r0,[r1,#0]
bx      lr
.size   main,$-main
.align  4
.L2:
.dw     out
```

Example 3: using input and output parameters

Add two C variables and assign the result to a third C variable. Registers are used for the input parameters (constraint `r`, `%1` for `a` and `%2` for `b` in the instruction template) and for the output parameter (constraint `r`, `%0` for `result` 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.

```
char a, b;
int result;

void main(void)
{
    a = 3;
    b = 4;
    __asm( "ADD %0,%1,%2" : "=r"(result): "r"(a), "r"(b) );
}
```

Generated assembly code:

```
main: .type func
ldr     r1,.L2
mov     r0,#3
strb    r0,[r1,#0]
mov     r2,#4
strb    r2,[r1,#1]
ADD     r0,r0,r2
str     r0,[r1,#4]
bx      lr
.size   main,$-main
.align  4
.L2:
.dw     a
```


Example 4: reserve 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 3*, but now register R0 is a reserved register. You can do this by adding a reserved register list (: "R0"). As you can see in the generated assembly code, register R0 is not used (the first register used is R1).

```
char a, b;
int result;

void main(void)
{
    a = 3;
    b = 4;
    __asm( "ADD %0,%1,%2" : "=r"(result): "r"(a), "r"(b) : "R0" );
}
```

Generated assembly code:

```
main: .type func
      ldr    r2, .L2
      mov    r1, #3
      strb   r1, [r2, #0]
      mov    r3, #4
      strb   r3, [r2, #1]
      ADD    r1, r1, r3
      str    r1, [r2, #4]
      bx     lr
      .size  main, $-main
      .align 4
.L2:
      .dw    a
```

Example 5: input and output are the same

If the input and output must be the same you must use a number constraint. The following example increments the value of the input variable `invar` and returns this value to `outvar`. Since the assembly instruction has to use only one register, the return value has to go in the same place as the input value. Parameter `%0` corresponds to `outvar`. To indicate that `invar` uses the same register as `outvar`, the input constraint `'0'` is used which indicates that `invar` also corresponds to `%0`.

```
int outvar;

void increment(int invar)
{
    __asm ("add %0, %1, #1": "=r"(outvar): "0"(invar) );
}
```

Generated assembly:

```
increment: .type func
          add r0, r0, #1
          ldr    r1, .L2
          str    r0, [r1, #0]
          bx     lr
          .size  increment, $-increment
          .align 4
.L2:
          .dw    outvar
```

1.5 Pragmas to Control the Compiler

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

For example, you can set a compiler option to specify which optimizations the compiler should perform. With the `#pragma optimize flags` you can set an optimization level for a specific part of the C source. This overrules the general optimization level that is set in the C compiler Optimization page in the Project Options dialog (command line option `--optimize (-O)`).

The general syntax for pragmas is:

```
#pragma pragma-spec [ON | OFF | DEFAULT | RESTORE]
```

or:

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

Pragmas marked with (*) accept the following special arguments:

`default` set the pragma to the initial value

`restore` restore the previous value of the pragma

Pragmas marked with (+) are boolean flags, and accept the following arguments:

`on` switch the flag on (same as without argument)

`off` switch the flag off

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

alias symbol=defined_symbol

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



See [assembler directive .EQU](#) in section 4.8.2, *Assembler Directives*, in Chapter *Assembly Language*.

call {near|far} ()*

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.



See [C compiler option --call \(-m\)](#) in section 6.1, *C Compiler Options*, in Chapter *Tool Options*.

extension isuffix () (+)*

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

extern symbol

Force an external reference (`.extern` assembler directive), even when the symbol is not used in the module.



See [assembler directive .EXTERN](#) in section 4.8.2, *Assembler Directives*, in Chapter *Assembly Language*.

inline
noinline
smartinline

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

```
int w,x,y,z;

#pragma inline
int add( int a, int b )
{
    int i=4;
    return( a + b );
}
#pragma noinline
void main( void )
{
    w = add( 1, 2 );
    z = add( x, y );
}
```

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



See section 1.8.3, *Inlining Functions: inline / __noinline*.

By default, small functions that are not too often called (from different locations), are inlined. This reduces execution speed at the cost of code size (C compiler option `-Oi`). With the `pragma noinline` / `pragma smartinline` you can temporarily disable this optimization.

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.



See for more information the C compiler options `--inline-max-incr` and `--inline-max-size` in section 6.1, *C Compiler Options* in Chapter *Tool Options*.

macro
nomacro (*) (+)

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

message "message" ...

Print the message string(s) on standard output.

optimize flags (*)
endoptimize

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



See section 2.6, *Compiler Optimizations* in Chapter *Using the Compiler* in the user's manual.

See C compiler option `--optimize (-O)` in section 6.1, *C Compiler Options*, in Chapter *Tool Options*.

profile [flag,...] (*)
endprofile

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



See C compiler option `--profile (-p)` in section 6.1, *C Compiler Options*, in Chapter *Tool Options*.

profiling (*) (+)

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

protect (*) (+)**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 [*flag,...*] (*)

Check for run-time errors. The pragma works the same as compiler option `--runtime (-r)`.



See [C compiler option `--runtime \(-r\)`](#) in section 6.1, *C Compiler Options*, in Chapter *Tool Options*.

section [*name*={*suffix*|-*f*|-*m*|-*fm*}] (*)**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`](#) in section 6.1, *C Compiler Options* in Chapter *Tool Options*.

See [assembler directive `.SECTION`](#) (Start or continue section), in section 4.8.2, *Assembler Directives*, in Chapter *Assembly Language*.

section_code_init (*) (+)**section_no_code_init**

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

section_const_init (*) (+)**section_no_const_init**

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

source (*) (+)**nosource**

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



See also [C compiler option `--source \(-s\)`](#)

stdinc (*) (+)

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.

<code>jump_switch</code>	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.
<code>binary_switch</code>	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.
<code>smart_switch</code>	Let the compiler decide the switch method used
<code>tbb_switch</code>	Force use of the <code>tbb</code> instruction. Uses a table of 8-bit jump offsets.
<code>tbh_switch</code>	Force use of the <code>tbh</code> instruction. Uses a table of 16-bit jump offsets.
<code>no_tbh_switch</code>	Same as <code>smart_switch</code> , but do not use the <code>tbh</code> instruction.



See Section 1.7, *Switch Statement*.

tradeoff level (*)

Specify tradeoff between speed (0) and size (4).



See also C compiler option `--tradeoff (-t)`

warning [number,...] (*)

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



See also C compiler option `--no-warnings (-w)`

weak symbol

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

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

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



See assembler directive **.WEAK** in Section 4.8.2, *Assembler Directives*, in Chapter *Assembly Language*.

1.6 Predefined Preprocessor Macros

In addition to the predefined macros required by the ISO C standard, such as `__DATE__` and `__FILE__`, the TASKING C compiler supports the predefined macros as defined in the table below. The macros are useful to create conditional C code.

Macro	Description
<code>__ARM__</code>	Expands to 1 for the ARM toolset, otherwise unrecognized as macro.
<code>__BIG_ENDIAN__</code>	Expands to 1 if big-endian mode is selected (<code>--endianness=big</code>), otherwise unrecognized as macro.
<code>__BUILD__</code>	Identifies the build number of the compiler, composed of decimal digits for the build number, three digits for the major branch number and three digits for the minor branch number. For example, if you use build 1.22.1 of the compiler, <code>__BUILD__</code> expands to 1022001. If there is no branch number, the branch digits expand to zero. For example, build 127 results in 127000000.
<code>__CARM__</code>	Expands to 1 for the ARM toolset, otherwise unrecognized as macro.
<code>__CPU__</code>	Expands to the ARM architecture name (option <code>--cpu=arch</code>).
<code>__CPU_arch__</code>	A symbol is defined depending on the option <code>--cpu=arch</code> . The <i>arch</i> is converted to uppercase. For example, if <code>--cpu=ARMv4T</code> is specified the symbol <code>__CPU_ARMV4T__</code> is defined. When no <code>--cpu</code> is supplied, symbol <code>__CPU_ARMV4T__</code> is the default.
<code>__DOUBLE_FP__</code>	Expands to 1 if you did <i>not</i> use option <code>--no-double</code> (Treat 'double' as 'float'), otherwise unrecognized as macro.
<code>__LITTLE_ENDIAN__</code>	Expands to 1 if little-endian mode is selected (<code>--endianness=little</code>), otherwise unrecognized as macro. This is the default.
<code>__REVISION__</code>	Identifies the revision number of the compiler. For example, if you use version 1.0r2 of the compiler, <code>__REVISION__</code> expands to 2.
<code>__SINGLE_FP__</code>	Expands to 1 if you used option <code>--no-double</code> (Treat 'double' as 'float'), otherwise unrecognized as macro.
<code>__TASKING__</code>	Identifies the compiler as a TASKING compiler. Expands to 1 if a TASKING compiler is used.
<code>__THUMB__</code>	Expands to 1 if you used option <code>--thumb</code> , otherwise unrecognized as macro.
<code>__VERSION__</code>	Identifies the version number of the compiler. For example, if you use version 1.0r2 of the compiler, <code>__VERSION__</code> expands to 1000 (dot and revision number are omitted, minor version number in 3 digits).

Table 1-3: Predefined preprocessor macros

Example

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

#endif
```

1.7 Switch Statement

The TASKING C compiler supports three ways of code generation for a switch statement: a jump chain (linear switch), a jump table or a 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 tries to use the switch method which uses the least space in ROM (table size in ROMDATA plus code to do the indexing).

For a switch with a long type argument, only binary search table code is used.

For an int type argument, a jump table switch is only possible when all case values are in the same 256 value range (the high byte value of all programmed cases are the same).

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.

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 `tbh` instruction gets priority over a normal switch table implementation.

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

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

Using a pragma cannot overrule the restrictions as described earlier.

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

Example

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

#pragma jump_switch

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

1.8 Functions

1.8.1 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. See the table below.

Parameter Type	Parameter Number			
	1	2	3	4
_Bool	r0	r1	r2	r3
char	r0	r1	r2	r3
short	r0	r1	r2	r3
int / long	r0	r1	r2	r3
float	r0	r1	r2	r3
32-bit pointer	r0	r1	r2	r3
32-bit struct	r0	r1	r2	r3
long long	r0r1	r1r2	r2r3	r3
double	r0r1	r1r2	r2r3	
64-bit struct	r0r1	r1r2	r2r3	

Table 1-4: Register usage for parameter passing



If a register corresponding to a parameter number is already in use the next register is used.

Example with three arguments

```
func1( int a, int b, int *c )
```

- a (first parameter) is passed in register r0.
- b (second parameter) is passed in register r1.
- c (third parameter) is passed in register r2.

Example with one long long/double arguments and one other argument

```
func2( long long d, char e )
```

- d (first parameter) is passed in register r0 and r1
- e (second parameter) is passed in register r2.

Example with two long long/double arguments and one other argument

```
func3( double f, long long g, char h )
```

- f (first parameter) is passed in register r0 and r1
- g (second parameter) is passed in register r2 and r3.
- h (third parameter) cannot be passed through registers anymore, and is passed via the stack.

1.8.2 Function Return Types

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

Return Type	Register
_Bool	r0
char	r0
short	r0
int / long	r0

Return Type	Register
float	r0
32-bit pointer	r0
32-bit struct	r0
long long	r0r1
double	r0r1
64-bit struct	r0r1

Table 1-5: Register usage for function return types

Objects larger than 64 bits are returned via the stack.

1.8.3 Inlining Functions: `inline` / `__noinline`

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:

```
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:

```
__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:

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

1.8.4 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.

The TASKING ARM C compiler recognizes the following intrinsic functions:

`__alloc`

```
void * volatile __alloc( __size_t size );
```

Allocate memory.

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

`__free`

```
void volatile __free( void *p );
```

Deallocates the memory pointed to by *p*. *p* must point to memory earlier allocated by a call to `__alloc()`.

Returns: nothing.

`__nop`

```
void __nop( void );
```

Generate NOP instructions.

Returns: nothing.

Example:

```
__nop(); /* generate NOP instruction */
```

`__get_return_address`

```
__codeptr volatile __get_return_address( void );
```

Used by the compiler for profiling when you compile with the `--profile (-p)` option.

Returns: return address of a function.

`__getpsr`

```
unsigned int volatile __getpsr( void );
```

Get the value of the SPSR status register.

Returns: the value of the status register SPSR

`__setpsr`

```
unsigned int volatile __setpsr( int set, int clear);
```

Set or clear bits in the SPSR status register.

Returns: the new value of the SPSR status register.

Example:

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

i = __setpsr (0, SR_F | SR_I);

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

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

__getcpsr

```
unsigned int volatile __getcpsr( void );
```

Get the value of the CPSR status register.

Returns: the value of the status register CPSR

__setcpsr

```
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.

__getapsr

```
unsigned int volatile __getapsr( void );
```

Get the value of the APSR status register.

Returns: the value of the status register APSR



This intrinsic is only available for the ARMv6-M and ARMv7-M (M-profile architectures).

__setapsr

```
unsigned int volatile __setapsr( int set, int clear);
```

Set or clear bits in the APSR status register.

Returns: the new value of the APSR status register.



This intrinsic is only available for the ARMv6-M and ARMv7-M (M-profile architectures).

__getipsr

```
unsigned int volatile __getipsr( void );
```

Get the value of the IPSR status register.

Returns: the value of the status register IPSR



This intrinsic is only available for the ARMv6-M and ARMv7-M (M-profile architectures).

__svc

```
void volatile __svc(int number);
```

Generates a supervisor call (software interrupt). *Number* must be a constant value.

Returns: nothing.

1.8.5 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 ARM supports seven types of exceptions. The next table lists the types of exceptions and the processor mode that is used to process that exception. 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.

Exception type	Mode	Normal address	High vector address	Function type qualifier
Reset	Supervisor	0x00000000	0xFFFF0000	
Undefined instructions	Undefined	0x00000004	0xFFFF0004	__interrupt_und
Supervisor call (software interrupt)	Supervisor	0x00000008	0xFFFF0008	__interrupt_svc
Prefetch abort	Abort	0x0000000C	0xFFFF000C	__interrupt_iabt
Data abort	Abort	0x00000010	0xFFFF0010	__interrupt_dabt
IRQ (interrupt)	IRQ	0x00000018	0xFFFF0018	__interrupt_irq
FIQ (fast interrupt)	FIQ	0x0000001C	0xFFFF001C	__interrupt_fiq

Table 1-6: Exception processing modes



ARMv6-M and ARMv7-M (M-profile architectures) have a different exception model. Read the *ARM Architecture Reference Manual* for details.

1.8.5.1 Defining an Exception Handler: __interrupt keywords

You can define six types of exception handlers with the function type qualifiers `__interrupt_und`, `__interrupt_svc`, `__interrupt_iabt`, `__interrupt_dabt`, `__interrupt_irq` and `__interrupt_fiq`. You can also use the general `__interrupt()` function qualifier.

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

```
void __interrupt_xxx
isr( void )
{
    ...
}

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

Example

```
void __interrupt_irq serial_receive( void )
{
    ...
}
```

Vector symbols

When you use one or more of these `__interrupt_xxx` 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.

Function type qualifier	Vector symbol	Vector symbol M-profile
<code>__interrupt_und</code>	<code>_vector_1</code>	-
<code>__interrupt_svc</code>	<code>_vector_2</code>	<code>_vector_11</code>
<code>__interrupt_iabt</code>	<code>_vector_3</code>	-
<code>__interrupt_dabt</code>	<code>_vector_4</code>	-
<code>__interrupt_irq</code>	<code>_vector_6</code>	-
<code>__interrupt_fiq</code>	<code>_vector_7</code>	-
<code>__interrupt(n)</code>	<code>_vector_n</code>	<code>_vector_n</code>

Note that the reset handler is designated by the symbol `_START` instead of `_vector_0` (`_vector_1` for M-profile architectures).

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, for example for different IRQ's or for different run-time phases of your application. Without the `__novector` function qualifier the compiler generates the `_vector_n` symbol multiple times, which results in a link error.

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

Enable interrupts in exception handlers (not for M-profile architectures)

Normally interrupts are disabled when an exception handler is entered. With the function qualifier `__nesting_enabled` you can force that the link register (LR) is saved and that interrupts are enabled. For example:

```
void __interrupt_svc __nesting_enabled svc( int n )
{
    if ( n == 2 )
    {
        __svc(3);
    }
    ...
}
```

1.8.5.2 Interrupt Frame: `__frame()`

With the function type qualifier `__frame()` 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:

```
void __interrupt_xxx
    __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_irq __frame(R4,R5,R6) void alarm( void )
{
    ...
}
```

1.9 Libraries

The TASKING compilers come with standard C libraries (ISO/IEC 9899:1999) and header files with the appropriate prototypes for the library functions. All standard C libraries are available in object format and in C or assembly source code.

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.



See section 3.2, *Library Functions*, in Chapter *Libraries*, for an extensive description of all standard C library functions.

1.9.1 Printf and Scanf Routines

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 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 indirect, the arguments are not known and always the `long long` version with floating-point support is used from the library.

Example

```
#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 1.5, *Pragmas to Control the Compiler* in the user's manual.



2 C++ Language

Summary

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

2.1 Introduction

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

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

- The C++ Programming Language (second edition) by Bjarne Strastrup (1991, Addison Wesley)
- ISO/IEC 14882:1998 C++ standard [ANSI] More information on the standards can be found at <http://www.ansi.org/>.

2.2 C++ Language Extension Keywords

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

Additionally the following language extensions are supported:

attributes

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

pragmas

The C++ compiler supports the same pragmas as the C compiler and some extra pragmas as explained in section 2.9, *Pragmas to Control the C++ Compiler*. Pragmas give directions to the code generator of the compiler.

2.3 C++ Dialect Accepted

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

Command line options are also available to enable and disable anachronisms and strict standard-conformance checking.

2.3.1 Standard Language Features Accepted

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

- The dependent statement of an `if`, `while`, `do-while`, or `for` is considered to be a scope, and the restriction on having such a dependent statement be a declaration is removed.
- The expression tested in an `if`, `while`, `do-while`, or `for`, as the first operand of a `"?"` operator, or as an operand of the `"&&"`, `"!:"`, or `"!"` operators may have a pointer-to-member type or a class type that can be converted to a pointer-to-member type in addition to the scalar cases permitted by the ARM.

- Qualified names are allowed in elaborated type specifiers.
- A global-scope qualifier is allowed in member references of the form `x::A::B` and `p->::A::B`.
- The precedence of the third operand of the `"?"` operator is changed.
- If control reaches the end of the `main()` routine, and `main()` has an integral return type, it is treated as if a `return 0;` statement were executed.
- Pointers to arrays with unknown bounds as parameter types are diagnosed as errors.
- A functional-notation cast of the form `A()` can be used even if `A` is a class without a (nontrivial) constructor. The temporary created gets the same default initialization to zero as a static object of the class type.
- A cast can be used to select one out of a set of overloaded functions when taking the address of a function.
- Template friend declarations and definitions are permitted in class definitions and class template definitions.
- Type template parameters are permitted to have default arguments.
- Function templates may have nontype template parameters.
- A reference to `const volatile` cannot be bound to an rvalue.
- Qualification conversions, such as conversion from `T**` to `T const * const *` are allowed.
- Digraphs are recognized.
- Operator keywords (e.g., `not`, `and`, `bitand`, etc.) are recognized.
- Static data member declarations can be used to declare member constants.
- When option `--wchar_t-keyword` is set, `wchar_t` is recognized as a keyword and a distinct type.
- `bool` is recognized.
- RTTI (run-time type identification), including `dynamic_cast` and the `typeid` operator, is implemented.
- Declarations in tested conditions (in `if`, `switch`, `for`, and `while` statements) are supported.
- Array `new` and `delete` are implemented.
- New-style casts (`static_cast`, `reinterpret_cast`, and `const_cast`) are implemented.
- Definition of a nested class outside its enclosing class is allowed.
- `mutable` is accepted on non-static data member declarations.
- Namespaces are implemented, including `using` declarations and directives. Access declarations are broadened to match the corresponding `using` declarations.
- Explicit instantiation of templates is implemented.
- The `typename` keyword is recognized.
- `explicit` is accepted to declare non-converting constructors.
- The scope of a variable declared in the `for-init-statement` of a `for` loop is the scope of the loop (not the surrounding scope).
- Member templates are implemented.
- The new specialization syntax (using `"template <>"`) is implemented.
- Cv-qualifiers are retained on rvalues (in particular, on function return values).
- The distinction between trivial and nontrivial constructors has been implemented, as has the distinction between PODs and non-PODs with trivial constructors.
- The linkage specification is treated as part of the function type (affecting function overloading and implicit conversions).
- `extern inline` functions are supported, and the default linkage for `inline` functions is external.
- A typedef name may be used in an explicit destructor call.
- Placement delete is implemented.
- An array allocated via a placement `new` can be deallocated via `delete`.
- Covariant return types on overriding virtual functions are supported.
- `enum` types are considered to be non-integral types.
- Partial specialization of class templates is implemented.
- Partial ordering of function templates is implemented.
- Function declarations that match a function template are regarded as independent functions, not as "guiding declarations" that are instances of the template.
- It is possible to overload operators using functions that take `enum` types and no `class` types.
- Explicit specification of function template arguments is supported.

- Unnamed template parameters are supported.
- The new lookup rules for member references of the form `x.A::B` and `p->A::B` are supported.
- The notation `:: template` (and `->template`, etc.) is supported.
- In a reference of the form `f()->g()`, with `g` a static member function, `f()` is evaluated. The ARM specifies that the left operand is not evaluated in such cases.
- `enum` types can contain values larger than can be contained in an `int`.
- Default arguments of function templates and member functions of class templates are instantiated only when the default argument is used in a call.
- String literals and wide string literals have `const` type.
- Class name injection is implemented.
- Argument-dependent (Koenig) lookup of function names is implemented.
- Class and function names declared only in unqualified friend declarations are not visible except for functions found by argument-dependent lookup.
- A `void` expression can be specified on a return statement in a `void` function.
- Function-try-blocks, i.e., try-blocks that are the top-level statements of functions, constructors, or destructors, are implemented.
- Universal character set escapes (e.g., `\uabcd`) are implemented.
- On a call in which the expression to the left of the opening parenthesis has class type, overload resolution looks for conversion functions that can convert the class object to pointer-to-function types, and each such pointed-to "surrogate function" type is evaluated alongside any other candidate functions.
- Dependent name lookup in templates is implemented. Nondependent names are looked up only in the context of the template definition. Dependent names are also looked up in the instantiation context, via argument-dependent lookup.
- Value-initialization is implemented. This form of initialization is indicated by an initializer of `()` and causes zeroing of certain POD-typed members, where the usual default-initialization would leave them uninitialized.
- A partial specialization of a class member template cannot be added outside of the class definition.
- Qualification conversions may be performed as part of the template argument deduction process.
- The `export` keyword for templates is implemented.

2.3.2 C++0x Language Features Accepted

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

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


```
template<typename T> struct S {};
S<S<int>>> s; // OK. No whitespace needed
               // between closing angle brackets.
```
- The friend class syntax is extended to allow nonclass types as well as class types expressed through a typedef or without an elaborated type name. For example:


```
typedef struct S ST;
class C {
    friend S;           // OK (requires S to be in scope).
    friend ST;          // OK (same as "friend S;").
    friend int;         // OK (no effect).
    friend S const;     // Error: cv-qualifiers cannot
                       // appear directly.
};
```
- Mixed string literal concatenations are accepted (a feature carried over from C99):


```
wchar_t *str = "a" L"b"; // OK, same as L"ab".
```
- Variadic macros and empty macro arguments are accepted, as in C99.
- A trailing comma in the definition of an enumeration type is silently accepted (a feature carried over from C99):


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

- If the command line option `--long-long` is specified, the type `long long` is accepted. Unsuffixed integer literals that cannot be represented by type `long`, but could potentially be represented by type `unsigned long`, have type `long long` instead (this matches C99, but not the treatment of the `long long` extension in C89 or default C++ mode).
- The keyword `typename` followed by a qualified-id can appear outside a template declaration.

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

2.3.3 Anachronisms Accepted

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

- `overload` is allowed in function declarations. It is accepted and ignored.
- Definitions are not required for static data members that can be initialized using default initialization. The anachronism does not apply to static data members of template classes; they must always be defined.
- The number of elements in an array may be specified in an array `delete` operation. The value is ignored.
- A single `operator++()` and `operator--()` function can be used to overload both prefix and postfix operations.
- The base class name may be omitted in a base class initializer if there is only one immediate base class.
- Assignment to `this` in constructors and destructors is allowed. This is allowed only if anachronisms are enabled and the "assignment to `this`" configuration parameter is enabled.
- A bound function pointer (a pointer to a member function for a given object) can be cast to a pointer to a function.
- A nested class name may be used as a non-nested class name provided no other class of that name has been declared. The anachronism is not applied to template classes.
- A reference to a non-const type may be initialized from a value of a different type. A temporary is created, it is initialized from the (converted) initial value, and the reference is set to the temporary.
- A reference to a non-const class type may be initialized from an rvalue of the class type or a derived class thereof. No (additional) temporary is used.
- A function with old-style parameter declarations is allowed and may participate in function overloading as though it were prototyped. Default argument promotion is not applied to parameter types of such functions when the check for compatibility is done, so that the following declares the overloading of two functions named `f`:

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

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

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

```
struct A {  
    A(int);  
    A operator=(A&);  
    A operator+(const A&);  
};  
main () {  
    A b(1);  
    b = A(1) + A(2); // Allowed as anachronism  
}
```

2.3.4 Extensions Accepted in Normal C++ Mode

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

- A `friend` declaration for a class may omit the `class` keyword:

```
class A {  
    friend B; // Should be "friend class B"  
};
```

- Constants of scalar type may be defined within classes:

```
class A {
    const int size = 10;
    int a[size];
};
```

- In the declaration of a class member, a qualified name may be used:

```
struct A {
    int A::f(); // Should be int f();
};
```

- The `restrict` keyword is allowed.
- Implicit type conversion between a pointer to an extern "C" function and a pointer to an extern "C++" function is permitted. Here's an example:

```
extern "C" void f(); // f's type has extern "C" linkage
void (*pf)() // pf points to an extern "C++" function
    = &f;    // error unless implicit conversion is
              // allowed
```

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

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

```
char *p = x ? "abc" : "def";
```

- Default arguments may be specified for function parameters other than those of a top-level function declaration (e.g., they are accepted on `typedef` declarations and on pointer-to-function and pointer-to-member-function declarations).
- Non-static local variables of an enclosing function can be referenced in a non-evaluated expression (e.g., a `sizeof` expression) inside a local class. A warning is issued.
- In default C++ mode, the friend class syntax is extended to allow nonclass types as well as class types expressed through a `typedef` or without an elaborated type name. For example:

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

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

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

- In default C++ mode, variadic macros are accepted. (This is a feature carried over from C99 and also available in GNU modes.)
- In default C++ mode, empty macro arguments are accepted (a feature carried over from C99).
- A trailing comma in the definition of an enumeration type is silently accepted (a feature carried over from C99):

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

2.4 GNU Extensions

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

- Extended designators are accepted
- Compound literals are accepted.

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

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

This can be useful in macro and template definitions.

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

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

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

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

- If the command line option `--long-long` is specified, the extensions for the `long long` and `unsigned long long` types are enabled.
- Zero-length array types (specified by `[0]`) are supported. These are complete types of size zero.
- C99-style flexible array members are accepted. In addition, the last field of a class type have a class type whose last field is a flexible array member. In GNU C++ mode, flexible array members are treated exactly like zero-length arrays, and can therefore appear anywhere in the class type.
- The C99 `_Pragma` operator is supported.
- The gcc built-in `<stdarg.h>` and `<varargs.h>` facilities (`__builtin_va_list`, `__builtin_va_arg`, ...) are accepted.
- The `sizeof` operator is applicable to `void` and to function types and evaluates to the value one.
- Variables can be redeclared with different top-level cv-qualifiers (the new qualification is merged into existing qualifiers). For example:

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

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

```
int counter __asm__("counter_v1") = 0;
```

- Register variables can be mapped on specific registers using the `asm` keyword.

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

- The keyword `inline` is ignored (with a warning) on variable declarations and on block-extern function declarations.
- Excess aggregate initializers are ignored with a warning.

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

- Expressions of types `void*`, `void const*`, `void volatile*` and `void const volatile*` can be dereferenced; the result is an lvalue.
- The `__restrict__` keyword is accepted. It is identical to the C99 `restrict` keyword, except for its spelling.
- Out-of-range floating-point values are accepted without a diagnostic. When IEEE floating-point is being used, the "infinity" value is used.
- Extended variadic macros are supported.
- Dollar signs (\$) are allowed in identifiers.
- Hexadecimal floating point constants are recognized.

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

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

- The `\e` escape sequence is recognized and stands for the ASCII "ESC" character.
- The address of a statement label can be taken by use of the prefix `&&` operator, e.g., `void *a = &&L`. A transfer to the address of a label can be done by the `"goto *"` statement, e.g., `goto *a`.
- Multi-line strings are supported, e.g.,

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

- ASCII "NULL" characters are accepted in source files.
- A source file can end with a backslash (`"\"`) character.
- Case ranges (e.g., `"case 'a' ... 'z:'"`) are supported.
- A number of macros are predefined in GNU mode. See section 2.10, [Predefined Macros](#).
- A predefined macro can be undefined.
- A large number of special functions of the form `__builtin_xyz` (e.g., `__builtin_alloca`) are predeclared.
- Some expressions are considered to be constant-expressions even though they are not so considered in standard C and C++. Examples include `"((char *)&((struct S *)0)->c[0]) - (char *)0"` and `"(int)"Hello" & 0"`.
- The macro `__GNUC__` is predefined to the major version number of the emulated GNU compiler. Similarly, the macros `__GNUC_MINOR__` and `__GNUC_PATCHLEVEL__` are predefined to the corresponding minor version number and patch level. Finally, `__VERSION__` is predefined to a string describing the compiler version.
- The `__thread` specifier can be used to indicate that a variable should be placed in thread-local storage (requires `gnu_version >= 30400`).
- An extern inline function that is referenced but not defined is permitted (with a warning).
- Trigraphs are ignored (with a warning).
- Non-standard casts are allowed in null pointer constants, e.g., `(int)(int *)0` is considered a null pointer constant in spite of the pointer cast in the middle.
- Statement expressions, e.g., `{int j; j = f(); j;}` are accepted. Branches into a statement expression are not allowed. In C++ mode, branches out are also not allowed. Variable-length arrays, destructible entities, try, catch, local non-POD class definitions, and dynamically-initialized local static variables are not allowed inside a statement expression.
- Labels can be declared to be local in statement expressions by introducing them with a `__label__` declaration.

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

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

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

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

```
int i;
(short)i = 0; // Accepted, cast is ignored; entire int is set
```

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

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

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

- If an explicit instantiation directive is preceded by the keyword `extern`, no (explicit or implicit) instantiation is for the indicated specialization.
- An explicit instantiation directive that names a class may omit the `class` keyword, and may refer to a typedef.
- An explicit instantiation or extern template directive that names a class is accepted in an invalid namespace.
- `std::type_info` does not need to be introduced with a special pragma.
- A special keyword `__null` expands to the same constant as the literal "0", but is expected to be used as a null pointer constant.
- When `gnu_version < 30400`, names from dependent base classes are ignored only if another name would be found by the lookup.

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

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

```
template <class T> struct C {
    struct A { int i; };
    struct B: public A {
        void f() {
            i = 0; // g++ uses A::i not C::i
        }
    };
    int i;
};
```

- A new operation in a template is always treated as dependent (when `gnu_version >= 30400`).

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

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

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

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

```
struct A {
    A(int) {};
```

```
};
A::A a(1);
```

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

```
template <int u1> struct A { };
template <> struct A<1> {
    template<class T> A(T i, int j);
};

template <> A<1>::A<1>(int i, int j) { }
    // accepted in g++ mode
```

- A difference in calling convention is ignored when redeclaring a typedef.

```
typedef void F();

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

- The macro `__GNUG__` is defined identically to `__GNUC__` (i.e., the major version number of the GNU compiler version that is being emulated).
- The macro `_GNU_SOURCE` is defined as "1".
- Guiding declarations (a feature present in early drafts of the standard, but not in the final standard) are disabled.
- Namespace `std` is predeclared.
- No connection is made between declarations of identical names in different scopes even when these names are declared `extern "C"`. E.g.,

```
extern "C" { void f(int); }
namespace N {
    extern "C" {
        void f() {} // Warning (not error) in g++ mode
    }
}
int main() { f(1); }
```

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

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

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

- The definition of a member of a class template that appears outside of the class definition may declare a nontype template parameter with a type that is different than the type used in the definition of the class template. A warning is issued (GNU version 30300 and below).

```
template <int I> struct A { void f(); };
template <unsigned int I> void A<I>::f(){}
```

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

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

- A friend declaration may refer to a member typedef.

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

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

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

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

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

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

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

- In a catch clause, an entity may be declared with the same name as the handler parameter.

```
try { }  
catch(int e) {  
    char e;  
}
```

- The diagnostic issued for an exception specification mismatch is reduced to a warning if the previous declaration was found in a system header.
- The exception specification for an explicit template specialization (for a function or member function) does not have to match the exception specification of the corresponding primary template.
- A template argument list may appear following a constructor name in constructor definition that appears outside of the class definition:

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

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

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


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

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

- Exception specifications are ignored on function definitions when support for exception handling is disabled (normally, they are only ignored on function declarations that aren't definitions).
- A friend declaration in a class template may refer to an undeclared template.

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

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

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

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

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

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

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

- Default arguments are retained as part of deduced function types.
- A namespace member may be redeclared outside of its namespace.
- A template may be redeclared outside of its class or namespace.

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

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

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

- A partial specialization may be declared after an instantiation has been done that would have used the partial specialization if it had been declared earlier. A warning is issued.

```
template <class T> class X {};
X<int*> xi;
template <class T> class X<T*> {};
```

- The "." or "->" operator may be used in an integral constant expression if the result is an integral or enumeration constant:

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

- Strong using-directives are supported.

```
using namespace debug __attribute__((strong));
```

- Partial specializations that are unusable because of nondeductible template parameters are accepted and ignored.

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

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

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

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

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

- When doing nonclass prototype instantiations (e.g., `gnu_version` 30400 and above), the severity of the diagnostic issued if a const template static data member is defined without an initializer is reduced to a warning.

```
template <class T> struct A {
    static const int i;
};
template <class T> const int A<T>::i;
```

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

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

The following GNU extensions are *not* currently supported:

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

2.5 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:

```
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:

```
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);
}
```

2.6 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, member functions of template classes, and static data members of template classes (hereafter referred to as template entities) are not necessarily done immediately, for several reasons:

- One would like to end up with only one copy of each instantiated entity across all the object files that make up a program. (This of course applies to entities with external linkage.)
- The language allows one to write a *specialization* of a template entity, i.e., a specific version to be used in place of a version generated from the template for a specific data type. (One could, for example, write a version of `Stack<int>`, or of just `Stack<int>::push`, that replaces the template-generated version; often, such a specialization provides a more efficient representation for a particular data type.) Since the compiler cannot know, when compiling a reference to a template entity, if a specialization for that entity will be provided in another compilation, it cannot do the instantiation automatically in any source file that references it.
- C++ templates can be *exported* (i.e., declared with the keyword `export`). Such templates can be used in a translation unit that does not contain the definition of the template to instantiate. The instantiation of such a template must be delayed until the template definition has been found.
- The language also dictates that template functions that are not referenced should not be compiled, that, in fact, such functions might contain semantic errors that would prevent them from being compiled. Therefore, a reference to a template class should not automatically instantiate all the member functions of that class.

(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.

2.6.1 Automatic Instantiation

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

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

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*.

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

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

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

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

If you are using Borland's compiler you must make sure that every compilation sees all the source code it needs to instantiate all the template entities referenced in that compilation. That is, one cannot refer to a template entity in a source file if a definition for that entity is not included by that source file. In practice, this means that either all the definition code is put directly in the `.h` files, or that each `.h` file includes an associated `.cc` (actually, `.cpp`) file.

Our approach is a little different. It requires that, for each instantiation of a non-exported template, there is some (normal, top-level, explicitly-compiled) source file that contains the definition of the template entity, a reference that causes the instantiation, and the declarations of any types required for the instantiation.³ This requirement can be met in various ways:

- The Borland convention: each `.h` file that declares a template entity also contains either the definition of the entity or includes another file containing the definition.
- Implicit inclusion: when the compiler sees a template declaration in a `.h` file and discovers a need to instantiate that entity, it is given permission to go off looking for an associated definition file having the same base name and a different suffix, and it implicitly includes that file at the end of the compilation. This method allows most programs written using the *cfront* convention to be compiled with our approach. See section 2.6.4, [Implicit Inclusion](#).
- The ad hoc approach: you make sure that the files that define template entities also have the definitions of all the available types, and add code or pragmas in those files to request instantiation of the entities there.

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

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

2.6.2 Instantiation Modes

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

`--instantiate=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.

² The actual implementation allows for several different suffixes and provides a command line option to change the suffixes sought.

³ Isn't this always the case? No. Suppose that file A contains a definition of class X and a reference to `Stack<X>::push`, and that file B contains the definition for the member function `push`. There would be no file containing both the definition of `push` and the definition of X.

--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 control program is called with 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.

2.6.3 Instantiation #pragma Directives

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

- The **instantiate** pragma causes a specified entity to be instantiated.
- The **do_not_instantiate** pragma suppresses the instantiation of a specified entity. It is typically used to suppress the instantiation of an entity for which a specific definition will be supplied.
- The **can_instantiate** pragma indicates that a specified entity can be instantiated in the current compilation, but need not be; it is used in conjunction with automatic instantiation, to indicate potential sites for instantiation if the template entity turns out to be required.

The argument to the instantiation pragma may be:

- a template class name `A<int>`
- a template class declaration `class A<int>`
- a member function name `A<int>::f`
- a static data member name `A<int>::i`
- a static data declaration `int A<int>::i`
- a member function declaration `void A<int>::f(int, char)`
- a template function declaration `char* f(int, float)`

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

```
#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.

```
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

```

#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.

2.6.4 Implicit Inclusion

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

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

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

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

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

The implicit inclusion mode can be turned on by the command line option `--implicit-include`. If this option is turned on, you cannot use exported templates.

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.

2.6.5 Exported Templates

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

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

export template<class T> T const& min(T const&, T const&);
int main()
{
    trace();
    return min(2, 3);
}

// File 2:
#include <stdio.h>
static void trace() { printf("File 2\n"); }

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

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

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

You cannot use exported templates together with the command line option **`--implicit-include`**.

2.6.5.1 Finding the Exported Template Definition

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

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

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

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

2.6.5.2 Secondary Translation Units

An instantiation of an exported template can be triggered by an explicit instantiation directive, or by the command line option `--instantiate=used`. In each case, the translation unit that contains the initial point of instantiation will be processed as the *primary translation unit*. Based on information it finds in the `.et` files, the C++ compiler will then load and parse the translation unit containing the definition of the template to instantiate. This is a *secondary translation unit*. The simultaneous processing of the primary and secondary translation units enables the C++ compiler to create instantiations of the exported templates (which can include entities from both translation units). This process may reveal the need for additional instantiations of exported templates, which in turn can cause additional secondary translation units to be loaded⁴.

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

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

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

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

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

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

2.6.5.3 Libraries with Exported Templates

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

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

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

1. For each `.et` file in the original source directory, copy the associated source file to the `export` directory.
2. Concatenate all of the `.et` files into a single `.et` file (e.g., `mylib.et`) in the `export` directory. The individual `.et` files could be copied to the `export` directory, but having all of the `.et` information in one file will make use of the library more efficient.
3. Create an `export_info` file in the `export` directory. The `export_info` file specifies the include search paths to be used when recompiling files in the `export` directory. If no `export_info` file is provided, the include search path used when compiling the client program that uses the library will also be used to recompile the library exported template files.

⁴ As a consequence, using exported templates may require considerably more memory than similar uses of regular (included) templates.

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

```
include=x
```

or

```
sys_include=x
```

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

```
# The include directories to be used for the xyz library

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

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

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

2.7 Inlining Functions

The C++ compiler supports a minimal form of function inlining. When the C++ compiler encounters a call of a function declared `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.

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.

2.8 Extern Inline Functions

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

When static functions are used, local static variables of the functions are promoted to global variables with specially encoded names, so that even though there may be multiple copies of the code, there is only one copy of such global variables. This mechanism does not strictly conform to the standard because the address of an extern inline function is not constant across translation units.

When the instantiation mechanism is used, the address of an extern inline function is constant across translation units, but at the cost of requiring the use of one of the template instantiation mechanisms, even for programs that don't use templates. Definitions of extern inline functions can be provided either through use of the automatic instantiation mechanism or by use of the `--instantiate=used` or `--instantiate=all` instantiation modes. There is no mechanism to manually control the definition of extern inline function bodies.

2.9 Pragas to Control the C++ Compiler

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

The syntax is:

```
#pragma pragma-spec
```

The C++ compiler supports the following pragmas and all C compiler pragmas that are described in section 1.5, [Pragas to Control the Compiler](#).

instantiate / do_not_instantiate / can_instantiate

These are template instantiation pragmas. They are described in detail in section 2.6.3, [Instantiation #pragma Directives](#).

hdrstop / no_pch

These are precompiled header pragmas. They are described in detail in section 2.11, [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`:

```
#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.

ident

This pragma is given in the form:

```
#pragma ident "string"
```

or

```
#ident "string"
```

2.10 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.

Macro	Description
<code>__ABI_COMPATIBILITY_VERSION</code>	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.
<code>__ABI_CHANGES_FOR_RTTI</code>	This macro is set to TRUE, meaning that the ABI changes for RTTI are implemented. This macro is used when building the C++ library.
<code>__ABI_CHANGES_FOR_ARRAY_NEW_AND_DELETE</code>	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.
<code>__ABI_CHANGES_FOR_PLACEMENT_DELETE</code>	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.
<code>__ARRAY_OPERATORS</code>	Defined when array new and delete are enabled. This is the default.
<code>__BASE_FILE__</code>	Similar to <code>__FILE__</code> but indicates the primary source file rather than the current one (i.e., when the current file is an included file).
<code>__BIG_ENDIAN__</code>	Expands to 1 if big-endian mode is selected (<code>--endianness=big</code>), otherwise unrecognized as macro.
<code>__BOOL</code>	Defined when <code>bool</code> is a keyword. This is the default.
<code>__BUILD__</code>	Identifies the build number of the C++ compiler, composed of decimal digits for the build number, three digits for the major branch number and three digits for the minor branch number. For example, if you use build 1.22.1 of the compiler, <code>__BUILD__</code> expands to 1022001. If there is no branch number, the branch digits expand to zero. For example, build 127 results in 127000000.
<code>__CHAR_MIN</code> / <code>__CHAR_MAX</code>	Used in <code>limits.h</code> to define the minimum/maximum value of a plain <code>char</code> respectively.
<code>__CPARM__</code>	Identifies the C++ compiler. You can use this symbol to flag parts of the source which must be recognized by the C++ compiler only. It expands to 1.
<code>__cplusplus</code>	Always defined.
<code>__CPU__</code>	Expands to a string with the CPU supplied with the option <code>--cpu</code> . When no <code>--cpu</code> is supplied, this symbol is not defined.
<code>__DATE__</code>	Defined to the date of the compilation in the form "Mmm dd yyyy".
<code>__DELTA_TYPE</code>	Defines the type of the offset field in the virtual function table. This macro is used when building the C++ library.
<code>__DOUBLE_FP__</code>	Expands to 1 if you did <i>not</i> use option <code>--no-double</code> (Treat 'double' as 'float'), otherwise unrecognized as macro.
<code>__embedded_cplusplus</code>	Defined as 1 in Embedded C++ mode.
<code>__EXCEPTIONS</code>	Defined when exception handling is enabled (<code>--exceptions</code>).
<code>__FILE__</code>	Expands to the current source file name.
<code>__func__</code>	Same as <code>__FUNCTION__</code> in GNU mode.
<code>__IMPLICIT_USING_STD</code>	Defined when the standard header files should implicitly do a using-directive on the <code>std</code> namespace (<code>--using-std</code>).
<code>__JMP_BUF_ELEMENT_TYPE</code>	Specifies the type of an element of the <code>setjmp</code> buffer. This macro is used when building the C++ library.
<code>__JMP_BUF_NUM_ELEMENTS</code>	Defines the number of elements in the <code>setjmp</code> buffer. This macro is used when building the C++ library.
<code>__LINE__</code>	Expands to the line number of the line where this macro is called.

Macro	Description
<code>__LITTLE_ENDIAN__</code>	Expands to 1 if little-endian mode is selected (<code>--endianness=little</code>), otherwise unrecognized as macro. This is the default.
<code>__NAMESPACES</code>	Defined when namespaces are supported (this is the default, you can disable support for namespaces with <code>--no-namespaces</code>).
<code>__NO_LONG_LONG</code>	Defined when the <code>long long</code> type is not supported. This is the default.
<code>__NULL_EH_REGION_NUMBER</code>	Defines the value used as the null region number value in the exception handling tables. This macro is used when building the C++ library.
<code>__PLACEMENT_DELETE</code>	Defined when placement delete is enabled.
<code>__PRETTY_FUNCTION__</code>	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.
<code>__PTRDIFF_MIN / __PTRDIFF_MAX</code>	Used in <code>stdint.h</code> to define the minimum/maximum value of a <code>ptrdiff_t</code> type respectively.
<code>__REGION_NUMBER_TYPE</code>	Defines the type of a region number field in the exception handling tables. This macro is used when building the C++ library.
<code>__REVISION__</code>	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: <code>v1.0r1</code> -> 1, <code>v1.0rb</code> -> -1
<code>__RTTI</code>	Defined when RTTI is enabled (<code>--rtti</code>).
<code>__RUNTIME_USES_NAMESPACES</code>	Defined when the run-time uses namespaces.
<code>__SIGNED_CHARS__</code>	Defined when plain <code>char</code> is signed.
<code>__SINGLE_FP__</code>	Expands to 1 if you used option <code>--no-double</code> (Treat 'double' as 'float'), otherwise unrecognized as macro.
<code>__SIZE_MIN / __SIZE_MAX</code>	Used in <code>stdint.h</code> to define the minimum/maximum value of a <code>size_t</code> type respectively.
<code>__STDC__</code>	Always defined, but the value may be redefined.
<code>__STDC_VERSION__</code>	Identifies the ISO-C version number. Expands to 199901L for ISO C99, but the value may be redefined.
<code>__STLP_NO_IOSTREAMS</code>	Defined when option <code>--io-streams</code> is not used. This disables I/O stream functions in the STLport C++ library.
<code>__TASKING__</code>	Always defined for the TASKING C++ compiler.
<code>__THUMB__</code>	Expands to 1 if you used option <code>--thumb</code> , otherwise unrecognized as macro.
<code>__TIME__</code>	Expands to the compilation time: "hh:mm:ss"
<code>__TYPE_TRAITS_ENABLED</code>	Defined when type traits pseudo-functions (to ease the implementation of ISO/IEC TR 19768; e.g., <code>__is_union</code>) are enabled. This is the default in C++ mode.
<code>__VAR_HANDLE_TYPE</code>	Defines the type of the variable-handle field in the exception handling tables. This macro is used when building the C++ library.
<code>__VERSION__</code>	Identifies the version number of the C++ compiler. For example, if you use version 2.1r1 of the compiler, <code>__VERSION__</code> expands to 2001 (dot and revision number are omitted, minor version number in 3 digits).
<code>__VIRTUAL_FUNCTION_INDEX_TYPE</code>	Defines the type of the virtual function index field of the virtual function table. This macro is used when building the C++ library.
<code>__VIRTUAL_FUNCTION_TYPE</code>	Defines the type of the virtual function field of the virtual function table. This macro is used when building the C++ library.
<code>__WCHAR_MIN / __WCHAR_MAX</code>	Used in <code>stdint.h</code> to define the minimum/maximum value of a <code>wchar_t</code> type respectively.
<code>__WCHAR_T</code>	Defined when <code>wchar_t</code> is a keyword.

Table 2-1: Predefined macros

2.11 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.

2.11.1 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:

```
#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, heres a more complicated example:

```
#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:

```
// 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:

```
// 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:

```
// 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

```
#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.

2.11.2 Manual Precompiled Header Processing

Command line option `--create-pch=filename` specifies that a precompiled header file of the specified name should be created.

Command line option `--use-pch=filename` 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.

2.11.3 Other Ways to Control Precompiled Headers

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

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

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

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

- **#pragma no_pch** may be used to suppress precompiled header processing for a given source file.
- Command line option `--pch-dir=directory-name` is used to specify the directory in which to search for and/or create a PCH file.

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

2.11.4 Performance Issues

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

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

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

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

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

```
#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.



3 Libraries

Summary

This chapter lists all library functions that you can call in your C source.

3.1 Introduction

This chapter contains an overview of all library functions that you can call in your C source. This includes all functions of the standard C library (ISO C99) 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 3.2, [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.

C Library

Libraries	Description
carm.lib cthumb.lib	C library, for ARM and Thumb instructions respectively (some functions also need the floating-point library)
carms.lib cthumbs.lib	Single precision C library (some functions also need the floating-point library)
fparm.lib fpthumb.lib	Floating-point library
rtarm.lib rtthumb.lib	Run-time library
pbarm.lib pbthumb.lib pcarm.lib pctthumb.lib pctarm.lib pctthumb.lib pdarm.lib pdthumb.lib ptarm.lib ptthumb.lib	Profiling libraries: pb = block/function counter pc = call graph pct = call graph and timing pd = dummy pt = function timing

Table 3-1: Overview of libraries

C++ Library

The TASKING C++ compiler supports the STLport C++ libraries. STLport is a multiplatform ISO C++ Standard Library implementation. It is a free, open-source product, which is delivered with the TASKING C++ compiler. The library supports standard templates and I/O streams.

The include files for the STLport C++ libraries are present in directory `include.stl` relative to the product installation directory.

You can find more information on the STLport library on the following site: <http://stlport.sourceforge.net/>

Also read the license agreement on <http://stlport.sourceforge.net/License.shtml>. This license agreement is applicable to the STLport C++ library only. All other product components fall under the TASKING license agreement.

For an STL Programmer's Guide you can see <http://www.sgi.com/tech/stl/index.html>.

The following C++ libraries are delivered with the product:

Libraries	Description
cparm[s][x].lib cpthumb[s][x].lib	C++ libraries for ARM and Thumb s = single precision floating-point x = exception handling
sttarmx.lib stlthumbx.lib	STLport C++ libraries for ARM and Thumb (with exception handling)

3.2 Library Functions

The following sections 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.

A number of wide-character functions are available as C source code, but have not been compiled with the C library. To use complete wide-character functionality, you must recompile the libraries with the macro `WCHAR_SUPPORT_ENABLED` and keep this macro also defined when compiling your own sources. (See [C compiler option `--define \(-D\)`](#) in section 6.1, *C Compiler Options*, in Chapter 6, *Tool options*.)

3.2.1 `assert.h`

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

3.2.2 `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:

```
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

<code>csin</code>	<code>csinf</code>	<code>csinl</code>	Returns the complex sine of z .
<code>ccos</code>	<code>ccosf</code>	<code>ccosl</code>	Returns the complex cosine of z .
<code>ctan</code>	<code>ctanf</code>	<code>ctanl</code>	Returns the complex tangent of z .
<code>casin</code>	<code>casinf</code>	<code>casinl</code>	Returns the complex arc sine $\sin^{-1}(z)$.
<code>cacos</code>	<code>cacosf</code>	<code>cacosl</code>	Returns the complex arc cosine $\cos^{-1}(z)$.
<code>catan</code>	<code>catanf</code>	<code>catanl</code>	Returns the complex arc tangent $\tan^{-1}(z)$.
<code>csinh</code>	<code>csinhf</code>	<code>csinhl</code>	Returns the complex hyperbolic sine of z .
<code>ccosh</code>	<code>ccoshf</code>	<code>ccoshl</code>	Returns the complex hyperbolic cosine of z .
<code>ctanh</code>	<code>ctanhf</code>	<code>ctanhl</code>	Returns the complex hyperbolic tangent of z .
<code>casinh</code>	<code>casinhf</code>	<code>casinhl</code>	Returns the complex arc hyperbolic sinus of z .
<code>cacosh</code>	<code>cacoshf</code>	<code>cacoshl</code>	Returns the complex arc hyperbolic cosinus of z .
<code>catanh</code>	<code>catanhf</code>	<code>catanhl</code>	Returns the complex arc hyperbolic tangent of z .

Exponential and logarithmic functions

<code>cexp</code>	<code>cexpf</code>	<code>cexpl</code>	Returns the result of the complex exponential function e^z .
<code>clog</code>	<code>clogf</code>	<code>clogl</code>	Returns the complex natural logarithm.

Power and absolute-value functions

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

Manipulation functions

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

3.2.3 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.

ctype.h	wctype.h	Description
<code>isalnum</code>	<code>iswalnum</code>	Returns a non-zero value when c is an alphabetic character or a number ([A-Z][a-z][0-9]).
<code>isalpha</code>	<code>iswalpha</code>	Returns a non-zero value when c is an alphabetic character ([A-Z][a-z]).
<code>isblank</code>	<code>iswblank</code>	Returns a non-zero value when c is a blank character (tab, space...)
<code>iscntrl</code>	<code>iswcntrl</code>	Returns a non-zero value when c is a control character.
<code>isdigit</code>	<code>iswdigit</code>	Returns a non-zero value when c is a numeric character ([0-9]).
<code>isgraph</code>	<code>iswgraph</code>	Returns a non-zero value when c is printable, but not a space.
<code>islower</code>	<code>iswlower</code>	Returns a non-zero value when c is a lowercase character ([a-z]).
<code>isprint</code>	<code>iswprint</code>	Returns a non-zero value when c is printable, including spaces.
<code>ispunct</code>	<code>iswpunct</code>	Returns a non-zero value when c is a punctuation character (such as '.', ',', '!').
<code>isspace</code>	<code>iswspace</code>	Returns a non-zero value when c is a space type character (space, tab, vertical tab, formfeed, linefeed, carriage return).
<code>isupper</code>	<code>iswupper</code>	Returns a non-zero value when c is an uppercase character ([A-Z]).
<code>isxdigit</code>	<code>iswxdigit</code>	Returns a non-zero value when c is a hexadecimal digit ([0-9][A-F][a-f]).
<code>tolower</code>	<code>towlower</code>	Returns c converted to a lowercase character if it is an uppercase character, otherwise c is returned.
<code>toupper</code>	<code>towupper</code>	Returns c converted to an uppercase character if it is a lowercase character, otherwise c is returned.
<code>_tolower</code>	-	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.
<code>_toupper</code>	-	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.
<code>isascii</code>		Returns a non-zero value when c is in the range of 0 and 127. This function is not defined in ISO C99.
<code>toascii</code>		Converts c to an ASCII value (strip highest bit). This function is not defined in ISO C99.

3.2.4 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 C99.

<code>_dbg_trap</code>	Returns a non-zero value when <code>c</code> is in the range of 0 and 127. This function is not defined in ISO C99.
<code>_argcv(const char *buf, size_t size)</code>	Converts <code>c</code> to an ASCII value (strip highest bit). This function is not defined in ISO C99.

3.2.5 errno.h

<code>int errno</code>	External variable that holds implementation defined error codes.
------------------------	--

The following error codes are defined as macros in `errno.h`:

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

Floating-point errors

<code>EDOM</code>	23	Argument too large
<code>ERANGE</code>	24	Result too large

Errors returned by printf/scanf

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

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

<code>EILSEQ</code>	29	Invalid or incomplete multibyte or wide character
---------------------	----	---

Errors set by RTOS

<code>EILSEQ</code>	30	Operation canceled
<code>ENODEV</code>	31	No such device

3.2.6 fcntl.h

The file `fcntl.h` contains the function `open()`, which calls the low level function `_open()`, and definitions of flags used by the low level function `_open()`. This header file is not defined in ISO C99.

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

3.2.7 fenv.h

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

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

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

<code>FE_DIVBYZERO</code>	<code>FE_INEXACT</code>	<code>FE_INVALID</code>
<code>FE_OVERFLOW</code>	<code>FE_UNDERFLOW</code>	<code>FE_ALL_EXCEPT</code>

<code>fegetround</code>	Returns the current rounding direction, represented as one of the values of the rounding direction macros. <i>(Not implemented)</i>
<code>fesetround</code>	Sets the current rounding directions. <i>(Not implemented)</i>

Currently no rounding mode macros are implemented.

3.2.8 float.h

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



`float.h` used to contain prototypes for the functions `copysign(f)`, `isinf(f)`, `isfinite(f)`, `isnan(f)` and `scalb(f)`. These functions have accordingly to the ISO C99 standard been moved to the header file `math.h`. See also section 3.2.15, [math.h](#) and [tgmath.h](#).

3.2.9 inttypes.h and stdint.h

The header files `stdint.h` and `inttypes.h` provide additional declarations for integer types and have various characteristics. The `stdint.h` header file contains basic definitions of integer types of certain sizes, and corresponding sets of macros. This header file clearly refers to the corresponding sections in the ISO 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.

<code>intmax_t imaxabs(intmax_t j);</code>	Returns the absolute value of <code>j</code>
<code>imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);</code>	Computes <code>numer/denom</code> and <code>numer % denom</code> . The result is stored in the <code>quot</code> and <code>rem</code> components of the <code>imaxdiv_t</code> structure type.

<code>intmax_t strtoimax(const char * restrict nptr, char ** restrict endptr, int base);</code>	Convert string to maximum sized integer. (Compare <code>strtoul</code>)
<code>uintmax_t strtoumax(const char * restrict nptr, char ** restrict endptr, int base);</code>	Convert string to maximum sized unsigned integer. (Compare <code>strtoul</code>)
<code>intmax_t wcstojmax(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);</code>	Convert wide string to maximum sized integer. (Compare <code>wctol</code>)
<code>uintmax_t wcstoumax(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);</code>	Convert wide string to maximum sized unsigned integer. (Compare <code>wctoul</code>)

3.2.10 `io.h`

The header file `io.h` contains definitions and prototypes for low level I/O functions. This header file is not defined in ISO/IEC9899.

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

3.2.11 `iso646.h`

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

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

3.2.12 `limits.h`

Contains the sizes of integral types, defined as macros.

3.2.13 `locale.h`

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

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

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

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

LC_ALL	0	LC_NUMERIC	3
LC_COLLATE	1	LC_TIME	4
LC_CTYPE	2	LC_MONETARY	5

```
struct 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.

3.2.14 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 3.2.23, [stdlib.h and wchar.h](#).

<code>malloc(size)</code>	Allocates space for an object with size <i>size</i> . The allocated space is not initialized. Returns a pointer to the allocated space.
<code>calloc(nobj, size)</code>	Allocates space for <i>n</i> objects with size <i>size</i> . The allocated space is initialized with zeros. Returns a pointer to the allocated space.
<code>free(*ptr)</code>	Deallocates the memory space pointed to by <i>ptr</i> which should be a pointer earlier returned by the <code>malloc</code> or <code>calloc</code> function.
<code>realloc(*ptr, size)</code>	Deallocates the old object pointed to by <i>ptr</i> and returns a pointer to a new object with size <i>size</i> , 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.

3.2.15 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

math.h			tgmath.h	Description
<code>sin</code>	<code>sinf</code>	<code>sinl</code>	<code>sin</code>	Returns the sine of <i>x</i> .
<code>cos</code>	<code>cosf</code>	<code>cosl</code>	<code>cos</code>	Returns the cosine of <i>x</i> .
<code>tan</code>	<code>tanf</code>	<code>tanl</code>	<code>tan</code>	Returns the tangent of <i>x</i> .
<code>asin</code>	<code>asinf</code>	<code>asinl</code>	<code>asin</code>	Returns the arc sine $\sin^{-1}(x)$ of <i>x</i> .
<code>acos</code>	<code>acosf</code>	<code>acosl</code>	<code>acos</code>	Returns the arc cosine $\cos^{-1}(x)$ of <i>x</i> .
<code>atan</code>	<code>atanf</code>	<code>atanl</code>	<code>atan</code>	Returns the arc tangent $\tan^{-1}(x)$ of <i>x</i> .
<code>atan2</code>	<code>atan2f</code>	<code>atan2l</code>	<code>atan2</code>	Returns the result of: $\tan^{-1}(y/x)$.
<code>sinh</code>	<code>sinhf</code>	<code>sinhl</code>	<code>sinh</code>	Returns the hyperbolic sine of <i>x</i> .
<code>cosh</code>	<code>coshf</code>	<code>coshl</code>	<code>cosh</code>	Returns the hyperbolic cosine of <i>x</i> .
<code>tanh</code>	<code>tanhf</code>	<code>tanh1</code>	<code>tanh</code>	Returns the hyperbolic tangent of <i>x</i> .
<code>asinh</code>	<code>asinhf</code>	<code>asinh1</code>	<code>asinh</code>	Returns the arc hyperbolic sinus of <i>x</i> .
<code>acosh</code>	<code>acoshf</code>	<code>acosh1</code>	<code>acosh</code>	Returns the non-negative arc hyperbolic cosinus of <i>x</i> .
<code>atanh</code>	<code>atanhf</code>	<code>atanhl</code>	<code>atanh</code>	Returns the arc hyperbolic tangent of <i>x</i> .

Exponential and logarithmic functions

All of these functions are new in ISO C99, except for `exp`, `log` and `log10`.

math.h			tgmath.h	Description
exp	expf	expl	exp	Returns the result of the exponential function e^x .
exp2	exp2f	exp2l	exp2	Returns the result of the exponential function 2^x . (Not implemented)
expm1	expm1f	expm1l	expm1	Returns the result of the exponential function e^x-1 (Not implemented)
log	logf	logl	log	Returns the natural logarithm $\ln(x)$, $x>0$.
log10	log10f	log10l	log10	Returns the base-10 logarithm of x , $x>0$.
log1p	log1pf	log1pl	log1p	Returns the base-e logarithm of $(1+x)$. $x < -1$. (Not implemented)
log2	log2f	log2l	log2	Returns the base-2 logarithm of x . $x>0$. (Not implemented)
ilogb	ilogbf	ilogbl	ilogb	Returns the signed exponent of x as an integer. $x>0$. (Not implemented)
logb	logbf	logbl	logb	Returns the exponent of x as a signed integer in value in floating-point notation. $x > 0$. (Not implemented)

frexp, ldexp, modf, scalbn, scalbln

math.h			tgmath.h	Description
frexp	frexpl	frexpf	frexp	Splits a float x into fraction f and exponent n , so that: $f = 0.0$ or $0.5 \leq f \leq 1.0$ and $f \cdot 2^n = x$. Returns f , stores n .
ldexp	ldexpl	ldexpf	ldexp	Inverse of <code>frexp</code> . Returns the result of $x \cdot 2^n$. (x and n are both arguments).
modf	modfl	modff	-	Splits a float x into fraction f and integer n , so that: $ f < 1.0$ and $f + n = x$. Returns f , stores n .
scalbn	scalbnl	scalbnf	scalbn	Computes the result of $x \cdot \text{FLT_RADIX}^n$. efficiently, not normally by computing FLT_RADIX^n explicitly.
scalbln	scalblnl	scalblnf	scalbln	Same as <code>scalbn</code> but with argument n as long int.

Rounding functions

math.h			tgmath.h	Description
ceil	ceilf	ceill	ceil	Returns the smallest integer not less than x, as a double.
floor	floorf	floorl	floor	Returns the largest integer not greater than x, as a double.
rint	rintl	rintf	rint	Returns the rounded integer value as an int according to the current rounding direction. See fenv.h. (Not implemented)
lrint	lrintf	lrintl	lrint	Returns the rounded integer value as a long int according to the current rounding direction. See fenv.h. (Not implemented)
llrint	llrintf	llrintl	llrint	Returns the rounded integer value as a long long int according to the current rounding direction. See fenv.h. (Not implemented)
nearbyint	nearbyintf nearbyintl		nearbyint	Returns the rounded integer value as a floating-point according to the current rounding direction. See fenv.h. (Not implemented)
round	roundl	roundf	round	Returns the nearest integer value of x as int. (Not implemented)
lround	lroundl	lroundf	lround	Returns the nearest integer value of x as long int. (Not implemented)
llround	llroundl	llroundf	llround	Returns the nearest integer value of x as long long int. (Not implemented)
trunc	trunc	truncf	trunc	Returns the truncated integer value x. (Not implemented)

Remainder after division

math.h			tgmath.h	Description
fmod	fmodl	fmodf	fmod	Returns the remainder r of $x-ny$. n is chosen as $\text{trunc}(x/y)$. r has the same sign as x .
remainder	remainderl remainderf		remainder	Returns the remainder r of $x-ny$. n is chosen as $\text{trunc}(x/y)$. r may not have the same sign as x . <i>(Not implemented)</i>
remquo	remquol	remquof	remquo	Same as remainder. In addition, the argument <code>*quo</code> is given a specific value (see ISO). <i>(Not implemented)</i>

Power and absolute-value functions

math.h			tgmath.h	Description
cbrt	cbrtl	cbrtf	cbrt	Returns the real cube root of x ($=x^{1/3}$). <i>(Not implemented)</i>
fabs	fabsl	fabsf	fabs	Returns the absolute value of x ($ x $). (<code>abs</code> , <code>labs</code> , <code>llabs</code> , <code>div</code> , <code>ldiv</code> , <code>lldiv</code> are defined in <code>stdlib.h</code>)
fma	fmal	fmaf	fma	Floating-point multiply add. Returns $x*y+z$. <i>(Not implemented)</i>
hypot	hypotl	hypotf	hypot	Returns the square root of x^2+y^2 .
pow	powl	powf	power	Returns x raised to the power y (x^y).
sqrt	sqrtl	sqrtf	sqrt	Returns the non-negative square root of x . $x \neq 0$.

Manipulation functions: copysign, nan, nextafter, nexttoward

math.h			tgmath.h	Description
copysign	copysignl copysignf		copysign	Returns the value of x with the sign of y .
nan	nanl	nanf	-	Returns a quiet NaN, if available, with content indicated through <code>tagp</code> . <i>(Not implemented)</i>
nextafter	nextafterl nextafterf		nextafter	Returns the next representable value in the specified format after x in the direction of y . Returns y if $x=y$. <i>(Not implemented)</i>
nexttoward	nexttowardl nexttowardf		nexttoward	Same as <code>nextafter</code> , except that the second argument in all three variants is of type long double. Returns y if $x=y$. <i>(Not implemented)</i>

Positive difference, maximum, minimum

math.h			tgmath.h	Description
fdim	fdiml	fdimf	fdim	Returns the positive difference between: $ x-y $. <i>(Not implemented)</i>
fmax	fmaxl	fmaxf	fmax	Returns the maximum value of their arguments. <i>(Not implemented)</i>
fmin	fminl	fminf	fmin	Returns the minimum value of their arguments. <i>(Not implemented)</i>

Error and gamma (Not implemented)

math.h			tgmath.h	Description
erf	erfl	erff	erf	Computes the error function of x. (Not implemented)
erfc	erfcl	erfcf	erc	Computes the complementary error function of x. (Not implemented)
lgamma	lgammal	lgammaf	lgamma	Computes the $\ast \log_e \Gamma(x) $ (Not implemented)
tgamma	tgammal	tgammaf	tgamma	Computes $\Gamma(x)$ (Not implemented)

Comparison macros

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

math.h	tgmath.h	Description
isgreater	–	Returns the value of $(x) > (y)$
isgreaterequal	–	Returns the value of $(x) \geq (y)$
isless	–	Returns the value of $(x) < (y)$
islessequal	–	Returns the value of $(x) \leq (y)$
islessgreater	–	Returns the value of $(x) < (y) \mid \mid (x) > (y)$
isunordered	–	Returns 1 if its arguments are unordered, 0 otherwise.

Classification macros

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

math.h	tgmath.h	Description
fpclassify	–	Returns the class of its argument: FP_INFINITE, FP_NAN, FP_NORMAL, FP_SUBNORMAL or FP_ZERO
isfinite	–	Returns a nonzero value if and only if its argument has a finite value
isinf	–	Returns a nonzero value if and only if its argument has an infinit value
isnan	–	Returns a nonzero value if and only if its argument has NaN value.
isnormal	–	Returns a nonzero value if an only if its argument has a normal value.
signbit	–	Returns a nonzero value if and only if its argument value is negative.

3.2.16 setjmp.h

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

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

3.2.17 signal.h

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

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

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

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

The next function determines how subsequent signals will be handled:

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

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

SIG_DFL	Default behaviour is used
SIG_IGN	The signal is ignored

The function returns the previous value of `signalfunction` for the specific signal, or `SIG_ERR` if an error occurs.

3.2.18 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:

<code>va_arg(ap, type)</code>	Returns the value of the next argument in the variable argument list. It's return type has the type of the given argument <code>type</code> . A next call to this macro will return the value of the next argument.
<code>va_copy(va_list dest, va_list src)</code>	This macro duplicates the current state of <code>src</code> in <code>dest</code> , creating a second pointer into the argument list. After this call, <code>va_arg()</code> may be used on <code>src</code> and <code>dest</code> independently.
<code>va_end(va_list ap)</code>	This macro must be called after the arguments have been processed. It should be called before the function using the macro 'va_start' is terminated (ANSI specification).
<code>va_start(va_list ap, lastarg);</code>	This macro initializes <code>ap</code> . After this call, each call to <code>va_arg()</code> will return the value of the next argument. In our implementation, <code>va_list</code> cannot contain any bit type variables. Also the given argument <code>lastarg</code> must be the last non-bit type argument in the list.

3.2.19 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.

```
#define bool                _Bool
#define true                1
#define false               0
#define __bool_true_false_are_defined 1
```

3.2.20 stddef.h

This header file defines the types for common use:

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

Besides these types, the following macros are defined:

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

3.2.21 `stdint.h`



See section 3.2.9, [inttypes.h and stdint.h](#)

3.2.22 `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. An **FILE** object is created with the function `fopen`. The pointer to this object is used as an argument in many of the in this header file. The **FILE** object can contain the following information:

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

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

Macros

<code>stdio.h</code> / <code>wchar.h</code>	Description
<code>NULL</code>	expands to the null pointer constant
<code>BUFSIZ</code>	Size of the buffer used by the <code>setbuf/setvbuf</code> function: 512
<code>EOF</code>	End of file indicator. Expands to <code>-1</code> .
<code>WEOF</code>	End of file indicator. Expands to <code>UINT_MAX</code> (defined in <code>limits.h</code>) NOTE: <code>WEOF</code> need not to be a negative number as long as its value does not correspond to a member of the wide character set. (Defined in <code>wchar.h</code>).
<code>FOPEN_MAX</code>	Number of files that can be opened simultaneously: 10
<code>FILENAME_MAX</code>	Maximum length of a filename: 100
<code>_IOFBF</code> <code>_IOLBF</code> <code>_IONBF</code>	Expand to an integer expression, suitable for use as argument to the <code>setvbuf</code> function.
<code>L_tmpnam</code>	Size of the string used to hold temporary file names: 8 (<code>tmpxxxxx</code>)
<code>TMP_MAX</code>	Maximum number of unique temporary filenames that can be generated: 0x8000
<code>SEEK_CUR</code> <code>SEEK_END</code> <code>SEEK_SET</code>	Expand to an integer expression, suitable for use as the third argument to the <code>fseek</code> function.
<code>stderr</code> <code>stdin</code> <code>stdout</code>	Expressions of type "pointer to FILE" that point to the FILE objects associated with standard error, input and output streams.

File access

stdio.h	Description
<code>fopen(name, mode)</code>	Opens a file for a given mode. Available modes are: "r" read; open text file for reading "w" write; create text file for writing; if the file already exists its contents is discarded "a" append; open existing text file or create new text file for writing at end of file "r+" open text file for update; reading and writing "w+" create text file for update; previous contents if any is discarded "a+" append; open or create text file for update, writes at end of file
<code>fclose(name)</code>	Flushes the data stream and closes the specified file that was previously opened with <code>fopen</code> .
<code>fflush(name)</code>	If stream is an output stream, any buffered but unwritten data is written. Else, the effect is undefined.
<code>freopen(name, mode, stream)</code>	Similar to <code>fopen</code> , but rather than generating a new value of type <code>FILE *</code> , the existing value is associated with a new stream.
<code>setbuf(stream, buffer)</code>	If <code>buffer</code> is <code>NULL</code> , buffering is turned off for the stream. Otherwise, <code>setbuf</code> is equivalent to: <code>(void) setvbuf(stream, buf, _IOFBF, BUFSIZ).</code>
<code>setvbuf(stream, buffer, mode, size)</code>	Controls buffering for the <i>stream</i> ; this function must be called before reading or writing. <i>Mode</i> can have the following values: <code>_IOFBF</code> causes full buffering <code>_IOLBF</code> causes line buffering of text files <code>_IONBF</code> causes no buffering If <i>buffer</i> is not <code>NULL</code> , it will be used as a buffer; otherwise a buffer will be allocated. <i>size</i> determines the buffer size.

Character input/output

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

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

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

Table 3-2: Printf conversion characters

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

The format string can contain :

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

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

- A '*', meaning that no assignment is done for this field.
- A number specifying the maximum field width.
- The conversion characters d, i, n, o, u and x may be 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.

Character	Scanned as
d	int, signed decimal.
i	int, the integer may be given octal (i.e. a leading 0 is entered) or hexadecimal (leading "0x" or "0X"), or just decimal.
o	int, unsigned octal.
u	int, unsigned decimal.
x	int, unsigned hexadecimal in lowercase or uppercase.
c	single character (converted to unsigned char).
s	char *, a string of non white space characters. The argument should point to an array of characters, large enough to hold the string and a terminating NULL character.

Character	Scanned as
f	float
e, E	float
g, G	float
a, A	float
n	int *, the number of characters written so far is written into the argument. No scanning is done.
p	pointer; hexadecimal 24-bit value which must be entered without 0x- prefix.
[...]	Matches a string of input characters from the set between the brackets. A NULL character is added to terminate the string. Specifying [...] includes the ']' character in the set of scanning characters.
[^...]	Matches a string of input characters not in the set between the brackets. A NULL character is added to terminate the string. Specifying [^...] includes the ']' character in the set.
%	Literal '%', no assignment is done.

Table 3-3: Scanf conversion characters

stdio.h	wchar.h	Description
<code>fscanf(stream, format, ...)</code>	<code>fwscanf(stream, format, ...)</code>	Performs a formatted read from the given <i>stream</i> . Returns the number of items converted successfully. (FSS implementation)
<code>scanf(format, ...)</code>	<code>wscanf(format, ...)</code>	Performs a formatted read from the <code>stdin</code> stream. Returns the number of items converted successfully. (FSS implementation)
<code>sscanf(*s, format, ...)</code>	<code>swscanf(*s, format, ...)</code>	Performs a formatted read from the string <i>s</i> . Returns the number of items converted successfully.
<code>vfscanf(stream, format, arg)</code>	<code>vfwscanf(stream, format, arg)</code>	Same as <code>fscanf/fwscanf</code> , but extra arguments are given as variable argument list <i>arg</i> . (See section 3.2.18, stdarg.h)
<code>vscanf(format, arg)</code>	<code>vwscanf(format, arg)</code>	Same as <code>scanf/wscanf</code> , but extra arguments are given as variable argument list <i>arg</i> . (See section 3.2.18, stdarg.h)
<code>vsscanf(s, format, arg)</code>	<code>vswscanf(s, format, arg)</code>	Same as <code>scanf/wscanf</code> , but extra arguments are given as variable argument list <i>arg</i> . (See section 3.2.18, stdarg.h)
<code>fprintf(stream, format, ...)</code>	<code>fwprintf(stream, format, ...)</code>	Performs a formatted write to the given <i>stream</i> . Returns EOF/WEOF on error. (FSS implementation)
<code>printf(format, ...)</code>	<code>wprintf(format, ...)</code>	Performs a formatted write to the stream <code>stdout</code> . Returns EOF/WEOF on error. (FSS implementation)
<code>sprintf(*s, format, ...)</code>	–	Performs a formatted write to string <i>s</i> . Returns EOF/WEOF on error.
<code>snprintf(*s, n, format, ...)</code>	<code>swprintf(*s, n, format, ...)</code>	Same as <code>sprintf</code> , but <i>n</i> specifies the maximum number of characters (including the terminating null character) to be written.
<code>vfprintf(stream, format, arg)</code>	<code>vfwprintf(stream, format, arg)</code>	Same as <code>fprintf/fwprintf</code> , but extra arguments are given as variable argument list <i>arg</i> . (See section 3.2.18, stdarg.h) (FSS implementation)
<code>vprintf(format, arg)</code>	<code>vwprintf(format, arg)</code>	Same as <code>printf/wprintf</code> , but extra arguments are given as variable argument list <i>arg</i> . (See section 3.2.18, stdarg.h) (FSS implementation)
<code>vsprintf(*s, format, arg)</code>	<code>vswprintf(*s, format, arg)</code>	Same as <code>sprintf/swprintf</code> , but extra arguments are given as variable argument list <i>arg</i> . (See section 3.2.18, stdarg.h) (FSS implementation)

Character input/output

stdio.h	wchar.h	Description
<code>fgetc(stream)</code>	<code>fgetwc(stream)</code>	Reads one character from <i>stream</i> . Returns the read character, or EOF/WEOF on error. (FSS implementation)
<code>getc(stream)</code>	<code>getwc(stream)</code>	Same as <code>fgetc/fgetwc</code> except that is implemented as a macro. (FSS implementation) NOTE: Currently #defined as <code>getchar()/getwchar()</code> because FILE I/O is not supported. Returns the read character, or EOF/WEOF on error.
<code>getchar(stdin)</code>	<code>getwchar(stdin)</code>	Reads one character from the <code>stdin</code> stream. Returns the character read or EOF/WEOF on error. Implemented as macro. (FSS implementation)
<code>fgets(*s,n,stream)</code>	<code>fgetws(*s,n,stream)</code>	Reads at most the next <i>n</i> -1 characters from the <i>stream</i> into array <i>s</i> until a newline is found. Returns <i>s</i> or NULL or EOF/WEOF on error. (FSS implementation)
<code>gets(*s,n,stdin)</code>	-	Reads at most the next <i>n</i> -1 characters from the <code>stdin</code> stream into array <i>s</i> . A newline is ignored. Returns <i>s</i> or NULL or EOF/WEOF on error. (FSS implementation)
<code>ungetc(c,stream)</code>	<code>ungetwc(c,stream)</code>	Pushes character <i>c</i> back onto the input <i>stream</i> . Returns EOF/WEOF on error.
<code>fputc(c,stream)</code>	<code>fputwc(c,stream)</code>	Put character <i>c</i> onto the given <i>stream</i> . Returns EOF/WEOF on error. (FSS implementation)
<code>putc(c,stream)</code>	<code>putwc(c,stream)</code>	Same as <code>fputc/fputwc</code> except that is implemented as a macro. (FSS implementation)
<code>putchar(c,stdout)</code>	<code>putwchar(c,stdout)</code>	Put character <i>c</i> onto the <code>stdout</code> stream. Returns EOF/WEOF on error. Implemented as macro. (FSS implementation)
<code>fputs(*s,stream)</code>	<code>fputws(*s,stream)</code>	Writes string <i>s</i> to the given <i>stream</i> . Returns EOF/WEOF on error.
<code>puts(*s)</code>	-	Writes string <i>s</i> to the <code>stdout</code> stream. Returns EOF/WEOF on error. (FSS implementation)

Direct input/output

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

Random access

stdio.h	Description
<code>fseek(stream,offset,origin)</code>	Sets the position indicator for <i>stream</i> . (FSS implementation)

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

<code>SEEK_SET</code>	0	<i>offset</i> characters from the beginning of the file
<code>SEEK_CUR</code>	1	<i>offset</i> characters from the current position in the file
<code>SEEK_END</code>	2	<i>offset</i> characters from the end of the file

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

Operations on files

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

Error handling

stdio.h	Description
<code>clearerr(stream)</code>	Clears the end of file and error indicators for <i>stream</i> .
<code>ferror(stream)</code>	Returns a non-zero value if the error indicator for <i>stream</i> is set.
<code>feof(stream)</code>	Returns a non-zero value if the end of file indicator for <i>stream</i> is set.
<code>perror(*s)</code>	Prints <i>s</i> and the error message belonging to the integer <code>errno</code> . (See section 3.2.5, errno.h)

3.2.23 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

<code>EXIT_SUCCESS</code> 0	Predefined exit codes that can be used in the <code>exit</code> function.
<code>EXIT_FAILURE</code> 1	
<code>RAND_MAX</code> 32767	Highest number that can be returned by the <code>rand/srand</code> function.
<code>MB_CUR_MAX</code> 1	Maximum number of bytes in a multibyte character for the extended character set specified by the current locale (category <code>LC_CTYPE</code> , see section 3.2.13, locale.h).

Numeric conversions

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

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

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

stdlib.h

```
float      strtod(*s,**endp)
double     strtod(*s,**endp)
long double strtold(*s,**endp)
```

wchar.h

```
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.

stdlib.h

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

wchar.h

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

Random number generation

```
rand          Returns a pseudo random integer in the range 0 to RAND_MAX.
srand(seed)   Same as rand but uses seed for a new sequence of pseudo random numbers.
```

Memory management

```
malloc(size)   Allocates space for an object with size size. The allocated space is not initialized. Returns a
                pointer to the allocated space.
calloc(nobj,size) Allocates space for n objects with size size. The allocated space is initialized with zeros. Returns
                a pointer to the allocated space.
free(ptr)      Deallocates the memory space pointed to by ptr which should be a pointer earlier returned by the
                malloc or calloc function.
realloc(ptr,size) Deallocates the old object pointed to by ptr and returns a pointer to a new object with size size,
                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 3.2.17, signal.h).
atexit(func)    Func points to a function that is called (without arguments) when the program normally
                terminates.
exit(status)    Causes normal program termination. Acts as if main() returns with status as the return value.
                Status can also be specified with the predefined macros EXIT_SUCCESS or EXIT_FAILURE.
_Exit(status)   Same as exit, but not registered by the atexit function or signal handlers registered by the
                signal function are called.
```

<code>getenv(*s)</code>	Searches an environment list for a string <i>s</i> . Returns a pointer to the contents of <i>s</i> . NOTE: this function is not implemented because there is no OS.
<code>system(*s)</code>	Passes the string <i>s</i> to the environment for execution. NOTE: this function is not implemented because there is no OS.

Searching and sorting

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

Integer arithmetic

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

Multibyte/wide character and string conversions

<code>mblen(*s,n)</code>	Determines the number of bytes in the multi-byte character pointed to by <i>s</i> . At most <i>n</i> characters will be examined. (See also <code>mbrlen</code> in section 3.2.27, wchar.h)
<code>mbtowc(*pwc,*s,n)</code>	Converts the multi-byte character in <i>s</i> to a wide-character code and stores it in <i>pwc</i> . At most <i>n</i> characters will be examined.
<code>wctomb(*s,wc)</code>	Converts the wide-character <i>wc</i> into a multi-byte representation and stores it in the string pointed to by <i>s</i> . At most <code>MB_CUR_MAX</code> characters are stored.
<code>mbstowcs(*pwcs,*s,n)</code>	Converts a sequence of multi-byte characters in the string pointed to by <i>s</i> into a sequence of wide characters and stores at most <i>n</i> wide characters into the array pointed to by <i>pwcs</i> . (See also <code>mbstowcs</code> in section 3.2.27, wchar.h)
<code>wcstombs(*s,*pwcs,n)</code>	Converts a sequence of wide characters in the array pointed to by <i>pwcs</i> into multi-byte characters and stores at most <i>n</i> multi-byte characters into the string pointed to by <i>s</i> . (See also <code>wcrtowmb</code> in section 3.2.27, wchar.h)

3.2.24 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

stdio.h	wchar.h	Description
<code>memcpy(*s1,*s2,n)</code>	<code>wmemcpy(*s1,*s2,n)</code>	Copies <i>n</i> characters from <i>*s2</i> into <i>*s1</i> and returns <i>*s1</i> . If <i>*s1</i> and <i>*s2</i> overlap the result is undefined.
<code>memmove(*s1,*s2,n)</code>	<code>wmemmove(*s1,*s2,n)</code>	Same as <code>memcpy</code> , but overlapping strings are handled correctly. Returns <i>*s1</i> .
<code>strcpy(*s1,*s2)</code>	<code>wscpy(*s1,*s2)</code>	Copies <i>*s2</i> into <i>*s1</i> and returns <i>*s1</i> . If <i>*s1</i> and <i>*s2</i> overlap the result is undefined.
<code>strncpy(*s1,*s2,n)</code>	<code>wcsncpy(*s1,*s2,n)</code>	Copies not more than <i>n</i> characters from <i>*s2</i> into <i>*s1</i> and returns <i>*s1</i> . If <i>*s1</i> and <i>*s2</i> overlap the result is undefined.

stdio.h	wchar.h	Description
<code>strcat(*s1,*s2)</code>	<code>wscat(*s1,*s2)</code>	Appends a copy of <i>s2</i> to <i>s1</i> and returns <i>s1</i> . If <i>s1</i> and <i>s2</i> overlap the result is undefined.
<code>strncat(*s1,*s2,n)</code>	<code>wcsncat(*s1,*s2,n)</code>	Appends not more than <i>n</i> characters from <i>s2</i> to <i>s1</i> and returns <i>s1</i> . If <i>s1</i> and <i>s2</i> overlap the result is undefined.

Comparison functions

stdio.h	wchar.h	Description
<code>memcmp(*s1,*s2,n)</code>	<code>wmemcmp(*s1,*s2,n)</code>	Compares the first <i>n</i> characters of <i>s1</i> to the first <i>n</i> characters of <i>s2</i> . Returns < 0 if <i>s1</i> < <i>s2</i> , 0 if <i>s1</i> = <i>s2</i> , or > 0 if <i>s1</i> > <i>s2</i> .
<code>strcmp(*s1,*s2)</code>	<code>wscmp(*s1,*s2)</code>	Compares string <i>s1</i> to string <i>s2</i> . Returns < 0 if <i>s1</i> < <i>s2</i> , 0 if <i>s1</i> = <i>s2</i> , or > 0 if <i>s1</i> > <i>s2</i> .
<code>strncmp(*s1,*s2,n)</code>	<code>wcsncmp(*s1,*s2,n)</code>	Compares the first <i>n</i> characters of <i>s1</i> to the first <i>n</i> characters of <i>s2</i> . Returns < 0 if <i>s1</i> < <i>s2</i> , 0 if <i>s1</i> = <i>s2</i> , or > 0 if <i>s1</i> > <i>s2</i> .
<code>strcoll(*s1,*s2)</code>	<code>wscoll(*s1,*s2)</code>	Performs a local-specific comparison between string <i>s1</i> and string <i>s2</i> according to the LC_COLLATE category of the current locale. Returns < 0 if <i>s1</i> < <i>s2</i> , 0 if <i>s1</i> = <i>s2</i> , or > 0 if <i>s1</i> > <i>s2</i> . (See section 3.2.13, locale.h)
<code>strxfrm(*s1,*s2,n)</code>	<code>wcsxfrm(*s1,*s2,n)</code>	Transforms (a local) string <i>s2</i> so that a comparison between transformed strings with <code>strcmp</code> gives the same result as a comparison between non-transformed strings with <code>strcoll</code> . Returns the transformed string <i>s1</i> .

Search functions

stdio.h	wchar.h	Description
<code>memchr(*s,c,n)</code>	<code>wmemchr(*s,c,n)</code>	Checks the first <i>n</i> characters of <i>s</i> on the occurrence of character <i>c</i> . Returns a pointer to the found character.
<code>strchr(*s,c)</code>	<code>wcschr(*s,c)</code>	Returns a pointer to the first occurrence of character <i>c</i> in string <i>s</i> or the null pointer if not found.
<code>strrchr(*s,c)</code>	<code>wcsrchr(*s,c)</code>	Returns a pointer to the last occurrence of character <i>c</i> in string <i>s</i> or the null pointer if not found.
<code>strspn(*s,*set)</code>	<code>wcsspn(*s,*set)</code>	Searches <i>s</i> for a sequence of characters specified in <i>set</i> . Returns the length of the first sequence found.
<code>strcspn(*s,*set)</code>	<code>wscspn(*s,*set)</code>	Searches <i>s</i> for a sequence of characters <i>not</i> specified in <i>set</i> . Returns the length of the first sequence found.
<code>strpbrk(*s,*set)</code>	<code>wcspbrk(*s,*set)</code>	Same as <code>strspn/wcsspn</code> but returns a pointer to the first character in <i>s</i> that also is specified in <i>set</i> .
<code>strstr(*s,*sub)</code>	<code>wcsstr(*s,*sub)</code>	Searches for a substring <i>sub</i> in <i>s</i> . Returns a pointer to the first occurrence of <i>sub</i> in <i>s</i> .
<code>strtok(*s,*delim)</code>	<code>wcstok(*s,*delim)</code>	A sequence of calls to this function breaks the string <i>s</i> into a sequence of tokens delimited by a character specified in <i>delim</i> . The token found in <i>s</i> is terminated with a null character. The function returns a pointer to the first position in <i>s</i> of the token.

Miscellaneous functions

stdio.h	wchar.h	Description
<code>memset(*s,c,n)</code>	<code>wmemset(*s,c,n)</code>	Fills the first <i>n</i> bytes of <i>s</i> with character <i>c</i> and returns <i>s</i> .
<code>strerror(errno)</code>	–	Typically, the values for <code>errno</code> come from <code>int errno</code> . This function returns a pointer to the associated error message. (See also section 3.2.5, errno.h)
<code>strlen(*s)</code>	<code>wcslon(*s)</code>	Returns the length of string <i>s</i> .

3.2.25 time.h and wchar.h

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

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

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

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

Time manipulation

<code>clock</code>	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 <code>clock</code> should be divided by the value defined as <code>CLOCKS_PER_SEC</code> <code>12000000</code>
<code>difftime(t1,t0)</code>	Returns the difference $t1-t0$ in seconds.
<code>mktime(tm *tp)</code>	Converts the broken-down time in the structure pointed to by <i>tp</i> , to a value of type <code>time_t</code> . The return value has the same encoding as the return value of the <code>time</code> function.
<code>time(*timer)</code>	Returns the current calendar time. This value is also assigned to <i>*timer</i> .

Time conversion

<code>asctime(tm *tp)</code>	Converts the broken-down time in the structure pointed to by <i>tp</i> into a string in the form <code>Mon Jan 21 16:15:14 2004\n\0</code> . Returns a pointer to this string.
<code>ctime(*timer)</code>	Converts the calendar time pointed to by <i>timer</i> to local time in the form of a string. This is equivalent to: <code>asctime(localtime(timer))</code>
<code>gmtime(*timer)</code>	Converts the calendar time pointed to by <i>timer</i> to the broken-down time, expressed as UTC. Returns a pointer to the broken-down time.
<code>localtime(*timer)</code>	Converts the calendar time pointed to by <i>timer</i> to the broken-down time, expressed as local time. Returns a pointer to the broken-down time.

Formatted time

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

<code>stdio.h</code>	<code>wchar.h</code>
<code>strftime(*s,smax,*fmt,tm *tp)</code>	<code>wstrftime(*s,smax,*fmt,tm *tp)</code>

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 3.2.13, [locale.h](#)). You can use the next conversion specifiers:

%a	abbreviated weekday name
%A	full weekday name
%b	abbreviated month name
%B	full month name
%c	local date and time representation (same as %a %b %e %T %Y)
%C	last two 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 the newline character
%p	local equivalent of AM or PM
%r	locale's 12-hour clock time; same as %I:%M:%S %p
%R	same as %H:%M
%S	second (00–59)
%t	replaced by horizontal tab character
%T	ISO 8601 time format: %H:%M:%S
%u	ISO 8601 weekday number (1–7), Monday as first day of the week
%U	week number of the year, Sunday as first day of the week (00–53)
%V	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
%%	%

3.2.26 `unistd.h`

The file `unistd.h` contains standard UNIX I/O functions. These functions are all implemented using the debugger's file system simulation. Except for `lstat` and `fstat` which are not implemented. This header file is not defined in ISO C99.

<code>access(*name, mode)</code>	Use the file system simulation of the debugger to check the permissions of a file on the host. <i>mode</i> specifies the type of access and is a bit pattern constructed by a logical OR of the following values: <div style="margin-left: 20px;"> <code>R_OK</code> Checks read permission. <code>W_OK</code> Checks write permission. <code>X_OK</code> Checks execute (search) permission. <code>F_OK</code> Checks to see if the file exists. (FSS implementation) </div>
<code>chdir(*path)</code>	Use the file system simulation feature of the debugger to change the current directory on the host to the directory indicated by <i>path</i> . (FSS implementation)
<code>close(fd)</code>	File close function. The given file descriptor should be properly closed. This function calls <code>_close()</code> . (FSS implementation)
<code>getcwd(*buf, size)</code>	Use the file system simulation feature of the debugger to retrieve the current directory on the host. Returns the directory name. (FSS implementation)

<code>lseek(fd, offset, whence)</code>	Moves read-write file offset. Calls <code>_lseek()</code> . (FSS implementation)
<code>read(fd, *buff, cnt)</code>	Reads a sequence of characters from a file. This function calls <code>_read()</code> . (FSS implementation)
<code>stat(*name, *buff)</code>	Use the file system simulation feature of the debugger to <code>stat()</code> a file on the host platform. (FSS implementation)
<code>lstat(*name, *buff)</code>	This function is identical to <code>stat()</code> , except in the case of a symbolic link, where the link itself is 'stat'-ed, not the file that it refers to. (Not implemented)
<code>fstat(fd, *buff)</code>	This function is identical to <code>stat()</code> , except that it uses a file descriptor instead of a name. (Not implemented)
<code>unlink(*name)</code>	Removes the named file, so that a subsequent attempt to open it fails. Calls <code>_unlink()</code> . (FSS implementation)
<code>write(fd, *buff, cnt)</code>	Write a sequence of characters to a file. Calls <code>_write()</code> . (FSS implementation)

3.2.27 wchar.h

Many in `wchar.h` represent the wide-character variant of other so these are discussed together. (See sections 3.2.22, [stdio.h and wchar.h](#), 3.2.23, [stdlib.h and wchar.h](#), 3.2.24, [string.h and wchar.h](#) and 3.2.25, [time.h and wchar.h](#)).

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

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

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

<code>mbsinit(*ps)</code>	Determines whether the object pointed to by <i>ps</i> , is an initial conversion state. Returns a non-zero value if so.
<code>mbsrtowcs(*pwcs, **src, n, *ps)</code>	Restartable version of <code>mbstowcs</code> . See section 3.2.23, stdlib.h and wchar.h . The initial conversion state is specified by <i>ps</i> . The input sequence of multibyte characters is specified indirectly by <i>src</i> .
<code>wcsrtombs(*s, **src, n, *ps)</code>	Restartable version of <code>wcstombs</code> . See section 3.2.23, stdlib.h and wchar.h . The initial conversion state is specified by <i>ps</i> . The input wide string is specified indirectly by <i>src</i> .
<code>mbrtowc(*pwc, *s, n, *ps)</code>	Converts a multibyte character <i>s</i> to a wide character <i>pwc</i> according to conversion state <i>ps</i> . See also <code>mbtowc</code> in section 3.2.23, stdlib.h and wchar.h .
<code>wcrtomb(*s, wc, *ps)</code>	Converts a wide character <i>wc</i> to a multi-byte character according to conversion state <i>ps</i> and stores the multi-byte character in <i>s</i> .
<code>btowc(c)</code>	Returns the wide character corresponding to character <i>c</i> . Returns <code>WEOF</code> on error.
<code>wctob(c)</code>	Returns the multi-byte character corresponding to the wide character <i>c</i> . The returned multi-byte character is represented as one byte. Returns <code>EOF</code> on error.
<code>mbrlen(*s, n, *ps)</code>	Inspects up to <i>n</i> bytes from the string <i>s</i> to see if those characters represent valid multibyte characters, relative to the conversion state held in <i>ps</i> .

3.2.28 wctype.h

Most in `wctype.h` represent the wide-character variant of declared in `ctype.h` and are discussed in section 3.2.3, [ctype.h](#) and [wctype.h](#). In addition, this header file provides extensible, locale specific, 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 3.2.13, [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.

Function	Equivalent to locale specific test
----------	------------------------------------

<code>iswalnum(wc)</code>	<code>iswctype(wc, wctype("alnum"))</code>
<code>iswalpha(wc)</code>	<code>iswctype(wc, wctype("alpha"))</code>
<code>iswcntrl(wc)</code>	<code>iswctype(wc, wctype("cntrl"))</code>
<code>iswdigit(wc)</code>	<code>iswctype(wc, wctype("digit"))</code>
<code>iswgraph(wc)</code>	<code>iswctype(wc, wctype("graph"))</code>
<code>iswlower(wc)</code>	<code>iswctype(wc, wctype("lower"))</code>
<code>iswprint(wc)</code>	<code>iswctype(wc, wctype("print"))</code>
<code>iswpunct(wc)</code>	<code>iswctype(wc, wctype("punct"))</code>
<code>iswspace(wc)</code>	<code>iswctype(wc, wctype("space"))</code>
<code>iswupper(wc)</code>	<code>iswctype(wc, wctype("upper"))</code>
<code>iswxdigit(wc)</code>	<code>iswctype(wc, wctype("xdigit"))</code>

`wctrans(*property)` Constructs a value of type `wctype_t` that describes a mapping between wide characters identified by the string **property*. If *property* identifies a valid mapping of wide characters according to the `LC_TYPE` category (see section 3.2.13, [locale.h](#)) of the current locale, a non-zero value is returned that can be used as an argument in the `towctrans` function.

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

Function	Equivalent to locale specific transformation
----------	--

<code>towlower(wc)</code>	<code>towctrans(wc, wctrans("tolower"))</code>
<code>towupper(wc)</code>	<code>towctrans(wc, wctrans("toupper"))</code>



4 Assembly Language

Summary

This chapter describes the most important aspects of the TASKING assembly language 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*.

4.1 Assembly Syntax

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

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

The syntax of an assembly *statement* is:

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

label A label is a special symbol which is assigned the value and type of the current program location counter. A label can consist of letters, digits and underscore characters (_). The first character cannot be a digit. 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.

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

1: b lp ; This is an endless loop
      ; using numeric labels
```

instruction 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 4.3, [Operands of an Assembly Instruction](#). The instructions are described in the target's *Core 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 4.10, [Generic Instructions](#).

directive With directives you can control the assembler from within the assembly source. Except for preprocessing directives, these must not start in the first column. Directives are described in section 4.8, [Assembler Directives](#).

macro_call A call to a previously defined macro. It must not start in the first column. See section 4.9 [Macro Operations](#).

comment Comment, preceded by a ; (semicolon).

You can use empty lines or lines with only comments.

4.2 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 4.6.3, *Expression Operators*. Other special assembler characters are:

Character	Description
;	Start of a comment
\	Line continuation character or Macro operator: argument concatenation
?	Macro operator: return decimal value of a symbol
%	Macro operator: return hex value of a symbol
^	Macro operator: override local label
"	Macro string delimiter or Quoted string .DEFINE expansion character
'	String constants delimiter
@	Start of a built-in assembly function
\$	Location counter substitution

Note that macro operators have a higher precedence than expression operators.

4.3 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:

Operand	Description
<i>symbol</i>	A symbolic name as described in section 4.4, <i>Symbol Names</i> . Symbols can also occur in expressions.
<i>register</i>	Any valid register as listed in section 4.5, <i>Registers</i> .
<i>expression</i>	Any valid expression as described in section 4.6, <i>Assembly Expressions</i> .
<i>address</i>	A combination of <i>expression</i> , <i>register</i> and <i>symbol</i> .

4.4 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 4.4.1, *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 beginning with a period (.) are reserved for the system (for example for directives or section names). Instructions are also reserved. The case of these built-in symbols is insignificant.

Examples

Valid symbol names:

```
loop_1
ENTRY
a_B_c
_aBC
```

Invalid symbol names:

```
1_loop      (starts with a number)
.DEFINE     (reserved directive name)
```

4.4.1 Predefined Preprocessor Symbols

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

Macro	Description
<code>__ASARM__</code>	Expands to 1 for the ARM toolset, otherwise unrecognized as macro.
<code>__BUILD__</code>	Identifies the build number of the assembler, composed of decimal digits for the build number, three digits for the major branch number and three digits for the minor branch number. For example, if you use build 1.22.1 of the compiler, <code>__BUILD__</code> expands to 1022001. If there is no branch number, the branch digits expand to zero. For example, build 127 results in 127000000.
<code>__REVISION__</code>	Identifies the revision number of the assembler. For example, if you use version 1.0r2 of the compiler, <code>__REVISION__</code> expands to 2.
<code>__TASKING__</code>	Identifies the assembler as a TASKING assembler. Expands to 1 if a TASKING compiler is used.
<code>__VERSION__</code>	Identifies the version number of the assembler. For example, if you use version 1.0r2 of the assembler, <code>__VERSION__</code> expands to 1000 (dot and revision number are omitted, minor version number in 3 digits).

Table 4-1: Assembler predefined preprocessor symbols

4.5 Registers

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

ARM registers

```
R0 .. R15
IP (alias for R12)   SP (alias for R13)
LR (alias for R14)   PC (alias for R15)
```

4.6 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 may contain user-defined labels (and their associated integer values), and any combination of integers 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.

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.

4.6.1 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.

Base	Description	Example
Binary	A 0b prefix followed by binary digits (0,1). Or use a b suffix	0b1101 11001010b
Hexadecimal	A 0x prefix followed by a hexadecimal digits (0-9, A-F, a-f). Or use a h suffix	0x12FF 0x45 0fa10h
Decimal, integer	Decimal digits (0-9).	12 1245

Table 4-2: Numeric constants

4.6.2 Strings

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

Strings constants in expressions are evaluated to a number (each character is replaced by its ASCII value). Strings in expressions can have a size of up to 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 `.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

```
'ABCD'           ; (0x41424344)
'''79'           ; to enclose a quote double it
"A\"BC"          ; or to enclose a quote escape it
'AB'+1           ; (0x4143) string used in expression
''               ; null string
```

4.6.3 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.

Type	Operator	Name	Description
	()	parenthesis	Expressions enclosed by parenthesis are evaluated first.
Unary	+	plus	Returns the value of its operand.
	-	minus	Returns the negative of its operand.
	~	complement	Returns complement, integer only
	!	logical negate	Returns 1 if the operands' value is 0; otherwise 0. For example, if <code>buf</code> is 0 then <code>!buf</code> is 1.
Arithmetic	*	multiplication	Yields the product of two operands.
	/	division	Yields the quotient of the division of the first operand by the second. With integers, the divide operation produces a truncated integer.
	%	modulo	Integer only: yields the remainder from a division of the first operand by the second.
	+ -	addition subtraction	Yields the sum of its operands. Yields the difference of its operands.
Shift	<<	shift left	Integer only: shifts the left operand to the left (zero-filled) by the number of bits specified by the right operand.
	>>	shift right	Integer only: shifts the left operand to the right (sign bit extended) by the number of bits specified by the right operand.
Relational	<	less than	Returns: an integer 1 if the indicated condition is TRUE. an integer 0 if the indicated condition is FALSE.
	<=	less or equal	
	>	greater than	
	>=	greater or equal	
	==	equal	
	!=	not equal	
Bitwise	&	AND	Integer only: yields bitwise AND
		OR	Integer only: yields bitwise OR
	^	exclusive OR	Integer only: yields bitwise exclusive OR
Logical	&&	logical AND	Returns an integer 1 if both operands are non-zero; otherwise, it returns an integer 0.
		logical OR	Returns an integer 1 if either of the operands is non-zero; otherwise, it returns an integer 1

Table 4-3: Assembly expression operators

4.7 Built-in Assembly Functions

The TASKING assemblers have 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 '@' 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.

4.7.1 Overview of Built-in Assembly Functions

The following table provides an overview of all built-in assembly functions. Next all functions are described into more detail. *expr* can be any assembly expression resulting in an integer value. Expressions are explained in section 4.6, [Assembly Expressions](#).

Overview of assembly functions

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

4.7.2 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 ADRL. 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 ADRL.

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:

```
.IF @ARG('TWIDDLE') ;is argument twiddle present?
.IF @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('processor_type')

With the @CPU function you can check whether the source code is being assembled for a certain processor type. The function evaluates to TRUE when the specified *processor_type* matches the processor type that was specified with the option **--cpu=cpu**.

This function is useful to create conditional code for several targets as shown in the example.

Example:

```
.IF @CPU('ARMv4')      ; true if you specified option --cpu=ARMv4
... ; code for the ARMv4
.ELIF @CPU('ARMv6M')   ; true if you specified option --cpu=ARMv6M
... ; code for the ARMv6-M
.ELSE
... ; code for other architectures
.ENDIF
```



C compiler option **--cpu** (Select architecture) in section 6.1, *C Compiler Options*, of Chapter *Tool Options*.

@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?
.ELSE
.ELSEIF @DEFINED(ANGLE)    ;does label ANGLE exist?
.ELSE

```

@LSB(expression)

Returns the *least* significant byte of the result of the *expression*.

The result of the expression is calculated as 16 bits.

@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).

@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 (bits 32..63) 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 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
```

@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 option **--thumb (-T)**. So, it does not depend on the **.CODE16**, **.CODE32**, **.ARM** or **.THUMB** directive.

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

4.8 Assembler Directives

An assembler directive is simply a message to the assembler. Assembler directives are not translated into machine instructions, but can produce data. There are three types 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 directives
- Data definition / Storage allocation directives
- 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. 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.

4.8.1 Overview of Assembler Directives

The following tables provide an overview of all assembler directives. For a detailed description of these directives, refer to section 4.8.2, [Detailed Description of Assembler Directives](#).

Overview of assembly control directives

Directive	Description
<code>.END</code>	Indicates the end of an assembly module
<code>.INCLUDE</code>	Include file
<code>.MESSAGE</code>	Programmer generated message

Overview of symbol definition directives

Directive	Description
<code>.EQU</code>	Set permanent value to a symbol
<code>.EXTERN</code>	Import global section symbol
<code>.GLOBAL</code>	Declare global section symbol
<code>.SECTION/.ENDSEC</code>	Start a new section
<code>.SET</code>	Set temporary value to a symbol
<code>.SIZE</code>	Set size of symbol in the ELF symbol table
<code>.SOURCE</code>	Specify name of original C source file
<code>.TYPE</code>	Set symbol type in the ELF symbol table
<code>.WEAK</code>	Mark a symbol as 'weak'

Overview of data definition / storage allocation directives

Directive	Description
<code>.ALIGN</code>	Align location counter
<code>.BS / .BSB / .BSH / .BSW / .BSD</code>	Define block storage (initialized)
<code>.DB</code>	Define byte
<code>.DH</code>	Define half word
<code>.DW</code>	Define word
<code>.DD</code>	Define double-word
<code>.DS / .DSB / .DSH / .DSW / .DSD</code>	Define storage
<code>.FLOAT / .DOUBLE</code>	Define a 32-bit / 64-bit floating-point constant
<code>.OFFSET</code>	Move location counter forwards

Overview of macro and conditional assembly directives

Directive	Description
<code>.DEFINE</code>	Define substitution string
<code>.BREAK</code>	Break out of current macro expansion
<code>.REPEAT / .ENDREP</code>	Repeat sequence of source lines
<code>.FOR / .ENDFOR</code>	Repeat sequence of source lines <i>n</i> times
<code>.IF / .ELIF / .ELSE</code>	Conditional assembly directive
<code>.ENDIF</code>	End of conditional assembly directive
<code>.MACRO / .ENDM</code>	Define macro
<code>.UNDEF</code>	Undefine <code>.DEFINE</code> symbol or macro

Overview of listing control assembly directives

Directive	Description
<code>.LIST / .NOLIST</code>	Print / do not print source lines to list file
<code>.PAGE</code>	Set top of page/size of page
<code>.TITLE</code>	Set program title in header of assembly list file

Overview of HLL directives

Directive	Description
<code>.CALLS</code>	Pass call tree information

ARM specific directives

Directive	Description
<code>.CODE16 / .CODE32</code>	Treat instructions as Thumb or ARM instructions using pre-UAL syntax
<code>.THUMB / .ARM</code>	Treat instructions as Thumb or ARM instructions using UAL syntax
<code>.LTORG</code>	Assemble current literal pool immediately

4.8.2 Detailed Description of Assembler Directives

.ALIGN

Syntax

.ALIGN *expression*

Description

With the `.ALIGN` directive you tell the assembler to align the location counter.

When the assembler encounters the `.ALIGN` directive, it moves the location counter forwards 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.

Examples

```
.SECTION .text
.ALIGN 16      ; the assembler aligns
instruction    ; this instruction at 16 MAUs and
               ; fills the 'gap' with NOP instructions.

.SECTION .text
.ALIGN 12      ; WRONG: not a power of two, the
instruction    ; assembler aligns this instruction at
               ; 16 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.

Example

```
.FOR MYVAR IN 10 TO 20
...    ;
...    ; assembly source lines
...    ;
.IF MYVAR > 15
    .BREAK
.ENDIF
.ENDREP
```


.BS/.BSB/.BSH/.BSW/.BSD

Syntax

`[label] .BS expression1[,expression2]`

`[label] .BSB expression1[,expression2]`

`[label] .BSH expression1[,expression2]`

`[label] .BSW expression1[,expression2]`

`[label] .BSD expression1[,expression2]`

Description

With the `.BS` directive (Block Storage) the assembler reserves a block of memory. The reserved block of memory is initialized to the value of `expression2`, or zero if omitted.

With `expression1` 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 `expression2` you can specify a value to initialize the block with. Only the least significant MAU of `expression2` is used. If you omit `expression2`, the default is zero.

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



You cannot initialize 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:

.BSB The `expression1` argument specifies the number of bytes to reserve.

.BSH The `expression1` argument specifies the number of half words to reserve (one half word is 16 bits).

.BSW The `expression1` argument specifies the number of words to reserve (one word is 32 bits).

.BSD The `expression1` argument specifies the number of double-words to reserve (one double-word is 64 bits).

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



.DS (Define Storage)

.CALLS

Syntax

```
.CALLS 'caller', 'callee'
```

or

```
.CALLS 'caller', "", stack0_usage[, stack1_usage]...
```

Description

The first syntax creates a call graph reference between *caller* and *callee*. The linker needs this information to build a call graph. If applicable the call graph is used to create a static stack overlay. *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 values specified are 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.

Normally `.CALLS` directives are automatically generated by the 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.

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

.THUMB

.CODE32

.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 assembler issues an error message if **.CODE16** is used in combination with option **--cpu=ARMv4**.

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) in Section 6.3, *Assembler Options*, of Chapter *Tool Options*.

.DB

Syntax

`[label] .DB argument[,argument]...`

Description

With the `.DB` directive (Define Byte) the assembler allocates and initializes one byte of memory for each *argument*.

An *argument* can be:

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

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

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

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

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

```
.DB 'R'           ; = 0x52
.DB 'AB',,, 'D'   ; = 0x41420043 (second argument is empty)
```

Example

```
TABLE:  .DB 14,253,0x62,'ABCD'
CHARS:  .DB 'A','B',,, 'C','D'
```

Related information



- .BS** (Block Storage)
- .DS** (Define Storage)
- .DH** (Define Half Word)
- .DW** (Define Word)
- .DD** (Define Double-Word)

.DD

Syntax

`[label] .DD argument[,argument]...`

Description

With the `.DD` directive (Define Double-Word) you allocate and initialize one double-word of memory for each *argument*.

One double-word is 64 bits.

An *argument* is:

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

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

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

Double-word arguments are stored as is. Floating-point values are not allowed. If the evaluated argument is too large to be represented in a double-word, the assembler issues a warning and truncates the value.

In case of character strings, each ASCII value of the character is stored in successive locations starting at the most significant byte of a double-word:

```
.DD 'AB',, 'D' => 0x4241000000000000
                  0x0000000000000000 (second argument is empty)
                  0x4400000000000000
```

Example

```
TABLE: .DD 14,253,0x62,'ABCD'
CHARS: .DD 'A','B',, 'C','D'
```

Related information



.BS (Block Storage)
.DS (Define Storage)
.DB (Define Byte)
.DH (Define Half Word)
.DW (Define Word)

.DEFINE

Syntax

```
.DEFINE symbol string
```

Description

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

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

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

Example

Suppose you defined the symbol `LEN` with the substitution string `"32"`:

```
.DEFINE LEN "32"
```

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

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

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

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

Related information



.UNDEF (Undefine a `.DEFINE` symbol or macro)

.MACRO/.ENDM (Define a macro)

.DH

Syntax

`[label] .DH argument[,argument]...`

Description

With the `.DH` directive (Define Half Word) you allocate and initialize a half word of memory for each *argument*.

A half word is 16 bits.

An *argument* is:

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

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

Multiple arguments are stored in successive half word address locations. If an argument is NULL, its corresponding address location is filled with zeros.

Half word arguments are stored as is. Floating-point values are not allowed.

If the evaluated argument is too large to be represented in a half word, the assembler issues a warning and truncates the value.

In case of single and multiple character strings, each ASCII value of the character is stored in successive locations starting at the most significant byte of a half word. The standard C escape sequences are allowed:

```
.DH 'AB',, 'D' => 0x4241
                  0x0000 (second argument is empty)
                  0x4400
```

Example

```
TABLE: .DH 14,253,0x62,'ABCD'
CHARS: .DH 'A','B',, 'C','D'
```

Related information



.BS (Block Storage)
.DS (Define Storage)
.DB (Define Byte)
.DW (Define Word)
.DD (Define Double-Word)

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

With the *expression* you specify the number of minimum addressable units (MAUs) that you want to reserve. The expression must evaluate to an integer larger than zero and cannot contain references to symbols that are not yet defined in the assembly source.

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



You cannot use the `.DS` directive in sections with attribute `init`. If you need to reserve *initialized* space in an `init` section, use the `.BS` directive instead.

The `.DSB`, `.DSH`, `.DSW` and `.DSD` directives are variants of the `.DS` directive:

.DSB The *expression* argument specifies the number of bytes to reserve.

.DSH The *expression* argument specifies the number of half words to reserve (one half word is 16 bits).

.DSW The *expression* argument specifies the number of words to reserve (one word is 32 bits).

.DSD The *expression* argument specifies the number of double-words to reserve (one double-word is 64 bits).

Example

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

Related information



.BS (Block Storage)

.DB (Define Byte)

.DH (Define Half Word)

.DW (Define Word)

.DD (Define Double-Word)

.DW

Syntax

`[label] .DW argument[,argument]...`

Description

With the `.DW` directive (Define Word) you allocate and initialize one word of memory for each *argument*.

One word is 32 bits.

An *argument* is:

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

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

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

Word arguments are stored as is. Floating-point values are not allowed. If the evaluated argument is too large to be represented in a word, the assembler issues a warning and truncates the value.

In case of single and multiple character strings, each ASCII value of the character is stored in successive locations starting at the most significant byte of a word. The standard C escape sequences are allowed:

```
.DW 'AB',, 'D' => 0x42410000
                  0x00000000 (second argument is empty)
                  0x44000000
```

Example

```
TABLE: .DW 14,253,0x62,'ABCD'
CHARS: .DW 'A','B',, 'C','D'
```

Related information



- .BS** (Block Storage)
- .DS** (Define Storage)
- .DB** (Define Byte)
- .DH** (Define Half Word)
- .DD** (Define Double-Word)

.END

Syntax

.END

Description

With the `.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.

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. Once defined, you cannot redefine the *symbol*. With the **.GLOBAL** directive you can define the symbol global.

Example

To assign the value 0x4000 permanently to the symbol `MYSYMBOL`:

```
MYSYMBOL .EQU 0x4000
```

You cannot redefine the used symbols.

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 symbol is referenced in the current module while it is defined outside the current module.

You must define the symbols either outside any module or declare it 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.

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. 12.457 and +0.27E-13 are legal floating-point constants.

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

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

Example

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

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

Related information



.DS (Define Storage)

.FOR/.ENDFOR

Syntax

```
[label] .FOR var IN expression[,expression]...
```

```
....  
.ENDFOR
```

or:

```
[label] .FOR var IN start TO end [STEP step]
```

```
....  
.ENDFOR
```

Description

With the `.FOR` / `.ENDFOR` directive you can repeat a sequence 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 loop 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 loop is repeated with each occurrence of the symbol `var` removed.
2. In the second method, the loop 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 loop 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.

```
.FOR VAR1 IN 1,2+3,4,12  
    .DB VAR1  
    .DW (VAR1*VAR1)  
.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.

```
.FOR VAR2 IN 1 to 0x10  
    .DW (VAR1*VAR1)  
.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 **-ig**.

With the `.GLOBAL` directive you declare one or 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.

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

```
.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 integer and cannot contain forward references. If *expression* evaluates to zero, the IF-condition is considered FALSE, any non-zero result of *expression* is considered as TRUE.

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

Example

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

```
.IF    TEST
... ; code for the test version
.ELIF DEMO
... ; code for the demo version
.ELSE
... ; code for the final version
.ENDIF
```

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

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

You can also define the symbols in Altium Designer as preprocessor macros in dialog **Project » Project Options » Assembler » Preprocessing** (assembler option `--define`).

Related information



Assembler option `--define` (Define preprocessor macro) in Section 6.3, *Assembler Options*, of Chapter *Tool Options*.

.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 (-I)**.
3. The path that is specified in the environment variable `A$target\INC` when the product was installed.
4. The default directory `...\ctarget\include`.

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** \(Add directory to include file search path\)](#) in Section 6.3, *Assembler Options*, of Chapter *Tool Options*.

.LIST/.NOLIST

Syntax

```
.NOLIST
.  
. ; assembly source lines  
.  
.LIST
```

Description

If you generate a list file (see assembler option **--list-file**), you can use the `.LIST` and `.NOLIST` directives to specify which source lines the assembler must write to the list file.

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

It is possible to nest the `.LIST`/`.NOLIST` directives.

Example

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

```
.SECTION .text  
... ; source line 1  
.NOLIST  
... ; source line 2  
.LIST  
... ; source line 3  
.ENDSEC  
.END
```

The assembler generates a list file with the following lines:

```
.SECTION .text  
... ; source line 1  
.LIST  
... ; source line 3  
.ENDSEC  
.END
```

Related information



[Assembler option **--list-file** \(Generate list file\)](#) in Section 6.3, *Assembler Options*, of Chapter *Tool Options*.

.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](#)
[LDR= Thumb generic](#)

.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.
- *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:

Operator	Name	Description
\	Macro argument concatenation	Concatenates a macro argument with adjacent alphanumeric characters.
?	Return decimal value of symbol	Substitutes the <code>?symbol</code> sequence with a character string that represents the decimal value of the symbol.
%	Return hex value of symbol	Substitutes the <code>%symbol</code> sequence with a character string that represents the hexadecimal value of the symbol.
"	Macro string delimiter	Allows the use of macro arguments as literal strings.
^	Macro local label override	Prevents name mangling on labels in macros.

Example

The macro definition:

```
macro_a .MACRO arg1,arg2           ;header
      .db arg1                     ;body
      .dw (arg1*arg2)
      .ENDM                        ;terminator
```

The macro call:

```
.section .data
macro_a 2,3

.endsec
```

The macro expands as follows:

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

Related information



.DEFINE (Define a substitution string)
Section 4.9, *Macro Operations*.

.MESSAGE

Syntax

.MESSAGE *type* [{*str|exp|symbol*}[, {*str|exp|symbol*}]...]

Description

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

With *type* you can specify the following types of messages:

I Information message. Error and warning counts are not affected and the assembler continues the assembling process.

W Warning message. Increments the warning count and the assembler continues the assembling process.

E Error message. Increments the error count and the assembler continues the assembling process.

F Fatal error message. The assembler immediately aborts the assembling process and generates no object file or list file.

The `.MESSAGE` directive is for example useful in combination with conditional assembly to indicate which part is assembled.

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
```

.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.

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
```

.PAGE

Syntax

.PAGE [*width,length,blanktop,blankbtm,blankleft*]

Description

If you generate a list file (see assembler option **--list-file**), you can use the **.PAGE** directive to format the generated list file.

width Number of characters on a line (1–255). Default is 132.

length Number of lines per page (10–255). Default is 66.

blanktop Number of blank lines at the top of the page. Default = 0.
Specify a value so that $blanktop + blankbtm \leq length - 10$.

blankbtm Number of blank lines at the bottom of the page. Default = 0.
Specify a value so that $blanktop + blankbtm \leq length - 10$.

blankleft Number of blank columns at the left of the page. Default = 0. Specify a value smaller than *width*.

If you use the **.PAGE** directive without arguments, it causes a 'formfeed': the next source line is printed on the next page in the list file.

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

A label is not allowed with this directive.

Example

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

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

.PAGE ,,5  ; insert five blank lines at the top. Note
           ; that you can omit the last two arguments.
```

Related information



.TITLE (Set program title in header of assembler list file)

Assembler option **--list-file** (Generate list file) in Section 6.3, *Assembler Options*, of Chapter *Tool Options*.

.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 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

Syntax

```
.SECTION name [,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:

Section Name	Description	Section Type
.text	Code sections	code
.data	Initialized data	data
.bss	Uninitialized data (cleared)	data
.rodata	ROM data (constants)	data
.debug	Debug sections	debug

Table 4-4: Predefined section names

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 `\Program Files\Altium Designer\System\Tasking\include.lsl`.

You can specify the following section attributes:

Example

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

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

.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 a 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
        .code32
        .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 "test.c"
```

Related information



.TITLE

Syntax

.TITLE [*title*]

Description

If you generate a list file (see 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.

Example

```
.TITLE "The best program"
```

In the header of each page in the assembler list file, the title of the program is printed. In this case: `The best program`

Related information



.PAGE (Format the assembler list file)

Assembler option **--list-file** (Generate list file) in Section 6.3, *Assembler Options*, of Chapter *Tool Options*.

.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.

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

Example

```
.UNDEF LEN
```

Undefines the **LEN** substitution string that was previously defined with the **.DEFINE** directive.

Related information



.DEFINE (Define substitution string)

.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)

4.9 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.

4.9.1 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.
- *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:

```
macro_name .MACRO [arg[,arg]...] [; comment]
.
    source statements
.
.ENDM
```

If the macro name is the same as an existing assembler directive or mnemonic opcode, the assembler replaces the directive or mnemonic opcode with the macro and issues a warning.

The arguments are symbolic names that the macro preprocessor replaces with the literal arguments when the macro is expanded (called). Each argument must follow the same rules as global symbol names. Argument names cannot start with a percent sign (%).

Example

Consider the following macro definition:

```
RESERV .MACRO val ; reserve space
    .DS val
.ENDM
```

After the following macro call:

```
.section .text
RESERV 8
.endsec
```

The macro expands to:

```
.DS 8
```

4.9.2 Calling a Macro

To invoke a macro, construct a source statement with the following format:

```
[label] macro_name [arg[,arg...]] [; 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.

arg 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.

4.9.3 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.

Operator	Name	Description
\	Macro argument concatenation	Concatenates a macro argument with adjacent alphanumeric characters.
?	Return decimal value of symbol	Substitutes the <i>?symbol</i> sequence with a character string that represents the decimal value of the symbol.
%	Return hex value of symbol	Substitutes the <i>%symbol</i> sequence with a character string that represents the hexadecimal value of the symbol.
"	Macro string delimiter	Allows the use of macro arguments as literal strings.
^	Macro local label override	Prevents name mangling on labels in macros.

**Example: Argument Concatenation Operator – **

Consider the following macro definition:

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

The macro is called as follows:

```
MAC_A 2,1
```

The macro expands as follows:

```
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:

```
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.

```
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:

```
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 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
```

4.9.4 Using the .FOR and .REPEAT Directives as Macros

The `.FOR` and `.REPEAT` directives are specialized macro forms to repeat a block of source statements. You can think of them as a simultaneous definition and call of an unnamed macro. The source statements between the `.FOR` and `.ENDFOR` directives and `.REPEAT` and `.ENDREP` directives follow the same rules as macro definitions.



For a detailed description of these directives, see section 4.8, [Assembler Directives](#).

4.9.5 Conditional Assembly

With the conditional assembly directives you can instruct the macro preprocessor to use a part of the code that matches a certain condition.

You can specify assembly conditions with arguments in the case of macros, or through definition of symbols via the `.DEFINE`, `.SET`, and `.EQU` directives.

The built-in functions of the assembler provide a versatile means of testing many conditions of the assembly environment.

You can use conditional directives also within a macro definition to check at expansion time if arguments fall within a range of allowable values. In this way macros become self-checking and can generate error messages to any desired level of detail.

The conditional assembly directive `.IF/.ENDIF` has the following form:

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

A section of a program that is to be conditionally assembled must be bounded by an `.IF`–`.ENDIF` directive pair. 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.

4.10 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).

4.10.1 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.

Instruction	Replacement
<i>ADRcond Rd,label</i>	<i>ADRcond Rd, PC, @ALUPCREL(label,0,1)</i>
<i>ADRLcond Rd,label</i>	<i>ADRcond Rd, PC, @ALUPCREL(label,0,0)</i> <i>ADRcond Rd, Rd, @ALUPCREL(label,1,1)</i>
<i>ADRLlcond Rd,label</i>	<i>ADRcond Rd, PC, @ALUPCREL(label,0,0)</i> <i>ADRcond Rd, Rd, @ALUPCREL(label,1,0)</i> <i>ADRcond Rd, Rd, @ALUPCREL(label,2,1)</i>

BX for ARMv4

The ARMv4 architecture does not support the BX instruction in hardware. If the option `--cpu=ARMv4` or `--cpu=ARMv4T` was specified, the assembler will emit a relocation type at the location of the BX instruction. The linker will replace the BX instruction by a MOV instruction if the option `--cpu=ARMv4` was specified.

Instruction	Replacement (by linker)	Description
<i>BXcond Rm</i>	<i>MOVcond PC,Rm</i>	Only if architecture is ARMv4

4.10.2 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.

Instruction	Replacement	Description
<i>LDRcond Rd,=expr</i>	<i>MOVcond Rd, #expr</i>	If <i>expr</i> can be encoded
	<i>MVNcond Rd, #@LSW(~(expr))</i>	If <i>~expr</i> can be encoded
	<i>LDRcond Rd,ltpool</i> ;; code <i>ltpool:</i> .DW <i>expr</i>	If <i>expr</i> is external or PC-relative, or cannot be encoded

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 `.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.

Instruction	Replacement
VLD $Rcond$ $Sd,=expr$	VLD $Rcond$ $Sd,ltpool$;; code $ltpool$: .FLOAT $expr$
VLD $Rcond$ $Dd,=expr$	VLD $Rcond$ $Dd,ltpool$;; code $ltpool$: .DOUBLE $expr$

MOV32 ARM and Thumb-2 generic

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

Instruction	Replacement	Description
MOV32 $cond$ $Rd,=expr$	MOVW $cond$ $Rd, \#@LSH(expr)$ MOVT $cond$ $Rd, \#@MSH(expr)$	If $expr$ is internal and absolute
	MOVW $cond$ $Rd, \#expr$ MOVT $cond$ $Rd, \#expr$	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 ($\sim value$) 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.

Instruction	Replacement (if #~imm or #~imm can be encoded)
ADD $cond$ $Rd,Rn,\#imm32$	SUB $cond$ $Rd,Rn,\#-(imm32)$
ADD $condS$ $Rd,Rn,\#imm32$	SUB $condS$ $Rd,Rn,\#-(imm32)$
ADDW $cond$ $Rd,Rn,\#imm12$	SUBW $cond$ $Rd,Rn,\#-(imm12)$
SUB $cond$ $Rd,Rn,\#imm32$	ADD $cond$ $Rd,Rn,\#-(imm32)$
SUB $condS$ $Rd,Rn,\#imm32$	ADD $condS$ $Rd,Rn,\#-(imm32)$
SUBW $cond$ $Rd,Rn,\#imm12$	ADDW $cond$ $Rd,Rn,\#-(imm12)$
ADC $cond$ $Rd,Rn,\#imm32$	SBC $cond$ $Rd,Rn,\#-(imm32)$
ADC $condS$ $Rd,Rn,\#imm32$	SBC $condS$ $Rd,Rn,\#-(imm32)$
SBC $cond$ $Rd,Rn,\#imm32$	ADC $cond$ $Rd,Rn,\#-(imm32)$
SBC $condS$ $Rd,Rn,\#imm32$	ADC $condS$ $Rd,Rn,\#-(imm32)$
AND $cond$ $Rd,Rn,\#imm32$	BIC $cond$ $Rd,Rn,\#@LSW(\sim(imm32))$
AND $condS$ $Rd,Rn,\#imm32$	BIC $condS$ $Rd,Rn,\#@LSW(\sim(imm32))$
BIC $cond$ $Rd,Rn,\#imm32$	AND $cond$ $Rd,Rn,\#@LSW(\sim(imm32))$
BIC $condS$ $Rd,Rn,\#imm32$	AND $condS$ $Rd,Rn,\#@LSW(\sim(imm32))$
CMN $cond$ $Rn,\#imm32$	CMP $cond$ $Rn,\#-(imm32)$
CMP $cond$ $Rn,\#imm32$	CMN $cond$ $Rn,\#-(imm32)$
MOV $cond$ $Rd,\#imm32$	MVN $cond$ $Rd,\#@LSW(\sim(imm32))$
MOV $condS$ $Rd,\#imm32$	MVN $condS$ $Rd,\#@LSW(\sim(imm32))$
MVN $cond$ $Rd,\#imm32$	MOV $cond$ $Rd,\#@LSW(\sim(imm32))$
MVN $condS$ $Rd,\#imm32$	MOV $condS$ $Rd,\#@LSW(\sim(imm32))$

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.

4.10.3 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.

Instruction	Replacement	Description
LDR <i>Rd</i> ,= <i>expr</i>	MOV <i>Rd</i> , # <i>expr</i>	If <i>expr</i> is in range
	LDR <i>Rd</i> , <i>ltpool</i> ;; code <i>ltpool</i> : .DW <i>expr</i>	If <i>expr</i> is external or PC-relative, or not in range

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.

Instruction	Replacement	Description
Bcond <i>label</i>	Binv_cond ~1 B <i>label</i> ~1:	If target <i>label</i> out-of-range

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.

Instruction	Replacement
ADD <i>Rd</i> , <i>Rn</i> ,# <i>imm</i>	SUB <i>Rd</i> , <i>Rn</i> ,#−(<i>imm</i>)
ADD <i>Rd</i> ,# <i>imm</i>	SUB <i>Rd</i> ,#−(<i>imm</i>)
ADD SP,# <i>imm</i>	SUB SP,#−(<i>imm</i>)
SUB <i>Rd</i> , <i>Rn</i> ,# <i>imm</i>	ADD <i>Rd</i> , <i>Rn</i> ,#−(<i>imm</i>)
SUB <i>Rd</i> ,# <i>imm</i>	ADD <i>Rd</i> ,#−(<i>imm</i>)
SUB SP,# <i>imm</i>	ADD SP,#−(<i>imm</i>)



5 Run-time Environment

Summary

This chapter describes the startup code used by the TASKING ARM C Compiler, the vector table, the stack layout and the heap.

5.1 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 closedown 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.asm` (ARM and Thumb), or `cstart_thumb2.asm` (Thumb2 specific) in the directory `lib\src`. This 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 Altium Designer or on the command line with linker option `--define`. If the default run-time startup code does not match your configuration, you need to modify the startup code accordingly.

The entry point of the startup code (reset handler) is label `_START`. 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:

1. Load the 'real' program address. This assures that the reset handler is immune for any ROM/RAM re-mapping.
2. Initialize the stack pointers for each processor mode. The stack pointers are loaded in memory by the stack address located at a linker generate symbol (for example `_lc_ub_stack`). These symbols are defined in the linker script file. See section 5.3, [Stack and Heap](#), for detailed information on the stack.
3. Call a user function which initializes hardware. The startup code calls the function `__init_hardware`. This function has an empty implementation in the run-time library, which you should change if certain hardware initializations, such as ROM/RAM re-mapping or MMU configuration, are required before calling the main application.
4. 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.
5. Initialize or copy the vector table. The startup code calls the function `__init_vector_table`. This function has a default implementation in the run-time library, which copies the vector table from ROM to RAM if necessary. You should only change it in very specific situations. For example, in case position dependent vectors are used (B instructions instead of LDR PC) and the vector table must be generated in RAM (or copied from ROM to RAM with patched offsets in the B instructions).
6. 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 `main`.
7. Load register r10 with the end of the user/system stack. This is needed in case stack overflow checking is enabled.
8. Switch to Thumb code if you specified [compiler option --thumb](#).
9. Initialize profiling if profiling is enabled. For an extensive description of profiling refer to Chapter 4, [Profiling](#), in the user's manual.
10. Initialize the `argc` and `argv` arguments to zero.
11. Call the entry point of your application with a call to function `main()`.

Exit code

When the C application 'returns', which is not likely to happen in an embedded environment, the program ends with an endless loop, at the assembly label `_exit`. When you use a debugger, it can be useful to set a breakpoint on this label to indicate that the program has reached the end, or that the library function `exit()` has been called.

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 linker command line option `--define` with the following syntax:

```
--define=symbol[=value]
```

In the startup file (`cstart.asm` and `cstart_thumb2.asm`) the following macro preprocessor symbols are used:

Define	Description
<code>__PROF_ENABLE__</code>	If defined, initialize profiling
<code>__POSIX__</code>	If defined, call <code>posix_main</code> instead of <code>main</code>

Table 5-1: Defines used in `cstart.asm`

The following table shows the linker labels and other labels used in the startup code.

Define	Description
<code>_START</code>	Start label, mentioned in LSL file (<code>arm_arch.lsl</code>)
<code>_Next</code>	Real program address
<code>main</code>	Start label user C program
<code>exit</code>	Start label of <code>exit()</code> function
<code>_exit</code>	<code>exit()</code> function returns to this place
<code>_lc_ub_stack</code>	User/system mode stack pointer
<code>_lc_ue_stack</code>	End of stack symbol, required by debugger
<code>_lc_ub_stack_und</code>	Undefined mode stack pointer (*)
<code>_lc_ub_stack_svc</code>	Supervisor mode stack pointer (*)
<code>_lc_ub_stack_abt</code>	Abort mode stack pointer (*)
<code>_lc_ub_stack_irq</code>	IRQ mode stack pointer (*)
<code>_lc_ub_stack_fiq</code>	FIQ mode stack pointer (*)
<code>_lc_ub_table</code>	ROM to RAM copy table
<code>__APPLICATION_MODE__</code>	Contains the processor mode, and the IRQ/FIQ interrupts mode (*)
<code>__init_hardware</code>	Start label of hardware initialization routine
<code>__init_vector_table</code>	Start label of vector table initialization

Table 5-2: Labels used in `cstart.asm` and `cstart_thumb2.asm`

(*) The labels marked with a * are available in `cstart.asm` only.

5.2 Reset Handler and Vector Table

Reset handler

As explained in the previous section the entry point of the startup code (reset handler) is label `_START`. 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 to specify a fixed address. In this case the linker determines the address and patches the vector table. There are however situations where you have to specify a fixed ROM address:

- If `_START` 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 re-maps this address during startup. Note that the reset handler in the run-time library is immune to this re-mapping because the first instruction in the startup code sets the program counter to the actual ROM address.
- When the reset handler is called from the vector table with a branch instruction (`B _START`) and the linker has located the reset handler at an address that is out-of-range of the branch instruction. When you specify a fixed ROM address you can make sure that the reset handler can be called from the vector table. Note however that you can prevent out-of-range branches by using a position independent vector table, which loads the handler addresses into the program counter by means of a PC-relative load from a literal pool.

To set a fixed ROM address in Altium Designer:

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **Processor** entry and select **Startup code / Vector table**.
3. Enter an address in the **Force reset handler at ROM address** field.

Vector table

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:

- In the dialog as described above, enter an address in the **Vector table ROM address** field.
For M-profile architectures also specify the **Number of vectors**.

The vector table size for M-profile architectures is calculated as 4 times the number of vectors. The default number of vectors is 16, but you can specify another value.

The linker can generate a vector table for you:

- In the **Startup code / Vector table** page, enable the option **Generate vector table in ROM**.

You can ask the linker to reserve space in RAM memory for a copy of the vector table at run-time at a certain address in memory. Typically this would be the address which will be the mapping of address `0x00000000` after ROM/RAM re-mapping. If you reserve space for a copy you can also let the startup code copy the table automatically from ROM to RAM, but only if position independent vectors are used.

- In the **Startup code / Vector table** page, enable the option **Reserve space for copy in RAM** and fill in the **RAM address**.
Optionally enable the option **Copy vector table to RAM**.

Refer to the run-time library implementation of the `__init_vector_table` routine in `lib\src\initvectortable.asm` or `initvectortable_thumb.asm` for more information.

Vector table versions (all architectures except M-profile)

You can select between two versions of the vector table: position dependent or position independent.

The *position dependent* table contains branch instructions to the handlers. The handlers must be located in-range of the branch instructions. The size of the table is 32 bytes.

The *position independent* table contains PC-relative load instructions of the PC. The handler addresses are in a literal pool (data pocket) following the vector table. There are no range restrictions. The size of the table and pool together is 64 bytes. A position independent table is recommended if the table is copied from ROM to RAM at run-time or if the ROM table is re-mapped to address `0x00000000` after startup.

- In the **Startup code / Vector table** page, select the **Vector type: position dependent** or **position independent**.

If you selected a position dependent vector table it is possible to locate the FIQ handler directly at the FIQ vector, since the FIQ vector is the last vector in the table. Doing so saves a branch instruction when servicing a fast interrupt. The generated vector table or the space reserved for the table will be 28 bytes instead of 32. This option is not available for a position independent vector table. Note that you need to use the `__at ()` attribute to specify the actual position of the FIQ handler.

- In the **Startup code / Vector table** page, enable or disable the option **Do not use FIQ vector**.

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:

Function type qualifier	Vector symbol
<code>__interrupt_und</code>	<code>_vector_1</code>
<code>__interrupt_svc</code>	<code>_vector_2</code>
<code>__interrupt_iabt</code>	<code>_vector_3</code>
<code>__interrupt_dabt</code>	<code>_vector_4</code>
<code>__interrupt_irq</code>	<code>_vector_6</code>
<code>__interrupt_fiq</code>	<code>_vector_7</code>

Table 5-3: Function qualifiers and vector symbols

Note that the reset handler is designated by the symbol `_START` instead of `_vector_0`. The fifth vector, with symbol `_vector_5` is reserved. You should use the same vector symbols in hand-coded assembly handlers. You may first want to generate an idle handler in C with the compiler and then use the result as a starting point for your assembly implementation. If the linker does not find the symbol for a handler, it will generate a loop for the corresponding vector, i.e. a jump to itself.

Note that if you have more than one handler for the same exception, for example for different IRQ's or for different run-time phases of your application, and you are using the `__interrupt_type` function qualifier of the compiler, you will need to specify the `__novector` attribute in order to prevent the compiler from generating the `_vector_nr` symbol multiple times, as this would lead to a link error.

Execution mode (all architectures except M-profile)

In Altium Designer you can define the execution mode in which the processor should run when your application's main program is called, together with the interrupt status (FIQ interrupts enabled/disabled, IRQ interrupts enabled/disabled). Based on these settings the linker will generate a symbol (`_APPLICATION_MODE`) which value is used in the startup code in the run-time library to set the value of the CPSR status register before calling your main function.

- In the **Startup code / Vector table** page, select the **Main application execution mode**. Optionally enable or disable the options **Enable IRQ interrupts** and **Enable FIQ interrupts**.

Preprocessor macros in `arm_arch.lsl`

The options you select in Altium Designer result in preprocessor macros that are used in the LSL file. Instead of using Altium Designer you can also define the macros by using the linker option `--define`.

Define	Description
<code>__START</code>	Reset handler ROM address
<code>__PROCESSOR_MODE</code>	Main application execution mode
<code>__IRQ_BIT</code>	Is 0 if IRQ interrupts enabled
<code>__FIQ_BIT</code>	Is 0 if FIQ interrupts enabled
<code>__APPLICATION_MODE</code>	Contains the processor mode, and the IRQ/FIQ interrupts mode
<code>__NO_AUTO_VECTORS</code>	If defined, the vector table will not be generated
<code>__NR_OF_VECTORS</code>	Number of vectors for M-profile architectures (default 16)
<code>__PIC_VECTORS</code>	Defined if position independent vectors are used
<code>__FIQ_HANDLER_INLINE</code>	Defined if you do not generate/reserve the FIQ vector
<code>__VECTOR_TABLE_ROM_ADDR</code>	ROM address of the vector table
<code>__VECTOR_TABLE_RAM_SPACE</code>	Defined if space must be reserved for a copy of the vector table in RAM
<code>__VECTOR_TABLE_RAM_ADDR</code>	RAM address of the copy of the vector table
<code>__VECTOR_TABLE_RAM_COPY</code>	Defined if the linker should copy the vector table to RAM

Table 5-4: Defines used in `arm_arch.lsl`

5.3 Stack and Heap

The stack is used for local automatic variables and function parameters. The following diagram shows the structure of a stack frame.

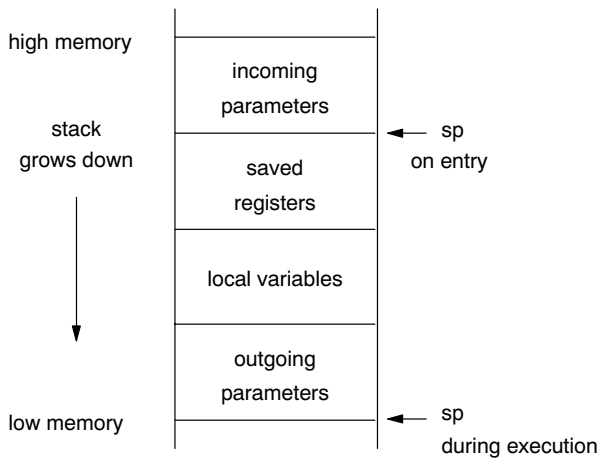


Figure 5-1: Stack diagram

The ARM hardware has separate stack pointers for each processor mode. These stack pointers should be initialized at run-time. This is taken care of by the startup code in the run-time library, by means of linker-generated symbols defined in the LSL file. See section 5.1, [Startup Code](#), for a list of these symbols.

You can define the values of these symbols in Altium Designer as follows.

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **Linker** entry and select **Stack/Heap**.
3. Make your changes.

For the user stack (used in USR mode and in SYS mode) you can specify the size and the location, and you can tell the linker to add unused memory to the stack. You can do the same for the heap. For the other stacks you can only specify their sizes. The linker will determine their locations, and their sizes are fixed. If you want another stack to be the primary stack, for example the supervisor stack, you need to change the LSL file.

The stack size is defined in the linker script file (`arm_arch.lsl` in directory `include.lsl`) with macros:

Define	Description
<code>__STACK</code>	Size of user stack (used in USR mode and in SYS mode)
<code>__STACK_FIQ</code>	FIQ mode stack size (*)
<code>__STACK_IRQ</code>	IRQ mode stack size (*)
<code>__STACK_SVC</code>	Supervisor mode stack size (*)
<code>__STACK_ABT</code>	Abort mode stack size (*)
<code>__STACK_UND</code>	Undefined mode stack size (*)
<code>__STACK_FIXED</code>	Defined if you do not expand the user stack if space is left
<code>__STACKADDR</code>	User stack start address

Table 5-5: Stack macros used in `arm_arch.lsl`

(*) The defines marked with a * are not used for M-profile architectures.

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.

Define	Description
<code>__HEAP</code>	Size of heap
<code>__HEAP_FIXED</code>	Defined if you do not expand the heap if space is left
<code>__HEAPADDR</code>	Heap start address

Table 5-6: Heap macros used in `arm_arch.lsl`



6 Tool Options

Summary

This chapter provides a detailed description of the options for the C compiler, C++ compiler, assembler, linker, control program, make program and the librarian.

6.1 C Compiler Options

Altium Designer uses a makefile to build your entire project. This means that in Altium Designer you cannot run the compiler separately. If you compile a single C source file from within Altium Designer, the file is also assembled. However, you can set options specific for the compiler.

Options in Altium Designer versus options on the command line

Most command line options have an equivalent option in Altium Designer but some options are only available on the command line (for example in a Windows Command Prompt). If there is no equivalent option in Altium Designer, you can specify a command line option in Altium Designer as follows:

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C Compiler** entry and select **Miscellaneous**.
3. Enter one or more command line options in the **Additional C compiler options** field.

Invocation syntax on the command line (Windows Command Prompt)

To call the compiler from the command line, use the following syntax:

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

The input *file* must be a C source file (.c or .ic).

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 double minus (––) characters. You can abbreviate long option names as long as the name is unique. You can mix short and long option names on the command line.

Options can have flags or sub-options. To switch a flag 'on', use a lowercase letter or a *+longflag*. To switch a flag off, use an uppercase letter or a *–longflag*. Separate *longflags* with commas. The 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: `--align-composites`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C Compiler** entry and select **Code Generation**.
3. Select the **Alignment of 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.

Related information



C Compiler: `--call (-m)`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C Compiler** entry and select **Code Generation**.
3. Set the option **Select call mode** to **Use 26-bit 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.

C Compiler: `--cert`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C Compiler** entry and select **CERT C Secure Code Checking**.
3. Make a selection from the **CERT C secure code checking** list.
4. If you select **Custom configuration**, expand the **Custom Configuration** entry and enable one or more individual recommendations/rules.

Command line syntax

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

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 10, [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 10, [CERT C Secure Coding Standard](#)

[C compiler option `--diag`](#) (Explanation of diagnostic messages)

C Compiler: `--check`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C Compiler** entry and select **Miscellaneous**.
3. Add the option `--check` to the **Additional C compiler options** field.

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.

Related information



Assembler option `--check` (Check syntax)

C Compiler: --cpu (-C)

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **Processor** entry and select **Processor Definition**.
3. Select a processor from the **Select processor** box.

Command line syntax

```
--cpu=[ARMv4|ARMv4T|ARMv5T|ARMv5TE|ARMv6M|ARMv7M|XS]
-C[ARMv4|ARMv4T|ARMv5T|ARMv5TE|ARMv6M|ARMv7M|XS]
```

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 compiler uses the appropriate instruction set. You choose one of the following architectures: ARMv4, ARMv4T, ARMv5T, ARMv5TE, ARMv6-M, ARMv7-M or XScale.

Example

To compile the file `test.c` for the ARMv4 processor type, enter the following on the command line:

```
carm --cpu=ARMv4 test.c
```



When you call the compiler from the command line, make sure you specify the same core type to the assembler to avoid conflicts!

Related information



–

C Compiler: `--debug-info (-g)`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C Compiler** entry and select **Debug Information**.
3. Enable the option **Generate symbolic debug information**.
4. Enable or disable the suboptions.

Command line syntax

`--debug-info[=suboption]`

`-g[suboption]`

You can set the following suboptions (when you specify `-g` without suboption, the default is `-gd`):

small	(1 c)	Emit small set of debug information.
default	(2 d)	Emit default symbolic debug information.
all	(3 a)	Emit full symbolic debug information.

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.

When you specify a high optimization level, debug comfort may decrease. Therefore, the compiler issues a warning if the chosen optimizations expect to affect ease of debugging.

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 over-sized assembler/object files.

Full debug information

With this information extra debug information is generated. In extra-ordinary cases you may use this debug information (for instance, if you use your own debugger which makes use of this information). With this suboption, the resulting assembler/object file increases significantly.

Related information



C Compiler: --define (-D)

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C Compiler** entry and select **Preprocessing**.
3. Select **User macro** and click on the down arrow in the right pane to expand macro input.
4. Click on an empty **Macro** field and enter a macro name. (Then click an empty cell to confirm)
5. Optionally, click in the **Value** field and enter a definition. (Then click an empty cell to confirm)

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.

On the command line, you can use the option **--define (-D)** multiple times. If the command line exceeds the length limit of the operating system, you can define the macros in an *option file* which you then must specify to the compiler with the option **--option-file=file (-f)**.

Defining macros with this option (instead of in the C source) is, for example, useful to compile conditional C source as shown in the example below.

Example

Consider the following C program with conditional code to compile a demo program and a real program:

```
void main( void )
{
    #if DEMO == 1
        demo_func();    /* compile for the demo program */
    #else
        real_func();    /* compile for the real program */
    #endif
}
```

You can now use a macro definition to set the DEMO flag:

<u>Macro</u>	<u>Value</u>
DEMO	1 (or empty)

On the command line, use the option as follows:

```
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 specify a macro with arguments. Macro definitions follow exactly the same rules as the **#define** statement in the C language.

<u>Macro</u>	<u>Value</u>
MAX(A,B)	((A) > (B) ? (A) : (B))

On the command line, use the option **-D** as follows:

```
carm -D"MAX(A,B)=((A) > (B) ? (A) : (B))" test.c
```

Note that the macro name and definition are placed between double quotes because otherwise the spaces would indicate a new option.

Related information



C compiler option `--undefine` (Undefine preprocessor macro)

C compiler option `--option-file` (Read options from file)

C Compiler: `--dep-file`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C Compiler** entry and select **Miscellaneous**.
3. Add the option `--dep-file` to the **Additional C compiler options** field.

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 (-Em)`, 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`.

Related information



C compiler option `--preprocess=+make` (Generate dependencies for make)

C Compiler: --diag

Menu entry

1. From the **View** menu, select **Workspace » Panels » System Messages**.

The Message pannel appears.

2. In the **Message** panel, right-click on the message you want more information on.

A popup menu appears.

3. Select **More Info**.

A Message Info box appears with additional information.

Command line syntax

`--diag=[format:]{all|msg[-msg],...}`

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 (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.

With this option the compiler does not compile any files.

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



C compiler option `--cert` (Enable individual CERT checks)

C Compiler: --endianness

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **Processor** entry and select **Processor Definition**.
3. Enable the option **Big-endian mode**.

Command line syntax

`--endianness=endianness`

`-B`

`--big-endian`

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



C Compiler: `--error-file`

Menu entry

Command line only.

Command line syntax

`--error-file[=file]`

Description

With this option the compiler redirects error messages to a file.

If you do not specify a filename, the error file will be named after the input file with extension `.err`.

Example

To write errors to `errors.err` instead of `stderr`, enter:

```
carm --error-file=errors.err test.c
```

Related information



C Compiler: --fpu

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C Compiler** entry and select **Floating-Point**.
3. Select an option from the **Floating-point unit (FPU) support** field.

Command line syntax

`--fpu=fpu`

You can specify the following arguments:

VFPv2	Compile for VFPv2 architecture
VFPv3	Compile for VFPv3 architecture
none	Compile for software FPU library

Description

With this option you define the kind of FPU support with which you create your application.

Related information



C Compiler: `--global-type-checking`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C Compiler** entry and select **Diagnostics**.
3. 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: --help (-?)

Menu entry

Command line only.

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
```

C Compiler: `--include-directory (-I)`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Select **Build Options**.
3. Add a pathname in the **Include files path** field.

If you enter multiple paths, separate them with a semicolon (;).

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 order in which the compiler searches for include files is:

1. The pathname in the C source file and the directory of the C source (only for `#include` files that are enclosed in `"`).
2. The path that is specified with this option.
3. The path that is specified in the environment variable `CARMINC` when the product was installed.
4. 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:

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

You can specify the include directory `myinclude` to the C compiler:

```
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: `--include-file (-H)`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C Compiler** entry and select **Preprocessing**.
3. Enter the name of the file in the **Include this file before source** field or click ... and select a file.

Command line syntax

```
--include-file=file,...  
-Hfile,...
```

Description

With this option (set at project level) you include one extra file at the beginning of each C source file in your project. On a document level (**Project » Document Options**), you can overrule this option with another file or no file at all.

The specified include file is included before all other includes. This is the same as specifying `#include "file"` at the very beginning of (each of) your C source files.

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)

Section 2.4, *How the Compiler Searches Include Files*, in chapter *Using the Compiler* of the user's manual.

C Compiler: `--inline`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C Compiler** entry and select **Optimization**.
3. 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.

Related information



C Compiler: `--inline-max-incr` / `--inline-max-size`

Menu entry

1. From the **Project** menu, select **Project Options...**
The *Project Options dialog box* appears.
2. Expand the **C Compiler** entry and select **Optimization**.
3. Set the option **Maximum code size increase caused by inlining** to a value.
4. Set the option **Maximum size for functions to always inline** to a value.

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 have only effect when you have enabled the inlining optimization (option `-Oi`).



Regardless of the optimization process, the compiler always inlines all functions that have the function qualifier `inline`.

With the option `--inline-max-size` you can specify the maximum size of functions that the compiler inlines as part of the optimization process. The compiler always inlines all functions that are smaller than the specified *threshold*. The threshold is measured in compiler internal units and the compiler uses this measure to decide which functions are small enough to inline. The default threshold is -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 --inline-max-incr=40 --inline-max-size=15 test.c
```

The compiler first inlines all functions with the function qualifier `inline` and all functions that are smaller than the specified threshold of 15. If the code size has still not increased with 40%, the compiler decides which other functions it can inline.

Related information



C compiler option `--optimize=+inline` (Optimization: automatic function inlining)

Section 1.8.3, *Inlining Functions*, in chapter *C Language*.

Section 2.6.3, *Optimize for Size or Speed*, in chapter *Using the Compiler* of the user's manual.

C Compiler: `--interwork`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C Compiler** entry and select **Code Generation**.
3. 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: `--iso` (`-c`)

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C Compiler** entry and select **Language**.
3. Select the ISO C standard **C90** or **C99**.

Command line syntax

```
--iso={90|99}  
-c{90|99}
```

Description

With this option you select the ISO C standard. The compiler checks the C source against this standard and may generate warnings or errors if you use C language that is not defined in the standard.

C90 is also referred to as the "ANSI C standard". C99 refers to the newer ISO/IEC 9899:1999 (E) standard. C99 is the default.

```
carm --iso=90 test.c
```

Related information



C compiler option `--language` (Language extensions)

C Compiler: `--keep-output-files (-k)`

Menu entry

Altium Designer *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.

Related information



C Compiler: --language (-A)

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C Compiler** entry and select **Language**.
3. Enable or disable the following options:
 - **Allow GNU C extensions**
 - **Allow C++ style comments in ISO C90 mode** (only available when **ISO C 90** is selected)
 - **Check assignment of string literal to non-const string pointer**
 - **Allow optimization across volatile access**

Command line syntax

```
--language=[flags]
-A[flags]
```

You can set the following flags:

+/-gcc	(g/G)	Enable a number of gcc extensions
+/-comments	(p/P)	Allow C++ style comments in C source code
+/-volatile	(v/V)	Do not optimize across volatile access
+/-strings	(x/X)	Relaxed const check for string literals

The option **--language (-A)** is the equivalent of **-AGPVX** which disables all language extensions.

The default is **-AGpVx**.

Description

With this option you control the language extensions the compiler accepts. Default the C compiler allows all language extensions.

With **Allow C++ style comments in ISO C90 mode (--language=+comments)**, you tell the compiler to allow C++ style comments (//) in ISO C90 mode (option **-c90**). In ISO C99 mode this style of comments is always accepted.

With **Check assignment of string literal to non-const string pointer (--language=-strings)** you tell the compiler to generate a warning when you assign a constant string to a non-constant string pointer. When you disable this option the following example produces no warning:

```
char *p;
void main( void ) { p = "hello"; }
```

With option **Allow GNU C extensions (--language=+gcc)** you tell the compiler to enable the following gcc languages extensions:

- The identifier `__FUNCTION__` expands to the current function name
- Alternative syntax for variadic macros
- Alternative syntax for designated initializers
- Allow zero sized arrays
- Allow empty struct/union
- Allow empty initializer list
- Allow initialization of static objects by compound literals
- The middle operand of a `? :` operator may be omitted
- Allow a compound statement inside braces as expression
- Allow arithmetic on void pointers and function pointers
- Allow a range of values after a single case label
- Additional preprocessor directive `#warning`
- Allow comma operator, conditional operator and cast as lvalue
- An inline function without "static" or "extern" will be global

- An "extern inline" function will not be compiled on its own
- An `__attribute__` directly following a struct/union definition relates to that tag instead of to the objects in the declaration.

For an exact description of these gcc extensions, please refer to the gcc info pages ([info gcc](#)).

When you disable the option **Allow optimization across volatile access** (with `--language=+volatile`), you tell the compiler to block optimizations when reading or writing a volatile object, by treating the access as a call to an unknown function. This way you can prevent for example that code below the volatile object is optimized away to somewhere above the volatile object.

Example:

```
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
             mov      r0,r3
.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
.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]
           bne     .L2
           str     r1,[r3,#0]
.L2:
           ldr     r0,[r3,#0]
           orr     r0,r0,#32768
           str     r0,[r3,#0]
           mov     r0,#1
           str     r0,[r2,#0]      ; <== Volatile access
           ldr     pc,[sp],#4
           .size   TestFunc,$-TestFunc
           .align  4
.L3:
           .dw     variable
           .dw     access
```

Example

```
carm -AGPx -c90 test.c
carm --language=-gcc,-comments,+strings --iso=90 test.c
```



C compiler option `--iso` (ISO C standard)

C Compiler: `--make-target`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C Compiler** entry and select **Miscellaneous**.
3. Add the option `--make-target` to the **Additional C compiler options** field.

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`.

Related information



C compiler option `--preprocess=+make` (Generate dependencies for make)

C compiler option `--dep-file` (Generate dependencies in a file)

C Compiler: `--mil` / `--mil-split`

Menu entry

Command line only.

Command line syntax

`--mil`

`--mil-split[=file,...]`

Description

With option `--mil` the C compiler skips the code generator phase and writes the optimized intermediate representation (MIL) to a file with the suffix `.mil`. The C compiler accepts `.mil` files as input files on the command line.

Option `--mil-split` does the same as option `--mil`, but in addition, the C compiler splits the MIL representation and writes it to separate files with suffix `.ms`. One file is written for each input file or MIL library specified on the command line. The `.ms` files are only updated on a change. The C compiler accepts `.ms` files as input files on the command line.

With option `--mil-split` you can perform application-wide optimizations during the frontend phase by specifying all modules at once, and still invoke the backend phase one module at a time to reduce the total compilation time.

Optionally, you can specify another filename for the `.ms` file the C compiler generates. Without an argument, the basename of the C source file is used to create the `.ms` filename. Note that if you specify a filename, you have to specify one filename for every input file.

Related information



Control program option `--mil-link` / `--mil-split`

C Compiler: `--misrac`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C Compiler** entry and select **MISRA-C**.
3. In the **MISRA-C checking** box select a MISRA-C configuration.
If you select Custom MISRA-C configuration:
4. In the left pane, expand the **MISRA-C** entry and select **Custom 2004** or **Custom 1998**.
5. Enable or disable the individual rules.

Command line syntax

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

Description

With this option you specify to the compiler which MISRA-C rules must be checked. With the option `--misrac=all` the compiler checks for all supported MISRA-C rules.

Example

```
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



C compiler option `--misrac-advisory-warnings`

C compiler option `--misrac-required-warnings`

Linker option `--misrac-report`

C Compiler: `--misrac-advisory-warnings` / `--misrac-required-warnings`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C Compiler** entry and select **MISRA-C**.
3. Enable one or both options **Turn advisory rule violation into warning** and **Turn required rule violation into warning**.

Command line syntax

`--misrac-advisory-warnings`

`--misrac-required-warnings`

Description

Normally, if an advisory rule or required rule is violated, the compiler generates an error. As a consequence, no output file is generated. With this option, the compiler generates a warning instead of an error.

Related information



C compiler option `--misrac`

Linker option `--misrac-report`

C Compiler: `--misrac-version`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog appears.
2. Expand the **C Compiler** entry and select **MISRA-C**.
3. Select the MISRA-C standard: **1998** or **2004**.

Command line syntax

`--misrac-version={1998|2004}`

Description

MISRA-C rules exist in two versions: MISRA-C:1998 and MISRA-C:2004. By default, the C source is checked against the MISRA-C:2004 rules. With this option you can specify to check against the MISRA-C:1998 rules.

Related information



See Chapter 11, [MISRA-C Rules](#), for a list of all supported MISRA-C rules.

C compiler option `--misrac`

C Compiler: --no-double (-F)

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C Compiler** entry and select **Floating-Point**.
3. Enable the option **Use single precision floating-point only**.

Command line syntax

--no-double

-F

Description

With this option you tell the compiler to treat variables of the type `double` as `float`. Because the float type takes less space, execution speed increases and code size decreases, both at the cost of less precision.

Related information



C Compiler: `--no-stdinc`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C Compiler** entry and select **Miscellaneous**.
3. Add the option `--no-stdinc` to the **Additional C compiler 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)

C Compiler: --no-warnings (-w)

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C Compiler** entry and select **Diagnostics**.
3. In the **Error reporting** field, select one of the following options:
 - **Report all warnings**
 - **Suppress all warnings**
 - **Suppress specific warnings**

*If you select **Suppress specific warnings**:*

4. Enter the numbers, separated by commas or as a range, of the warnings you want to suppress.

Command line syntax

```
--no-warnings[=number[-number],...]  
-w[number[-number],...]
```

Description

With this option you can suppress 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 warning is suppressed.
- You can specify the option **--no-warnings=number** multiple times.

Example

To suppress warnings 135 and 136, enter 135, 136 in the **Specific warnings to suppress** field, or enter the following on the command line:

```
carml test.c --no-warnings=135,136
```

Related information



C compiler option **--warnings-as-errors** (Treat warnings as errors)

C Compiler: --optimize (-O)

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C Compiler** entry and select **Optimization**.
3. Select an optimization level in the **Optimization level** box.
4. If you select **Custom Optimization**, enable or disable the optimizations you want.
5. In addition, in the **Size/speed trade-off** field, select a level between **fully optimize for size** or **fully optimize for speed**.

Command line syntax

--optimize[=flags]

-O[flags]

Use the following options for predefined sets of flags:

--optimize=0	(-O0)	No optimization Alias for: -OaBCEFGIKLOPSUY
--optimize=1	(-O1)	Few optimizations Alias for: -OabcefgIKLOPSUY
--optimize=2	(-O2)	Medium optimization (default) Alias for: -OabcefgkllopsUy
--optimize=3	(-O3)	Full optimization Alias for: -Oabcefgiklopsuy

You can enable the following individual optimizations:

+/-coalesce	(a/A)	Coalescer (remove unnecessary moves)
+/-ipro	(b/B)	Interprocedural Register Optimization
+/-cse	(c/C)	Common subexpression elimination (CSE)
+/-expression	(e/E)	Expression simplification
+/-flow	(f/F)	Control flow simplification (optimization and code reordering)
+/-glo	(g/G)	Generic assembly code optimizations
+/-inline	(i/I)	Function inlining
+/-schedule	(k/K)	Instruction scheduler
+/-loop	(l/L)	Loop transformations
+/-forward	(o/O)	Forward store
+/-propagate	(p/P)	Constant propagation
+/-subscript	(s/S)	Subscript strength reduction
+/-unroll	(u/U)	Unroll small loops
+/-peephole	(y/Y)	Peephole optimizations

For an extensive description of these optimizations, please refer to section 2.6, *Compiler Optimizations* in chapter *Using the Compiler* of the user's manual.

Description

The TASKING C compilers offer four optimization levels and a custom level, at each level a specific set of optimizations is enabled.

- **No optimization (-O0)**: No optimizations are performed except for the coalescer (to allow better debug information). 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.
- **Few optimizations (-O1)**: 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.
- **Medium optimization (-O2)**: Enables more optimizations to reduce code size and/or execution time. This is the default optimization level.

- **Full optimization (-O3):** This is the highest optimization level. Use this level to decrease execution time to meet your real-time requirements.
- **Custom optimization (-Ox/X):** you can enable/disable specific optimizations.

With these options you can control the level of optimization. The default optimization level is **Medium optimization** (option **-O2** or **-O** or **-OabcefglklopsUy**).

You can overrule these settings in your C source file with the pragma pair `#pragma optimize flag` and `#pragma endoptimize`.



In addition to the command line option **--optimize (-O)**, 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 -O2 test.c
carm --optimize=2 test.c
carm -O test.c
carm --optimize test.c
carm -OabcefglklopsUy test.c
carm --optimize=+coalesce,+ipro,+cse,+expression,+flow,+glo,
    -inline,+schedule,+loop,+forward,+propagate,+subscript,
    -unroll,+peephole test.c
```

Related information



Section 2.6, [Compiler Optimizations](#), in chapter *Using the Compiler* of the user's manual.

C compiler option **--tradeoff (-t)** (Trade off between speed (**-t0**) and size (**-t4**))

C Compiler: `--option-file (-f)`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C Compiler** entry and select **Miscellaneous**.
3. Add the option `--option-file` to the **Additional C compiler options** field.

Be aware that the options in the option file are added to the C compiler options you have set in the other dialogs. Only in extraordinary cases you may want to use them in combination. Altium Designer automatically saves the options with your project.

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 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:

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

Specify the option file to the C compiler:

```
carm --option-file=myoptions
```

This is equivalent to the following command line:

```
carm -g -DDEMO=1 test.c
```

Related information



C Compiler: --output (-o)

Menu entry

Altium Designer 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: `--preprocess` (`-E`)

Menu entry

1. From the **Project** menu, select **Project Options...**
The *Project Options dialog box* appears.
2. Expand the **C Compiler** entry and select **Preprocessing**.
3. Enable the option **Store the C Compiler preprocess output (<file>.pre)**.

Command line syntax

```
--preprocess[=flags]
-E[flags]
```

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

<code>+/-comments</code>	<code>(c/C)</code>	Keep comments from the C source in the preprocessed output
<code>+/-includes</code>	<code>(i/I)</code>	Generate a list of included source files
<code>+/-list</code>	<code>(l/L)</code>	Generate a list of macro definitions
<code>+/-make</code>	<code>(m/M)</code>	Generate dependency lines that can be used for the makefile
<code>+/-noline</code>	<code>(p/P)</code>	Strip #line source position info (lines starting with #line)

The compiler sends the preprocessed file to stdout. To capture the information in a file, specify an output file with the option `--output`.

Description

When compiling, each file is preprocessed first. With this option you can store the result of preprocessed C files. Altium Designer stores the preprocessed file in a file called *name.pre* (where *name* is the name of the C source file being compiled). C comments are not preserved (similar to `-ECILMP`).

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.

Related information



C compiler option `--make-target` (Specify target name for `-Em` output)

C Compiler: --profile (-p)

Menu entry

1. From the **Project** menu, select **Project Options...**
The *Project Options dialog box* appears.
2. Expand the **C Compiler** entry and select **Debug Information**.
3. Enable the option **Generate profiling information**.
4. Enable one or more of the following suboptions to select which profiles should be obtained:
 - **Block counters** (not in combination with **Call graph** or **Function timers**)
 - **Call graph**
 - **Function counters**
 - **Function timers**



Note that the more detailed information you request, the larger the overhead in terms of execution time, code size and heap space needed. The option **Generate Debug information** (**--debug** or **-g**) does not affect profiling, execution time or code size.

Command line syntax

```
--profile[=flags]
-p[flags]
```

Use the following option for a predefined set of flags:

```
--profile=g      (-pg)      profiling with call graph and function timers
                        Alias for: -pBcFt
```

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

+/-block	(b/B)	block counters
+/-callgraph	(c/C)	call graph
+/-function	(f/F)	function counters
+/-static	(s/S)	static profile generation
+/-time	(t/T)	function timers

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.



For an extensive description of profiling refer to Chapter 4, [Profiling](#), in the user's manual.

With this option, the compiler adds the extra code to your application that takes care of the profiling process. 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).

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.



If you use the profiling option, you must link the corresponding libraries too! Refer to Section 6.4, [Linking with Libraries](#) in Chapter *Using the Linker* of the user's manual, for an overview of the (profiling) libraries. When you use Altium Designer, automatically the correct libraries are linked.

Example

```
carm --profile=+block test.c
```

In this case you must link the library `pbarm.lib`.

Related information



Chapter 4, [Profiling](#) in the user's manual.

C Compiler: --rename-sections (-R)

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C Compiler** entry and select **Miscellaneous**.
3. Add the option **--rename-sections** to the **Additional C compiler options** field.

Command line syntax

```
--rename-sections=[name]={suffix|-f|-m|-fm}
-R[name]={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.

With the section *name* you select which sections are renamed. With *suffix* you specify the suffix part which will be attached to the existing name. The following name values have special meaning:

With the suboption **-f**, the compiler uses the function name (only for code).

With the suboption **-m**, the compiler uses the name of the current module.

With the suboption **-fm** (or **-mf**), the compiler uses the name of the current module for data sections and the function name for code sections.

If you do not specify a section name, all sections will receive the specified suffix.

```
carm --rename-sections=.data=NEW test.c
```

To add the name of the current module name as suffix to all data sections, resulting in `.data.test`):

```
carm --rename-sections=.data=-m test.c
```



Assembler directive **.SECTION**

C Compiler: --runtime (-r)

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C Compiler** entry and select **Debug Information**.
3. Enable the option **Run-time checks**.
4. Enable one or more of the following suboptions to select which run-time checks should be performed:
 - **Bounds checking**
 - **Report unhandled case in a switch**
 - **Malloc consistency checks**
 - **Stack overflow check**
 - **Division by zero check**

Command line syntax

```
--runtime[=flags]  
-r[flags]
```

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

+/-bounds	(b/B)	bounds checking
+/-case	(c/C)	report unhandled case in a switch
+/-malloc	(m/M)	malloc consistency checks
+/-stack	(s/S)	check for stack overflow
+/-zero	(z/Z)	check for divide by zero

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 original application 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



C Compiler: `--signed-bitfields`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C Compiler** entry and select **Language**.
3. Enable the option **Treat 'int' bit-fields as signed**.

Command line syntax

`--signed-bitfields`

Description

For bit-fields it depends on the implementation whether a plain `int` is treated as `signed int` or `unsigned int`. By default an `int` bit-field is treated as `unsigned int`. This offers the best performance. With this option you tell the compiler to treat `int` bit-fields as `signed int`. In this case, you can still add the keyword `unsigned` to treat a particular `int` bit-field as `unsigned`.

Related information



C Compiler: `--silicon-bug`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C Compiler** entry and select **Miscellaneous**.
3. Add the option `--silicon-bug` to the **Additional C compiler options** field.

Command line syntax

`--silicon-bug[=arg,...]`

You can give the following argument:

602117 workaround for 602117

Description

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

If you do not specify an argument, then all silicon bug workarounds are enabled.

Related information



C Compiler: `--source` (`-s`)

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C Compiler** entry and select **Miscellaneous**.
3. Enable the option **Merge C source code with assembly in output file (.src)**.

Command line syntax

`--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



C Compiler: --static

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C Compiler** entry and select **Miscellaneous**.
3. Add the option **--static** to the **Additional C compiler options** field.

Command line syntax

--static

Description

With this option, the compiler treats external definitions at file scope (except for `main`) as if they were declared `static`. As a result, unused functions will be eliminated, and the alias checking algorithm assumes that objects with static storage cannot be referenced from functions outside the current module.



On the command line this option only makes sense when you specify all modules of an application on the command line.

Example

```
carm --static module1.c module2.c module3.c ...
```

Related information



-

C Compiler: `--stdout (-n)`

Menu entry

Command line only.

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: `--thumb`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C Compiler** entry and select **Code Generation**.
3. Enable the option **Generate Thumb code**.

Command line syntax

`--thumb`

Description

With this option you tell the compiler to generate 16-bit thumb instructions.

Related information



C compiler option `--interwork` (Generate interworking code)

C Compiler: `--tradeoff (-t)`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C Compiler** entry and select **Optimization**.
3. In the **Size/speed trade-off** field, select a level between **fully optimize for size** or **fully optimize for speed**.

Command line syntax

```
--tradeoff={0|1|2|3|4}  
-t{0|1|2|3|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 more speed (`--tradeoff=0`).



If you have not used the option `--optimize`, the compiler uses the default optimization. In this case it is still useful to specify a trade-off level.

Related information



C compiler option `--optimize` (Specify optimization level)

C Compiler: --uchar (-u)

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C Compiler** entry and select **Language**.
3. Enable the option **Treat 'char' variables as unsigned**.

Command line syntax

--uchar
-u

Description

By default char is the same as specifying signed char. With this option char is the same as unsigned char.

Related information



C Compiler: `--unaligned-access`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C Compiler** entry and select **Miscellaneous**.
3. Add the option `--unaligned-access` to the **Additional C compiler 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: `--undefine` (`-U`)

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C Compiler** entry and select **Miscellaneous**.
3. Add the option `--undefine` to the **Additional C compiler options** field.

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.

However, the following predefined ISO C standard macros cannot be undefined:

<code>__FILE__</code>	current source filename
<code>__LINE__</code>	current source line number (int type)
<code>__TIME__</code>	hh:mm:ss
<code>__DATE__</code>	mmm dd yyyy
<code>__STDC__</code>	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)

C Compiler: `--version` (`-V`)

Menu entry

Command line only.

Command line syntax

`--version`
`-V`

Description

Displays version information of the compiler. The compiler ignores all other options or input files.

Related information



C Compiler: `--warnings-as-errors`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C Compiler** entry and select **Diagnostics**.
3. In the **Treat warnings as errors**. field, select **None**, **Treat all warnings as errors** or **Treat specific warnings as errors**.

*If you select **Treat specific warnings as errors**:*

4. Enter the numbers, separated by commas or as a range, of the warnings you want to treat 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 as errors. This means that the exit status of the compiler will be non-zero after one or more compiler warnings. As a consequence, the compiler now also stops after encountering a warning.

You can also limit this option to specific warnings by specifying a comma-separated list of warning numbers or ranges.

Related information



C compiler option `--no-warnings` (Suppress some or all warnings)

6.2 C++ Compiler Options

Altium Designer uses a makefile to build your entire project. This means that in Altium Designer you cannot run the C++ compiler separately. If you compile a single C++ source file from within Altium Designer, the file is also compiled and assembled. However, you can set options specific for the C++ compiler.

Options in Altium Designer versus options on the command line

Most command line options have an equivalent option in Altium Designer but some options are only available on the command line (for example in a Windows Command Prompt). If there is no equivalent option in Altium Designer, you can specify a command line option in Altium Designer as follows:

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Enter one or more command line options in the **Additional C++ compiler options** field.

Invocation syntax on the command line (Windows Command Prompt)

To call the C++ compiler from the command line, use the following syntax:

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

The input *file* must be a C++ source file (.cc, .cxx or .cpp).

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 double minus (--) characters. You can abbreviate long option names as long as the name is unique. You can mix short and long option names on the command line.

If a long 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 sub-options. To switch a flag 'on', use a lowercase letter or a *+longflag*. To switch a flag off, use an uppercase letter or a *-longflag*. Separate *longflags* with commas. The 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 options **--define** and **--undefine** 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 option **--output-file**.

C++ Compiler: `--alternative-tokens`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--alternative-tokens` to the **Additional C++ compiler options** field.

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: `--anachronisms`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C++ Compiler** entry and select **C++ Language**.
3. Enable the option **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 2.3.3, *Anachronisms Accepted* in chapter *C++ Language*.

C++ Compiler: `--base-assign-op-is-default`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--base-assign-op-is-default` to the **Additional C++ compiler options** field.

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: `--building-runtime`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--building-runtime` to the **Additional C++ compiler options** field.

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: `--c++0x`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--c++0x` to the **Additional C++ compiler options** field.

Command line syntax

`--c++0x`

Description

Enable the C++ extensions that are defined by the latest C++ working paper.

Related information



C++ Compiler: `--check`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--check` to the **Additional C++ compiler options** field.

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.

Related information



C compiler option `--check` (Check syntax)

Assembler option `--check` (Check syntax)

C++ Compiler: `--context-limit`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--context-limit` to the **Additional C++ compiler options** field.

Command line syntax

`--context-limit=number` (Default: 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: `--cpu (-C)`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **Processor** entry and select **Processor Definition**.
3. Select a processor from the **Select processor** box.

Command line syntax

```
--cpu=[ARMv4|ARMv4T|ARMv5T|ARMv5TE|ARMv6M|ARMv7M|XS]  
-C[ARMv4|ARMv4T|ARMv5T|ARMv5TE|ARMv6M|ARMv7M|XS]
```

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: ARMv4, ARMv4T, ARMv5T, ARMv5TE, ARMv6-M, ARMv7-M or XScale.

The macro `__CPU__` is set to the string supplied to this option.

Example

To compile the file `test.cc` for the ARMv4 processor type, enter the following on the command line:

```
cparm --cpu=ARMv4 test.cc
```

Related information



C++ Compiler: `--create-pch`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Precompiled Headers**.
3. Enter a filename in the **Create precompiled header file** field.

Command line syntax

`--create-pch=filename`

Description

If other conditions are satisfied, create a precompiled header file with the specified name. If `--pch` (automatic PCH mode) or `--use-pch` appears on the command line following this option, its effect is erased.

Example

To create a precompiled header file with the name `test.pch`, enter

```
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 2.11, *Precompiled Headers* in chapter *C++ Language*.

C++ Compiler: `--define (-D)`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C++ Compiler** entry and select **Preprocessing**.
3. Select **User macro** and click on the down arrow in the right pane to expand macro input.
4. Click on an empty **Macro** field and enter a macro name. (Then click an empty cell to confirm)
5. Optionally, click in the **Value** field and enter a definition. (Then click an empty cell to confirm)

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, as in `#define`. If you only specify a macro name (no macro definition), the macro expands as `'1'`. You can specify as many macros as you like.

Function-style macros can be defined by appending a macro parameter list to *macro_name*.

On the command line, you can use the option `--define (-D)` multiple times. If the command line exceeds the length limit of the operating system, you can define the macros in an *option file* which you then must specify to the C++ compiler with the option `--option-file=file (-f)`.

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 definition can be tested by the preprocessor with `#if`, `#ifdef` and `#ifndef`, for conditional compilations.

Example

Consider the following program with conditional code to compile a demo program and a real program:

```
void main( void )  
{  
  #if DEMO == 1  
    demo_func();    /* compile for the demo program */  
  #else  
    real_func();    /* compile for the real program */  
  #endif  
}
```

You can now use a macro definition to set the DEMO flag:

```
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.

```
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: `--dep-file`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--dep-file` to the **Additional C++ compiler options** field.

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 (-Em)`, 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: `--dollar`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--dollar` to the **Additional C++ compiler options** field.

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: `--embedded-c++`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C++ Compiler** entry and select **C++ Language**.
3. 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: `--endianness`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **Processor** entry and select **Processor Definition**.
3. Enable the option **Big-endian mode**.

Command line syntax

`--endianness=endianness`

`-B`

`--big-endian`

You can specify the following *endianness*:

big	(b)	Big endian
little	(l)	Little endian (default)

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.

Related information



C++ Compiler: `--error-file`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--error-file` to the **Additional C++ compiler options** field.

Command line syntax

`--error-file[=file]`

Description

With this option the C++ compiler redirects error messages to a file. If you do not specify a filename, the error file will be named after the input file with extension `.ecp`.

Example

To write errors to `errors.ecp` instead of `stderr`, enter:

```
cparm --error-file=errors.ecp test.cc
```

Related information



C++ Compiler: `--error-limit` (`-e`)

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--error-limit` to the **Additional C++ compiler options** field.

Command line syntax

`--error-limit=number` (Default: 100)
`-enumber`

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: `--exceptions` (`-x`)

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C++ Compiler** entry and select **C++ Language**.
3. Enable the option **Support for exception handling**.

Command line syntax

`--exceptions`
`-x`

Description

With this option you enable support for exception handling in the C++ compiler.

The macro `__EXCEPTIONS` is defined when exception handling support is enabled.

Related information



C++ Compiler: `--exported-template-file`

Menu entry

-

Command line syntax

`--exported-template-file=file`

Description

This option specifies the name to be used for the exported template file used for processing of exported templates.

This option is supplied for use by the control program that invokes the C++ compiler and is not intended to be used by end-users.

Related information



-

C++ Compiler: `--extended-variadic-macros`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--extended-variadic-macros` to the **Additional C++ compiler options** field.

Command line syntax

`--extended-variadic-macros`

Default: macros with a variable number of arguments are not allowed.

Description

Allow macros with a variable number of arguments (implies `--variadic-macros`) and allow the naming of the variable argument list.

Related information



C++ compiler option `--variadic-macros` (Allow variadic macros)

C++ Compiler: `--force-vtbl`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. In the **Virtual function tables field**, select **Force definition of virtual function tables**.

Command line syntax

`--force-vtbl`

Description

Force definition of virtual function tables in cases where the heuristic used by the C++ compiler to decide on definition of virtual function tables provides no guidance.

Related information



C++ compiler option `--suppress-vtbl` (Suppress definition of virtual function tables)

C++ Compiler: `--friend-injection`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--friend-injection` to the **Additional C++ compiler options** field.

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: --g++

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C++ Compiler** entry and select **C++ Language**.
3. Enable the option **Allow GNU C++ extensions**.

Command line syntax

--g++

Description

Enable GNU C++ compiler language extensions.

Related information



Section 2.4, [GNU Extensions](#) in chapter *C++ Language*.

C++ Compiler: `--gnu-version`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--gnu-version` to the **Additional C++ compiler options** field.

Command line syntax

`--gnu-version=version`

Default: 30300 (version 3.3.0)

Description

It depends on the GNU C++ compiler version if a particular GNU extension is supported or not. With this option you set the GNU C++ compiler version that should be emulated in GNU C++ mode. Version *x.y.z* of the GNU C++ compiler is represented by the value $x*10000+y*100+z$.

Example

To specify version 3.4.1 of the GNU C++ compiler, enter:

```
cparm --g++ --gnu-version=30401 test.cc
```

Related information



Section 2.4, [GNU Extensions](#) in chapter *C++ Language*.

C++ Compiler: `--guiding-decls`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--guiding-decls` to the **Additional C++ compiler options** field.

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:

```
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: --help (-?)

Menu entry

Command line only.

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. 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:

```
cparm -?  
cparm --help  
cparm
```

The following invocation displays an extended list of the available options:

```
cparm --help=options
```

Related information



C++ Compiler: `--implicit-extern-c-type-conversion`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--implicit-extern-c-type-conversion` to the **Additional C++ compiler options** field.

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: `--implicit-include`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. 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 2.6.4, *Implicit Inclusion* in chapter *C++ Language*.

C++ Compiler: `--incl-suffixes`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--incl-suffixes` to the **Additional C++ compiler options** field.

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 3.3, [How the C++ Compiler Searches Include Files](#) in chapter *Using the C++ Compiler* of the user's manual.

C++ Compiler: `--include-directory (-I)`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Select **Build Options**.
3. Add a pathname in the **Include files path** field.

If you enter multiple paths, separate them with a semicolon (;).

Command line syntax

```
--include-directory=path,...
-Ipath,...
```

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 3.3, *How the C++ Compiler Searches Include Files* in chapter *Using the C++ Compiler* of the user's manual.

C++ Compiler: `--include-file` (`-H`)

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C++ Compiler** entry and select **Preprocessing**.
3. Enter the name of the file in the **Include this file before source** field or click ... and select a file.

Command line syntax

`--include-file=file,...`
`-Hfile,...`

Description

With this option (set at project level) you include one extra file at the beginning of each C++ source file in your project. On a document level (**Project » Document Options**), you can overrule this option with another file or no file at all.

The specified include file is included at the beginning of the compilation. This is the same as specifying `#include "file"` at the very beginning of (each of) your C++ source files.

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 3.3, [How the C++ Compiler Searches Include Files](#) in chapter *Using the C++ Compiler* of the user's manual.

C++ Compiler: `--include-macros-file`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--include-macros-file` to the **Additional C++ compiler options** field.

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 3.3, [How the C++ Compiler Searches Include Files](#) in chapter *Using the C++ Compiler* of the user's manual.

C++ Compiler: `--init-priority`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--init-priority` to the **Additional C++ compiler options** field.

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: `--instantiate`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Select an instantiation mode in the **Instantiation mode of external template entities** box

Command line syntax

`--instantiate=mode`

`-tmode`

You can specify the following modes:

used (default)
all
local

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 <code>--instantiate=used</code> except that the functions are given internal linkage. This is intended to provide a very simple mechanism for those getting started with templates. The compiler will instantiate the functions that are used in each compilation unit as local functions, and the program will link and run correctly (barring problems due to multiple copies of local static variables). However, one may end up with many copies of the instantiated functions, so this is not suitable for production use.

You cannot use `--instantiate=local` in conjunction with automatic template instantiation.

Related information



C++ compiler option `--no-auto-instantiation` (Disable automatic C++ instantiation)
 Section 2.6, *Template Instantiation* in chapter *C++ Language*.

C++ Compiler: --io-streams

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C++ Compiler** entry and select **C++ Language**.
3. Enable the option **Support for I/O streams**.

Command line syntax

--io-streams

Description

As I/O streams require substantial resources they are disabled by default. Use this option to enable I/O streams support in the C++ library.

This option also enables exception handling.

Related information



C++ Compiler: `--late-tiebreaker`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--late-tiebreaker` to the **Additional C++ compiler options** field.

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: --list-file (-L)

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option **--list-file** to the **Additional C++ compiler options** field.

Command line syntax

--list-file=file
-Lfile

Default: -1

Description

Generate raw listing information in the *file*. This information is likely to be used to generate a formatted listing. The raw listing file contains raw source lines, information on transitions into and out of include files, and diagnostics generated by the C++ compiler.

Each line of the listing file begins with a key character that identifies the type of line, as follows:

- N:** A normal line of source; the rest of the line is the text of the line.
- X:** The expanded form of a normal line of source; the rest of the line is the text of the line. This line appears following the N line, and only if the line contains non-trivial modifications (comments are considered trivial modifications; macro expansions, line splices, and trigraphs are considered non-trivial modifications). Comments are replaced by a single space in the expanded-form line.
- S:** A line of source skipped by an #if or the like; the rest of the line is text. Note that the #else, #elif, or #endif that ends a skip is marked with an N.
- L:** An indication of a change in source position. The line has a format similar to the # line-identifying directive output by the C preprocessor, that is to say

L *line_number* "*file-name*" *key*

where *key* is,

- 1** for entry into an include file;
- 2** for exit from an include file;

and omitted otherwise.

The first line in the raw listing file is always an L line identifying the primary input file. L lines are also output for #line directives (*key* is omitted). L lines indicate the source position of the following source line in the raw listing file.

- R, W, E, or C:** an indication of a diagnostic (**R** for remark, **W** for warning, **E** for error, and **C** for catastrophic error). The line has the form

S "*file-name*" *line_number* *column-number* *message-text*

where *S* is R, W, E, or C, as explained above. Errors at the end of file indicate the last line of the primary source file and a column number of zero. Command line errors are catastrophes with an empty file name (""), and a line and column number of zero. Internal errors are catastrophes with position information as usual, and message-text beginning with (internal error). When a diagnostic displays a list (e.g., all the contending routines when there is ambiguity on an overloaded call), the initial diagnostic line is followed by one or more lines with the same overall format (code letter, file name, line number, column number, and message text), but in which the code letter is the lower case version of the code letter in the initial line. The source position in such lines is the same as that in the corresponding initial line.

Example

To write raw listing information to the file `test.lst`, enter:

```
cparm --list-file=test.lst test.cc
```

Related information



C++ Compiler: `--long-lifetime-temps`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--long-lifetime-temps` to the **Additional C++ compiler options** field.

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: --long-long

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option **--long-long** to the **Additional C++ compiler options** field.

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: `--make-target`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C Compiler** entry and select **Miscellaneous**.
3. Add the option `--make-target` to the **Additional C compiler options** field.

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`.

Related information



C++ compiler option `--preprocess=+make` (Generate dependencies for make)

C++ compiler option `--dep-file` (Generate dependencies in a file)

C++ Compiler: `--multibyte-chars`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--multibyte-chars` to the **Additional C++ compiler options** field.

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: `--namespaces`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--namespaces` to the **Additional C++ compiler options** field.

Command line syntax

`--namespaces`

`--no-namespaces`

Default: namespaces are supported.

Description

With option `--no-namespaces` you disable support for namespaces.

When you used option `--embedded-c++`, namespaces are disabled. With option `--namespaces` you can enable support for namespaces in this case.

The macro `__NAMESPACES` is defined when namespace support is enabled.

Related information



C++ compiler option `--embedded-c++` (Embedded C++ compliancy tests)

C++ compiler option `--using-std` (Implicit use of the std namespace)

Section 2.5, *Namespace Support* in chapter *C++ Language*.

C++ Compiler: `--no-arg-dep-lookup`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--no-arg-dep-lookup` to the **Additional C++ compiler options** field.

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: `--no-array-new-and-delete`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--no-array-new-and-delete` to the **Additional C++ compiler options** field.

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: --no-auto-instantiation

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option **--no-auto-instantiation** to the **Additional C++ compiler options** field.

Command line syntax

--no-auto-instantiation

Default: the C++ compiler automatically instantiates templates.

Description

With this option automatic instantiation of templates is disabled.

Related information



C++ compiler option **--instantiate** (Instantiation mode)
Section 2.6, [Template Instantiation](#) in chapter *C++ Language*.

C++ Compiler: `--no-bool`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--no-bool` to the **Additional C++ compiler options** field.

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: `--no-class-name-injection`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--no-class-name-injection` to the **Additional C++ compiler options** field.

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: `--no-const-string-literals`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--no-const-string-literals` to the **Additional C++ compiler options** field.

Command line syntax

`--no-const-string-literals`

Default: C++ string literals and wide string literals are `const` (as required by the standard).

Description

With this option C++ string literals and wide string literals are non-`const` (as was true in earlier versions of the C++ language).

Related information



C++ Compiler: `--no-dep-name`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--no-dep-name` to the **Additional C++ compiler options** field.

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: `--no-distinct-template-signatures`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--no-distinct-template-signatures` to the **Additional C++ compiler options** field.

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: `--no-double` (-F)

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--no-double` to the **Additional C++ compiler options** field.

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: `--no-enum-overloading`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--no-enum-overloading` to the **Additional C++ compiler options** field.

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: `--no-explicit`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--no-explicit` to the **Additional C++ compiler options** field.

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: `--no-export`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--no-export` to the **Additional C++ compiler options** field.

Command line syntax

`--no-export`

Default: exported templates (declared with the keyword `export`) are allowed.

Description

Disable recognition of exported templates. This option requires that dependent name processing be done, and cannot be used with implicit inclusion of template definitions.

Related information



Section 2.6.5, [Exported Templates](#) in chapter *C++ Language*.

C++ Compiler: `--no-extern-inline`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--no-extern-inline` to the **Additional C++ compiler options** field.

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 2.8, [Extern Inline Functions](#) in chapter *C++ Language*.

C++ Compiler: `--no-for-init-diff-warning`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--no-for-init-diff-warning` to the **Additional C++ compiler options** field.

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: `--no-implicit-typename`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--no-implicit-typename` to the **Additional C++ compiler options** field.

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: --no-inlining

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Disable the option **Minimal inlining of function calls**.

Command line syntax

--no-inlining

Description

Disable minimal inlining of function calls.

Related information



C++ Compiler: `--nonconst-ref-anachronism`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--nonconst-ref-anachronism` to the **Additional C++ compiler options** field.

Command line syntax

`--nonconst-ref-anachronism`

`--no-nonconst-ref-anachronism` (default)

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 option `--anachronisms`.

Related information



C++ compiler option `--anachronisms` (Enable C++ anachronisms)
Section 2.3.3, *Anachronisms Accepted* in chapter *C++ Language*.

C++ Compiler: `--nonstd-qualifier-deduction`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--nonstd-qualifier-deduction` to the **Additional C++ compiler options** field.

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: `--nonstd-using-decl`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--nonstd-using-decl` to the **Additional C++ compiler options** field.

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: `--no-parse-templates`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--no-parse-templates` to the **Additional C++ compiler options** field.

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: `--no-pch-messages`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--no-pch-messages` to the **Additional C++ compiler options** field.

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 2.11, *Precompiled Headers* in chapter *C++ Language*.

C++ Compiler: `--no-preprocessing-only`

Menu entry

Altium Designer 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: `--no-stdinc` / `--no-stdstlinc`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--no-stdinc` or `--no-stdstlinc` to the **Additional C++ compiler options** field.

Command line syntax

`--no-stdinc`
`--no-stdstlinc`

Description

With option `--no-stdinc` you tell the C++ compiler not to look in the default `include` directory relative to the installation directory, when searching for standard include files.

With option `--no-stdstlinc` you tell the C++ compiler not to look in the default `include.stl` directory relative to the installation directory, when searching for standard STL include files.

This way the C++ compiler only searches in the include file search paths you specified.

Related information



Section 3.3, [How the C++ Compiler Searches Include Files](#) in chapter *Using the C++ Compiler* of the user's manual.

C++ Compiler: `--no-typename`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--no-typename` to the **Additional C++ compiler options** field.

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: `--no-use-before-set-warnings` (-j)

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Diagnostics**.
3. 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: `--no-warnings (-w)`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Diagnostics**.
3. Enable the option **Suppress all warnings**.

Command line syntax

`--no-warnings`
`-w`

Description

With this option you suppress all warning messages. Error messages are still issued.

Related information



C++ compiler option `--warnings-as-errors` (Treat warnings as errors)

C++ Compiler: `--old-for-init`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--old-for-init` to the **Additional C++ compiler options** field.

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: `--old-line-commands`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--old-line-commands` to the **Additional C++ compiler options** field.

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: `--old-specializations`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--old-specializations` to the **Additional C++ compiler options** field.

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-file (-f)

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option **--option-file** to the **Additional C compiler options** field.

Be aware that the options in the option file are added to the C++ compiler options you have set in the other dialogs. Only in extraordinary cases you may want to use them in combination. Altium Designer automatically saves the options with your project.

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:

```
--embedded-c++  
--define=DEMO=1  
test.cc
```

Specify the option file to the C++ compiler:

```
cparm --option-file=myoptions
```

This is equivalent to the following command line:

```
cparm --embedded-c++ --define=DEMO=1 test.cc
```

Related information



C++ Compiler: --output (-o)

Menu entry

Altium Designer names the output file always after the C++ source file.

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option **--old-specializations** to the **Additional C++ compiler options** field.

Command line syntax

--output=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 only)

C++ Compiler: --pch

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Precompiled Headers**.
3. Enable the option **Automatically use and/or create a 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 2.11, *Precompiled Headers* in chapter *C++ Language*.

C++ Compiler: `--pch-dir`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C++ Compiler** entry and select **Precompiled Headers**.
3. Enter a path in the **Precompiled header file directory** field.

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 2.11, *Precompiled Headers* in chapter *C++ Language*.

C++ Compiler: `--pch-verbose`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--pch-verbose` to the **Additional C++ compiler options** field.

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 2.11, *Precompiled Headers* in chapter *C++ Language*.

C++ Compiler: `--pending-instantiations`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--pending-instantiations` to the **Additional C++ compiler options** field.

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 2.6, [Template Instantiation](#) in chapter *C++ Language*.

C++ Compiler: --preprocess (-E)

Menu entry

1. From the **Project** menu, select **Project Options...**
The *Project Options dialog box* appears.
2. Expand the **C++ Compiler** entry and select **Preprocessing**.
3. Enable the option **Store the C++ compiler preprocess output (<file>.pre)**.

Command line syntax

```
--preprocess[=flags]
-E[flags]
```

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

+/-comments	(c/C)	Keep comments from the C source in the preprocessed output
+/-make	(m/M)	Generate dependency lines that can be used for the makefile
+/-noline	(p/P)	Strip #line source position info (lines starting with #line)

Description

With this option you tell the C++ compiler to preprocess the C++ source. Under Altium Designer the C++ compiler sends the preprocessed output to the file *name.pre* (where *name* is the name of the C++ source file to compile). Altium Designer also compiles the C++ source.

On the command line, the C++ compiler sends the preprocessed file to stdout. To capture the information in a file, specify an output file with the option **--output**.

With **--preprocess=+comments** you tell the preprocessor to keep the comments from the C++ source file in the preprocessed output.

With **--preprocess=+make** the C++ compiler will generate dependency lines that can be used in a Makefile. The preprocessor output is discarded. 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 **--make-target** (Specify target name for **-Em** output)

C++ Compiler: `--remarks (-r)`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Diagnostics**.
3. 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 3.4, [C++ Compiler Error Messages](#) in chapter *Using the C++ Compiler* of the user's manual.

C++ Compiler: `--remove-unnneeded-entities`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--remove-unnneeded-entities` to the **Additional C++ compiler options** field.

Command line syntax

`--remove-unnneeded-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: `--rtti`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **C++ Language**.
3. Enable the option **Support for RTTI (run-time type information)**.

Command line syntax

`--rtti`

Default: RTTI (run-time type information) features are disabled.

Description

Enable support for RTTI (run-time type information) features: `dynamic_cast`, `typeid`.

The macro `__RTTI` is defined when RTTI support is enabled.

Related information



C++ Compiler: `--schar` (`-s`)

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--schar` to the **Additional C++ compiler options** field.

Command line syntax

`--schar`
`-s`

Description

With this option `char` is the same as `signed char`.

When plain `char` is signed, the macro `__SIGNED_CHARS__` is defined.

Related information



[C++ compiler option `--uchar`](#) (Plain `char` is unsigned)

C++ Compiler: `--signed-bitfields`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--signed-bitfields` to the **Additional C++ compiler options** field.

Command line syntax

`--signed-bitfields`

Description

For bit-fields it depends on the implementation whether a plain `int` is treated as `signed int` or `unsigned int`. By default an `int` bit-field is treated as `unsigned int`. This offers the best performance. With this option you tell the C++ compiler to treat `int` bit-fields as `signed int`. In this case, you can still add the keyword `unsigned` to treat a particular `int` bit-field as `unsigned`.

Related information



C compiler option `--signed-bitfields`

C++ Compiler: `--special-subscript-cost`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--special-subscript-cost` to the **Additional C++ compiler options** field.

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:

```
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: `--strict (-A)`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **C++ Language**.
3. Disable the option **Allow non-ANSI/ISO C++ features**.

Command line syntax

`--strict`
`-A`

Default: non-ANSI/ISO C++ features are enabled.

Description

Enable strict ANSI/ISO mode, which provides diagnostic messages when non-standard features are used, and disables features that conflict with ANSI/ISO C or C++. All ANSI/ISO violations are issued as errors.

Example

To enable strict ANSI mode, with error diagnostic messages, enter:

```
cparm --strict test.cc
```

Related information



C++ compiler option `--strict-warnings` (Strict ANSI/ISO mode with warnings)

C++ Compiler: `--strict-warnings` (`-a`)

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--strict-warnings` to the **Additional C++ compiler options** field.

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: `--suppress-vtbl`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. In the **Virtual function tables field**, select **Suppress definition of virtual function tables**.

Command line syntax

`--suppress-vtbl`

Description

Suppress definition of virtual function tables in cases where the heuristic used by the C++ compiler to decide on definition of virtual function tables provides no guidance. The virtual function table for a class is defined in a compilation if the compilation contains a definition of the first non-inline non-pure virtual function of the class. For classes that contain no such function, the default behavior is to define the virtual function table (but to define it as a local static entity). The `--suppress-vtbl` option suppresses the definition of the virtual function tables for such classes, and the `--force-vtbl` option forces the definition of the virtual function table for such classes. `--force-vtbl` differs from the default behavior in that it does not force the definition to be local.

Related information



C++ compiler option `--force-vtbl` (Force definition of virtual function tables)

C++ Compiler: --sys-include

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option **--sys-include** to the **Additional C++ compiler options** field.

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 3.3, [How the C++ Compiler Searches Include Files](#) in chapter *Using the C++ Compiler* of the user's manual.

C++ Compiler: `--template-directory`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--template-directory` to the **Additional C++ compiler options** field.

Command line syntax

`--template-directory=directory,...`

Description

Specifies a directory name to be placed on the exported template search path. The directories are used to find the definitions of exported templates (`.et` files) and are searched in the order in which they are specified on the command line. The current directory is always the first entry on the search path.

Example

To add the directory `export` to the exported template search path, enter:

```
cparm --template-directory=export test.cc
```

Related information



Section 2.6.5, [Exported Templates](#) in chapter *C++ Language*.

C++ Compiler: `--thumb`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C Compiler** entry and select **Code Generation**.
3. Enable the option **Generate Thumb code**.

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: `--timing`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--timing` to the **Additional C++ compiler options** field.

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: `--trace-includes`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Diagnostics**.
3. Enable the option **List all visited #included files**.

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: `--type-traits-helpers`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--type-traits-helpers` to the **Additional C++ compiler options** field.

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: --uchar (-u)

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option **--uchar** to the **Additional C++ compiler options** field.

Command line syntax

--uchar
-u

Description

By default char is the same as specifying signed char. With this option char is the same as unsigned char.

Related information



C++ compiler option **--schar** (Plain char is signed)

C++ Compiler: **--undefine (-U)**

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option **--undefine** to the **Additional C++ compiler options** field.

Command line syntax

```
--undefine=macro_name  
-Umacro_name
```

Description

Remove any initial definition of identifier 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 2.10, [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 2.10, [Predefined Macros](#) in chapter *C++ Language*.

C++ Compiler: `--use-pch`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C++ Compiler** entry and select **Precompiled Headers**.
3. 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 2.11, *Precompiled Headers* in chapter *C++ Language*.

C++ Compiler: `--using-std`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--using-std` to the **Additional C++ compiler options** field.

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 2.5, [Namespace Support](#) in chapter *C++ Language*.

C++ Compiler: `--variadic-macros`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option `--variadic-macros` to the **Additional C++ compiler options** field.

Command line syntax

`--variadic-macros`

Default: macros with a variable number of arguments are not allowed.

Description

Allow macros with a variable number of arguments.

Related information



C++ compiler option `--extended-variadic-macros` (Allow extended variadic macros)

C++ Compiler: `--version` (`-V`)

Menu entry

Command line only.

Command line syntax

`--version`
`-V`

Description

Display version information. The C++ compiler ignores all other options or input files.

Related information



C++ Compiler: `--warnings-as-errors`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C++ Compiler** entry and select **Diagnostics**.
3. Enable the option **Treat warnings as errors**.

Command line syntax

`--warnings-as-errors`

Description

If the C++ compiler encounters an error, it stops compiling. With this option you tell the C++ compiler to treat all warnings as errors. This means that the exit status of the C++ compiler will be non-zero after one or more compiler warnings. As a consequence, the C++ compiler now also stops after encountering a warning.

Related information



C++ compiler option `--no-warnings` (Suppress all warnings)

C++ Compiler: `--wchar_t-keyword`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **C++ Compiler** entry and select **C++ Language**.
3. Enable the option **Allow the 'wchar_t' keyword**.

Command line syntax

`--wchar_t-keyword`

Default: `wchar_t` is not recognized as a keyword.

Description

Enable recognition of `wchar_t` as a keyword.

The macro `_WCHAR_T` is defined when `wchar_t` is recognized as a keyword.

Related information



C++ Compiler: --xref-file (-X)

Menu entry

1. From the **Project** menu, select **Project Options...**
The *Project Options dialog box* appears.
2. Expand the **C++ Compiler** entry and select **Miscellaneous**.
3. Add the option **--xref-file** to the **Additional C++ compiler options** field.

Command line syntax

```
--xref-file=file
-Xfile
```

Description

Generate cross-reference information in a *file*. For each reference to an identifier in the source program, a line of the form

```
symbol_id name X file-name line-number column-number
```

is written, where *X* is

- D** for definition;
- d** for declaration (that is, a declaration that is not a definition);
- M** for modification;
- A** for address taken;
- U** for used;
- C** for changed (but actually meaning used and modified in a single operation, such as an increment);
- R** for any other kind of reference, or
- E** for an error in which the kind of reference is indeterminate.

symbol-id is a unique decimal number for the symbol. The fields of the above line are separated by tab characters.

Related information



6.3 Assembler Options

Altium Designer uses a makefile to build your entire project. This means that in Altium Designer you cannot run the assembler separately. If you want assembly results, you must compile a single C source file from within Altium Designer, the file is then also assembled. However, you can set options specific for the assembler.

Options in Altium Designer versus options on the command line

Most command line options have an equivalent option in Altium Designer but some options are only available on the command line (for example in a Windows Command Prompt). If there is no equivalent option in Altium Designer, you can specify a command line option in Altium Designer as follows:

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **Assembler** entry and select **Miscellaneous**.
3. Enter one or more command line options in the **Additional assembler options** field.

Invocation syntax on the command line (Windows Command Prompt)

To call the assembler from the command line, use the following syntax:

```
asarm [ [option]... [file]... ]...
```

The input *file* must be an assembly source file (.asm or .src).

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 double minus (--) characters. You can abbreviate long option names as long as the name is unique. You can mix short and long option names on the command line.

Options can have flags or sub-options. To switch a flag 'on', use a lowercase letter or a *+longflag*. To switch a flag off, use an uppercase letter or a *-longflag*. Separate *longflags* with commas. The following two invocations are equivalent:

```
asarm -Ogs test.src
```

```
asarm --optimize=+generics,+instr-size test.src
```

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

Assembler: **--case-insensitive (-c)**

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **Assembler** entry and select **Miscellaneous**.
3. Disable the option **Assemble case sensitive**.

Command line syntax

--case-insensitive
-c

Description

With this option you tell the assembler not to distinguish between upper and lower case characters. By default the assembler considers upper and lower case characters as different characters.



Disabling the option **Assemble case sensitive** in Altium Designer is the same as specifying the option **--case-insensitive** on the command line.

Assembly source files that are generated by the compiler must always be assembled 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`.

Related information



–

Assembler: `--check`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **Assembler** entry and select **Miscellaneous**.
3. Add the option `--check` to the **Additional assembler options** field.

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.

Related information



C compiler option `--check` (Check syntax)

Assembler: --cpu (-C)

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **Processor** entry and select **Processor Definition**.
3. Select a processor from the **Select processor** box.

Command line syntax

```
--cpu=[ARMv4|ARMv4T|ARMv5T|ARMv5TE|ARMv6M|ARMv7M|XS]
-C[ARMv4|ARMv4T|ARMv5T|ARMv5TE|ARMv6M|ARMv7M|XS]
```

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. If the architecture is ARMv4 the linker replaces BX instructions by MOV PC instructions. The default architecture is ARMv4T and the complete list of supported architectures is: ARMv4, ARMv4T, ARMv5T, ARMv5TE, ARMv6-M, ARMv7-M or XScale.

Assembly code can check the value of the option by means of the built-in function @CPU (). Architecture ARMv4 does not support the Thumb instruction set. Architecture profile ARMv7-M only supports the Thumb-2 instruction set, i.e. it has no ARM execution state.



When you call the assembler from the command line, make sure you specify the same core type to the compiler to avoid conflicts!

Related information



[Assembly function @CPU \(\)](#)

Assembler: `--debug-info (-g)`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **Assembler** entry and select **Debug Information**.
3. Select which debug information to include: **Automatic HLL or assembly level debug information**, **Custom debug information** or **No debug information**.

*If you select **Custom debug information**:*

4. Select which Custom debug information to include: **Assembler source line information**, **Pass HLL debug information**, or **None**.
5. Enable or disable the option **Assembler local symbols information**.

Command line syntax

```
--debug-info[=flag]
-g[flag]
```

You can set the following flags:

<code>+/-asm</code>	<code>(a/A)</code>	Assembly source line information
<code>+/-hll</code>	<code>(h/H)</code>	Pass high level language debug information (HLL)
<code>+/-local</code>	<code>(l/L)</code>	Assembler local symbols debug information
<code>+/-smart</code>	<code>(s/S)</code>	Smart debug information

Description

With this option you tell the assembler which kind of debug information to emit in the object file.

If you do not use this option, the default is `--debug-info=+hll`. If you specify `--debug-info` without any flags, the default is `--debug-info=+smart`.

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: --define (-D)

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **Assembler** entry and select **Preprocessing**.
3. Click on **User macro**, click on the down arrow in the right pane to expand macro input.
4. Click on an empty **Macro** field and enter a macro name. (Then click outside the cell to confirm)
5. Optionally, click in the **Value** field and enter a definition. (Then click outside the cell to confirm)

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. 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 assembler with the option **--option-file=file (-f)**.

Defining macros with this option (instead of in the assembly source) is, for example, useful in combination with conditional assembly as shown in the example below.



This option has the same effect as defining symbols via the `.DEFINE`, `.SET`, and `.EQU` directives. (similar to `#define` in the C language). With the `.MACRO` directive you can define more complex macros.

Example

Consider the following assembly program with conditional code to assemble a demo program and a real program:

```
.IF DEMO == 1
...      ; instructions for demo application
.ELSE
...      ; instructions for the real application
.ENDIF
```

You can now use a macro definition to set the DEMO flag:

Macro	Value
DEMO	1 (or empty)

```
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** (Read options from file)

Assembler: --diag

Menu entry

1. From the **View** menu, select **Workspace Panels » System » Messages**.

The Messages panel appears.

2. In the **Messages** panel, right-click on the message you want more information on.

A popup menu appears.

3. Select **More Info**.

A Message Info box appears with additional information.

Command line syntax

`--diag=[format:]{all|nr,...}`

Description

With this option you can ask for an extended description of error messages in the format you choose. The output is directed to `stdout` (normally your screen) and in the format you specify. You can specify the following formats: **html**, **rtf** or **text** (default). To create a file with the descriptions, you must redirect the output.

With the suboption **all**, the descriptions of all error messages are given. If you want the description of one or more selected error messages, you can specify the error message numbers, separated by commas.

With this option the assembler does not assemble any files.

Example

To display an explanation of message number 241, enter:

```
asarm --diag=241
```

This results in the following message and explanation:

```
W241: additional input files will be ignored
```

```
The assembler supports only a single input file. All other input files are ignored.
```

To write an explanation of all errors and warnings in HTML format to file `aserrors.html`, use redirection and enter:

```
asarm --diag=html:all > aserrors.html
```

Related information



Assembler: **--emit-locals**

Menu entry

Command line only.

Command line syntax

--emit-locals[=*flag*,...]

You can set the following flags (when you specify no flags, the default is **Es**):

+/-eques	(e/E)	emit local EQU symbols
+/-symbols	(s/S)	emit local non-EQU symbols

Description

With the option **--emit-locals=+eques** the assembler also emits local EQU symbols to the object file. Normally, only global symbols and non-EQU local symbols are emitted. Having local symbols in the object file can be useful for debugging.

Related information



Assembler: --endianness

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **Processor** entry and select **Processor Definition**.
3. Enable the option **Big-endian mode**.

Command line syntax

`--endianness=endianness`

`-B`

`--big-endian`

You can specify the following *endianness*:

big	(b)	Big endian
little	(l)	Little endian (default)

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: `--error-file`

Menu entry

Command line only.

Command line syntax

`--error-file[=file]`

Description

With this option the assembler redirects error messages to a file.

If you do not specify a filename, the error file will be named after the input file with extension `.ers`.

Example

To write errors to `errors.err` instead of `stderr`, enter:

```
asarm --error-file=errors.err test.src
```

Related information



Assembler: **--error-limit**

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **Assembler** entry and select **Miscellaneous**.
3. Add the option **--error-limit** to the **Additional assembler options** field.

Command line syntax

--error-limit=number

Description

With this option you tell the assembler to only emit the specified maximum number of errors. Without this option (same as 0) the assembler emits all errors.

Related information



Assembler: --help (-?)

Menu entry

Command line only.

Command line syntax

```
--help[=options]  
-?
```

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
```

Assembler: **--include-directory (-I)**

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Select **Build Options**.
3. Add a pathname in the **Include Files Path** field.

If you enter multiple paths, separate them with a semicolon (;).

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 `include` directory relative to the installation directory.

Example

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

```
.INCLUDE 'myinc.inc'
```

You can call the assembler as follows:

```
asarm --include-directory=c:\proj\include test.src
```

First the assembler looks in the directory where `test.src` is located for the file `myinc.inc`. If it does not find the file, it looks in the directory `c:\proj\include` for the file `myinc.inc` (this option). 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 (-H)** (Include file before source)

Assembler: `--include-file (-H)`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **Assembler** entry and select **Preprocessing**.
3. Enter the name of the file in the **Include this file before source** field or click ... and select a file.

Command line syntax

`--include-file=file,...`
`-Hfile,...`

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 test1.src
```

The file `myinc.inc` is included at the beginning of `test1.src` before it is assembled.

Related information



[Assembler option `--include-directory`](#) (Include files path)

Section 5.4, [How the Assembler Searches Include Files](#), in chapter *Using the Assembler* of the user's manual.

Assembler: --inversions

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **Assembler** entry and select **Miscellaneous**.
3. 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

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: --keep-output-files (-k)

Menu entry

Altium Designer *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: **--list-file (-l)**

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **Assembler** entry and select **List File**.
3. Enable **Generate list file**.
4. In the **List file format** section, enable or disable the types of information to be included.

Command line syntax

--list-file[=file]
-l[file]

Description

With this option you tell the assembler to generate a list file. A list file shows the generated object code and the relative addresses. Note that the assembler generates a relocatable object file with relative addresses.

With the optional *file* you can specify an alternative name for the list file. By default, the name of the list file is the basename of the source file with the extension `.lst`.

Related information



On the command line you can use the option **--list-format (-L)** to specify which types of information should be included in the list file.

Assembler: --list-format (-L)

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **Assembler** entry and select **List File**.
3. Enable **Generate list file**.
4. In the **List file format** section, enable or disable the types of information to be included.

Command line syntax

--list-format=flags
-Lflags

You can set the following flags:

0	Same as -LDEGILMNPQRSVWXYZ (all options disabled)
1	Same as -Ldegilmnpqrsvwxyz (all options enabled)
+/-section	(d/D) Section directives (.SECTION)
+/-symbol	(e/E) Symbol definition directives
+/-generic-expansion	(g/G) Generic instruction expansion
+/-generic	(i/I) Generic instructions
+/-line	(l/L) C preprocessor #line directives
+/-macro	(m/M) Macro/dup definitions (e.g. .MACRO)
+/-empty-line	(n/N) Empty source lines (newline)
+/-conditional	(p/P) Conditional assembly (.IF, .ELSE, .ENDIF)
+/-equate	(q/Q) Assembler .EQU and .SET directives
+/-relocations	(r/R) Relocation characters ('r')
+/-hll	(s/S) HLL symbolic debug information (.SYMB)
+/-equate-values	(v/V) Assembler .EQU and .SET values
+/-wrap-lines	(w/W) Wrapped source lines
+/-macro-expansion	(x/X) Macro expansions
+/-cycle-count	(y/Y) Cycle counts
+/-macro-expansion	(z/Z) Define expansions

Default: -LdEGiIlMnPqrsVwXyZ

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: **--no-warnings (-w)**

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **Assembler** entry and select **Diagnostics**.
3. Enable one of the options:
 - **Report all warnings**
 - **Suppress all warnings**
 - **Suppress specific warnings**

*If you select **Suppress specific warnings**:*

4. Enter the numbers, separated by commas, of the warnings you want to suppress.

Command line syntax

```
--no-warnings[=number,...]  
-w[number,...]
```

Description

With this option you can suppresses all warning messages or specific warning messages.

- If you do not specify this option, all warnings are reported.
- If you specify this option but without numbers, all warnings are suppressed.
- If you specify this option with a number, only the specified warning is suppressed.
You can specify the option **--no-warnings=number** multiple times.

Example

To suppress warnings 135 and 136, enter **135, 136** in the **Specific warnings to suppress** field, or enter the following on the command line:

```
asarm test.src --no-warnings=135,136
```

Related information



Assembler option **--warnings-as-errors** (Treat warnings as errors)

Assembler: --old-syntax

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **Assembler** entry and select **Miscellaneous**.
3. 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



Assembler: --optimize (-O)

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **Assembler** entry and select **Optimization**.
3. Enable or disable the optimization options:
 - **Generic instructions**
 - **Jump chains**
 - **Instruction size**

Command line syntax

-Oflags

--optimize=flags

You can set the following flags:

+/-generics	(g/G)	Allow generic instructions
+/-jumpchains	(j/J)	Jump chains
+/-instr-size	(s/S)	Optimize instruction size

Default: **--optimize=gJs**

Description

Allow generic instructions

If you use generic instructions in your assembly source, the assembler can optimize them by replacing it with the fastest or shortest possible variant of that instruction. By default this option is enabled. If you turn off this optimization, the assembler generates an error on generic instructions. Be aware that the compiler also generates generic instructions!

Jump chains

With this optimization, the assembler replaces chained jumps by a single jump instruction. For example, a jump from *a* to *b* immediately followed by a jump from *b* to *c*, is replaced by a jump from *a* to *c*.

Optimize instruction size

With this optimization the assembler tries to find the shortest possible operand encoding for instructions.

Related information



Section 5.5, [Assembler Optimizations](#) in chapter *Using the Assembler* of the user's manual.

Assembler: --option-file (-f)

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **Assembler** entry and select **Miscellaneous**.
3. Add the option **--option-file** to the **Additional assembler options** field.

Be aware that the options in the option file are added to the assembler options you have set in the other dialogs. Only in extraordinary cases you may want to use them in combination.

Command line syntax

```
--option-file=file,...
-f file,...
```

Description

This option is primarily intended for command line use. Instead of typing all options on the command line, you can create an option file which contains all options and flags you want to specify. With this option you specify the option file to the assembler.

Use an option file when the command line would exceed the limits of the operating system, or just to store options and save typing.

Option files can also be generated on the fly, for example by the make utility. You can specify the option **--option-file** multiple times.

Format of an option file

- Multiple arguments on one line in the option file are allowed.
- To include whitespace in an argument, surround the argument with single or double quotes.
- If you want to use single quotes as part of the argument, surround the argument by double quotes and vice versa:

```
"This has a single quote ' embedded"
```

```
'This has a double quote " embedded'
```

```
'This has a double quote " and a single quote "'" embedded"
```

- When a text line reaches its length limit, use a backslash 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:

```
-gaL
test.src
```

Specify the option file to the assembler:

```
asarm --option-file=myoptions
```

This is equivalent to the following command line:

```
asarm -gaL test.src
```

Related information



Assembler: `--output (-o)`

Menu entry

Altium Designer names the output file always after the source 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: `--page-length`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **Assembler** entry and select **Miscellaneous**.
3. Add the option `--page-length` to the **Additional assembler options** field.

Command line syntax

`--page-length=number`

Default: 72

Description

If you generate a list file with the assembler option `--list-file`, (`-l`), 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: `--page-width`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **Assembler** entry and select **Miscellaneous**.
3. Add the option `--page-width` to the **Additional assembler options** field.

Command line syntax

`--page-width=number`

Default: 132

Description

If you generate a list file with the assembler option `--list-file`, (`-l`), 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: `--preprocess` (`-E`)

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **Assembler** entry and select **Miscellaneous**.
3. Add the option `--preprocess` to the **Additional assembler options** field.

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: `--preprocessor-type (-m)`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **Assembler** entry and select **Miscellaneous**.
3. Add the option `--preprocessor-type` to the **Additional assembler options** field.

Command line syntax

`--preprocessor-type={none|tasking}`
`-m{n|t}` Default: `-mt`

Description

With this option you select the preprocessor that the assembler will use. By default, the assembler uses the TASKING preprocessor.

When the assembly source file does not contain any preprocessor symbols, you can specify to the assembler not to use a preprocessor.

Related information



Assembler: --relaxed

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **Assembler** entry and select **Miscellaneous**.
3. 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

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: `--section-info (-t)`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **Assembler** entry and select **List File**.
3. Enable **Generate list file**.
4. Enable the option **Display section information**.

Command line syntax

```
--section-info[=flags]  
-t[flags]
```

You can set the following flags:

<code>+/-console</code>	<code>(c/C)</code>	Display section information on <code>stdout</code> .
<code>+/-list</code>	<code>(l/L)</code>	Write section information to the list file.

Description

With this option you tell the assembler to display section information. For each section its memory space, size, total cycle counts and name is listed on `stdout` and/or in the list file.

The cycle count consists of two parts: the total accumulated count for the section and the total accumulated count for all repeated instructions. In the case of nested loops it is possible that the total supersedes the section total.

Without arguments this option is the same as `--section-info=cl`.



With `--section-info=l`, the assembler writes the section information to the list file. You must specify this option in combination with the option `--list-file` (generate list file).

Example

```
asarm --list-file --section-info=+console,+list test.src
```

The assembler generates a list file and writes the section information to this file. The section information is also displayed on `stdout`.

Related information



Assembler option `--list-file` (generate list file)

Assembler: `--symbol-scope (-i)`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **Assembler** entry and select **Miscellaneous**.
3. Select the default label mode: **Local** or **Global**.

Command line syntax

```
--symbol-scope={global|local}  
-i{g|l}                (Default: -il)
```

Description

With this option you tell the assembler how to treat symbols that you have not specified explicitly as global or local. By default the assembler treats all symbols as local symbols unless you have defined them explicitly as global.

Related information



Assembler: **--thumb (-T)**

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **Assembler** entry and select **Miscellaneous**.
3. Add the option **--thumb** to the **Additional assembler options** field.

Command line syntax

--thumb
-T

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. The assembler will complain if **--thumb** is used in combination with **--cpu=ARMv4**. Specifying **--thumb** with **--cpu=ARMv7M** 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](#) / [.THUMB](#) / [.ARM](#)

Assembler: `--version` (`-V`)

Menu entry

Command line only.

Command line syntax

`--version`
`-V`

Description

Displays version information of the assembler. The assembler ignores all other options or input files.

Related information



Assembler: **--verbose (-v)**

Menu entry

Command line only.

Command line syntax

--verbose
-v

Description

With this option you put the assembler in verbose mode. The assembler prints the filenames and the assembly passes while it processes the files so you can monitor the current status of the assembler.

Related information



Assembler: `--warnings-as-errors`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **Assembler** entry and select **Diagnostics**.
3. 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 compiler warnings. As a consequence, the assembler now also stops after encountering a warning.

You can also limit this option to specific warnings by specifying a comma-separated list of warning numbers.

Related information



Assembler option `--no-warnings` (Suppress some or all warnings)

6.4 Linker Options

Altium Designer uses a *makefile* to build your entire project. This means that you cannot run the linker separately. However, you can set options specific for the linker.

Options in Altium Designer versus options on the command line

Most command line options have an equivalent option in Altium Designer but some options are only available on the command line (for example in a Windows Command Prompt). If there is no equivalent option in Altium Designer, you can specify a command line option in Altium Designer as follows:

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **Linker** entry and select **Miscellaneous**.
3. Enter one or more command line options in the **Additional Linker options** field.

Invocation syntax on the command line (Windows Command Prompt)

The invocation syntax on the command line is:

```
lkarm [ [option]... [file]... ]...
```

When you are linking multiple files, either relocatable object files (`.obj` or `.o`), `.out` files or libraries (`.lib` or `.a`), it is important to specify the files in the right order. Files with extensions other than `.obj`, `.o`, `.out`, `.lib` or `.a` are not recognized as input files.

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 double minus (`--`) characters. You can abbreviate long option names as long as the name is unique. You can mix short and long option names on the command line.

Options can have flags or sub-options. To switch a flag 'on', use a lowercase letter or a *+longflag*. To switch a flag off, use an uppercase letter or a *-longflag*. Separate *longflags* with commas. The following two invocations are equivalent:

```
lkarm -mfk test.obj
```

```
lkarm --map-file-format=+files,+link test.obj
```

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

Linker: --case-insensitive

Menu entry

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

Command line syntax

--case-insensitive

Description

With this option you tell the linker not to distinguish between upper and lower case characters in symbols. By default the linker considers upper and lower case characters as different characters.



Disabling the option **Link case sensitive** in Altium Designer is the same as specifying the option **--case-insensitive** on the command line.

Assembly source files that are generated by the compiler must *always* be assembled and thus linked case sensitive. When you have written your own assembly code and specified to assemble it case insensitive, you must also link the `.obj` file case insensitive.

Related information



–

Linker: **--chip-output (-c)**

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **Linker** entry and select **Output Format**.
3. Enable the options **Intel HEX records** and/or **Motorola S-records**.

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

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). In Altium Designer you cannot specify the address size because Altium Designer always uses the default values.

Description

With this option you specify the Intel Hex or Motorola S-record output format for loading into a PROM-programmer. The linker generates a file for each ROM memory defined in the LSL file, where sections are located:

```
memory memname  
{ type=rom; }
```

The name of the file is the name of the Altium Designer project or, on the command line, the name of the memory device that was emitted with extension `.hex` or `.sre`. Optionally, you can specify a *basename* which prepends the generated file name.



The linker always outputs a debugging file in ELF/DWARF format and optionally an absolute object file in Intel Hex-format and/or Motorola S-record format.

Example

To generate Intel Hex output files for each defined memory, enter the following on the command line:

```
lkarm --chip-output=myfile:IHEX test1.obj
```

In this case, this generates the file `myfile_memname.hex`

Related information



Linker option **--output** (Output file)

Section 8.2, [Motorola S-Record Format](#),
Section 8.3, [Intel Hex Record Format](#), in Chapter *Object File Formats*.

Linker: --cpu (-C)

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **Processor** entry and select **Processor Definition**.
3. Select a processor from the **Select processor** box.

Command line syntax

```
--cpu=[ARMv4|ARMv4T|ARMv5T|ARMv5TE|ARMv6M|ARMv7M|XS]
-C[ARMv4|ARMv4T|ARMv5T|ARMv5TE|ARMv6M|ARMv7M|XS]
```

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. If the architecture is ARMv4 the linker will replace BX instructions in ARMv4T code by MOV PC instructions. If the architecture is ARMv5T, ARMv5TE or XScale the linker will replace unconditional BL instructions by BLX instructions if the branch target requires a state change between ARM and Thumb or vice versa. The default architecture is ARMv4T and the complete list of supported architectures is: ARMv4, ARMv4T, ARMv5T, ARMv5TE, ARMv6-M, ARMv7-M, XScale.

Architecture ARMv4 does not support the Thumb instruction set. Architecture ARMv7-M only supports the Thumb-2 instruction set.

Related information



Linker: **--define (-D)**

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **Linker** entry and select **Miscellaneous**.
3. Click on **User macro**, click on the down arrow in the right pane to expand macro input.
4. Click on an empty **Macro** field and enter a macro name. (Then click outside the cell to confirm)
5. Optionally, click in the **Value** field and enter a definition. (Then click outside the cell to confirm)

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** 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=file (-f)**.

The definition can be tested by the preprocessor with **#if**, **#ifdef** and **#ifndef**, for conditional locating.

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 -darm.lsl -D__STACK=32k  
-D__START=0x00000000
```

or using the long option names:

```
lkarm -otest.abs -ls1-file=arm.lsl --define=__STACK=32k  
--define=__START=0x00000000
```

Related information



Linker option **--option-file** (Read options from file)

Linker: --diag

Menu entry

1. From the **View** menu, select **Workspace Panels » System » Messages**.

The Messages panel appears.

2. In the **Messages** panel, right-click on the message you want more information on.

A popup menu appears.

3. Select **More Info**.

A Message Info box appears with additional information.

Command line syntax

--diag=[format:]{all|nr,...}

Description

With this option you can ask for an extended description of error messages in the format you choose. The output is directed to `stdout` (normally your screen) and in the format you specify. You can specify the following formats: **html**, **rtf** or **text** (default). To create a file with the descriptions, you must redirect the output.

With the suboption **all**, the descriptions of all error messages are given. If you want the description of one or more selected error messages, you can specify the error message numbers, separated by commas.

With this option the 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 `lerrors.html`, enter:

```
lkarm --diag=html:all > lerrors.html
```

Related information



Linker: `--endianness`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **Processor** entry and select **Processor Definition**.
3. Enable the option **Big-endian mode**.

Command line syntax

`--endianness=endianness`

`-B`

`--big-endian`

You can specify the following *endianness*:

big	(b)	Big endian
little	(l)	Little endian (default)

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: `--error-file`

Menu entry

-

Command line syntax

`--error-file[=file]`

Description

With this option the linker redirects error messages to a file.

If you do not specify a filename, the error file is `lkarm.elk`.

Example

To write errors to `errors.elk` instead of `stderr`, enter:

```
lkarm --error-file=errors.elk test.obj
```

Related information



Linker: **--error-limit**

Menu entry

-

Command line syntax

--error-limit=*number*

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



-

Linker: `--extern (-e)`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **Linker** entry and select **Miscellaneous**.
3. Add the option `--extern` to the **Additional linker options** field.

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 6.4, [Linking with Libraries](#), in chapter *Using the Linker* of the user's manual.

Linker: **--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: `--global-type-checking`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **Linker** entry and select **Miscellaneous**.
3. Add the option `--global-type-checking` to the **Additional linker options** field.

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 2 or DWARF 3 debug information.

This check should give the same result as the C compiler when you use MIL linking.

Related information



Linker: **--help (-?)**

Menu entry

-

Command line syntax

```
--help[=options]  
-?
```

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
```


Linker: **--hex-format**

Menu entry

-

Command line syntax

--hex-format=flag,...

You can set the following flag:

+/-start-address (s/S) Emit start address record

Default: **--hex-format=s**

Description

With this option you can specify to emit or omit the start address record from the hex file.

Related information



Linker option **--output** (Output file)

Linker: `--hex-record-size`

Menu entry

-

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 8.3, *Intel Hex Record Format*, in Chapter *Object File Formats*.

Linker: `--import-object`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **Linker** entry and select **Miscellaneous**.
3. Add one or more files to the **Import the following data object files** field.

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 6.7, [Importing Binary Files](#).

Linker: `--include-directory (-I)`

Menu entry

-

Command line syntax

`--include-directory=path,...`
`-Ipath,...`

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 `myls1.lsl` contains the following line:

```
#include "myinc.inc"
```

You can call the linker as follows:

```
lkarm --include-directory=c:\proj\include --ls1-file=myls1.lsl test.obj
```

First the linker looks in the directory where `myls1.lsl` is located for the file `myinc.inc`. If it does not find the file, it looks in the directory `c:\proj\include` for the file `myinc.inc` (this option). Finally it looks in the directory `$(PRODDIR)\include.lsl`.

Related information



Linker: **--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 -otest.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 6.5, [Incremental Linking](#) in chapter *Using the Linker* of the user's manual.

Linker: `--keep-output-files` (`-k`)

Menu entry

Altium Designer *always* removes the output files when errors occurred.

Command line syntax

`--keep-output-files`
`-k`

Description

If an error occurs during linking, the resulting output file may be incomplete or incorrect. With this option you keep the generated output files when an error occurs.

By default the linker removes the generated output file when an error occurs. This is useful when you use the make utility. If the erroneous files are not removed, the make utility may process corrupt files on a subsequent invocation.

Use this option when you still want to use the generated file. For example when you know that a particular error does not result in a corrupt object file, or when you want to inspect the output file, or send it to Altium support.

Related information



Linker: `--library (-l)`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **Linker** entry and select **Libraries**.
3. Enable the option **Link default C libraries**.

Command line syntax

`--library=name`
`-lname`

Description

With this option you tell the linker 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):

```
lkarm test.obj mylib.lib --library=carm
```

The linker links the file `test.obj` and first looks in `mylib.lib` (in the current directory only), then in the system library `carm.lib` to resolve unresolved symbols.

Related information



Linker option `--library-directory` (Additional search path for system libraries)

Section 6.4, [Linking with Libraries](#), in chapter *Using the Linker* of the user's manual.

Linker: **--library-directory (-L) / --ignore-default-library-path**

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Open the **Build Options** page.
3. Add a pathname in the **Library files path** field.

If you enter multiple paths, separate them with a semicolon (;).

Command line syntax

```
--library-directory=dir
-Ldir

--ignore-default-library-path
-L
```

Description

With this option you can specify the path(s) where your system libraries, specified with the **--library** option, are located. If you want to specify multiple paths, use the option **--library-directory** for each separate path.

The default path is \$(PRODDIR)\carm\lib.

If you specify only **-L** (without a pathname) or the long option **--ignore-default-library-path**, the linker will not search the default path and also not in the paths specified in the environment variable **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 **--library** option is:

1. The path that is specified with the **--library-directory** option.
2. The path that is specified in the environment variable **LIBARM**.
3. The default directory \$(PRODDIR)\carm\lib (or a processor specific sub-directory).

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 variable **LIBARM**.

Then the linker looks in the default directory \$(PRODDIR)\carm\lib for libraries.

Related information



Linker option **--library** (Link system library)

Section 6.4.1, [How the linker searches libraries](#) in chapter *Using the Linker* of the user's manual.

Linker: `--link-only`

Menu entry

-

Command line syntax

`--link-only`

Description

With this option you suppress the locating phase. The linker stops after linking and informs you about unresolved references.

Related information



Control program option `-cl` (Stop after linking)

Linker: `--long-branch-veneers`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **Linker** entry and select **Miscellaneous**.
3. Enable the option **Generate long-branch veneers**.

Command line syntax

`--long-branch-veneers`

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. You cannot use this option with the ARMv6-M architecture profile.

Related information



Linker: `--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=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 6.9, *Controlling the Linker with a Script*, in chapter *Using the Linker* of the user's manual.

Linker: `--lsl-dump`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **Linker** entry and select **Miscellaneous**.
3. Enable the option **Dump processor and memory info from LSL file**.

Command line syntax

`--lsl-dump[=file]`

Description

With this option you tell the linker to dump the LSL part of the map file in a separate file, independent of the option `--map-file` (generate map file). If you do not specify a filename, the file `lktarget.ldf` is used.

Related information



Linker option `--map-file-format` (Map file formatting)

Linker: `--lsl-file (-d)`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **Linker** entry and select **Miscellaneous**.
3. Enable the option **Use project specific LSL file**.
4. In the **LSL file** field, type a name or click ... and select an LSL file.

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. You can specify the existing file `arm.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 6.9, *Controlling the Linker with a Script*, in chapter *Using the Linker* of the user's manual.

Linker: --map-file (-M)

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **Linker** entry and select **Map File**.
3. Enable the option **Generate a memory map file (.map)**.
4. In the **Map file format** section, enable or disable the information you want to be included in the map file.

Command line syntax

```
--map-file[=file]  
-M[file]
```

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 **-o** option, the linker uses the same basename as the output file with the extension `.map`. If you did not specify the **-o** option, the linker uses the file `task1.map`. Altium Designer names the `.map` file after the project.

A linker map file is a text file that shows how the linker has mapped the sections and symbols from the various object files (`.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



With the option **--map-file-format** (map file formatting) you can specify which parts you want to place in the map file.

Section 7.2, [Linker Map File Format](#), in Chapter *List File Formats*.

Linker: `--map-file-format (-m)`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **Linker** entry and select **Map File**.
3. Enable the option **Generate a map file (.map)**.
4. In the **Map file format** section, enable or disable the information you want to be included in the map file.

Command line syntax

`--map-file-format=flags`
`-mflags`

You can specify the following formats:

0		Same as <code>-mcDfikLMNoQrSU</code> (link information)
1		Same as <code>-mCDfiKIMNoQRSU</code> (locate information)
2		Same as <code>-mcdfiklmNoQrSu</code> (most information)
<code>+/-callgraph</code>	<code>(c/C)</code>	Call graph information
<code>+/-removed</code>	<code>(d/D)</code>	Removed sections information
<code>+/-files</code>	<code>(f/F)</code>	Processed files information
<code>+/-invocation</code>	<code>(i/I)</code>	Invocation and tool information
<code>+/-link</code>	<code>(k/K)</code>	Link result information
<code>+/-locate</code>	<code>(l/L)</code>	Locate result information
<code>+/-memory</code>	<code>(m/M)</code>	Memory usage information
<code>+/-nonalloc</code>	<code>(n/N)</code>	Non alloc information
<code>+/-overlay</code>	<code>(o/O)</code>	Overlay information
<code>+/-statics</code>	<code>(q/Q)</code>	Module local symbols
<code>+/-crossref</code>	<code>(r/R)</code>	Cross references information
<code>+/-lsl</code>	<code>(s/S)</code>	Processor and memory information
<code>+/-rules</code>	<code>(u/U)</code>	Locate rules

Description

With this option you specify which information you want to include in the map file. Use this option in combination with the option `--map-file (-M)`.

If you do not specify this option, the linker uses the default: `--map-file-format=2`.

Related information



Linker option `--map-file` (Generate map file)

Linker: `--misra-c-report`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **C Compiler** entry and select **MISRA-C**.
3. Select a MISRA-C configuration.
4. Enable the option **Produce a MISRA-C report**.

Command line syntax

`--misra-c-report[=file]`

Description

With this option you tell the linker to create a MISRA-C Quality Assurance report. This report lists the various modules in the project with the respective MISRA-C settings at the time of compilation. If you do not specify a filename, the file *name.mcr* is used.

Related information



Compiler option `--misrac`

Linker: --munch

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog appears.
2. Expand the **Linker** entry and select **Miscellaneous**.
3. Add the option **--munch** to the **Additional linker options** field.

Command line syntax

--munch

Description

With this option you tell the linker to activate the muncher in the pre-locate phase.

Related information



Linker: `--non-romable`

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **Linker** entry and select **Miscellaneous**.
3. Add the option `--non-romable` to the **Additional linker options** field.

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.

Related information



Linker: --no-rescan

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **Linker** entry and select **Libraries**.
3. *Disable* the option **Rescan libraries to solve unresolved externals**.

Command line syntax

--no-rescan

Description

When the linker processes a library it searches for symbol definitions that are referenced by the objects and libraries processed so far. If the library contains a definition for an unresolved reference the linker extracts the object that contains the definition from the library. The linker processes object files and libraries in the order in which they appear on the command line.

When all objects and libraries are processed the linker checks if there are unresolved symbols left. If so, the default behavior of the linker is to rescan all libraries in the order given 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: **--no-rom-copy (-N)**

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **Linker** entry and select **Miscellaneous**.
3. Add the option **--no-rom-copy** to the **Additional linker options** field.

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 section. However, no entries to copy data sections from ROM to RAM are placed in the copy table.

The data sections are initialized when the application is downloaded. The data sections are not re-initialized when the application is restarted.

Related information



Linker: **--no-warnings (-w)**

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **Linker** entry and select **Diagnostics**.
3. Set **Error reporting** to one of the following values:
 - **Report all warnings**
 - **Suppress all warnings**
 - **Suppress specific warnings.**

*If you select **Suppress specific warnings**:*

4. Enter the numbers, separated by commas, of the warnings you want to suppress.

Command line syntax

```
--no-warnings[=number,...]
-w[number,...]
```

Description

With this option you can suppresses all warning messages or specific warning messages.

- If you do not specify this option, all warnings are reported.
- If you specify this option but without numbers, all warnings are suppressed.
- If you specify this option with a number, only the specified warning is suppressed. You can specify the option **--no-warnings=number** multiple times.

Example

To suppress warnings 135 and 136, enter **135, 136** in the **Specific warnings to suppress** field, or enter the following on the command line:

```
lkarm --no-warnings=135,136 test.obj
```

Related information



Linker option **--warnings-as-errors** (Treat warnings as errors)

Linker: **--optimize (-O)**

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **Linker** entry and select **Optimization**.
3. Select an optimization level in the **Optimization level** box.
*If you select **Custom Optimization**:*
4. Enable the optimizations you want.

Command line syntax

--optimize[=flags]
-O[flags]

Use the following options for predefined sets of flags:

--optimize=0	(-O0)	No optimization Alias for: -OCLTX
--optimize=1	(-O1)	Default optimization Alias for: -OCLtXY
--optimize=2	(-O2)	All optimizations Alias for: -Ocltxy

You can set the following flags:

+/-delete-unreferenced-sections	(c/C)	Delete unreferenced sections from the output file
+/-first-fit-decreasing	(l/L)	Use a 'first fit decreasing' algorithm to locate unrestricted sections in memory.
+/-copytable-compression	(t/T)	Emit smart restrictions to reduce copy table size
+/-delete-duplicate-code	(x/X)	Delete duplicate code sections from the output file
+/-delete-duplicate-data	(y/Y)	Delete duplicate constant data from the output file

Description

With this option you can control the level of optimization the linker performs. If you do not use this option, **--optimize=1** is the default.

Related information



Section 6.8, [Linker Optimizations](#), in chapter *Using the Linker* of the user's manual.

Linker: --option-file (-f)

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **Linker** entry and select **Miscellaneous**.
3. Add the option **--option-file** to the **Additional linker options** field.

Be aware that the options in the option file are added to the linker options you have set in the other dialogs. Only in extraordinary cases you may want to use them in combination. Altium Designer automatically saves the options with your project.

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 length of the command line would exceed the limits of the operating system, or just to store options and save typing.

You can specify the option **--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:

```
-Mymap      (generate a map file)
test.obj    (input file)
-Lc:\mylibs (additional search path for system libraries)
```

Specify the option file to the linker:

```
lkarm --option-file=myoptions
```

This is equivalent to the following command line:

```
lkarm -Mymap test.obj -Lc:\mylibs
```

Related information



Linker: --output (-o)

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **Linker** entry and select **Output Format**.
3. Enable one or more output formats

Command line syntax

```
--output=[filename][:format[:addr_size]]...  
-o[filename][:format[:addr_size]]...
```

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, named after the first input file with extension `.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) or **4** bytes (S3 records, default).

The name of the output file will be *filename* with the extension `.hex` or `.sre` and contains the code and data allocated in the default address space. If they exist, any other address spaces are also emitted whereas their output files are named *filename_spacename.hex* (`.sre`).



Use option **--chip-output (-c)** to create Intel Hex or Motorola S-record output files for each chip defined in the LSL file (suitable for loading into a PROM-programmer).

Example

To create the output file `myfile.hex` of the default address space:

```
lkarm test.obj --output=myfile.hex:IHEX
```

Related information



Linker option **--chip-output** (Generate an output file for each chip)

Linker: `--print-mangled-symbols` (-P)

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog appears.
2. Expand the **Linker** entry and select **Miscellaneous**.
3. Add the option `--print-mangled-symbols` to the **Additional linker options** field.

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: **--strip-debug (-S)**

Menu entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Expand the **Linker** entry and select **Miscellaneous**.
3. *Disable* the option **Include 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: `--user-provided-initialization-code (-i)`

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **Linker** entry and select **Miscellaneous**.
3. Add the option `--user-provided-initialization-code` to the **Additional linker options** field.

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: **--verbose (-v) / --extra-verbose (-vv)**

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **Linker** entry and select **Miscellaneous**.
3. Add the option **--verbose** or **--extra-verbose** to the **Additional linker options** field.

Command line syntax

--verbose / --extra-verbose
-v / -vv

Description

With this option you put the linker in *verbose* mode. The linker prints the link phases while it processes the files. In the *extra verbose* mode, the linker also prints the filenames and it shows which objects are extracted from libraries. With this option you can monitor the current status of the linker.

Related information



Linker: `--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: **--warnings-as-errors**

Menu entry

1. From the **Project** menu, select **Project Options...**

The Project Options dialog box appears.

2. Expand the **Linker** entry and select **Diagnostics**.
3. 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)

6.5 Control Program Options

The control program is a tool to facilitate use of the toolset from the command line. Therefore you can only call the control program from the command line. The invocation syntax is:

```
ccarm [option]... [file]...
```

Options

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**.

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 double minus (--) characters. You can abbreviate long option names as long as the name is unique. You can mix short and long option names on the command line.

Options can have flags or sub-options. To switch a flag 'on', use a lowercase letter or a *+longflag*. To switch a flag off, use an uppercase letter or a *-longflag*. Separate *longflags* with commas. The following two invocations are equivalent:

```
ccarm -Wc-Oac test.c  
ccarm --pass-c=--optimize=+coalescer,+cse test.c
```

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

Control Program: `--address-size`

Command line syntax

`--address-size=addr_size`

Description

If you specify IHEX or SREC with the control option `--format`, you can additionally specify the record length to be emitted in the output files.

With this option you can specify the size of addresses in bytes (record length). For Intel Hex you can use the values: **1**, **2**, and **4** (default). For Motorola S-records you can specify: **2** (S1 records), **3** (S2 records) 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)

[Linker option `--output`](#) (Specify an output object file)

Control Program: `--check`

Command line syntax

`--check`

Description

With this option you can check the source code for syntax errors, without generating code. This saves time in developing your application.

The compiler/assembler reports any warnings and/or errors.

Related information



C compiler option `--check` (Check syntax)

Assembler option `--check` (Check syntax)

Control Program: `--cpu (-C)`

Command line syntax

```
--cpu=[ARMv4|ARMv4T|ARMv5T|ARMv5TE|ARMv6M|ARMv7M|XS]  
-C[ARMv4|ARMv4T|ARMv5T|ARMv5TE|ARMv6M|ARMv7M|XS]
```

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. If the architecture is ARMv4 the linker replaces BX instructions by MOV PC instructions. The default architecture is ARMv4T and the complete list of supported architectures is: ARMv4, ARMv4T, ARMv5T, ARMv5TE, ARMv6-M, ARMv7-M or XScale.

Assembly code can check the value of the option by means of the built-in function `@CPU()`. Architecture ARMv4 does not support the Thumb instruction set. Architecture profile ARMv7-M only supports the Thumb-2 instruction set, i.e. it has no ARM execution state.

Related information



C compiler option `--cpu` (Select architecture)

Assembler option `--cpu` (Select architecture)

Control Program: **--create** (**-cc/-cl/-cm/-co/-cs**)

Command line syntax

--create[=*stage*]
-c[*stage*]

You can specify the following stages (if you omit the *stage*, the default is **--create=object**):

intermediate-c	(c)	Stop after C++ files are compiled to intermediate C files (<i>.ic</i>)
relocatable	(l)	Stop after the files are linked to a linker object file (<i>.out</i>)
mil	(m)	Stop after C files are compiled to MIL (<i>.mil</i>)
object	(o)	Stop after the files are assembled to objects (<i>.obj</i>)
assembly	(s)	Stop after C files are compiled to assembly (<i>.src</i>)

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.

Related information



Linker option **--link-only** (Link only, no locating)

Control Program: `--debug-info (-g)`

Command line syntax

`--debug-info`
`-g`

Description

With this option you tell the control program to include debug information in the generated object file.

Related information



Control Program: **--define (-D)**

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. On the command line, use the option **--define** multiple times. If the command line exceeds the length limit of the operating system, you can define the macros in an *option file* which you then must specify to the control program with the option **--option-file=file (-f)**.

Defining macros with this option (instead of in the C source) is, for example, useful to compile or assemble conditional source as shown in the example below.

The control program passes the option **--define (-D)** to the compiler and the assembler.

Example

Consider the following C program with conditional code to compile a demo program and a real program:

```
void main( void )
{
    #if DEMO == 1
        demo_func();    /* compile for the demo program */
    #else
        real_func();    /* compile for the real program */
    #endif
}
```

You can now use a macro definition to set the DEMO flag. With the control program this looks as follows:

```
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 -D"MAX(A,B)=((A) > (B) ? (A) : (B))" test.c
```

Related information



[Control Program option **--undefine**](#) (Undefine preprocessor macro)

[Control Program option **--option-file**](#) (Read options from file)

Control Program: **--dep-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

```
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: --diag

Command line syntax

```
--diag=[format:]{all|nr,...}
```

Description

With this option you can ask for an extended description of error messages in the format you choose. The output is directed to `stdout` (normally your screen) and in the format you specify. You can specify the following formats: **html**, **rtf** or **text** (default). To create a file with the descriptions, you must redirect the output.

With the suboption **all**, the descriptions of all error messages are given. If you want the description of one or more selected error messages, you can specify the error message numbers, separated by commas.

With this option the control program does not process any files.

Example

To display an explanation of message number 103, enter:

```
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`, enter:

```
ccarm --diag=html:all > ccerrors.html
```

Related information



-

Control Program: `--dry-run (-n)`

Command line syntax

`--dry-run`
`-n`

Description

With this option you put the control program *verbose* mode. The control program prints the invocations of the tools it would use to process the files without actually performing the steps.

Related information



Control Program option `--verbose (-v)` (Verbose output)

Control Program: `--endianness`

Command line syntax

`--endianness=endianness`

`-B`

`--big-endian`

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: `--error-file`

Command line syntax

`--error-file`

Description

With this option the control program tells the compiler, assembler and linker to redirect error messages to a file.

The error file will be named after the input file with extension `.err` (for compiler) or `.ers` (for assembler). For the linker, the error file is `lkarm.elk`.

Example

To write errors to error files instead of `stderr`, enter:

```
ccarm --error-file -t test.c
```

Related information



[Control Program option `--warnings-as-errors`](#) (Treat warnings as errors)

Control Program: `--exceptions`

Command line syntax

`--exceptions`

Description

With this option you enable support for exception handling in the C++ compiler.

Related information



Control Program: **--force-c**

Command line syntax

--force-c

Description

With this option you tell the control program to treat all `.cc` files as C files instead of C++ files. This means that the control program does not call the C++ compiler and forces the linker to link C libraries.

Related information



Control program option **--force-c++** (Force C++ compilation and linking)

Control Program: `--force-c++`

Command line syntax

`--force-c++`

Description

With this option you tell the control program to treat all `.c` files as C++ files instead of C files. This means that the control program calls the C++ compiler prior to the C compiler and forces the linker to link C++ libraries.

Related information



Control program option `--force-c` (Treat C++ files as C files)

Control Program: `--force-munch`

Command line syntax

`--force-munch`

Description

With this option you force the control program to activate the muncher in the pre-locate phase.

Related information



Control Program: **--format**

Command line syntax

--format=format

You can specify the following formats:

IEEE IEEE-695
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 an 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)

Linker option **--output** (Specify an output object file)

Linker option **--chip-output** (Generate hex file for each chip)

Control Program: **--global-type-checking**

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 **--global-type-checking** (Global type checking)

Control Program: --help (-?)

Command line syntax

```
--help[=options]  
-?
```

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
```

Control Program: `--include-directory (-I)`

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.

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)

Section 2.4, [How the Compiler Searches Include Files](#), in chapter *Using the Compiler* of the user's manual.

Control Program: `--instantiate`

Command line syntax

`--instantiate=mode`

You can specify the following modes:

used (default)
all
local

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



Control program option `--no-auto-instantiation` (Disable automatic C++ instantiation)
 Section 2.6, [Template Instantiation](#) in chapter *C++ Language*.

Control Program: `--io-streams`

Command line syntax

`--io-streams`

Description

As I/O streams require substantial resources they are disabled by default. Use this option to enable I/O streams support in the C++ library.

This option also enables exception handling.

Related information



Control Program: `--iso`

Command line syntax

`--iso={90|99}`

Description

With this option you specify to the control program against which ISO standard it should check your C source. C90 is also referred to as the "ANSI C standard". C99 refers to the newer ISO/IEC 9899:1999 (E) standard and is the default.



Independant of the chosen ISO standard, the control program always links libraries with C99 support.

Example

To compile the file `test.c` conform the ISO C90 standard:

```
ccarm --iso=90 test.c
```

Related information



C compiler option `--iso` (ISO C standard)

Control Program: `--keep-output-files (-k)`

Command line syntax

`--keep-output-files`
`-k`

Description

If an error occurs during the compilation, assembling or linking process, the resulting output file may be incomplete or incorrect. With this option you keep the generated output files when an error occurs.

By default the control program removes generated output files when an error occurs. This is useful when you use the make utility. If the erroneous files are not removed, the make utility may process corrupt files on a subsequent invocation.

Use this option when you still want to use the generated files. For example when you know that a particular error does not result in a corrupt file, or when you want to inspect the output file, or send it to Altium support.

Related information



Control Program: `--keep-temporary-files (-t)`

Menu Entry

1. From the **Project** menu, select **Project Options...**
The Project Options dialog box appears.
2. Select **Build Options**.
3. Enable the option **Keep temporary files that are generated during a compile**.

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.

Related information



Control Program: **--library (-l)**

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):

```
ccarm test.obj mylib.lib --library=carm
```

The linker links the file `test.obj` and first looks in `mylib.lib` (in the current directory only), then in the system library `carm.lib` to resolve unresolved symbols.

Related information



[Linker option **--library-directory**](#) (Additional search path for system libraries)

Section 6.4, [Linking with Libraries](#), in chapter *Using the Linker* of the user's manual.

Control Program: **--library-directory (-L) / --ignore-default-library-path**

Command line syntax

```
--library-directory=dir
-Ldir

--ignore-default-library-path
-L
```

Description

With this option you can specify the path(s) where your system libraries, specified with the **--library** option, are located. If you want to specify multiple paths, use the option **--library-directory** for each separate path.

By default path this is \$(PRODDIR)\carm\lib directory.

If you specify only **-L** (without a pathname) or the long option **--ignore-default-library-path**, the linker will not search the default path and also not in the paths specified in the environment variable **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 **--library** option is:

1. The path that is specified with the **--library-directory** option.
2. The path that is specified in the environment variable **LIBARM**.
3. The default directory \$(PRODDIR)\carm\lib (or a processor specific sub-directory).

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 variable **LIBARM**.

Then the linker looks in the default directory \$(PRODDIR)\carm\lib for libraries.

Related information



Linker option **--library** (Link system library)

Control Program: `--list-files`

Command line syntax

`--list-files[=name]`

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 *name* 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 *name*, or you specify more than one input files, the control program names the generated list file(s) after the specified input file(s) with extension `.lst`.

Example

This example generates the list files `1.lst` and `2.lst` for `1.c` and `2.c`. If in this example also a *name* had been specified, it would be ignored because two input files are specified.

```
ccarm 1.c 2.c --list-files
```

Related information



[Assembler option `--list-file`](#) (Generate list file)

[Assembler option `--list-format`](#) (List file formatting options)

Control Program: `--lsl-file (-d)`

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 and derivative definition describe the core's hardware architecture and its internal memory.
- the board specification describes the physical memory available in the system.
- the section layout definition describes how to locate sections in memory.

With this option you specify a linker script file via the control program to the linker. If you do not specify this option, the linker does not use a script file. You can specify the existing file `arm.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 6.9, [Controlling the Linker with a Script](#), in chapter *Using the Linker* of the user's manual.

Control Program: **--make-target**

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: **--mil-link** / **--mil-split**

Command line syntax

--mil-link
--mil-split

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.

Related information



C compiler option **--mil** / **--mil-split**

Control Program: `--no-auto-instantiation`

Command line syntax

`--no-auto-instantiation`

Default: the C++ compiler automatically instantiates templates.

Description

With this option automatic instantiation of templates is disabled.

Related information



Control program option `--instantiate` (Set instantiation mode)
Section 2.6, [Template Instantiation](#) in chapter *C++ Language*.

Control Program: `--no-default-libraries`

Command line syntax

`--no-default-libraries`

Description

By default the control program specifies the standard C libraries (C99) and run-time library to the linker. With this option you tell the control program *not* to specify the standard C libraries and run-time library to the linker.

In this case you must specify the libraries you want to link to the linker with the option `-llibrary_name`. The control program recognizes the option `-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 (`libmy.a`) and avoid unresolved externals:

```
ccarm --no-default-libraries -lmy test.c
```

Related information



Linker option `--library (-l)` (Add library)

Control Program: `--no-double (-F)`

Command line syntax

`--no-double`

`-F`

Description

With this option you tell the compiler to treat variables of the type `double` as `float`. Because the `float` type takes less space, execution speed increases and code size decreases, both at the cost of less precision.

Related information



Control Program: --no-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: **--no-preprocessing-only**

Command line syntax

--no-preprocessing-only

Description

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.

Example

```
ccarm --preprocess=+comments,-make,-noline --no-preprocessing-only test.c
```

Related information



Control program option **--preprocess / -E**

Control Program: **--no-warnings (-w)**

Command line syntax

```
--no-warnings[=number[-number],...]  
-w[number[-number],...]
```

Description

With this option you can suppress all warning messages or specific C compiler warning messages:

- If you do not specify this option, all warnings are reported.
- If you specify this option but without numbers, all warnings are suppressed.
- If you specify this option with a number or a range, only the specified C compiler warning is suppressed.
You can specify the option **--no-warnings=*number*** multiple times.

Related information



Control Program: `--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 control program.

Use an option file when the command line would exceed the limits of the operating system, or just to store options and save typing.

You can specify the option `--option-file` multiple times.

Format of an option file

- Multiple arguments on one line in the option file are allowed.
- To include whitespace in an argument, surround the argument with single or double quotes.
- If you want to use single quotes as part of the argument, surround the argument by double quotes and vice versa:

```
"This has a single quote ' embedded"
```

```
'This has a double quote " embedded'
```

```
'This has a double quote " and a single quote ''' embedded"
```

- When a text line reaches its length limit, use a `\` to continue the line. Whitespace between quotes is preserved.

```
"This is a continuation \  
line"
```

```
-> "This is a continuation line"
```

- It is possible to nest command line files up to 25 levels.

Example

Suppose the file `myoptions` contains the following lines:

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

Specify the option file to the control program:

```
ccarm --option-file=myoptions
```

This is equivalent to the following command line:

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

Related information



Control Program: `--output (-o)`

Command line syntax

`--output=file`
`-o file`

Description

Default, the control program generates a file with the same basename as the first specified input file. With this option you specify another name for the resulting absolute object file.

Example

```
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: **--pass (-W)**

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	(-Wc++option)	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.

Related information



Control Program: **--preprocess (-E)**

Command line syntax

```
--preprocess[=flags]
-E[flags]
```

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

+/-comments	(c/C)	Keep comments from the C source in the preprocessed output
+/-includes	(i/I)	Generate a list of included source files
+/-list	(l/L)	Generate a list of macro definitions
+/-make	(m/M)	Generate dependency lines that can be used for the makefile
+/-noline	(p/P)	Strip #line source position info (lines starting with #line)

Description

With this option you tell the control program 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 being compiled). Altium Designer 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

```
ccarm --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 **--no-preprocessing-only**

Control program option **--make-target** (Specify target name for **-Em** output)

Control program option **--dep-file** (Generate dependencies in a file)

Control Program: `--profile (-p)`

Command line syntax

```
--profile[=flags]
-p[flags]
```

Use the following option for a predefined set of flags:

```
--profile=g      (-pg)      profiling with call graph and function timers
                        Alias for: -pBCfST
```

You can set the following flags (when you specify `-p` without flags, the default is `-pBCfST`):

<code>+/-block</code>	<code>(b/B)</code>	block counters
<code>+/-callgraph</code>	<code>(c/C)</code>	call graph
<code>+/-function</code>	<code>(f/F)</code>	function counters
<code>+/-static</code>	<code>(s/S)</code>	static profile generation
<code>+/-time</code>	<code>(t/T)</code>	function timers

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.



For an extensive description of profiling refer to Chapter 4, *Profiling* in the user's manual.

With this option, the compiler adds the extra code to your application that takes care of the profiling process. You can obtain the following profiling data (see flags above):

Block counters (not in combination with Call graph or Time)

This will instrument the code to perform basic block counting. As the program runs, it counts the number of executions of each branch in an `if` statement, each iteration of a `for` loop, and so on. Note that though you can combine Block counters with Function counters, this has no effect because Function counters is only a subset of Block counters.

Call graph (not in combination with Block counters)

This will instrument the code to reconstruct the run-time call graph. As the program runs it associates the caller with the gathered profiling data.

Function counters

This will instrument the code to perform function call counting. This is a subset of the basic Block counters.

Time (not in combination with Block counters)

This will instrument the code to measure the time spent in a function. This includes the time spent in all sub functions (callees).



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 Debug information** (`-g` or `--debug`) does not affect profiling, execution time or code size.



The control program automatically specifies the corresponding profiling libraries to the linker.

Example

To generate block count information for the module `test.c` during execution, compile as follows:

```
ccarm --profile=+block test.c
```

In this case the library `pbarm.lib` is linked.

Related information



Chapter 4, *Profiling* in the user's manual.

Control Program: `--show-c++-warnings`

Command line syntax

`--show-c++-warnings`

Description

The C++ compiler may generate a compiled C++ file (.ic) that causes warnings during compilation or assembling. With this option you tell the control program to show these warnings. By default, warnings in C++ generated files are suppressed.

Related information



Control Program: `--signed-bitfields`

Command line syntax

`--signed-bitfields`

Description

For bit-fields it depends on the implementation whether a plain `int` is treated as `signed int` or `unsigned int`. By default an `int` bit-field is treated as `unsigned int`. This offers the best performance. With this option you tell the compiler to treat `int` bit-fields as `signed int`. In this case, you can still add the keyword `unsigned` to treat a particular `int` bit-field as `unsigned`.

This option is passed to both the C++ compiler and the C compiler.

Related information



Control Program: `--static`

Command line syntax

`--static`

Description

This option is directly passed to the compiler.

With this option, the compiler treats external definitions at file scope (except for `main`) as if they were declared `static`. As a result, unused functions will be eliminated, and the alias checking algorithm assumes that objects with static storage cannot be referenced from functions outside the current module.

This option only makes sense when you specify all modules of an application on the command line.

Example

```
ccarm --static module1.c module2.c module3.c
```

Related information



Control Program: **--thumb**

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.

Related information



Control Program: `--uchar (-u)`

Command line syntax

`-u`
`--uchar`

Description

By default `char` is the same as specifying `signed char`. With this option `char` is the same as `unsigned char`. This option is passed to both the C++ compiler and the C compiler.

Related information



Control Program: **--undefine (-U)**

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. However, you cannot undefine predefined ISO C standard macros.

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)

Control Program: **--verbose (-v)**

Command line syntax

--verbose
-v

Description

With this option you put the control program in verbose mode. With the option **-v** the control program performs its tasks while it prints the steps it performs to `stdout`.

Related information



Control Program option **-n (--dry-run)** (Verbose output and suppress execution)

Control Program: `--version` (`-V`)

Command line syntax

`--version`
`-V`

Description

Display version information. The control program ignores all other options or input files.

Related information



Control Program: **--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, only the specified control program warning is 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 all warnings)

Control program option **--pass** (Pass option to tool)

6.6 Make Utility Options

When you build a project in Altium Designer, Altium Designer generates a makefile and uses the make utility **tmk** 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:

```
tmk [option...] [target...] [macro=def]
```

This section describes all options for the make utility. The make utility is a command line tool so there are no equivalent options in Altium Designer.

Defining Macros

Command line syntax

macro=*definition*

Description

With this argument you can define a macro and specify it to the make utility.

A macro definition remains in existence during the execution of the makefile, even when the makefile recursively calls the make utility again. In the recursive call, the macro acts as an environment variable. This means that it is overruled by definitions in the recursive call. Use the option **-e** to prevent this.

You can specify as many macros as you like. If the command line exceeds the limit of the operating system, you can define the macros in an *option file* which you then must specify to the make utility with the option **-m file**.

Defining macros on the command line is, for example, useful in combination with conditional processing as shown in the example below.

Example

Consider the following makefile with conditional rules to build a demo program and a real program:

```
ifdef DEMO      # the value of DEMO is of no importance
    real.abs : real.obj main.obj
                lkarm demo.obj main.obj -darm.lsl -lcarm -lfparm
else
    real.abs : real.obj main.obj
                lkarm real.obj main.obj -darm.lsl -lcarm -lfparm
endif
```

You can now use a macro definition to set the DEMO flag:

```
tmk real.abs DEMO=1
```

In both cases the absolute object file `real.abs` is created but depending on the DEMO flag it is linked with `demo.obj` or with `real.obj`.

Related information



Make utility option **-e** (Environment variables override macro definitions)

Make utility option **-m** (Name of invocation file)

Make Utility: -?

Command line syntax

-?

Description

Displays an overview of all command line options.

Example

The following invocation displays a list of the available command line options:

```
tmk -?
```

Related information



Make Utility: -a

Command line syntax

-a

Description

Normally the make utility rebuilds only those files that are out of date. With this option you tell the make utility to rebuild *all* files, without checking whether they are out of date.

Example

```
tmk -a
```

Rebuilds all your files, regardless of whether they are out of date or not.

Related information



Make Utility: **-c**

Command line syntax

-c

Description

Altium Designer uses this option for the graphical version of the make utility when you create sub-projects. In this case the make utility calls another instance of the make utility for the sub-project. With the option **-c**, the make utility runs as a child process of the current make.

The option **-c** overrides the option **-err**.

Example

```
tmk -c
```

The make utility runs its commands as a child processes.

Related information



Make Utility: **-D/-DD**

Command line syntax

-D
-DD

Description

With the option **-D** the make utility prints every line of the makefile to standard output as it is read by **tmk**.

With the option **-DD** not only the lines of the makefile are printed but also the lines of the `tmk.mk` file (implicit rules).

Example

```
tmk -D
```

Each line of the makefile that is read by the make utility is printed to standard output (usually your screen).

Related information



Make Utility: **-d/-dd**

Command line syntax

-d
-dd

Description

With the option **-d** the make utility shows which files are out of date and thus need to be rebuild. The option **-dd** gives more detail than the option **-d**.

Example

```
tmk -d
```

Shows which files are out of date and rebuilds them.

Related information



Make Utility: **-e**

Command line syntax

-e

Description

If you use macro definitions, they may overrule the settings of the environment variables.

With the option **-e**, the settings of the environment variables are used even if macros define otherwise.

Example

```
tmk -e
```

The make utility uses the settings of the environment variables regardless of macro definitions.

Related information



Make Utility: **-err**

Command line syntax

-err *file*

Description

With this option the make utility redirects error messages and verbose messages to a specified file.

With the option **-s** the make utility only displays error messages.

Example

```
tmk -err error.txt
```

The make utility writes messages to the file `error.txt`.

Related information



[Make utility option **-s**](#) (Do not print commands before execution)

Make Utility: **-f**

Command line syntax

-f *my_makefile*

Description

Default the make utility uses the file `makefile` to build your files.

With this option you tell the make utility to use the specified file instead of the file `makefile`. Multiple **-f** options act as if all the makefiles were concatenated in a left-to-right order.

Example

```
tmk -f mymake
```

The make utility uses the file `mymake` to build your files.

Related information



Make Utility: -G

Command line syntax

-G *path*

Description

Normally you must call the make utility **tmk** from the directory where your makefile and other files are stored.

With the option **-G** you can call the make utility from within another directory. The *path* is the path to the directory where your makefile and other files are stored and can be absolute or relative to your current directory.

Example

Suppose your makefile and other files are stored in the directory `..\myfiles`. You can call the make utility, for example, as follows:

```
tmk -G ..\myfiles
```

Related information



Make Utility: -i

Command line syntax

-i

Description

When an error occurs during the make process, the make utility exits with a certain exit code.

With the option -i, the make utility exits without an error code, even when errors occurred.

Example

```
tmk -i
```

The make utility exits without an error code, even when an error occurs.

Related information



Make Utility: -K

Command line syntax

-K

Description

With this option the make utility keeps temporary files it creates during the make process. The make utility stores temporary files in the directory that you have specified with the environment variable TMPDIR or in the default 'temp' directory of your system when the TMPDIR environment variable is not specified.

Example

```
tmk -K
```

The make utility preserves all temporary files.

Related information



Make Utility: **-k**

Command line syntax

-k

Description

When during the make process the make utility encounters an error, it stops rebuilding your files.

With the option **-k**, the make utility only stops building the target that produced the error. All other targets defined in the makefile are built.

Example

```
tmk -k
```

If the make utility encounters an error, it stops building the current target but proceeds with the other targets that are defined in the makefile.

Related information



Make utility option **-S** (Undo the effect of **-k**)

Make Utility: -m

Command line syntax

`-m file`

Description

Instead of typing all options on the command line, you can create an option file which contains all options and flags you want to specify. With this option you specify the option file to the make utility.

Use an option file when the command line would exceed the limits of the operating system, or just to store options and save typing.

You can specify the option `-m` multiple times.

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 backslash to continue the line. Whitespace between quotes is preserved.

```
"This is a continuation \  
line"
```

```
-> "This is a continuation line"
```

- It is possible to nest command line files up to 25 levels.

Example

Suppose the file `myoptions` contains the following lines:

```
-k  
-err errors.txt  
test.abs
```

Specify the option file to the make utility:

```
tmk -m myoptions
```

This is equivalent to the following command line:

```
tmk -k -err errors.txt test.abs
```

Related information



-

Make Utility: **-n**

Command line syntax

-n

Description

With this option you tell the make utility to perform a *dry run*. The make utility shows what it would do but does not actually perform these tasks.

This option is for example useful to quickly inspect what would happen if you call the make utility.

Example

```
tmk -n
```

The make utility does not perform any tasks but displays what it would do if called without the option **-n**.

Related information



[Make utility option **-s**](#) (Do not print commands before execution)

Make Utility: -p

Command line syntax

-p

Description

Normally, if a command in a target rule in a makefile returns an error or when the target construction is interrupted, the make utility removes that target file. With this option you tell the make utility to make all target files precious. This means that dependency files are never removed.

Example

```
tmk -p
```

The make utility never removes target dependency files.

Related information



Make Utility: -q

Command line syntax

-q

Description

With this option the make utility does not perform any tasks but only returns an exit code. A zero status indicates that all target files are up to date, a non-zero status indicates that some or all target files are out of date.

Example

```
tmk -q
```

The make utility only returns an exit code that indicates whether all target files are up to date or not. It does not rebuild any files.

Related information



Make Utility: -r

Command line syntax

-r

Description

When you call the make utility, it first reads the implicit rules from the file `tmk.mk`, then it reads the makefile with the rules to build your files. (The file `tmk.mk` is located in the `\etc` directory of the toolset.)

With this option you tell the make utility *not* to read `tmk.mk` and to rely fully on the make rules in the makefile.

Example

```
tmk -r
```

The make utility does not read the implicit make rules in `tmk.mk`.

Related information



Make Utility: -S

Command line syntax

-S

Description

With this option you cancel the effect of the option **-k**. This is only necessary in a recursive make where the option **-k** might be inherited from the top-level make via MAKEFLAGS or if you set the option **-k** in the environment variable MAKEFLAGS.

Example

```
tmk -S
```

The effect of the option **-k** is cancelled so the make utility stops with the make process after it encounters an error.

The option **-k** in this example may have been set with the environment variable MAKEFLAGS or in a recursive call to **tmk** in the makefile.

Related information



[Make utility option -k](#) (On error, abandon the work for the current target only)

Make Utility: -s

Command line syntax

-s

Description

With this option you tell the make utility to perform its tasks without printing the commands it executes. Error messages are normally printed.

Example

```
tmk -s
```

The make utility rebuilds your files but does not print the commands it executes during the make process.

Related information



[Make utility option -n](#) (Perform a dry run)

Make Utility: -t

Command line syntax

-t

Description

With this option you tell the make utility to *touch* the target files, bringing them up to date, rather than performing the rules to rebuild them.

Example

```
tmk -t
```

The make utility updates out-of-date files by giving them a new date and time stamp. The files are not actually rebuild.

Related information



Make Utility: -time

Command line syntax

-time

Description

With this option you tell the make utility to display the current date and time on standard output.

Example

```
tmk -time
```

The make utility displays the current date and time and updates out-of-date files.

Related information



Make Utility: -V

Command line syntax

-V

Description

Display version information. The make utility ignores all other options or input files.

Example

```
tmk -V
```

The make utility displays the version information but does not perform any tasks.

Related information



Make Utility: -W

Command line syntax

`-W target`

Description

With this option the make utility considers the specified target file always as up to date and will not rebuild it.

Example

```
tmk -W test.abs
```

The make utility rebuilds out of date targets in the makefile except the file `test.abs` which is considered now as up to date.

Related information



Make Utility: -x

Command line syntax

-x

Description

With this option the make utility shows extended error messages. Extended error messages give more detailed information about the exit status of the make utility after errors. Altium Designer uses this option for the graphical version of make.

Example

```
tmk -x
```

If errors occur, the make utility gives extended information.

Related information



6.7 Librarian Options

The librarian **tlb** is a tool to build library files and it offers the possibility to replace, extract and remove modules from an existing library.

You can only call the librarian from the command line. The invocation syntax is:

```
tlb key_option [sub_option...] library [object_file]
```

This section describes all options for the make utility. Suboptions can only be used in combination with certain key options. Keyoptions and their suboptions are therefor described together. The miscellaneous options can always be used and are also described separately.

The librarian is a command line tool so there are no equivalent options in Altium Designer.

Description	Option	Suboption
Main functions (key options)		
Replace or add an object module	-r	-a -b -c -u -v
Extract an object module from the library	-x	-o -v
Delete object module from library	-d	-v
Move object module to another position	-m	-a -b -v
Print a table of contents of the library	-t	-s0 -s1
Print object module to standard output	-p	
Suboptions		
Append or move new modules after existing module <i>name</i>	-a name	
Append or move new modules before existing module <i>name</i>	-b name	
Create library without notification if library does not exist	-c	
Preserve last-modified date from the library	-o	
Print symbols in library modules	-s{0 1}	
Replace only newer modules	-u	
Verbose	-v	
Miscellaneous		
Display options	-?	
Display version header	-V	
Read options from <i>file</i>	-f file	
Suppress warnings above level <i>n</i>	-wn	

Table 6–1: Overview of librarian options and suboptions

Librarian: -?

Command line syntax

-?

Description

Displays an overview of all command line options.

Example

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

```
tlb -?  
tlb
```

Related information



Librarian: -d

Command line syntax

-d [-v]

Description

Delete the specified object modules from a library. With the suboption **-v** the librarian shows which files are removed.

-v Verbose: the librarian shows which files are removed.

Example

```
tlb -d mylib.lib obj1.obj obj2.obj
```

The librarian deletes `obj1.obj` and `obj2.obj` from the library `mylib.lib`.

```
tlb -d -v mylib.lib obj1.obj obj2.obj
```

The librarian deletes `obj1.obj` and `obj2.obj` from the library `mylib.lib` and displays which files are removed.

Related information



Librarian: -f

Command line syntax

-f *file*

Description

Instead of typing all options on the command line, you can create an option file which contains all options and flags you want to specify. With this option you specify the option file to the librarian **tlb**.

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 **-f** 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 backslash 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 librarian:

```
tlb -f myoptions
```

This is equivalent to the following command line:

```
tlb -x mylib.lib obj1.obj -w5
```


Librarian: -m

Command line syntax

-m [**-a** *posname*] [**-b** *posname*]

Description

Move the specified object modules to another position in the library.

The ordering of members in a library can make a difference in how programs are linked if a symbol is defined in more than one member.

Default, the specified members are moved to the end of the archive. Use the suboptions **-a** or **-b** to move them to a specified place instead.

-a *posname* Move the specified object module(s) after the existing module *posname*.

-b *posname* Move the specified object module(s) before the existing module *posname*.

Example

Suppose the library `mylib.lib` contains the following objects (see option **-t**):

```
obj1.obj  
obj2.obj  
obj3.obj
```

To move `obj1.obj` to the end of `mylib.lib`:

```
tlb -m mylib.lib obj1.obj
```

To move `obj3.obj` just before `obj2.obj`:

```
tlb -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



Librarian option **-t** (Print library contents)

Librarian: -p

Command line syntax

-p

Description

Print the specified object module(s) in the library to standard output.

This option is only useful when you redirect or pipe the output to other files or tools that serve your own purposes. Normally you do not need this option.

Example

```
tlb -p mylib.lib obj1.obj > file.obj
```

The librarian 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



Librarian: -r

Command line syntax

```
-r [-a posname] [-b posname] [-c] [-u] [-v]
```

Description

You can use the option **-r** for several purposes:

- Adding new objects to the library
- Replacing objects in the library with the same object of a newer date
- Creating a new library

The option **-r** normally *adds* a new module to the library. However, if the library already contains a module with the specified name, the existing module is *replaced*. If you specify a library that does not exist, the librarian *creates* a new library with the specified name.

If you add a module to the library without specifying the suboption **-a** or **-b**, the specified module is added at the end of the archive. Use the suboptions **-a** or **-b** to insert them to a specified place instead.

-a posname Add the specified object module(s) after the existing module *posname*.

-b posname Add the specified object module(s) before the existing module *posname*.

-c Create a new library without checking whether it already exists. If the library already exists, it is overwritten.

-u Insert the specified object module only if it is newer than the module in the library.

-v Verbose: the librarian shows which files are removed.



The suboptions **-a** or **-b** have no effect when an object is added to the library.

Examples

Suppose the library `mylib.lib` contains the following objects (see option **-t**):

```
obj1.obj
```

To add `obj2.obj` to the end of `mylib.lib`:

```
tlb -r mylib.lib obj2.obj
```

To insert `obj3.obj` just before `obj2.obj`:

```
tlb -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:

```
tlb -r obj1.obj newlib.lib
```

The librarian 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 suboption **-c**:

```
tlb -r -c obj1.obj mylib.lib
```

The librarian overwrites the library `mylib.lib` and adds the object `obj1.obj` to it. The new library `mylib.lib` only contains `obj1.obj`.

Related information



[Librarian option `-t`](#) (Print library contents)

Librarian: -t

Command line syntax

-t [-s0|-s1]

Description

Print a table of contents of the library to standard out. With the suboption **-s** the librarian displays all symbols per object file.

-s0 Displays per object the library in which it resides, the name of the object itself and all symbols in the object.

-s1 Displays only the symbols of all object files in the library.

Example

```
tlb -t mylib.lib
```

The librarian prints a list of all object modules in the library `mylib.lib`.

```
tlb -t -s0 mylib.lib
```

The librarian prints per object all symbols in the library. This looks like:

```
prolog.obj
  symbols:
mylib.lib:prolog.obj:___Qabi_callee_save
mylib.lib:prolog.obj:___Qabi_callee_restore
div16.obj
  symbols:
mylib.lib:div16.obj:___udiv16
mylib.lib:div16.obj:___div16
mylib.lib:div16.obj:___urem16
mylib.lib:div16.obj:___rem16
```

Related information



Librarian: -V

Command line syntax

-V

Description

Display version information. The librarian ignores all other options or input files.

Example

```
tlb -V
```

The librarian displays version information but does not perform any tasks.

Related information



-

Librarian: **-w**

Command line syntax

-w*level*

Description

With this suboption you tell the librarian 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:

```
tlb -x -w5 mylib.lib obj1.obj
```

Related information



Librarian: **-x**

Command line syntax

-x [-o] [-v]

Description

Extract an existing module from the library.

- o** Give the extracted object module the same date as the last-modified date that was recorded in the library.
Without this suboption it receives the last-modified date of the moment it is extracted.
- v** Verbose: the librarian shows which files are extracted.

Examples

To extract the file `obj1.obj` from the library `mylib.lib`:

```
tlb -x mylib.lib obj1.obj
```

If you do not specify an object module, all object modules are extracted:

```
tlb -x mylib.lib
```

Related information





7 List File Formats

Summary

This chapter describes the format of the assembler list file and the linker map file.

7.1 Assembler List File Format

The assembler list file is an additional output file of the assembler that contains information about the generated code.

The list file consists of a page header and a source listing.

Page header

The page header is repeated on every page:

```
TASKING target Assembler vx.yrz Build nnn SN 00000000
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. With the assembler directives `.LIST/ .NOLIST` and `.PAGE`, and with the [assembler option `-Lflag \(--list-format\)`](#) you can format the list file.



See Section 4.8.2, [Assembler Directives](#) in Chapter *Assembly Language* and Section 6.3, [Assembler Options](#) in Chapter *Tools Options*.

The fourth line contains the headings of the columns for the source listing.

Source listing

The following is a sample part of a listing. An explanation of the different columns follows below.

```
ADDR CODE      CYCLES  LINE SOURCE LINE
                                1      ; Module start
                                .
                                .
0000 08009FE5  1      16      ldr      r0,.L2
0004 001090E5  1      17      ldr      r1,[r0,#0]
0008 04009FE5  1      18      ldr      r0,.L2+4
000C rrrrrrrEA  3      19      b        printf
                                .
                                .
0000                                38      .ds      2
|  RESERVED
0001
```

The meaning of the different columns is:

ADDR This column contains the memory address. The address is a hexadecimal number that represents the offset from the beginning of a relocatable section or the absolute address for an absolute section. The address only appears on lines that generate object code.

CODE	This is the object code generated by the assembler for this source line, displayed in hexadecimal format. The displayed code need not be the same as the generated code that is entered in the object module. The code can also be relocatable code. In this case the letter 'r' is printed for the relocatable code part in the listing. For lines that allocate space, the code field contains the text "RESERVED". For lines that initialize a buffer, the code field lists one value followed by the word "REPEATS".
CYCLES	The first number in this column is the number of instruction cycles needed to execute the instruction(s) as generated in the CODE field. The second number is the accumulated cycle count of this section.
LINE	This column contains the line number. This is a decimal number indicating each input line, starting from 1 and incrementing with each source line.
SOURCE LINE	This column contains the source text. This is a copy of the source line from the assembly source file.



For the `.SET` and `.EQU` directives the ADDR and CODE columns do not apply. The symbol value is listed instead.

Related information

See section 5.6, [Generating a List File](#), in Chapter *Using the Assembler* of the user's manual for more information on how to generate a list file and specify the amount of list file information.

7.2 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.

With the linker option `--map-file-format` (map file formatting) you can specify which parts of the map file you want to see.

Example (part of) linker map file

```
***** Tool and Invocation *****
+-----+
| tool   | TASKING ARM object linker vx.yrz Build 062 |
| path   | <installation dir>\System\Tasking\carm\bin\lkarm.exe |
| arguments | -o hello.abs -fmk2052c.tmp |
| task   | task1 |
+-----+

***** Used Resources *****

* Memory usage in bytes
=====
+-----+
| Memory | Code | Data | Reserved | Free | Total |
|=====|
| system:xram | 0x0 | 0x00180 | 0x08028 | 0x77e58 | 0x80000 |
| system:xrom | 0x01050 | 0x0010a | 0x0 | 0x0eea6 | 0x10000 |
|-----|
| Total   | 0x01050 | 0x0028a | 0x08028 | 0x86cfe | 0x90000 |
+-----+

* Space usage in bytes
=====
+-----+
| Space | Native used Rom | Native used Ram | Foreign used | Reserved | Free Rom | Free Ram | Total |
|=====|
| system:sw:linear | 0x0115a | 0x00180 | 0x0 | 0x08028 | 0x0eea6 | 0x77e58 | 0x90000 |
|-----|
| Total | 0x0115a | 0x00180 | - | - | - | - | - |
| Largest gap | - | - | - | - | 0x0eea4 | 0x77e58 | - |
+-----+

Note:
When spaces share memory with each other, some space can be consumed by sections
located in other spaces. In the table above we call this foreign used space as
opposed to native used space.

* Estimated stack usage
=====
+-----+
| Stack Name | Used |
|=====|
| stack 0 | 0x000000fc |
|-----|
| recursive | no |
+-----+

***** Processed Files *****
+-----+
| File | From archive | Symbol causing the extraction |
|=====|
| cstart.obj | carm.lib | _START |
| hello.obj | | |
| printf.obj | carm.lib | printf |
+-----+
```

***** Link Result *****

[in] File	[in] Section	[in] Size (MAU)	[out] Offset	[out] Section	[out] Size (MAU)
hello.obj	.text (2)	0x00000018	0x00000000	.text (2)	0x00000018
cstart.obj	.text.cstart (296)	0x000000d0	0x00000000	.text.cstart (296)	0x000000d0
printf.obj	.text.libc (57)	0x0000004c	0x00000000	.text.libc (57)	0x0000004c

***** Module Local Symbols *****

* Scope "hello.c"

=====

Name	Space	addr	Space
hello.c	0x0	-	
.rodata	0x0000111c	system:sw:linear	
.rodata	0x00001118		
.text	0x00000134		
.text	0x00000108		

***** Cross References *****

Definition file	Definition section	Symbol	Referenced in
cstart.obj	.text.cstart (296)	_START	
printf.obj	.text.libc (57)	printf	hello.obj

* Undefined symbols

=====

Symbol	Referenced in
_init	hello.obj

***** Call Graph *****

```

main
|
+-- printloop
| |
| +-- printf *
|
+-- printf *

printf
|
+-- _doprint
|
|   +-- _doprint_int.c:_emitchar *
|   |
|   +-- _doprint_int.c:_putnumber
|   |
|   +-- _doprint_int.c:_emitchar *
|   |
|   +-- _doprint_int.c:_putstring *
|   |
|   +-- strlen *
|   |
|   +-- _doprint_int.c:_ltoa
|   |
|   +-- _doprint_int.c:_putstring *
|   |
|   +-- _doflt

```

***** Locate Result *****

* Task entry address

=====

symbol	_START

* Sections

=====

+ Space system:sw:linear (MAU = 8bit)

Chip	Group	Section	Size (MAU)	Space addr	Chip addr
system:xrom		_vector_0 (338)	0x00000004	0x00000020	0x00000020
system:xrom		[.data.libc] (322)	0x000000c8	0x00000040	0x00000040
system:xrom		.text (2)	0x00000018	0x00000134	0x00000134
system:xrom		.rodata (5)	0x0000000e	0x0000111c	0x0000111c
system:xrom		table (321)	0x00000030	0x0000112c	0x0000112c
system:xram		stack_fiq (315)	0x00000008	0x01000000	0x0
system:xram		stack (314)	0x00008000	0x010001a8	0x000001a8

* Symbols (sorted on name)

=====

Name	Space addr	Space
_APPLICATION_MODE_	0x0000001f	system:sw:linear
_Exit	0x00000f34	
_START	0x00000f38	
main	0x00000134	

* Symbols (sorted on address)

=====

Space addr	Name	Space
0x0000001f	_APPLICATION_MODE_	system:sw:linear
0x00000134	main	
0x00000f34	_Exit	
0x00000f38	_START	

***** Processor and Memory *****

***** Locate Rules *****

Address space	Type	Properties	Sections
system:sw:linear	absolute	0x00000020	_vector_0 (338)
system:sw:linear	contiguous		stack_fiq (315) stack_irq (316) stack_svc (317) stack_abt (318) stack_und (319)
system:sw:linear	clustered		.data.libc (65)
system:sw:linear	clustered		[.data.libc] (322)
system:sw:linear	unrestricted		.text.cstart (296)
system:sw:linear	unrestricted		.rodata (5)
system:sw:linear	unrestricted		table (321)
system:sw:linear	unrestricted		stack (314)

***** Removed Sections *****

Section	File	Library	Symbol	Reason
.text.libc (24)	printf_int.obj	cam.lib	dummy	unreferenced
.text.libc (76)	fgetc.obj	cam.lib	fgetc	unreferenced
.text.libc (90)	ungetc.obj	cam.lib	ungetc	unreferenced
.text.libc (97)	_filbuf.obj	cam.lib	_filbuf	unreferenced

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.

Memory	The names of the memory as defined in the linker script file (*.ls1).
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 (*.ls1). The names are constructed of the derivative name followed by a colon ':', the core name, another colon ':' and the space name.
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 (*.ls1).
Used	An estimation of the stack usage. The linker calculates the required stack size by using information (.CALLS directives) generated by the compiler. If for example recursion is detected, the calculated stack size is inaccurate, therefore this is an estimation only. The calculated stack size is supposed to be smaller than the actual allocated stack size. If that is not the case, then a warning is given.

Processed Files

This part of the map file shows all processed files. This also includes object files that are extracted from a library, with the symbol that led to the extraction.

Link Result

This part of the map file shows per object file how the link phase has mapped the sections from the various object files (.obj) to output sections.

[in] File	The name of an input object file.
[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.

Locate Result: Sections

This part of the map file shows the absolute position of each section in the absolute object file. It is organized per address space, memory chip and group and sorted on space address.

+ Space	The names of the address spaces as defined in the linker script file (*.lsl). The names are constructed of the derivative name followed by a colon ':', the core name, another colon ':' and the space name.
Chip	The names of the memory chips as defined in the linker script file (*.lsl) in the memory definitions.
Group	Sections can be ordered in groups. These are the names of the groups as defined in the linker script file (*.lsl) with the keyword group in the section_layout definition. The name that is displayed is the name of the deepest nested group.
Section	The name and id of the section. The number between '()' uniquely identifies the section. Names within square brackets [] will be copied during initialization from ROM to the corresponding section name in RAM.
Size (MAU)	The size of the section in minimum addressable units.
Space addr	The absolute address of the section in the address space.
Chip addr	The absolute offset of the section from the start of a memory chip.

Locate Result: Symbols

This part of the map file lists all external symbols per address space name, both sorted on address and sorted on symbol name.

Name	The name of the symbol.
Space addr	The absolute address of the section in the address space.
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.

Processor and Memory

This part of the map file shows the processor and memory information of the linker script file.

By default this part is not shown in the map file. You have to turn this part on manually with [linker option `--map-file-format=+lsl`](#) (processor and memory info). You can print this information to a separate file with [linker option `--lsl-dump`](#).

Locate Rules

This part of the map file shows the rules the linker uses to locate sections.

Address space	The names of the address spaces as defined in the linker script file (*.lsl). The names are constructed of the derivative name followed by a colon ':', the core name, another colon ':' and the space name.
Type	The rule type: <ul style="list-style-type: none"> ordered/contiguous/clustered/unrestricted <p>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.</p> absolute <p>The section must be located at the address shown in the Properties column.</p>

ranged	The section must be located anywhere in the address ranges shown in the Properties column; end addresses are not included in the range.						
floating	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.						
ranged float	Both the ranged and the floating restriction apply. In the Properties column the range restriction is listed first, followed by the floating restriction between parenthesis.						
ballooned	After locating all sections, the largest remaining gap in the space is used completely for the stack and/or heap.						
Properties	The contents depends on the Type column.						
Sections	<p>The sections to which the rule applies; restrictions between sections are shown in this column:</p> <table><tr><td><</td><td>ordered</td></tr><tr><td> </td><td>contiguous</td></tr><tr><td>+</td><td>clustered</td></tr></table> <p>For contiguous sections, the linker uses the section order as shown here. Clustered sections can be located in any relative order.</p>	<	ordered		contiguous	+	clustered
<	ordered						
	contiguous						
+	clustered						

Removed Sections

This part of the map file contains information about the sections that are removed from the absolute object file. For example, unreferenced sections.

Related information



Section 6.11, [Generating a Map File](#), in Chapter *Using the Linker* of the user's manual.
[Linker option --map-file](#) (Generate map file)



8 Object File Formats

Summary

This chapter describes the formats of several object files.

8.1 ELF/DWARF Object Format

The TASKING ARM toolset by default produces objects in the ELF/DWARF 2 format.

For a complete description of the ELF and DWARF formats, please refer to the *Tool Interface Standard (TIS)*.

8.2 Motorola S-Record Format

With the linker option `-ofilename:SREC` option the linker produces output in Motorola S-record format with three types of S-records: S0, S3 and S7. With the options `-ofilename:SREC:2` or `-ofilename:SREC:3` option you can force other types of S-records. They have the following layout:

S0 - record

'S' '0' <length_byte> <2 bytes 0> <comment> <checksum_byte>

A linker generated S-record file starts with a S0 record with the following contents:

```
length_byte  : $08
comment      : lkarm
checksum     : $E0

      l k a r m
S00800006C6B61726DE0
```

The S0 record is a comment record and does not contain relevant information for program execution.

The length_byte represents the number of bytes in the record, not including the record type and length byte.

The checksum is calculated by first adding the binary representation of the bytes following the record type (starting with the length_byte) to just before the checksum. Then the one's complement is calculated of this sum. The least significant byte of the result is the checksum. The sum of all bytes following the record type is 0FFH.

S1 - record

With the linker option `-ofilename:SREC:2`, the actual program code and data is supplied with S1 records, with the following layout:

'S' '1' <length_byte> <address> <code bytes> <checksum_byte>

This record is used for 2-byte addresses.

Example:

```
S1130250F03EF04DF0ACE8A408A2A013EDFCDB00E6
| | |                                     |_ checksum
| | |   |_ code
| |_ address
|_ length
```

The linker has an option that controls the length of the output buffer for generating S1 records. The default buffer length is 32 code bytes.

The checksum calculation of S1 records is identical to S0.

S2 - record

With the linker option `-ofilename:SREC:3`, the actual program code and data is supplied with S2 records, with the following layout:

'S' '2' <length_byte> <address> <code bytes> <checksum_byte>

This record is used for 3-byte addresses.

Example:

```
S213FF002000232222754E00754F04AF4FAE4E22BF
| | |                                     |_ checksum
| | |   |_ code
| |_ address
|_ length
```

The linker has an option that controls the length of the output buffer for generating S2 records. The default buffer length is 32 code bytes.

The checksum calculation of S2 records is identical to S0.

S3 – record

With the linker option **-ofilename:SREC:4**, which is the default, the actual program code and data is supplied with S3 records, with the following layout:

```
'S' '3' <length_byte> <address> <code bytes> <checksum_byte>
```

The linker generates 4-byte addresses by default.

Example:

```
S3070000FFFE6E6825
| |      | | checksum
| |      | | code
| |      | | address
| |      | | length
```

The linker has an option that controls the length of the output buffer for generating S3 records.

The checksum calculation of S3 records is identical to S0.

S7 – record

With the linker option **-ofilename:SREC:4**, which is the default, at the end of an S-record file, the linker generates an S7 record, which contains the program start address. S7 is the corresponding termination record for S3 records.

Layout:

```
'S' '7' <length_byte> <address> <checksum_byte>
```

Example:

```
S70500000000FA
| |      | | checksum
| |      | | address
| |      | | length
```

The checksum calculation of S7 records is identical to S0.

S8 – record

With the linker option **-ofilename:SREC:3**, at the end of an S-record file, the linker generates an S8 record, which contains the program start address.

Layout:

```
'S' '8' <length_byte> <address> <checksum_byte>
```

Example:

```
S804FF0003F9
| |      | | checksum
| |      | | address
| |      | | length
```

The checksum calculation of S8 records is identical to S0.

S9 – record

With the linker option **-ofilename:SREC:2**, at the end of an S-record file, the linker generates an S9 record, which contains the program start address. S9 is the corresponding termination record for S1 records.

Layout:

```
'S' '9' <length_byte> <address> <checksum_byte>
```

Example:

```
S9030210EA
| |  |__checksum
| |  |__address
|_  |__length
```

The checksum calculation of S9 records is identical to S0.

8.3 Intel Hex Record Format

Intel Hex records describe the hexadecimal object file format for 8-bit, 16-bit and 32-bit microprocessors. The hexadecimal object file is an ASCII representation of an absolute binary object file. There are six different types of records:

- Data Record (8-, 16, or 32-bit formats)
- End of File Record (8-, 16, or 32-bit formats)
- Extended Segment Address Record (16, or 32-bit formats)
- Start Segment Address Record (16, or 32-bit formats)
- Extended Linear Address Record (32-bit format only)
- Start Linear Address Record (32-bit format only)

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:

:	<i>length</i>	<i>offset</i>	<i>type</i>	<i>content</i>	<i>checksum</i>
---	---------------	---------------	-------------	----------------	-----------------

Where:

: is the record header.

length is the record length which specifies the number of bytes of the *content* field. This value occupies one byte (two hexadecimal digits). The linker outputs records of 255 bytes (32 hexadecimal digits) or less; that is, *length* is never greater than FFH.

offset is the starting load offset specifying an absolute address in memory where the data is to be located when loaded by a tool. This field is two bytes long. This field is only used for Data Records. In other records this field is coded as four ASCII zero characters ('0000').

type is the record type. This value occupies one byte (two hexadecimal digits). The record types are:

Byte Type	Record type
00	Data
01	End of File
02	Extended segment address (not used)
03	Start segment address (not used)
04	Extended linear address (32-bit)
05	Start linear address (32-bit)

content is the information contained in the record. This depends on the record type.

checksum is the record checksum. The linker computes the checksum by first adding the binary representation of the previous bytes (from *length* to *content*). The linker then computes the result of sum modulo 256 and subtracts the remainder from 256 (two's complement). Therefore, the sum of all bytes following the header is zero.

Extended Linear Address Record

The Extended Linear Address Record specifies the two most significant bytes (bits 16–31) of the absolute address of the first data byte in a subsequent Data Record:

:	02	0000	04	<i>upper_address</i>	<i>checksum</i>
---	----	------	----	----------------------	-----------------

The 32-bit absolute address of a byte in a Data Record is calculated as:

$$(address + offset + index) \text{ modulo } 4G$$

where:

address is the base address, where the two most significant bytes are the *upper_address* and the two least significant bytes are zero.

offset is the 16-bit offset from the Data Record.

index is the index of the data byte within the Data Record (0 for the first byte).

Example:

```
:0200000400FFFB
| | | | | _ checksum
| | | | | _ upper_address
| | | | | _ type
| | | | | _ offset
| | | | | _ length
```

Data Record

The Data Record specifies the actual program code and data.

:	<i>length</i>	<i>offset</i>	00	<i>data</i>	<i>checksum</i>
---	---------------	---------------	----	-------------	-----------------

The *length* byte specifies the number of *data* bytes. The linker has an option that controls the length of the output buffer for generating Data records. The default buffer length is 32 bytes.

The *offset* is the 16-bit starting load offset. Together with the address specified in the Extended Address Record it specifies an absolute address in memory where the data is to be located when loaded by a tool.

Example:

```
:0F00200000232222754E00754F04AF4FAE4E22C3
| | | | | _ checksum
| | | | | _ data
| | | | | _ type
| | | | | _ offset
| | | | | _ length
```

Start Linear Address Record

The Start Linear Address Record contains the 32-bit program execution start address.

Layout:

:	04	0000	05	<i>address</i>	<i>checksum</i>
---	----	------	----	----------------	-----------------

Example:

```
:0400000500FF0003F5
| | | | | _ checksum
| | | | | _ address
| | | | | _ type
| | | | | _ offset
| | | | | _ length
```

End of File Record

The hexadecimal file always ends with the following end-of-file record:

```
:00000001FF
| | | | _ checksum
| | | | _ type
| | _ offset
| _ length
```




9 Linker Script Language

Summary

This chapter describes the syntax of the linker script language (LSL)

9.1 Introduction

To make full use of the linker, you can write a script with information about the architecture of the target processor and locating information. The language for the script is called the *Linker Script Language* (LSL). This chapter first describes the structure of an LSL file. The next section contains a summary of the LSL syntax. Finally, in the remaining sections, the semantics of the Linker Script Language is explained.

The TASKING linker is a target independent linker/locator that can simultaneously link and locate all programs for all cores available on a target board. The target board may be of arbitrary complexity. A simple target board may contain one standard processor with some external memory that executes one task. A complex target board may contain multiple standard processors and DSPs combined with configurable IP-cores loaded in an FPGA. Each core may execute a different program, and external memory may be shared by multiple cores.

LSL serves two purposes. First it enables you to specify the characteristics (that are of interest to the linker) of your specific target board and of the cores installed on the board. Second it enables you to specify how sections should be located in memory.

9.2 Structure of a Linker Script File

A script file consists of several definitions. The definitions can appear in any order.

The architecture definition (required)

In essence an *architecture definition* describes how the linker should convert logical addresses into physical addresses for a given type of core. If the core supports multiple address spaces, then for each space the linker must know how to perform this conversion. In this context a physical address is an offset on a given internal or external bus. Additionally the architecture definition contains information about items such as the (hardware) stack and the vector table.

This specification is normally written by Altium. The architecture definition of the LSL file should not be changed by you unless you also modify the core's hardware architecture. If the LSL file describes a multi-core system an architecture definition must be available for each different type of core.



See section 9.5, [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. A derivative definition must be present in an LSL file. Microcontrollers and DSPs often have internal memory and I/O sub-systems apart from one or more cores. The design of such a chip is called a *derivative*.

When you design an FPGA together with a PCB, the components on the FPGA become part of the board design and there is no need to distinguish between internal and external memory. For this reason you probably do not need to work with derivative definitions at all. There are, however, two situations where derivative definitions are useful:

1. When you re-use an FPGA design for several board designs it may be practical to write a derivative definition for the FPGA design and include it in the project LSL file.
2. 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 9.6, [Semantics of the Derivative Definition](#) for a detailed description of LSL in the derivative definition.

The processor definition

The *processor definition* describes an instance of a derivative. Typically the processor definition instantiates one derivative only (single-core processor). A processor that contains multiple cores having the same (homogeneous) or different (heterogeneous) architecture can also be described by instantiating multiple derivatives of the same or different types in separate processor definitions.

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.



See section 9.7, [Semantics of the Board Specification](#) for a detailed description of LSL in the processor definition.

The memory and bus definitions (optional)

Memory and bus definition are used within the context of a derivative definition to specify internal memory and on-chip 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 9.7.3, [Defining External Memory and Buses](#), for more information on how to specify the external physical memory layout. *Internal* memory for a processor should be defined in the derivative definition for that processor.

The board specification

The processor definition and memory and bus definitions together form a *board specification*. LSL provides language constructs to easily describe single-core and heterogeneous or homogeneous multi-core systems. The board specification describes all characteristics of your target board's system buses, memory devices, I/O sub-systems, and cores that are of interest to the linker. Based on the information provided in the board specification the linker can for each core:

- convert a logical address to 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, from the board specification the linker can deduce which physical memory is (still) available while locating the section.



See section 9.9, [Semantics of the Section Layout Definition](#), for more information on how to locate a section at a specific place in memory.

Skeleton of a Linker Script File

The skeleton of a linker script file now looks as follows:

```
architecture architecture_name
{
    architecture definition
}

derivative derivative_name
{
    derivative definition
}
```

```

processor processor_name
{
    processor definition
}

memory definitions and/or bus definitions

section_layout space_name
{
    section placement statements
}

```

9.3 Syntax of the Linker Script Language

9.3.1 Preprocessing

When the linker loads an LSL file, the linker processes it with a C-style preprocessor. As such, it strips C and C++ comments. You can use the standard ISO C preprocessor directives, such as `#include`, `#define`, `#if/#else/#endif`.

For example:

```
#include "arch.lsl"
```

Preprocess and include the file `arch.lsl` at this point in the LSL file.

9.3.2 Lexical Syntax

The following lexicon is used to describe the syntax of the Linker Script Language:

<code>A ::= B</code>	= <i>A</i> is defined as <i>B</i>
<code>A ::= B C</code>	= <i>A</i> is defined as <i>B</i> and <i>C</i> ; <i>B</i> is followed by <i>C</i>
<code>A ::= B C</code>	= <i>A</i> is defined as <i>B</i> or <i>C</i>
<code>⁰ ¹</code>	= zero or one occurrence of <i>B</i>
<code>^{>=0}</code>	= zero or more occurrences of <i>B</i>
<code>^{>=1}</code>	= one or more occurrences of <i>B</i>
 <i>IDENTIFIER</i> = a character sequence starting with 'a'-'z', 'A'-'Z' or '_'. Following characters may also be digits and dots '.'	
<i>STRING</i>	= sequence of characters not starting with \n, \r or \t
<i>DQSTRING</i>	= " <i>STRING</i> " (double quoted string)
<i>OCT_NUM</i>	= octal number, starting with a zero (06, 045)
<i>DEC_NUM</i>	= decimal number, not starting with a zero (14, 1024)
<i>HEX_NUM</i>	= hexadecimal number, starting with '0x' (0x0023, 0xFF00)

OCT_NUM, *DEC_NUM* and *HEX_NUM* can be followed by a **k** (kilo), **M** (mega), or **G** (giga).

Characters in **bold** are characters that occur literally. Words in *italics* are higher order terms that are defined in the same or in one of the other sections.

To write comments in LSL file, you can use the C style `/* */` or C++ style `/**/`.

9.3.3 Identifiers

```
arch_name      ::= IDENTIFIER
bus_name       ::= IDENTIFIER
core_name      ::= IDENTIFIER
derivative_name ::= IDENTIFIER
file_name      ::= DQSTRING
group_name     ::= IDENTIFIER
mem_name       ::= IDENTIFIER
proc_name      ::= IDENTIFIER
section_name   ::= DQSTRING
space_name     ::= IDENTIFIER
stack_name     ::= section_name
symbol_name    ::= DQSTRING
```

9.3.4 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
```

9.3.5 Built-in Functions

```

function_call    ::= absolute ( expr )
                  | addressof ( addr_id )
                  | exists ( section_name )
                  | max ( expr , expr )
                  | min ( expr , expr )
                  | sizeof ( size_id )

addr_id          ::= sect : section_name
                  | group : group_name

size_id          ::= 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.

You can use the following built-in functions in expressions. All functions return a numerical value. This value is a 64-bit signed integer.

absolute()

```
int absolute( expr )
```

Converts the value of *expr* to a positive integer.

```
absolute( "labelA"-"labelB" )
```

addressof()

```
int addressof( addr_id )
```

Returns the address of *addr_id*, which is a named section or group. To get the offset of the section with the name *asect*:

```
addressof( sect: "asect" )
```



This function only works in assignments.

exists()

```
int exists( section_name )
```

The function returns 1 if the section *section_name* exists in one or more object file, 0 otherwise. If the section is not present in input object files, but generated from LSL, the result of this function is undefined.

To check whether the section *mysection* exists in one of the object files that is specified to the linker:

```
exists( "mysection" )
```

max()

```
int max( expr, expr )
```

Returns the value of the expression that has the largest value. To get the highest value of two symbols:

```
max( "sym1" , "sym2" )
```

min()

```
int min( expr, expr )
```

Returns the value of the expression that has the smallest value. To get the lowest value of two symbols:

```
min( "sym1" , "sym2" )
```

sizeof()

```
int sizeof( size_id )
```

Returns the size of the object (group, section or memory) the identifier refers to. To get the size of the section "asection":

```
sizeof( sect: "asection" )
```



The **group** and **sect** arguments only work in assignments. The **mem** argument can be used anywhere in section layouts.

9.3.6 LSL Definitions in the Linker Script File

```
description      ::= <definition>>=1
definition       ::= architecture_definition
                  | derivative_definition
                  | board_spec
                  | section_definition
                  | section_setup
```

- At least one *architecture_definition* must be present in the LSL file.

9.3.7 Memory and Bus Definitions

```
mem_def          ::= memory mem_name { <mem_descr ;>=0 }
```

- A *mem_def* defines a *memory* with the *mem_name* as a unique name.

```
mem_descr        ::= type = <reserved>0|1 mem_type
                  | mau = expr
                  | size = expr
                  | speed = number
                  | fill <= fill_values>0|1
                  | mapping
```

- A *mem_def* contains exactly one **type** statement.
- A *mem_def* contains exactly one **mau** statement (non-zero size).
- A *mem_def* contains exactly one **size** statement.
- A *mem_def* contains zero or one **speed** statement (if absent, the default speed value is 1).
- A *mem_def* contains zero or one **fill** statement.
- A *mem_def* contains at least one *mapping*.

```
mem_type         ::= rom          // attrs = rx
                  | ram          // attrs = rw
                  | nvram        // attrs = rwx
```

```
fill_values      ::= expr
                  | [ expr <, expr>=0 ]
```

```
bus_def          ::= bus bus_name { <bus_descr ;>=0 }
```

- A *bus_def* statement defines a *bus* with the given *bus_name* as a unique name within a core architecture.

```

bus_descr ::= mau = expr
            | width = expr // bus width, nr
            | // of data bits
            | mapping // legal destination
            | // 'bus' only

```

- The **mau** and **width** statements appear exactly once in a *bus_descr*. The default value for **width** is the **mau** size.
- The bus width must be an integer times the bus MAU size.
- The MAU size must be non-zero.
- A bus can only have a *mapping* on a destination *bus* (through **dest** = **bus:**).

```

mapping ::= map ( map_descr <, map_descr>*=0 )

```

```

map_descr ::= dest = destination
            | dest_dbits = range
            | dest_offset = expr
            | size = expr
            | src_dbits = range
            | src_offset = expr

```

- A *mapping* requires at least the **size** and **dest** statements.
- Each *map_descr* can occur only once.
- You can define multiple mappings from a single source.
- Overlap between source ranges or destination ranges is not allowed.
- If the **src_dbits** or **dest_dbits** statement is not present, its value defaults to the **width** value if the source/destination is a bus, and to the **mau** size otherwise.

```

destination ::= space : space_name
                | bus : <proc_name |
                    core_name :>0|1 bus_name

```

- A *space_name* refers to a defined address space.
- A *proc_name* refers to a defined processor.
- A *core_name* refers to a defined core.
- A *bus_name* refers to a defined bus.
- The following mappings are allowed (source to destination)
 - space => space
 - space => bus
 - bus => bus
 - memory => bus

```

range ::= expr .. expr

```

- With address ranges, the end address is not part of the range.

9.3.8 Architecture Definition

```

architecture_definition ::= architecture arch_name
                             <( parameter_list )>0|1
                             <extends arch_name
                             <( argument_list )>0|1 >0|1
                             { arch_spec>*=0 }

```

- An *architecture_definition* defines a core *architecture* with the given *arch_name* as a unique name.
- At least one *space_def* and at least one *bus_def* have to be present in an *architecture_definition*.
- An *architecture_definition* that uses the **extends** construct defines an architecture that *inherits* all elements of the architecture defined by the second *arch_name*. The *parent architecture* must be defined in the LSL file as well.

```

parameter_list ::= parameter <, parameter>*=0

```

```

parameter ::= IDENTIFIER <= expr>0|1

```

argument_list ::= **expr** <, **expr**>>=0

arch_spec ::= **bus_def**
| **space_def**
| **endianness_def**

space_def ::= **space** **space_name** { <**space_descr**>>=0 }

- A **space_def** defines an address space with the given **space_name** as a unique name within an architecture.

space_descr ::= **space_property** ;
| **section_definition** //no space ref
| **vector_table_statement**
| **reserved_range**

space_property ::= **id** = **number** // as used in object
| **mau** = **expr**
| **align** = **expr**
| **page_size** = **expr** <[**range**] <| [**range**]>>=0 >0|1
| **page**
| **direction** = **direction**
| **stack_def**
| **heap_def**
| **copy_table_def**
| **start_address**
| **mapping**

- A **space_def** contains exactly one **id** and one **mau** statement.
- A **space_def** contains at most one **align** statement.
- A **space_def** contains at most one **page_size** statement.
- A **space_def** contains at least one mapping.

stack_def ::= **stack** **stack_name** (**stack_heap_descr**
| **stack_heap_descr** >>=0)

- A **stack_def** defines a stack with the **stack_name** as a unique name.

heap_def ::= **heap** **heap_name** (**stack_heap_descr**
| **stack_heap_descr** >>=0)

- A **heap_def** defines a heap with the **heap_name** as a unique name.

stack_heap_descr ::= **min_size** = **expr**
| **grows** = **direction**
| **align** = **expr**
| **fixed**
| **id** = **expr**

- The **min_size** statement must be present.
- You can specify at most one **align** statement and one **grows** statement.
- Each stack definition has its own unique **id**, the number specified corresponds to the index in the **.CALLS** directive as generated by the compiler. If the **id** is omitted, the **id** is 0 (zero).

direction ::= **low_to_high**
| **high_to_low**

- If you do not specify the **grows** statement, the stack and grow **low-to-high**.

copy_table_def ::= **copytable** <(**copy_table_descr**
| **copy_table_descr** >>=0)>0|1

- A **space_def** contains at most one **copytable** statement.
- If the architecture definition contains more than one address space, exactly one copy table must be defined in one of the spaces. If the architecture definition contains only one address space, a copy table definition is optional (it will be generated in the space).


```

copy_table_descr ::= align = expr
                  | copy_unit = expr
                  | dest <space_name>0|1 = space_name
                  | page

```

- The **copy_unit** is defined by the size in MAUs in which the startup code moves data.
- The **dest** statement is only required when the startup code initializes memory used by another processor that has no access to ROM.
- A *space_name* refers to a defined address space.

```

start_addr ::= start_address ( start_addr_descr
                              <, start_addr_descr>>=0 )

```

```

start_addr_descr ::= run_addr = expr
                  | symbol = symbol_name

```

- A *symbol_name* refers to the section that contains the startup code.

```

vector_table_statement
    ::= vector_table section_name
       ( vecttab_spec <, vecttab_spec>>=0 )
       { <vector_def>>=0 }

```

```

vecttab_spec ::= vector_size = expr
              | size = expr
              | id_symbol_prefix = symbol_name
              | run_addr = addr_absolute
              | template = section_name
              | template_symbol = symbol_name
              | vector_prefix = section_name
              | fill = vector_value
              | no_inline
              | copy

```

```

vector_def ::= vector ( vector_spec <, vector_spec>>=0 )

```

```

vector_spec ::= id = vector_id_spec
              | fill = vector_value

```

```

vector_id_spec ::= number
                | [ range ] <, [ range ]>>=0

```

```

vector_value ::= symbol_name
              | [ number <, number>>=0 ]
              | loop <[ expr ]>>0|1

```

```

reserved_range ::= reserved expr .. expr ;

```

- The end address is not part of the range.

```

endianness_def ::= endianness { <endianness_type;>>=1 }

```

```

endianness_type ::= big
                  | little

```

9.3.9 Derivative Definition

```

derivative_definition
    ::= derivative derivative_name
       <( parameter_list )>>0|1
       <extends derivative_name
         <( argument_list )>>0|1 >>0|1
         { <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*.

9.3.10 Processor Definition and Board Specification

```
board_spec ::= proc_def
                | bus_def
                | mem_def
```

```
proc_def ::= processor proc_name
             { proc_descr ; }
```

```
proc_descr ::= derivative = derivative_name
              <( argument_list )>0|1
```

- A *proc_def* defines a *processor* with the *proc_name* as a unique name.
- If you do not explicitly define a processor for a derivative in an LSL file, the linker defines a processor with the same name as that derivative.
- A *derivative_name* refers to a defined derivative.
- A *proc_def* contains exactly one **derivative** statement.

9.3.11 Section Layout Definition and Section Setup

```
section_definition ::= section_layout <space_ref>0|1
                      <( locate_direction )>0|1
                      { <section_statement>*=0 }
```

- A section definition inside a space definition does not have a *space_ref*.
- All global section definitions have a *space_ref*.

```
space_ref ::= <proc_name>0|1 : <core_name>0|1
             : space_name
```

- If more than one processor is present, the *proc_name* must be given for a global section layout.
- If the section layout refers to a processor that has more than one core, the *core_name* must be given in the *space_ref*.
- A *proc_name* refers to a defined processor.
- A *core_name* refers to a defined core.
- A *space_name* refers to a defined address space.

```
locate_direction ::= direction = direction
```

```
direction ::= low_to_high
              | high_to_low
```

- A section layout contains at most one **direction** statement.
- If you do not specify the **direction** statement, the locate direction of the section layout is **low-to-high**.

```
section_statement ::= simple_section_statement ;
                   | aggregate_section_statement
```

simple_section_statement

```
 ::= assignment
    | select_section_statement
    | special_section_statement
```

assignment ::= *symbol_name* *assign_op* *expr*

assign_op ::= =
| :=

select_section_statement

```
 ::= select <ref_tree>0|1 <section_name>0|1
    <section_selections>0|1
```

- Either a *section_name* or at least one *section_selection* must be defined.

section_selections

```
 ::= ( section_selection
    <, section_selection>>=0 )
```

section_selection

```
 ::= attributes = < <+|-> attribute>>0
```

- *+attribute* means: select all sections that have this attribute.
- *-attribute* means: select all sections that do not have this attribute.

special_section_statement

```
 ::= heap stack_name <size_spec>0|1
    | stack stack_name <size_spec>0|1
    | copytable
    | reserved section_name <reserved_specs>0|1
```

- Special sections cannot be selected in load-time groups.

size_spec ::= (**size** = *expr*)

reserved_specs ::= (*reserved_spec* <, *reserved_spec*>^{>=0})

```
 reserved_spec ::= attributes
    | fill_spec
    | size = expr
    | alloc_allowed = absolute | ranged
```

- If a **reserved** section has attributes **r**, **rw**, **x**, **rx** or **rwrx**, and no fill pattern is defined, the section is filled with zeros. If no attributes are set, the section is created as a scratch section (attributes **ws**, no image).

fill_spec ::= **fill** = *fill_values*

```
 fill_values ::= expr
    | [ expr <, expr>>=0 ]
```

aggregate_section_statement

```
 ::= { <section_statement>>=0 }
    | group_descr
    | if_statement
    | section_creation_statement
```

```
 group_descr ::= group <group_name>0|1 <( group_specs )>0|1
    section_statement
```

- 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
```

- The **allow-cross-references** property is only allowed for *overlay* groups.
- Sub groups inherit all properties from a parent group.

```
group_alignment ::= align = expr
attributes      ::= attributes = <attribute>>=1
attribute       ::= r // readable sections
                  | w // writable sections
                  | x // executable code sections
                  | i // initialized sections
                  | s // scratch sections
                  | b // blanked (cleared) sections

group_load_address ::= load_addr <= load_or_run_addr>0|1
group_page         ::= page <= expr>0|1
                  | page_size = expr <[ range ] <| [ range ]>>=0 >0|1
group_run_address  ::= run_addr <= load_or_run_addr>0|1
group_type         ::= clustered
                  | contiguous
                  | ordered
                  | overlay
```

- For *non-contiguous* groups, you can only specify *group_alignment* and *attributes*.
- The **overlay** keyword also sets the **contiguous** property.
- The **clustered** property cannot be set together with **contiguous** or **ordered** on a single group.

```
load_or_run_addr ::= addr_absolute
                  | addr_range <| addr_range>>=0
```

```
addr_absolute    ::= expr
                  | memory_reference [ expr ]
```

- An absolute address can only be set on *ordered* groups.

```
addr_range       ::= [ expr .. expr ]
                  | memory_reference
                  | memory_reference [ expr .. expr ]
```

- The parent of a group with an *addr_range* or **page** restriction cannot be **ordered**, **contiguous** or **clustered**.
- The end address is not part of the range.

```
memory_reference ::= mem : <proc_name :>0|1 mem_name
```

- A *proc_name* refers to a defined processor.
- A *mem_name* refers to a defined memory.

```
if_statement     ::= if ( expr ) section_statement
                  <else section_statement>0|1
```

```

section_creation_statement
    ::= section section_name ( section_specs )
       { <section_statement2>*>=0 }

section_specs
    ::= section_spec <, section_spec >*>=0

section_spec
    ::= attributes
       | fill_spec
       | size = expr
       | blocksize = expr
       | overflow = section_name

section_statement2
    ::= select_section_statement ;
       | group_descr2
       | { <section_statement2>*>=0 }

group_descr2
    ::= group <group_name>*>0 | 1
       ( group_specs2 )
       section_statement2

group_specs2
    ::= group_spec2 <, group_spec2 >*>=0

group_spec2
    ::= group_alignment
       | attributes
       | load_addr

section_setup
    ::= section_setup space_ref
       { <section_setup_item>*>=0 }

section_setup_item
    ::= vector_table_statement
       | reserved_range
       | stack_def ;
       | heap_def ;

```

9.4 Expression Evaluation

Only *constant* expressions are allowed, including sizes, but not addresses, of sections in object files.

All expressions are evaluated with 64-bit precision integer arithmetic. The result of an expression can be absolute or relocatable. A symbol you assign is created as an absolute symbol.

9.5 Semantics of the Architecture Definition

Keywords in the architecture definition

```
architecture
  extends
endianness      big  little
bus
  mau
  width
  map
space
  id
  mau
  align
  page_size
  page
  direction      low_to_high  high_to_low
  stack
    min_size
    grows        low_to_high  high_to_low
    align
    fixed
    id
  heap
    min_size
    grows        low_to_high  high_to_low
    align
    fixed
    id
  copytable
    align
    copy_unit
    dest
    page
  vector_table
    vector_size
    size
    id_symbol_prefix
    run_addr
    template
    template_symbol
    vector_prefix
    fill
    no_inline
    copy
    vector
      id
      fill      loop
reserved
start_address
  run_addr
  symbol
map
```

```

map
    dest          bus space
    dest_dbits
    dest_offset
    size
    src_dbits
    src_offset

```

9.5.1 Defining an Architecture

With the keyword **architecture** you define an architecture and assign a unique name to it. The name is used to refer to it at other places in the LSL file:

```

architecture name
{
    definitions
}

```

If you are defining multiple core architectures that show great resemblance, you can define the common features in a parent core architecture and extend this with a child core architecture that contains specific features. The child inherits all features of the parent. With the keyword **extends** you create a child core architecture:

```

architecture name_child_arch extends name_parent_arch
{
    definitions
}

```

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.

```

architecture name_child_arch (parm1,parm2=1)
    extends name_parent_arch (arguments)
{
    definitions
}

```

9.5.2 Defining Internal Buses

With the **bus** keyword you define a bus (the combination of data and corresponding address bus). The bus name is used to identify a bus and does not conflict with other identifiers. Bus descriptions in an architecture definition or derivative definition define *internal* buses. Some internal buses are used to communicate with the components outside the core or processor. Such buses on a processor have physical pins reserved for the number of bits specified with the **width** statements.

- The **mau** field specifies the MAU size (Minimum Addressable Unit) of the data bus. This field is required.
- The **width** field specifies the width (number of address lines) of the data bus. The default value is the MAU size.
- The **map** keyword specifies how this bus maps onto another bus (if so). Mappings are described in section 9.5.4, [Mappings](#).

```

bus bus_name
{
    mau = 8;
    width = 8;
    map ( map_description );
}

```

9.5.3 Defining Address Spaces

With the **space** keyword you define a logical address space. The space name is used to identify the address space and does not conflict with other identifiers.

- The **id** field defines how the addressing space is identified in object files. In general, each address space has a unique ID. The linker locates sections with a certain ID in the address space with the same ID. This field is required. In IEEE this ID is specified explicitly for sections and symbols, ELF sections map by default to the address space with ID 1. Sections with one of the special names defined in the ABI (Application Binary Interface) may map to different address spaces.
- The **mau** field specifies the MAU size (Minimum Addressable Unit) of the space. This field is required.
- The **align** value must be a power of two. The linker uses this value to compute the start addresses when sections are concatenated. An align value of *n* means that objects in the address space have to be aligned on *n* MAUs.
- The **page_size** field sets the page 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 9.9.2, [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 9.5.4, [Mappings](#).

Stacks and heaps

- The **stack** keyword defines a stack in the address space and assigns a name to it. The architecture definition must contain at least one stack definition. Each stack of a core architecture must have a unique name. See also the **stack** keyword in section 9.9.3, [Creating or Modifying Special Sections](#).

The stack is described in terms of a minimum size (**min_size**) and the direction in which the stack grows (**grows**). This can be either from **low_to_high** addresses (stack grows upwards, this is the default) or from **high_to_low** addresses (stack grows downwards). The **min_size** is required.

By default, the linker tries to maximize the size of the stacks and heaps. After locating all sections, the largest remaining gap in the space is used completely for the stacks and heaps. If you specify the keyword **fixed**, you can disable this so-called 'balloon behavior'. The size is also fixed if you used a stack or heap in the software layout definition in a restricted way. For example when you override a stack with another size or select a stack in an ordered group with other sections.

The **id** keyword matches stack information generated by the compiler with a stack name specified in LSL. This value assigned to this keyword is strongly related to the compiler's output, so users are not supposed to change this configuration.

Optionally you can specify an alignment for the stack with the argument **align**. This alignment must be equal or larger than the alignment that you specify for the address space itself.

- The **heap** keyword defines a heap in the address space and assigns a name to it. The definition of a heap is similar to the definition of a stack. See also the **heap** keyword in section 9.9.3, [Creating or Modifying Special Sections](#).



See section 9.9, [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 9.9.2, [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 "vtable" (**vector_size**= m , **size**= n , **run_addr**= x , ...)

See the following example of a vector table definition:

```
vector_table "vtable" (vector_size = 4, size = 256, run_addr=0,
    template=".text.vector_template",
    template_symbol="_lc_vector_target",
    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=0, fill="_START");
    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 handler section name + symbol name for the target code must be supplied in the LSL file.

fill=[value(s)], fills the vector with the specified MAU values.

With **fill=loop** the vector jumps to itself. With the optional **[offset]** you can specify an offset from the vector table entry.

Reserved address ranges

- The **reserved** keyword specifies to reserve a part of an address space even if not all of the range is covered by memory. See also the **reserved** keyword in section 9.9.3, [Creating or Modifying Special Sections](#).

Start address

- The **start_address** keyword specifies the start address for the position where the C startup code is located. When a processor is reset, it initializes its program counter to a certain start address, sometimes called the *reset vector*. In the architecture definition, you must specify this start address in the correct address space in combination with the name of the label in the application code which must be located here.

The **run_addr** argument specifies the start address (reset vector). If the core starts executing using an entry from a vector table, and directly jumps to the start label, you should omit this argument.

The **symbol** argument specifies the name of the label in the application code that should be located at the specified start address. The **symbol** argument is required. The linker will resolve the start symbol and use its value after locating for the start address field in IEEE-695 files and Intel Hex files. If you also specified the **run_addr** argument, the start symbol (label) must point to a section. The linker locates this section such that the start symbol ends up on the start address.

```
space space_name
{
    id = 1;
    mau = 8;
    align = 8;
    page_size = 1;
    stack_name (min_size = 1k, grows = low_to_high);
    reserved start_address .. end_address;
    start_address ( run_addr = 0x0000,
                   symbol = "start_label" )
    map ( map_description );
}
```

9.5.4 Mappings

You can use a mapping when you define a space, bus or memory. With the **map** field you specify how addresses from the source (space, bus or memory) are translated to addresses of a destination (space, bus). The following mappings are possible:

- space => space
- space => bus
- bus => bus
- memory => bus

With a mapping you specify a range of source addresses you want to map (specified by a source offset and a size), the destination to which you want to map them (a bus or another address space), and the offset address in the destination.

- The **dest** argument specifies the destination. This can be a **bus** or another address **space** (only for a space to space mapping). This argument is required.
- The **src_offset** argument specifies the offset of the source addresses. In combination with size, this specifies the range of address that are mapped. By default the source offset is 0x0000.
- The **size** argument specifies the number of addresses that are mapped. This argument is required.
- The **dest_offset** argument specifies the position in the destination to which the specified range of addresses is mapped. By default the destination offset is 0x0000.

If you are mapping a bus to another bus, the number of data lines of each bus may differ. In this case you have to specify a range of source data lines you want to map (**src_dbits** = *begin..end*) and the range of destination data lines you want to map them to (**dest_dbits** = *first..last*).

- The **src_dbits** argument specifies a range of data lines of the source bus. By default all data lines are mapped.
- The **dest_dbits** argument specifies a range of data lines of the destination bus. By default, all data lines from the source bus are mapped on the data lines of the destination bus (starting with line 0).

From space to space

If you map an address space to another address space (nesting), you can do this by mapping the subspace to the containing larger space. In this example a small space of 64k is mapped on a large space of 16M.

```
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.

9.6 Semantics of the Derivative Definition

Keywords in the derivative definition

```

derivative
  extends
core
  architecture
bus
  mau
  width
  map
memory
  type          reserved rom ram nvram
  mau
  size
  speed
  fill
  map
section_layout
section_setup

  map
    dest          bus space
    dest_dbits
    dest_offset
    size
    src_dbits
    src_offset

```

9.6.1 Defining a Derivative

With the keyword **derivative** you define a derivative and assign a unique name to it. The name is used to refer to it at other places in the LSL file:

```

derivative name
{
  definitions
}

```

If you are defining multiple derivatives that show great resemblance, you can define the common features in a parent derivative and extend this with a child derivative that contains specific features. The child inherits all features of the parent (cores and memories). With the keyword **extends** you create a child derivative:

```

derivative name_child_deriv extends name_parent_deriv
{
  definitions
}

```

As with a core architecture, a derivative can have any number of parameters. These are identifiers which get values assigned on instantiation or extension of the derivative. You can use them in any expression within the derivative definition.

```

derivative name_child_deriv (parm1,parm2=1)
  extends name_parent_derivh (arguments)
{
  definitions
}

```

9.6.2 Instantiating Core Architectures

With the keyword **core** you instantiate a core architecture in a derivative.

- With the keyword **architecture** you tell the linker that the given core has a certain architecture. The architecture name refers to an existing architecture definition in the same LSL file.

For example, if you have two cores (called `mycore_1` and `mycore_2`) that have the same architecture (called `mycorearch`), you must instantiate both cores as follows:

```
core mycore_1
{
    architecture = mycorearch;
}

core mycore_2
{
    architecture = mycorearch;
}
```

If the architecture definition has parameters you must specify the arguments that correspond with the parameters. For example `mycorearch1` expects two parameters which are used in the architecture definition:

```
core mycore
{
    architecture = mycorearch1 (1,2);
}
```

9.6.3 Defining Internal Memory and Buses

With the **memory** keyword you define physical memory that is present on the target board. The memory name is used to identify the memory and does not conflict with other identifiers. It is common to define internal memory (on-chip) in the derivative definition. External memory (off-chip memory) is usually defined in the board specification (See section 9.7.3, [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
 - **nvr**: non volatile ram – writing is possible both at load-time and run-time

The optional **reserved** qualifier before the memory type, tells the linker not to locate any section in the memory by default. You can locate sections in such memories using an absolute address or range restriction (see subsection [Locating a group](#) in section 9.9.2, [Creating and Locating Groups of Sections](#)).

- The **mau** field specifies the MAU size (Minimum Addressable Unit) of the memory. This field is required.
- The **size** field specifies the size in MAU of the memory. This field is required.
- The **speed** field specifies a symbolic speed for the memory (1..4): 1 is the fastest, 4 the slowest. The linker uses the relative speed of the memories in such a way, that optimal speed is achieved. The default speed is 1.
- The **map** field specifies how this memory maps onto an (internal) bus. Mappings are described in section 9.5.4, [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.

```
memory mem_name
{
    type = rom;
    mau = 8;
    fill = 0xaa;
    size = 64k;
    speed = 2;
    map ( map_description );
}
```

With the **bus** keyword you define a bus in a derivative definition. Buses are described in section 9.5.2, [Defining Internal Buses](#).

9.7 Semantics of the Board Specification

Keywords in the board specification

```
processor
    derivative
bus
    mau
    width
    map
memory
    type          reserved rom ram nvram
    mau
    size
    speed
    fill
    map
    map
        dest      bus space
        dest_dbits
        dest_offset
        size
        src_dbits
        src_offset
```

9.7.1 Defining a Processor

If you have a target board with multiple processors that have the same derivative, you need to instantiate each individual processor in a processor definition. This information tells the linker which processor has which derivative and enables the linker to distinguish between the present processors.



If you use processors that all have a unique derivative, you may omit the processor definitions. In this case the linker assumes that for each derivative definition in the LSL file there is one processor. The linker uses the derivative name also for the processor.

With the keyword **processor** you define a processor. You can freely choose the processor name. The name is used to refer to it at other places in the LSL file:

```
processor proc_name
{
    processor definition
}
```

9.7.2 Instantiating Derivatives

With the keyword **derivative** you tell the linker that the given processor has a certain derivative. The derivative name refers to an existing derivative definition in the same LSL file.

For examples, if you have two processors on your target board (called `myproc_1` and `myproc_2`) that have the same derivative (called `myderiv`), you must instantiate both processors as follows:

```
processor myproc_1
{
    derivative = myderiv;
}

processor myproc_2
{
    derivative = myderiv;
}
```

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);
}
```

9.7.3 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;
    speed = 2;
    map ( map_description );
}
```



For a description of the keywords, see section 9.6.3, [Defining Internal Memory and Buses](#).

With the keyword **bus** you define a bus (the combination of data and corresponding address bus). The bus name is used to identify a bus and does not conflict with other identifiers. Bus descriptions at the global scope (outside any definition) define *external* buses. These are buses that are present on the target board.

```
bus bus_name
{
    mau = 8;
    width = 8;
    map ( map_description );
}
```



For a description of the keywords, see section 9.5.2, [Defining Internal Buses](#).

You can connect off-chip memory to any derivative: you need to map the off-chip memory to a bus and map that bus on the internal bus of the derivative you want to connect it to.

9.8 Semantics of the Section Setup Definition

Keywords in the section setup definition

```

section_setup
  stack
    min_size
    grows          low_to_high  high_to_low
    align
    fixed
    id
  heap
    min_size
    grows          low_to_high  high_to_low
    align
    fixed
    id
  vector_table
    vector_size
    size
    id_symbol_prefix
    run_addr
    template
    template_symbol
    vector_prefix
    fill
    no_inline
    copy
    vector
      id
      fill          loop
  reserved

```

9.8.1 Setting up a Section

With the keyword **section_setup** you can define stacks, heaps, vector tables, and/or reserved address ranges outside their address space definition.

```

section_setup ::my_space
{
  vector table statements
  reserved address range
  stack definition
  heap definition
}

```



See the subsections [Stacks and heaps](#), [Vector table](#) and [Reserved address ranges](#) in section 9.5.3, [Defining Address Spaces](#), for details on the keywords **stack**, **heap**, **vector_table** and **reserved**.

9.9 Semantics of the Section Layout Definition

Keywords in the section layout definition

```

section_layout
    direction      low_to_high  high_to_low
group
    align
    attributes     + -  r w x b i s
    copy
    nocopy
    fill
    ordered
    contiguous
    clustered
    overlay
    allow_cross_references
    load_addr
        mem
    run_addr
        mem
    page
    page_size
    priority
select
stack
    size
heap
    size
reserved
    size
    attributes     r w x
    fill
    alloc_allowed absolute ranged
copytable
section
    size
    blocksize
    attributes     r w x
    fill
    overflow

if
else

```

9.9.1 Defining a Section Layout

With the keyword **section_layout** you define a section layout for exactly one address space. In the section layout you can specify how input sections are placed in the address space, relative to each other, and what the absolute run and load addresses of each section will be.

You can define one or more section definitions. Each section definition arranges the sections in one address space. You can precede the address space name with a processor name and/or core name, separated by colons. You can omit the processor name and/or the core name if only one processor is defined and/or only one core is present in the processor. A reference to a space in the only core of the only processor in the system would look like `":my_space"`. A reference to a space of the only core on a specific processor in the system could be `my_chip:my_space`. The next example shows a section definition for sections in the `my_space` address space of the processor called `my_chip`:

```
section_layout my_chip::my_space ( locate_direction )
{
    section statements
}
```

With the optional keyword **direction** you specify whether the linker starts locating sections from **low_to_high** (default) or from **high_to_low**. In the second case the linker starts locating sections at the highest addresses in the address space but preserves the order of sections when necessary (one processor and core in this example).

```
section_layout ::my_space ( direction = high_to_low )
{
    section statements
}
```



If you do not explicitly tell the linker how to locate a section, the linker decides on the basis of the section attributes in the object file and the information in the architecture definition and memory parts where to locate the section.

9.9.2 Creating and Locating Groups of Sections

Sections are located per group. A group can contain one or more (sets of) input sections as well as other groups. Per group you can assign a mutual order to the sets of sections and locate them into a specific memory part.

```
group ( group_specifications )
{
    section_statements
}
```

With the *section_statements* you generally select sets of sections to form the group. This is described in subsection [Selecting sections for a group](#).

Instead of selecting sections, you can also modify special sections like stack and heap or create a reserved section. This is described in section 9.9.3, [Creating or Modifying Special Sections](#).

With the *group_specifications* you actually locate the sections in the group. This is described in subsection [Locating a group](#).

Selecting sections for a group

With the **select** keyword you can select one or more sections for the group. You can select a section by name or by attributes. If you select a section by name, you can use a wildcard pattern:

```
"*"      matches with all section names
"?"      matches with a single character in the section name
"\ "     takes the next character literally
"[abc]"  matches with a single 'a', 'b' or 'c' character
"[a-z]"  matches with any single character in the range 'a' to 'z'
```

```
group ( ... )
{
    select "mysection";
    select "*";
}
```

The first **select** statement selects the section with the name "mysection". The second **select** statement selects all sections that were not selected yet.

A section is selected by the first **select** statement that matches, in the union of all section layouts for the address space. Global section layouts are processed in the order in which they appear in the LSL file. Internal core architecture section layouts always take precedence over global section layouts.

- The **attributes** field selects all sections that carry (or do not carry) the given attribute. With **+attribute** you select sections that have the specified attribute set. With **-attribute** you select sections that do not have the specified attribute set. You can specify one or more of the following attributes:
 - **r** readable sections
 - **w** writable sections

- **x** executable sections
- **i** initialized sections
- **b** sections that should be cleared at program startup
- **s** scratch sections (not cleared and not initialized)

To select all read-only sections:

```
group ( ... )
{
    select (attributes = +r-w);
}
```



Keep in mind that all section selections are restricted to the address space of the section layout in which this group definition occurs.

- With the **ref_tree** field you can select a group of related sections. The relation between sections is often expressed by means of references. By selecting just the 'root' of tree, the complete tree is selected. This is for example useful to locate a group of related sections in special memory (e.g. fast memory). The (referenced) sections must meet the following conditions in order to be selected:

1. The sections are within the section layout's address space
2. The sections match the specified attributes
3. The sections have no absolute restriction (as is the case for all wildcard selections)

For example, to select the code sections referenced from `foo1`:

```
group refgrp (ordered, contiguous, run_addr=mem:ext_c)
{
    select ref_tree "foo1" (attributes=+x);
}
```

If section `foo1` references `foo2` and `foo2` references `foo3`, then all these sections are selected by the selection shown above.

Locating a group

```
group group_name ( group_specifications )
{
    section_statements
}
```

With the *group_specifications* you actually define how the linker must locate the group. You can roughly define three things: 1) assign properties to the group like alignment and read/write attributes, 2) define the mutual order in the address space for sections in the group and 3) restrict the possible addresses for the sections in a group.

The linker creates labels that allow you to refer to the begin and end address of a group from within the application software. Labels `_lc_gb_group_name` and `_lc_ge_group_name` mark the begin and end of the group respectively, where the begin is the lowest address used within this group and the end is the highest address used. Notice that a group not necessarily occupies all memory between begin and end address. The given label refers to where the section is located at run-time (versus load-time).

1. Assign properties to the group like alignment and read/write attributes.
These properties are assigned to all sections in the group (and subgroups) and override the attributes of the input sections.
 - The **align** field tells the linker to align all sections in the group and the group as a whole according to the align value. By default the linker uses the largest alignment constraint of either the input sections or the alignment of the address space.
 - The **attributes** field tells the linker to assign one or more attributes to all sections in the group. This overrules the default attributes. By default the linker uses the attributes of the input sections. You can set the **r**, **w** or **rw** attributes and you can switch between the **b** and **s** attributes.
 - The **copy** field tells the linker to locate a read-only section in RAM and generate a ROM copy and a copy action in the copy table. This property makes the sections in the group writable which causes the linker to generate ROM copies for the sections.

- The effect of the **nocopy** field is the opposite of the copy field. It prevents the linker from generating ROM copies of the selected sections.

2. Define the mutual order of the sections in the group.

By default, a group is *unrestricted* which means that the linker has total freedom to place the sections of the group in the address space.

- The **ordered** keyword tells the linker to locate the sections in the same order in the address space as they appear in the group (but not necessarily adjacent).

Suppose you have an ordered group that contains the sections 'A', 'B' and 'C'. By default the linker places the sections in the address space like 'A' – 'B' – 'C', where section 'A' gets the lowest possible address. With **direction=high_to_low** in the **section_layout** space properties, the linker places the sections in the address space like 'C' – 'B' – 'A', where section 'A' gets the highest possible address.

- The **contiguous** keyword tells the linker to locate the sections in the group in a single address range. Within a contiguous group the input sections are located in arbitrary order, however the group occupies one contiguous range of memory. Due to alignment of sections there can be 'alignment gaps' between the sections.

When you define a group that is both **ordered** and **contiguous**, this is called a *sequential* group. In a sequential group the linker places sections in the same order in the address space as they appear in the group and it occupies a contiguous range of memory.

- The **clustered** keyword tells the linker to locate the sections in the group in a number of *contiguous* blocks. It tries to keep the number of these blocks to a minimum. If enough memory is available, the group will be located as if it was specified as **contiguous**. Otherwise, it gets split into two or more blocks.

If a contiguous or clustered group contains *alignment gaps*, the linker can locate sections that are not part of the group in these gaps. To prevent this, you can use the **fill** keyword. If the group is located in RAM, the gaps are treated as reserved (scratch) space. If the group is located in ROM, the alignment gaps are filled with zeros by default. You can however change the fill pattern by specifying a bit pattern. The result of the expression, or list of expressions, is used as values to write to memory, each in MAU.

- The **overlay** keyword tells the linker to overlay the sections in the group. The linker places all sections in the address space using a contiguous range of addresses. (Thus an overlay group is automatically also a contiguous group.) To overlay the sections, all sections in the overlay group share the same run-time address.

For each input section within the overlay the linker automatically defines two symbols. The symbol **_lc_cb_section_name** is defined as the load-time start address of the section. The symbol **_lc_ce_section_name** is defined as the load-time end address of the section. C (or assembly) code may be used to copy the overlaid sections.

If sections in the overlay group contain references between groups, the linker reports an error. The keyword **allow_cross_references** tells the linker to accept cross-references. Normally, it does not make sense to have references between sections that are overlaid.

```
group ovl (overlay)
{
    group a
    {
        select "my_ovl_p1";
        select "my_ovl_p2";
    }
    group b
    {
        select "my_ovl_q1";
    }
}
```



It may be possible that one of the sections in the overlay group already has been defined in another group where it received a load-time address. In this case the linker does not overrule this load-time address and excludes the section from the overlay group.

3. Restrict the possible addresses for the sections in a group.

The load-time address specifies where the group's elements are loaded in memory at download time. The run-time address specifies where sections are located at run-time, that is when the program is executing. If you do not explicitly restrict the address in the LSL file, the linker assigns addresses to the sections based on the restrictions relative to other sections in the LSL file and section alignments. The program is responsible for copying overlay sections at appropriate moment from its load-time location to its run-time location (this is typically done by the startup code).

- The **run_addr** keyword defines the run-time address. If the run-time location of a group is set explicitly, the given order between groups specify whether the run-time address propagates to the parent group or not. The location of the sections a group can be restricted either to a single absolute address, or to a number of address ranges (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 (run_addr = 0xa00f0000)
```

You can use the '[offset]' variant to locate the group at the given absolute offset in memory:

```
group (run_addr = mem:A[0x1000])
```

A range can be an absolute space address range, written as [*expr* .. *expr*], a complete memory device, written as **mem:mem_name**, or a memory address range,

```
mem:mem_name[expr .. expr]
```

```
group (run_addr = mem:my_dram)
```

You can use the '|' to specify an address range of more than one physical memory device:

```
group (run_addr = mem:A | mem:B)
```

- The **load_addr** keyword changes the meaning of the section selection in the group: the linker selects the load-time ROM copy of the named section(s) instead of the regular sections. Just like **run_addr** you can specify an absolute address or an address range.

The **load_addr** keyword itself (without an assignment) specifies that the group's position in the LSL file defines its load-time address.

```
group (load_addr)
    select "mydata"; // select ROM copy of mydata: "[mydata]"
```

The load-time and run-time addresses of a group cannot be set at the same time. If the load-time property is set for a group, the group (only) restricts the positioning at load-time of the group's sections. It is not possible to set the address of a group that has a not-unrestricted parent group.

The properties of the load-time and run-time start address are:

- At run-time, before using an element in an overlay group, the application copies the sections from their load location to their run-time location, but only if these two addresses are different. For non-overlay sections this happens at program start-up.
- The start addresses cannot be set to absolute values for unrestricted groups.
- For non-overlay groups that do not have an overlay parent, the load-time start address equals the run-time start address.
- For any group, if the run-time start address is not set, the linker selects an appropriate address.
- 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 9.5.3, [Defining Address Spaces](#).

```
group (page, ... )
group (page = 3, ...)
```

- 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**.

```
group (priority=2)
{
    select "importantcode1";
    select "importantcode2";
}
```

9.9.3 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**.

```
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 9.5.3, [Defining Address Spaces](#).

Heap

- The keyword **heap** tells the linker to reserve a dynamic memory range for the `malloc()` function. Optionally you can assign a name to the heap section. With the keyword **size** you can change the size for the heap. If the **size** is not specified, the linker uses the size given by the **min_size** argument as defined for the heap in the architecture definition. Normally the linker automatically tries to maximize the size, unless you specified the keyword **fixed**.

```
group ( ... )
{
    heap "myheap" ( size = 2k );
}
```

The linker creates two labels to mark the begin and end of the heap, `_lc_ub_heap_name` for the begin of the heap and `_lc_ue_heap_name` for the end of the heap. The linker allocates space for the heap when a reference to either of the section labels exists in one of the input object files.

Reserved section

- The keyword **reserved** tells the linker to create an area or section of a given size. The linker will not locate any other sections in the memory occupied by a reserved section, with some exceptions. Optionally you can assign a name to a reserved section. With the keyword **size** you can specify a size for a given reserved area or section.

```
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.

Properties set in LSL		Resulting section properties		
attributes	filled	access	memory	content
x	yes		<rom>	executable
r	yes	r	<rom>	data
r	no	r	<rom>	scratch
rx	yes	r	<rom>	executable
rw	yes	rw	<ram>	data
rw	no	rw	<ram>	scratch
rwX	yes	rw	<ram>	executable

```
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 start, and **_lc_ue_name** for the end of the reserved section.

Output sections

- The keyword **section** tells the linker to accumulate sections obtained from object files ("input sections") into an output section of a fixed size in the locate phase. You can select the input sections with **select** statements. You can use groups inside output sections, but you can only set the **align**, **attributes** and **load_addr** attributes.

The **fill** field contains a bit pattern that the linker writes to all unused space in the output section. When all input sections have an image (code/data) you must specify a fill pattern. If you do not specify a fill pattern, all input sections must be scratch sections. The fill pattern is aligned at the start of the output section.

As with a reserved section you can use the **attributes** field to set the access type of the output section.

```
group ( ... )
{
    section "myoutput" ( size = 4k, attributes = rw, fill = 0xaa )
    {
        select "myinput1";
        select "myinput2";
    }
}
```

The available room for input sections is determined by the **size**, **blocksize** and **overflow** fields. With the keyword **size** you specify the fixed size of the output section. Input sections are placed from output section start towards higher addresses (offsets). When the end of the output section is reached and one or more input sections are not yet placed, an error is emitted. If however, the **overflow** field is set to another output section, remaining sections are located as if they were selected for the overflow output section.

```
group ( ... )
{
    section "tsk1_data" (size=4k, attributes=rw, fill=0,
                       overflow = "overflow_data")
    {
        select ".data.tsk1.*"
    }
    section "tsk2_data" (size=4k, attributes=rw, fill=0,
                       overflow = "overflow_data")
    {
        select ".data.tsk2.*"
    }
    section "overflow_data" (size=4k, attributes=rx,
                           fill=0)
    {
    }
}
```

With the keyword **blocksize**, the size of the output section will adapt to the size of its content. For example:

```
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 4k, if the content is 11k, the section will be 12k, 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 start, and **_lc_ue_name** for the end of the output section.

Copy table

- The keyword **copytable** tells the linker to select a section that is used as *copy table*. The content of the copy table is created by the linker. It contains the start address and length of all sections that should be initialized by the startup code.

The linker creates two labels to mark the begin and end of the section, **_lc_ub_table** for the start, and **_lc_ue_table** for the end of the copy table. The linker generates a copy table when a reference to either of the section labels exists in one of the input object files.

9.9.4 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
}
```

9.9.5 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 );
}
```




10 CERT C Secure Coding Standard

Summary

This chapter contains an overview of the CERT C Secure Coding Standard recommendations and rules that are supported by the TASKING VX-toolset.

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 see the [CERT C Secure Coding Standard](http://www.cert.org/secure-coding) 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.

10.1 Preprocessor (PRE)

PRE01-C

Use parentheses within macros around parameter names

Parenthesize all parameter names in macro definitions to avoid precedence problems.

PRE02-C

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:

```
#define EOF -1          // should be (-1)
int getchar(void);
void f(void)
{
    if (getchar() EOF) // != operator omitted
    {
        /* ... */
    }
}
```

- PRE10-C** Wrap multi-statement macros in a do-while loop
- When multiple statements are used in a macro, enclose them in a `do-while` statement, so the macro can appear safely inside `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.
- PRE11-C** 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.

10.2 Declarations and Initialization (DCL)

- DCL30-C** 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.
- DCL31-C** 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 VX-toolset 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:
- ```
void my_function(int a);
int main(void)
{
 int (*new_function)(int a) = my_function;
 return (*new_function)(10); /* the behavior is undefined */
}
```

## 10.3 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<br>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<br>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 <code>offsetof()</code> on bit-field members or invalid types<br>The behavior of the <code>offsetof()</code> macro is undefined when the member designator parameter designates a bit-field.                                                                                                                                                                                   |

## 10.4 Integers (INT)

---

|         |                                                                                                                                                                                                                                                                                                                                                               |
|---------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| INT30-C | Ensure that unsigned integer operations do not wrap<br>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<br>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<br>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.                                                                         |

## 10.5 Floating Point (FLP)

---

|         |                                                                                                                                                                                                  |
|---------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| FLP30-C | Do not use floating point variables as loop counters<br>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<br>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<br>Conversion from integral types to floating point types without sufficient precision can lead to loss of precision.  |

## 10.6 Arrays (ARR)

---

|         |                                                                                                                                                                                                                                                                                                                        |
|---------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ARR01-C | Do not apply the <code>sizeof</code> operator to a pointer when taking the size of an array<br>A function parameter declared as an array, is converted to a pointer by the compiler. Therefore, the <code>sizeof</code> 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<br>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<br>Reading or writing of data outside the bounds of an array may lead to incorrect program behavior or execution of arbitrary code.                                                                                                                           |

## 10.7 Characters and Strings (STR)

---

|         |                                                                                                                                                                                                                                                             |
|---------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| STR30-C | Do not attempt to modify string literals<br>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<br>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<br>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<br>The compiler issues this warning when the character buffer initialized by a string literal does not provide enough room for the terminating null character.              |

## 10.8 Memory Management (MEM)

---

|         |                                                                                                                                                                                                                                                                                                                                                                                                                                          |
|---------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| MEM00-C | Allocate and free memory in the same module, at the same level of abstraction<br>The compiler issues this warning when the result of the call to <code>malloc()</code> , <code>calloc()</code> or <code>realloc()</code> is discarded, and therefore not <code>free()</code> d, resulting in a memory leak.                                                                                                                              |
| MEM08-C | Use <code>realloc()</code> only to resize dynamically allocated arrays<br>Only use <code>realloc()</code> 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<br>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<br>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<br>The result of <code>realloc()</code> is assigned to the original pointer, without checking for failure. As a result, the original block of memory is lost when <code>realloc()</code> fails.                                                                                                                                                                                               |
| MEM33-C | Use the correct syntax for flexible array members<br>Use the ISO C99 syntax for flexible array members instead of an array member of size 1.                                                                                                                                                                                                                                                                                             |
| MEM34-C | Only free memory allocated dynamically<br>Freeing memory that is not allocated dynamically can lead to corruption of the heap data structures.                                                                                                                                                                                                                                                                                           |
| MEM35-C | Allocate sufficient memory for an object<br>The compiler issues this warning when the size of the object(s) allocated by <code>malloc()</code> , <code>calloc()</code> or <code>realloc()</code> is smaller than the size of an object pointed to by the result pointer. This may be caused by a <code>sizeof</code> expression with the wrong type or with a pointer type instead of the object type.                                   |

## 10.9 Environment (ENV)

---

|         |                                                                                                                                                                                                                                                                              |
|---------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ENV32-C | All <code>atexit</code> handlers must return normally<br>The compiler issues this warning when an <code>atexit()</code> handler is calling a function that does not return. No <code>atexit()</code> registered handler should terminate in any way other than by returning. |
|---------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

## 10.10 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.

## 10.11 Miscellaneous (MSC)

---

**MSC32-C** Ensure your random number generator is properly seeded

Ensure that the random number generator is properly seeded by calling `srand()`.





### Summary

This chapter contains an overview of the supported and unsupported MISRA-C rules.

### 11.1 MISRA-C:1998

This section lists all supported and unsupported MISRA-C:1998 rules.



See also section 2.7.2, *C Code Checking: MISRA-C*, in Chapter *Using the Compiler* of the *User's Manual*.



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
- x** 2. (A) Other languages should only be used with an interface standard
3. (A) Inline assembly is only allowed in dedicated C functions
- x** 4. (A) Provision should be made for appropriate run-time checking
5. (R) Only use characters and escape sequences defined by ISO C
- x** 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'
- x** 15. (A) Floating-point implementations should comply with a standard
16. (R) The bit representation of floating-point numbers shall not be used  
A violation is reported when a pointer to a floating-point type is converted to a pointer to an integer type.
17. (R) "typedef" names shall not be reused
18. (A) Numeric constants should be suffixed to indicate type  
A violation is reported when the value of the constant is outside the range indicated by the suffixes, if any.
19. (R) Octal constants (other than zero) shall not be used
20. (R) All object and function identifiers shall be declared before use
21. (R) Identifiers shall not hide identifiers in an outer scope
22. (A) Declarations should be at function scope where possible

- x 23. (A) All declarations at file scope should be static where possible
- 24. (R) Identifiers shall not have both internal and external linkage
- x 25. (R) Identifiers with external linkage shall have exactly one definition
- 26. (R) Multiple declarations for objects or functions shall be compatible
- x 27. (A) External objects should not be declared in more than one file
- 28. (A) The "register" storage class specifier should not be used
- 29. (R) The use of a tag shall agree with its declaration
- 30. (R) All automatics shall be initialized before being used  
This rule is checked using worst-case assumptions. This means that violations are reported not only for variables that are guaranteed to be uninitialized, but also for variables that are uninitialized on some execution paths.
- 31. (R) Braces shall be used in the initialization of arrays and structures
- 32. (R) Only the first, or all enumeration constants may be initialized
- 33. (R) The right hand operand of && or || shall not contain side effects
- 34. (R) The operands of a logical && or || shall be primary expressions
- 35. (R) Assignment operators shall not be used in Boolean expressions
- 36. (A) Logical operators should not be confused with bitwise operators
- 37. (R) Bitwise operations shall not be performed on signed integers
- 38. (R) A shift count shall be between 0 and the operand width minus 1  
This violation will only be checked when the shift count evaluates to a constant value at compile time.
- 39. (R) The unary minus shall not be applied to an unsigned expression
- 40. (A) "sizeof" should not be used on expressions with side effects
- x 41. (A) The implementation of integer division should be documented
- 42. (R) The comma operator shall only be used in a "for" condition
- 43. (R) Don't use implicit conversions which may result in information loss
- 44. (A) Redundant explicit casts should not be used
- 45. (R) Type casting from any type to or from pointers shall not be used
- 46. (R) The value of an expression shall be evaluation order independent  
This rule is checked using worst-case assumptions. This means that a violation will be reported when a possible alias may cause the result of an expression to be evaluation order dependent.
- 47. (A) No dependence should be placed on operator precedence rules
- 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



- 102. (A) No more than 2 levels of pointer indirection should be used  
A violation is reported when a pointer with three or more levels of indirection is declared.
- 103. (R) No relational operators between pointers to different objects  
In general, checking whether two pointers point to the same object is impossible. The compiler will only report a violation for a relational operation with incompatible pointer types.
- 104. (R) Non-constant pointers to functions shall not be used
- 105. (R) Functions assigned to the same pointer shall be of identical type
- 106. (R) Automatic address may not be assigned to a longer lived object
- 107. (R) The null pointer shall not be de-referenced  
A violation is reported for every pointer dereference that is not guarded by a NULL pointer test.
- 108. (R) All struct/union members shall be fully specified
- 109. (R) Overlapping variable storage shall not be used  
A violation is reported for every 'union' declaration.
- 110. (R) Unions shall not be used to access the sub-parts of larger types  
A violation is reported for a 'union' containing a 'struct' member.
- 111. (R) bit-fields shall have type "unsigned int" or "signed int"
- 112. (R) bit-fields of type "signed int" shall be at least 2 bits long
- 113. (R) All struct/union members shall be named
- 114. (R) Reserved and standard library names shall not be redefined
- 115. (R) Standard library function names shall not be reused
- x 116. (R) Production libraries shall comply with the MISRA-C restrictions
- x 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/atol/atoi 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



See also section 2.7.2, *C Code Checking: MISRA-C*, in Chapter *Using the Compiler* of the User's manual.

## 11.2 MISRA-C:2004

This section lists all supported and unsupported MISRA-C:2004 rules.



See also section 2.7.2, *C Code Checking: MISRA-C*, in Chapter *Using the Compiler* of the *User's Manual*.



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

- 1.1 (R) All code shall conform to ISO 9899:1990 "Programming languages – C", amended and corrected by ISO/IEC 9899/COR1:1995, ISO/IEC 9899/AMD1:1995, and ISO/IEC 9899/COR2:1996.
- 1.2 (R) No reliance shall be placed on undefined or unspecified behavior.
- x** 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/compiler/assemblers conform.
- x** 1.4 (R) The compiler/linker shall be checked to ensure that 31 character significance and case sensitivity are supported for external identifiers.
- x** 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

- x** 3.1 (R) All usage of implementation-defined behavior shall be documented.
- x** 3.2 (R) The character set and the corresponding encoding shall be documented.
- x** 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.
- x** 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.
- x 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.
- x 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.
- x 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.
- x 8.8 (R) An external object or function shall be declared in one and only one file.
- x 8.9 (R) An identifier with external linkage shall have exactly one external definition.
- x 8.10 (R) All declarations and definitions of objects or functions at file scope shall have internal linkage unless external linkage is required.
- 8.11 (R) The `static` storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage.
- 8.12 (R) When an array is declared with external linkage, its size shall be stated explicitly or defined implicitly by initialization.

**Initialization**

- 9.1 (R) 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.
- 9.2 (R) Braces shall be used to indicate and match the structure in the non-zero initialization of arrays and structures.
- 9.3 (R) In an enumerator list, the "=" construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.

**Arithmetic type conversions**

- 10.1 (R) The value of an expression of integer type shall not be implicitly converted to a different underlying type if:
  - a) it is not a conversion to a wider integer type of the same signedness, or
  - b) the expression is complex, or
  - c) the expression is not constant and is a function argument, or
  - d) the expression is not constant and is a return expression.
- 10.2 (R) The value of an expression of floating type shall not be implicitly converted to a different type if:
  - a) it is not a conversion to a wider floating type, or
  - b) the expression is complex, or
  - c) the expression is a function argument, or
  - d) the expression is a return expression.
- 10.3 (R) The value of a complex expression of integer type may only be cast to a type that is narrower and of the same signedness as the underlying type of the expression.
- 10.4 (R) The value of a complex expression of floating type may only be cast to a narrower floating type.
- 10.5 (R) 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.
- 10.6 (R) A "U" suffix shall be applied to all constants of `unsigned` type.

**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.
- x 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.
- x 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.

# Index

## Symbols

- `#define`, 6–68
- `#include`, system include directory, 6–147
- `#undef`, 6–154
- `__ABI_CHANGES_FOR_ARRAY_NEW_AND_DELETE`, 2–22
- `__ABI_CHANGES_FOR_PLACEMENT_DELETE`, 2–22
- `__ABI_CHANGES_FOR_RTTI`, 2–22
- `__ABI_COMPATIBILITY_VERSION`, 2–22
- `align()`, 1–2
- `ARM__`, 1–12
- `__ARRAY_OPERATORS`, 2–22
- `__ASARM__`, 4–3
- `asm`, syntax, 1–4
- `at()`, 1–3
- `__BASE_FILE__`, 2–22
- `__BIG_ENDIAN__`, 1–12, 2–22
- `__BUILD__`, 1–12, 2–22, 4–3
- `__CARM__`, 1–12
- `__CHAR_MAX`, 2–22
- `__CHAR_MIN`, 2–22
- `__CPARM__`, 2–22
- `__cplusplus`, 2–22
- `__CPU__`, 1–12, 2–22
- `__CPU_arch__`, 1–12
- `__DATE__`, 2–22
- `__DELTA_TYPE`, 2–22
- `__DOUBLE_FP__`, 1–12, 2–22
- `__embedded_cplusplus`, 2–22
- `__EXCEPTIONS`, 2–22
- `__FILE__`, 2–22
- `__func__`, 2–22
- `__IMPLICIT_USING_STD`, 2–22
- `interrupt()`, 1–18
- `__interrupt_dabt`, 1–18
- `__interrupt_fiq`, 1–18
- `__interrupt_iabt`, 1–18
- `__interrupt_irq`, 1–18
- `__interrupt_svc`, 1–18
- `__interrupt_und`, 1–18
- `__JMP_BUF_ELEMENT_TYPE`, 2–22
- `__JMP_BUF_NUM_ELEMENTS`, 2–22
- `__LINE__`, 2–22
- `__LITTLE_ENDIAN__`, 1–12, 2–23
- `__NAMESPACES`, 2–23
- `__nesting_enabled`, 1–19
- `__NO_LONG_LONG`, 2–23
- `__noinline`, 1–15
- `__novector`, 1–19
- `__NULL_EH_REGION_NUMBER`, 2–23
- `__packed__`, 1–2
- `__PLACEMENT_DELETE`, 2–23
- `__PRETTY_FUNCTION__`, 2–23
- `__PTRDIFF_MAX`, 2–23
- `__PTRDIFF_MIN`, 2–23
- `__REGION_NUMBER_TYPE`, 2–23
- `__REVISION__`, 1–12, 2–23, 4–3
- `__RTTI`, 2–23
- `__RUNTIME_USES_NAMESPACES`, 2–23
- `__SIGNED_CHARS__`, 2–23
- `__SINGLE_FP__`, 1–12, 2–23
- `__SIZE_MAX`, 2–23
- `__SIZE_MIN`, 2–23
- `__STDC__`, 2–23
- `__STDC_VERSION__`, 2–23
- `__TASKING__`, 1–12, 2–23, 4–3
- `__THUMB__`, 1–12, 2–23
- `__TIME__`, 2–23
- `__TYPE_TRAITS_ENABLED`, 2–23
- `__unaligned`, 1–2
- `__VAR_HANDLE_TYPE`, 2–23
- `__VERSION__`, 1–12, 2–23, 4–3
- `__VIRTUAL_FUNCTION_INDEX_TYPE`, 2–23
- `__VIRTUAL_FUNCTION_TYPE`, 2–23
- `__WCHAR_MAX`, 2–23
- `__WCHAR_MIN`, 2–23
- `__BOOL`, 2–22
- `_close`, 3–7
- `_Complex`, 3–3
- `_Exit`, 3–19
- `_Imaginary`, 3–3
- `_IOFBF`, 3–14
- `_IOLBF`, 3–14
- `_IONBF`, 3–14
- `_lseek`, 3–7
- `_open`, 3–7
- `_read`, 3–7
- `_START`, 5–1, 5–2
- `_STLP_NO_IOSTREAMS`, 2–23
- `_tolower`, 3–4
- `_unlink`, 3–7
- `_WCHAR_T`, 2–23
- `_write`, 3–7

## A

- `abort`, 3–19
- `abs`, 3–20
- Absolute Address, 1–3
- `access`, 3–23
- `acos` functions, 3–8
- `acosh` functions, 3–8
- Address spaces, 9–15
- `alias`, 1–8
- `align`, 4–11
- Alignment, 1–2
  - composite types*, 6–3
- Alignment gaps, 9–28
- Alternative C++ tokens, 6–59
- `alupcrel`, 4–6
- Anachronism, 2–4
- Anachronisms, 6–60, 6–117
- Architecture definition, 9–1, 9–14

- arg, [4-6](#)
- Argument, [3-4](#)
- arm, [4-15](#)
- array new and delete, [6-102](#)
- asctime, [3-22](#)
- asin functions, [3-8](#)
- asinh functions, [3-8](#)
- Assembler directives
  - [.align](#), [4-11](#)
  - [.arm](#), [4-15](#)
  - [.break](#), [4-12](#)
  - [.bs](#), [4-13](#)
  - [.bsb](#), [4-13](#)
  - [.bsd](#), [4-13](#)
  - [.bsh](#), [4-13](#)
  - [.bsw](#), [4-13](#)
  - [.calls](#), [4-14](#)
  - [.code16](#), [4-15](#)
  - [.code32](#), [4-15](#)
  - [.db](#), [4-16](#)
  - [.dd](#), [4-17](#)
  - [.define](#), [4-18](#)
  - [.dh](#), [4-19](#)
  - [.double](#), [4-25](#)
  - [.ds](#), [4-20](#)
  - [.dsb](#), [4-20](#)
  - [.dsd](#), [4-20](#)
  - [.dsh](#), [4-20](#)
  - [.dsw](#), [4-20](#)
  - [.dw](#), [4-21](#)
  - [.end](#), [4-22](#)
  - [.equ](#), [4-23](#)
  - [.extern](#), [4-24](#)
  - [.float](#), [4-25](#)
  - [.for/.endfor](#), [4-26](#)
  - [.global](#), [4-27](#)
  - [.if/.elif/.else/.endif](#), [4-28](#)
  - [.include](#), [4-29](#)
  - [.list/.nolist](#), [4-30](#)
  - [.ltorg](#), [4-31](#)
  - [.macro/.endm](#), [4-32](#)
  - [.message](#), [4-34](#)
  - [.offset](#), [4-35](#)
  - [.page](#), [4-36](#)
  - [.repeat/.endrep](#), [4-37](#)
  - [.section](#), [4-38](#)
  - [.set](#), [4-39](#)
  - [.size](#), [4-40](#)
  - [.source](#), [4-41](#)
  - [.thumb](#), [4-15](#)
  - [.title](#), [4-42](#)
  - [.type](#), [4-43](#)
  - [.undef](#), [4-44](#)
- ARM specific (overview), [4-10](#)
- assembly control (overview), [4-9](#)
- conditional assembly (overview), [4-10](#)
- data definition (overview), [4-10](#)
- detailed description, [4-11](#)
- HLL (overview), [4-10](#)
- listing control (overview), [4-10](#)
- macros (overview), [4-10](#)
- overview, [4-9](#)
- storage allocation (overview), [4-10](#)
- symbol definitions (overview), [4-9](#)
- weak, [4-45](#)
- Assembler options, [6-162](#)
  - [-?](#), [6-173](#)
  - [--case-insensitive](#), [6-163](#)
  - [--check](#), [6-164](#)
  - [--cpu](#), [6-165](#)
  - [--debug-info](#), [6-166](#)
  - [--define](#), [6-167](#)
  - [--diag](#), [6-168](#)
  - [--emit-locals](#), [6-169](#)
  - [--endianness](#), [6-170](#)
  - [--error-file](#), [6-171](#)
  - [--error-limit](#), [6-172](#)
  - [--help](#), [6-173](#)
  - [--include-directory](#), [6-174](#)
  - [--include-file](#), [6-175](#)
  - [--inversions](#), [6-176](#)
  - [--list-file](#), [6-178](#)
  - [--list-format](#), [6-179](#)
  - [--no-warnings](#), [6-180](#)
  - [--old-syntax](#), [6-181](#)
  - [--optimize](#), [6-182](#)
  - [--option-file](#), [6-183](#)
  - [--output](#), [6-184](#)
  - [--page-length](#), [6-185](#)
  - [--page-width](#), [6-186](#)
  - [--preprocess](#), [6-187](#)
  - [--preprocessor-type](#), [6-188](#)
  - [--relaxed](#), [6-189](#)
  - [--section-info](#), [6-190](#)
  - [--symbol-scope](#), [6-191](#)
  - [--thumb](#), [6-192](#)
  - [--verbose](#), [6-194](#)
  - [--version](#), [6-193](#)
  - [--warnings-as-errors](#), [6-195](#)
  - [-B \(--big-endian\)](#), [6-170](#)
  - [-C](#), [6-165](#)
  - [-c](#), [6-163](#)
  - [-D](#), [6-167](#)
  - [-E](#), [6-187](#)
  - [-f](#), [6-183](#)
  - [-g](#), [6-166](#)
  - [-H](#), [6-175](#)
  - [-I](#), [6-174](#)
  - [-i](#), [6-191](#)
  - [-k](#), [6-177](#)
  - [-k \(--keep-output-files\)](#), [6-177](#)
  - [-L](#), [6-179](#)
  - [-l](#), [6-178](#)
  - [-m](#), [6-188](#)
  - [-O](#), [6-182](#)
  - [-o](#), [6-184](#)

- T, 6-192
- t, 6-190
- V, 6-193
- v, 6-194
- w, 6-180
- debug information, 6-166, 6-169
- diagnostics, 6-172, 6-180, 6-195
- inverse instructions, 6-176
- list file, 6-178, 6-179, 6-190
- optimization, 6-182
- preprocessing, 6-167, 6-175, 6-187, 6-188
- relaxed 2-operand syntax, 6-189
- Thumb instructions, 6-192
- Assembler significant characters, 4-2
- Assembly, Programming in C, 1-4
- Assembly expressions, 4-3
- Assembly functions, 4-5
  - @alupcrel, 4-6
  - @arg, 4-6
  - @bigendian, 4-6
  - @cnt, 4-7
  - @cpu, 4-7
  - @defined, 4-7
  - @lsb, 4-7
  - @lsh, 4-7
  - @lsw, 4-7
  - @msb, 4-7
  - @msh, 4-7
  - @msw, 4-8
  - @pre\_uai, 4-8
  - @strcat, 4-8
  - @strcmp, 4-8
  - @strlen, 4-8
  - @strpos, 4-8
  - @thumb, 4-8
  - detailed description, 4-6
  - overview, 4-5
- Assembly syntax, 4-1
- atan functions, 3-8
- atan2 functions, 3-8
- atanh functions, 3-8
- atexit, 3-19
- atof, 3-19
- atoi, 3-19
- atol, 3-19
- atoll, 3-19

## B

- bigendian, 4-6
- Binary search table, 1-13
- binary\_switch, 1-10
- Board specification, 9-2, 9-22
- bool keyword, 6-104
- break, 4-12
- bs, 4-13
- bsb, 4-13
- bsd, 4-13
- bsearch, 3-20

- bsh, 4-13
- bsw, 4-13
- btowc, 3-24
- BUFSIZ, 3-13
- Build options, 6-216
  - include files path, 6-18, 6-34, 6-87, 6-174, 6-212
- Bus definition, 9-2
- Buses, 9-15

## C

- C compiler options, 6-1
  - , 6-17
  - align-composites, 6-3
  - big-endian, 6-13
  - call, 6-4
  - cert, 6-5
  - check, 6-6
  - cpu, 6-7
  - debug-info, 6-8
  - define, 6-9
  - dep-file, 6-11
  - diag, 6-12
  - endianness, 6-13
  - error file, 6-14
  - global-type-checking, 6-16
  - help, 6-17
  - include-directory, 6-18
  - include-file, 6-19
  - inline, 6-20
  - inline-max-incr, 6-21
  - inline-max-size, 6-21
  - interwork, 6-22
  - iso, 6-23
  - keep-output-files, 6-24
  - language, 6-25
  - make-target, 6-28
  - mil, 6-29
  - mil-split, 6-29
  - misrac, 6-30
  - misrac-advisory-warnings, 6-31
  - misrac-required-warnings, 6-31
  - no-double, 6-33
  - no-stdinc, 6-34
  - no-warnings, 6-35
  - optimize, 6-36
  - option-file, 6-38
  - output, 6-39
  - preprocess, 6-40
  - profile, 6-41
  - rename-sections, 6-43
  - runtime, 6-44
  - signed-bitfields, 6-46
  - silicon-bug, 6-47
  - source, 6-48
  - static, 6-49
  - stdout, 6-50
  - thumb, 6-51
  - tradeoff, 6-52

--uchar, [6-53](#)  
--unaligned-access, [6-54](#)  
--undefine, [6-55](#)  
--version, [6-56](#)  
--warnings-as-errors, [6-57](#)  
-A, [6-25](#)  
-B, [6-13](#)  
-C, [6-7](#)  
-c, [6-23](#)  
-D, [6-9](#)  
-E, [6-40](#)  
-F, [6-33](#)  
-f, [6-38](#)  
-f (--option-file), [6-15](#)  
-g, [6-8](#)  
-H, [6-19](#)  
-I, [6-18](#)  
-k, [6-24](#)  
-m, [6-4](#)  
-n, [6-50](#)  
-O, [6-36](#)  
-o, [6-39](#)  
-p, [6-41](#)  
-R, [6-43](#)  
-r, [6-44](#)  
-s, [6-48](#)  
-t, [6-52](#)  
-U, [6-55](#)  
-u, [6-53](#)  
-V, [6-56](#)  
-w, [6-35](#)  
debug information, [6-8](#)  
diagnostics, [6-35](#), [6-57](#)  
language, [6-23](#), [6-25](#), [6-46](#), [6-53](#)  
MISRA-C, [6-30](#)  
optimization, [6-36](#), [6-52](#)  
preprocessing, [6-9](#), [6-19](#), [6-40](#), [6-55](#)  
C++, language extensions, [2-1](#)  
C++ compiler options, [6-58](#)  
-?, [6-83](#)  
--alternative-tokens, [6-59](#)  
--anachronisms, [6-60](#)  
--base-assign-op-is-default, [6-61](#)  
--big-endian, [6-72](#)  
--building-runtime, [6-62](#)  
--c++0x, [6-63](#)  
--check, [6-64](#)  
--context-limit, [6-65](#)  
--cpu, [6-66](#)  
--create-pch, [6-67](#)  
--define, [6-68](#)  
--dep-file, [6-69](#)  
--dollar, [6-70](#)  
--embedded-c++, [6-71](#)  
--endianness, [6-72](#)  
--error-file, [6-73](#)  
--error-limit, [6-74](#)  
--exceptions, [6-75](#)  
--exported-template-file, [6-76](#)  
--extended-variadic-macros, [6-77](#)  
--force-vtbl, [6-78](#)  
--friend-injection, [6-79](#)  
--g++, [6-80](#)  
--gnu-version, [6-81](#)  
--guiding-decls, [6-82](#)  
--help, [6-83](#)  
--implicit-extern-c-type-conversion, [6-84](#)  
--implicit-include, [6-85](#)  
--incl-suffixes, [6-86](#)  
--include-directory, [6-87](#)  
--include-file, [6-88](#)  
--include-macros-file, [6-89](#)  
--init-priority, [6-90](#)  
--instantiate, [6-91](#)  
--io-streams, [6-92](#)  
--late-tiebreaker, [6-93](#)  
--list-file, [6-94](#)  
--long-lifetime-temps, [6-96](#)  
--long-long, [6-97](#)  
--make-target, [6-98](#)  
--multibyte-chars, [6-99](#)  
--namespaces, [6-100](#)  
--no-arg-dep-lookup, [6-101](#)  
--no-array-new-and-delete, [6-102](#)  
--no-auto-instantiation, [6-103](#)  
--no-bool, [6-104](#)  
--no-class-name-injection, [6-105](#)  
--no-const-string-literals, [6-106](#)  
--no-dep-name, [6-107](#)  
--no-distinct-template-signatures, [6-108](#)  
--no-double, [6-109](#)  
--no-enum-overloading, [6-110](#)  
--no-explicit, [6-111](#)  
--no-export, [6-112](#)  
--no-extern-inline, [6-113](#)  
--no-for-init-diff-warning, [6-114](#)  
--no-implicit-typename, [6-115](#)  
--no-inlining, [6-116](#)  
--no-nonconst-ref-anachronism, [6-117](#)  
--no-parse-templates, [6-120](#)  
--no-pch-messages, [6-121](#)  
--no-preprocessing-only, [6-122](#)  
--no-stdinc, [6-123](#)  
--no-stdstlinc, [6-123](#)  
--no-type-traits-helpers, [6-152](#)  
--no-typename, [6-124](#)  
--no-use-before-set-warnings, [6-125](#)  
--no-warnings, [6-126](#)  
--nonconst-ref-anachronism, [6-117](#)  
--nonstd-qualifier-deduction, [6-118](#)  
--nonstd-using-decl, [6-119](#)  
--old-for-init, [6-127](#)  
--old-line-commands, [6-128](#)  
--old-specializations, [6-129](#)  
--option-file, [6-130](#)  
--output, [6-132](#)

- `--pch`, 6-133
- `--pch-dir`, 6-134
- `--pch-verbose`, 6-135
- `--pending-instantiations`, 6-136
- `--preprocess`, 6-137
- `--remarks`, 6-138
- `--remove-unneeded-entities`, 6-139
- `--rtti`, 6-140
- `--schar`, 6-141
- `--signed-bitfields`, 6-142
- `--special-subscript-cost`, 6-143
- `--strict`, 6-144
- `--strict-warnings`, 6-145
- `--suppress-vtbl`, 6-146
- `--sys-include`, 6-147
- `--template-directory`, 6-148
- `--thumb`, 6-149
- `--timing`, 6-150
- `--trace-includes`, 6-151
- `--type-traits-helpers`, 6-152
- `--uchar`, 6-153
- `--undefine`, 6-154
- `--use-pch`, 6-155
- `--using-std`, 6-156
- `--variadic-macros`, 6-157
- `--version`, 6-158
- `--warnings-as-errors`, 6-159
- `--wchar_t-keyword`, 6-160
- `--xref-file`, 6-161
- `-A`, 6-144
- `-a`, 6-145
- `-B`, 6-72
- `-C`, 6-66
- `-D`, 6-68
- `-E`, 6-137
- `-e`, 6-74
- `-F`, 6-109
- `-f`, 6-130
- `-H`, 6-88
- `-I`, 6-87
- `-j`, 6-125
- `-L`, 6-94
- `-o`, 6-132
- `-r`, 6-138
- `-s`, 6-141
- `-t`, 6-91
- `-U`, 6-154
- `-u`, 6-153, 6-158
- `-V`, 6-158
- `-w`, 6-126
- `-X`, 6-161
- `-x`, 6-75
- diagnostics, 6-159
- preprocessing, 6-68, 6-88, 6-137
- C++ dialect, 2-1
  - accepted, 2-1
  - anachronisms accepted, 2-4
  - C++0x language features accepted, 2-3
    - normal C++ mode extensions, 2-4
    - standard language features accepted, 2-1
- C++ language features
  - accepted, 2-1
  - C++0x, 2-3
- C++ style comments, 6-25
- C++ working paper, 6-63
- cabs, 3-4
- cacos, 3-3
- cacosh, 3-3
- call, 1-8
- Call graph, 4-14
- calloc, 3-8, 3-19
- calls, 4-14
- carg, 3-4
- casin, 3-3
- casinh, 3-3
- catan, 3-3
- catanh, 3-3
- cbrt functions, 3-10
- ccos, 3-3
- ccosh, 3-3
- ceil functions, 3-9
- CERT C, 10-1
- CERT C secure coding, 6-5
- cexp, 3-3
- chdir, 3-23
- Check source code, 6-6, 6-64, 6-164, 6-241
- cimag, 3-4
- Class name injection, 6-105
- clearerr, 3-18
- clock, 3-22
- clock\_t, 3-22
- CLOCKS\_PER\_SEC, 3-22
- clog, 3-3
- close, 3-23
- cnt, 4-7
- Code checking, CERT C, 10-1
- Code compaction, 6-20
- code16, 4-15
- code32, 4-15
- Command file, 6-305
- Comment, 4-2
- Comments, 9-3
- compiler options, --misrac-version, 6-32
- complex, 3-3
- Conditional assembly, 4-49
- Conditional make rules, 6-292
- conj, 3-4
- Conjugate value, 3-4
- const, string literals, 6-106
- Context limit, 6-65
- Control program, passing options, 6-278
- Control program options, 6-239
  - ?, 6-257
  - adress-size, 6-240
  - check, 6-241
  - cpu, 6-242



- create, [6-243](#)
- debug-info, [6-244](#)
- define, [6-245](#)
- dep-file, [6-246](#)
- diag, [6-247](#)
- dry-run, [6-248](#)
- endianness, [6-249](#)
- error file, [6-250](#)
- exceptions, [6-251](#)
- force-c, [6-252](#)
- force-c++, [6-253](#)
- force-munch, [6-254](#)
- format, [6-255](#)
- global-type-checking, [6-256](#)
- help, [6-257](#)
- ignore-default-library-path, [6-265](#)
- include-directory, [6-258](#)
- instantiate, [6-259](#)
- io-streams, [6-260](#)
- iso, [6-261](#)
- keep-output-files, [6-262](#)
- keep-temporary-files, [6-263](#)
- library, [6-264](#)
- library-directory, [6-265](#)
- lsl-file, [6-267](#)
- make-target, [6-268](#)
- mil-link, [6-269](#)
- mil-split, [6-269](#)
- no-auto-instantiation, [6-270](#)
- no-default-libraries, [6-271](#)
- no-double, [6-272](#)
- no-map-file, [6-273](#)
- no-preprocessing-only, [6-274](#)
- no-warnings, [6-275](#)
- option file, [6-276](#)
- output, [6-277](#)
- pass, [6-278](#)
- preprocess, [6-279](#)
- profile, [6-280](#)
- show-c++-warnings, [6-282](#)
- signed-bitfields, [6-283](#)
- static, [6-284](#)
- thumb, [6-285](#)
- undefine, [6-287](#)
- verbose, [6-288](#)
- version, [6-289](#)
- warnings-as-errors, [6-290](#)
- B (--big-endian), [6-249](#)
- C, [6-242](#)
- c, [6-243](#)
- D, [6-245](#)
- d, [6-267](#)
- E, [6-279](#)
- F, [6-272](#)
- f, [6-276](#)
- g, [6-244](#)
- I, [6-258](#)
- k, [6-262](#)

- L, [6-265](#)
- l, [6-264](#)
- l (--library), [6-266](#)
- n, [6-248](#)
- o, [6-277](#)
- p, [6-280](#)
- t, [6-263](#)
- U, [6-287](#)
- u (--uchar), [6-286](#)
- V, [6-289](#)
- v, [6-288](#)
- W, [6-278](#)
- w, [6-275](#)
- preprocessing, [6-245](#)
- Copy assignment operator, [6-61](#)
- Copy table, [9-16](#), [9-32](#)
- copysign functions, [3-10](#)
- cos functions, [3-8](#)
- cosh functions, [3-8](#)
- cpow, [3-4](#)
- cproj, [3-4](#)
- cpu, [4-7](#)
- CPU functional problem workarounds, [6-47](#)
- creal, [3-4](#)
- Cross-reference file, [6-161](#)
- csin, [3-3](#)
- csinh, [3-3](#)
- csqrt, [3-4](#)
- ctan, [3-3](#)
- ctanh, [3-3](#)
- ctime, [3-22](#)

## D

- Data types, [1-2](#)
- db, [4-16](#)
- dd, [4-17](#)
- Debug info, [6-244](#)
- Debug information, [6-169](#)
- define, [4-18](#)
- defined, [4-7](#)
- Defining a macro, [4-46](#)
- Dependent name processing, [6-107](#)
- Derivative definition, [9-1](#), [9-20](#)
- dh, [4-19](#)
- difftime, [3-22](#)
- Digraph, [6-59](#)
- Directives, [4-1](#)
- div, [3-20](#)
- Dollar signs, [6-70](#)
- double, [4-25](#)
- Double as float, [6-272](#)
- ds, [4-20](#)
- dsb, [4-20](#)
- dsd, [4-20](#)
- dsh, [4-20](#)
- dsw, [4-20](#)
- dw, [4-21](#)



**E**

ELF/DWARF object format, 8-1  
 Embedded C++, 6-71  
 end, 4-22  
 Endianness, 6-13, 6-72, 6-170, 6-202, 6-249  
 endprofile, 1-9  
 endprotect, 1-10  
 Entities, remove unneeded, 6-139  
 enum overloading, 6-110  
 EOF, 3-13  
 equ, 4-23  
 erf functions, 3-11  
 erfc functions, 3-11  
 errno, 3-5  
 Error limit, 6-74  
 Error output file, 6-73  
 Exception handler, 1-18  
   *defining*, 1-18  
 Exception handling, 6-75  
 exit, 3-19  
 EXIT\_FAILURE, 3-18  
 EXIT\_SUCCESS, 3-18  
 exp functions, 3-9  
 exp2 functions, 3-9  
 explicit specifier, 6-111  
 expm1 functions, 3-9  
 export keyword, 2-18, 6-112  
 Exported templates, 2-18  
 Expressions, 4-3  
   *absolute*, 4-3  
   *relative*, 4-3  
   *relocatable*, 4-3  
 extension isuffix, 1-8  
 Extensions to C++, 2-1  
 extern, 1-8, 4-24  
 extern C, 6-84  
 extern C++, 6-84  
 extern inline, 6-113  
 Extern inline functions, 2-20

**F**

fabs functions, 3-10  
 fclose, 3-14  
 fdim functions, 3-10  
 FE\_ALL\_EXCEPT, 3-6  
 FE\_DIVBYZERO, 3-6  
 FE\_INEXACT, 3-6  
 FE\_INVALID, 3-6  
 FE\_OVERFLOW, 3-6  
 FE\_UNDERFLOW, 3-6  
 feenableexcept, 3-6  
 fegetenv, 3-6  
 fegetexceptflag, 3-6  
 feholdexcept, 3-6  
 feof, 3-18  
 feraiseexcept, 3-6  
 ferror, 3-18  
 fesetenv, 3-6

fesetexceptflag, 3-6  
 fetestexcept, 3-6  
 feupdateenv, 3-6  
 fflush, 3-14  
 fgetc, 3-17  
 fgetpos, 3-18  
 fgets, 3-17  
 fgetwc, 3-17  
 fgetws, 3-17  
 FILENAME\_MAX, 3-13  
 float, 4-25  
 Floating-point, single precision, 6-109  
 Floating-point unit, 6-15  
 floor functions, 3-9  
 fma functions, 3-10  
 fmax functions, 3-10  
 fmin functions, 3-10  
 fmod functions, 3-10  
 fopen, 3-14  
 FOPEN\_MAX, 3-13  
 for-init statement, 6-114, 6-127  
 for/endfor, 4-26  
 fpclassify, 3-11  
 fprintf, 3-16  
 fputc, 3-17  
 fputs, 3-17  
 fputwc, 3-17  
 fputws, 3-17  
 fread, 3-17  
 free, 3-8, 3-19  
 freopen, 3-14  
 frexp functions, 3-9  
 Friend injection, 6-79  
 fscanf, 3-16  
 fseek, 3-17  
 fsetpos, 3-18  
 fstat, 3-24  
 ftell, 3-18  
 Function, syntax, 4-5  
 Function inlining, 1-15  
 Function names, unqualified, 6-101  
 Function qualifiers  
   *\_\_frame()*, 1-19  
   *\_\_interrupt()*, 1-18  
   *\_\_interrupt\_dabt*, 1-18  
   *\_\_interrupt\_fiq*, 1-18  
   *\_\_interrupt\_iabt*, 1-18  
   *\_\_interrupt\_irq*, 1-18  
   *\_\_interrupt\_svc*, 1-18  
   *\_\_interrupt\_und*, 1-18  
   *\_\_nesting\_enabled*, 1-19  
   *\_\_novector*, 1-19  
 fwprintf, 3-16  
 fwrite, 3-17  
 fwscanf, 3-16

**G**

Generic instructions, 4-1, 4-51

getc, 3-17  
getchar, 3-17  
getcwd, 3-23  
getenv, 3-20  
gets, 3-17  
getwc, 3-17  
getwchar, 3-17  
global, 4-27  
Global type checking, 6-16, 6-207  
gmtime, 3-22  
GNU extensions, 2-5, 6-80  
    *compiler version*, 6-81  
Guiding declarations, 6-82

## H

Header files, 3-3  
    *alert.h*, 3-3  
    *complex.h*, 3-3  
    *ctype.h*, 3-4  
    *dbg.h*, 3-5  
    *errno.h*, 3-5  
    *fcntl.h*, 3-6  
    *fenv.h*, 3-6  
    *float.h*, 3-6  
    *inttypes.h*, 3-6  
    *io.h*, 3-7  
    *iso646.h*, 3-7  
    *limits.h*, 3-7  
    *locale.h*, 3-7  
    *malloc.h*, 3-8  
    *math.h*, 3-8  
    *setjmp.h*, 3-11  
    *signal.h*, 3-12  
    *stdarg.h*, 3-12  
    *stdbool.h*, 3-12  
    *stddef.h*, 3-12  
    *stdint.h*, 3-6  
    *stdio.h*, 3-13  
    *stdlib.h*, 3-18  
    *string.h*, 3-20  
    *tgmath.h*, 3-8  
    *time.h*, 3-22  
    *unistd.h*, 3-23  
    *wchar.h*, 3-13, 3-20, 3-22, 3-24  
    *wctype.h*, 3-4, 3-25  
Header stop, 2-24, 2-26  
Heap, 9-16  
    *begin of*, 5-6  
    *end of*, 5-6  
hypot functions, 3-10

## I

I/O streams, 6-92  
if/elif/else/endif, 4-28  
ilogb functions, 3-9  
imaginary, 3-3  
imaxabs, 3-6  
imaxdiv, 3-6

Implicit inclusion, 2-17  
include, 4-29  
Include directory, 6-18, 6-34, 6-87, 6-174, 6-212, 6-258  
Include file, 6-19, 6-88, 6-175  
    *macros*, 6-89  
Include files, suffix, 6-86  
Inline assembly, `__asm`, 1-4  
Inline function, 6-113  
Inline functions, 1-15  
inline/ noline / smartinline, 1-9  
Inlining, 6-20, 6-116  
Inlining functions in C++, 2-20  
Input specification, 4-1  
Instantiation, 2-14  
    *pending*, 6-136  
    *template*, 6-91  
Instantiation mode, 2-15  
Instantiation pragmas, 2-16  
Instructions, 4-1  
    *generic*, 4-1, 4-51  
Intel hex, record type, 8-5  
Intel Hex record format, 8-5  
Interrupt frame, 1-19  
Interrupt functions, 1-18  
    *\_\_frame()*, 1-19  
    *\_\_interrupt()*, 1-18  
    *\_\_interrupt\_dabt*, 1-18  
    *\_\_interrupt\_fiq*, 1-18  
    *\_\_interrupt\_iabt*, 1-18  
    *\_\_interrupt\_irq*, 1-18  
    *\_\_interrupt\_svc*, 1-18  
    *\_\_interrupt\_und*, 1-18  
Interrupt service routine, 1-18  
Intrinsic functions, 1-16  
isalnum, 3-4  
isalpha, 3-4  
isblank, 3-4  
iscntrl, 3-4  
isdigit, 3-4  
isfinite, 3-11  
isgraph, 3-4  
isgreater, 3-11  
isgreaterequal, 3-11  
isinf, 3-11  
isless, 3-11  
islessequal, 3-11  
islessgreater, 3-11  
islower, 3-4  
isnan, 3-11  
isnormal, 3-11  
ISO C standard, selecting, 6-23, 6-261  
isprint, 3-4  
ispunct, 3-4  
isspace, 3-4  
isunordered, 3-11  
isupper, 3-4  
iswalnum, 3-4, 3-25  
iswalpha, 3-4, 3-25

iswblank, 3-4  
 iswcntrl, 3-4, 3-25  
 iswctype, 3-25  
 iswdigit, 3-4, 3-25  
 iswgraph, 3-4, 3-25  
 iswlower, 3-4, 3-25  
 iswprint, 3-4, 3-25  
 iswpunct, 3-4, 3-25  
 iswspace, 3-4, 3-25  
 iswupper, 3-4, 3-25  
 iswxdigit, 3-4  
 iswxdigit, 3-25  
 isxdigit, 3-4

## J

Jump chain, 1-13  
 Jump table, 1-13  
 jump\_switch, 1-10

## K

Keyword  
   *bool*, 6-104  
   *explicit*, 6-111  
   *export*, 2-18, 6-112  
   *typename*, 6-124  
   *wchar\_t*, 6-160

## L

L\_tmpnam, 3-13  
 Label, 4-2  
 Labels, 4-1  
 labs, 3-20  
 ldexp functions, 3-9  
 ldiv, 3-20  
 lgamma functions, 3-11  
 Librarian options  
   -?, 6-318  
   -d, 6-319  
   -f, 6-320  
   -m, 6-321  
   -p, 6-322  
   -r, 6-323  
   -t, 6-325  
   -V, 6-326  
   -w, 6-327  
   -x, 6-328  
   *add module*, 6-323  
   *create library*, 6-323  
   *delete module*, 6-319  
   *extract module*, 6-328  
   *move module*, 6-321  
   *print list of objects*, 6-325  
   *print list of symbols*, 6-325  
   *print module*, 6-322  
   *replace module*, 6-323  
   *warning level*, 6-327  
 Libraries, linking, 6-206, 6-227  
 Library, specifying, 6-215, 6-216, 6-264, 6-265, 6-266

Lifetime, 6-96  
 linear\_switch, 1-10  
 Linker macro, 6-200  
 Linker options, 6-196  
   -?, 6-208  
   --case-insensitive, 6-197  
   --chip-output, 6-198  
   --cpu, 6-199  
   --define, 6-200  
   --diag, 6-201  
   --endianness, 6-202  
   --error-file, 6-203  
   --error-limit, 6-204  
   --extern, 6-205  
   --extra-verbose, 6-236  
   --first-library-first, 6-206  
   --global-type-checking, 6-207  
   --help, 6-208  
   --hex-format, 6-209  
   --hex-record-size, 6-210  
   --ignore-default-library-path, 6-216  
   --import-object, 6-211  
   --include-directory, 6-212  
   --incremental, 6-213  
   --keep-output-files, 6-214  
   --library, 6-215  
   --library-directory, 6-216  
   --link-only, 6-217  
   --long-branch-veneers, 6-218  
   --lsl-check, 6-219  
   --lsl-dump, 6-220  
   --lsl-file, 6-221  
   --map-file, 6-222  
   --map-file-format, 6-223  
   --misra-c-report, 6-224  
   --munch, 6-225  
   --no-rescan, 6-227  
   --no-rom-copy, 6-228  
   --no-warnings, 6-229  
   --non-romable, 6-226  
   --optimize, 6-230  
   --option-file, 6-231  
   --output, 6-232  
   --print-mangled-symbols, 6-233  
   --strip-debug, 6-234  
   --user-provided-initialization-code, 6-235  
   --verbose, 6-236  
   --version, 6-237  
   --warnings-as-errors, 6-238  
   -B (--big-endian), 6-202  
   -C, 6-199  
   -c, 6-198  
   -D, 6-200  
   -d, 6-221  
   -e, 6-205  
   -f, 6-231  
   -I, 6-212  
   -i, 6-235

- [-k, 6-214](#)
- [-L, 6-216](#)
- [-l, 6-215](#)
- [-M, 6-222](#)
- [-m, 6-223](#)
- [-N, 6-228](#)
- [-O, 6-230](#)
- [-o, 6-232](#)
- [-P, 6-233](#)
- [-r, 6-213](#)
- [-S, 6-234](#)
- [-V, 6-237](#)
- [-v, 6-236](#)
- [-vv, 6-236](#)
- [-w, 6-229](#)
- [diagnostics, 6-229, 6-238](#)
- [libraries, 6-215, 6-227](#)
- [Map File, 6-222](#)
- [miscellaneous, 6-197, 6-200, 6-205, 6-211, 6-220, 6-221](#)
- [optimization, 6-230](#)
- [output format, 6-198](#)
- Linker script file, [6-219, 6-220](#)
  - [architecture definition, 9-1](#)
  - [board specification, 9-2](#)
  - [bus definition, 9-2](#)
  - [derivative definition, 9-1](#)
  - [memory definition, 9-2](#)
  - [preprocessing, 9-3](#)
  - [processor definition, 9-2](#)
  - [section layout definition, 9-2](#)
  - [specifying, 6-221, 6-267](#)
  - [structure, 9-1](#)
- List file, [6-178, 6-179](#)
  - [C++, 6-94](#)
- [list/nolist, 4-30](#)
- [llabs, 3-20](#)
- [lldiv, 3-20](#)
- [llrint functions, 3-9](#)
- [llround functions, 3-9](#)
- [Local label override operator, 4-49](#)
- [localeconv, 3-8](#)
- [localtime, 3-22](#)
- [log functions, 3-9](#)
- [log10 functions, 3-9](#)
- [log1p functions, 3-9](#)
- [log2 functions, 3-9](#)
- [logb functions, 3-9](#)
- [Long-branch veneers, 6-218](#)
- [longjmp, 3-11](#)
- [Lookup of names used in templates, 6-107](#)
- [Lookup of unqualified function names, 6-101](#)
- [lrint functions, 3-9](#)
- [lround functions, 3-9](#)
- [lsb, 4-7](#)
- [lseek, 3-24](#)
- [lsh, 4-7](#)
- [LSL expression evaluation, 9-13](#)
- LSL functions
  - [absolute\(\), 9-5](#)
  - [addressof\(\), 9-5](#)
  - [exists\(\), 9-5](#)
  - [max\(\), 9-5](#)
  - [min\(\), 9-6](#)
  - [sizeof\(\), 9-6](#)
- LSL keywords
  - [align, 9-16, 9-27](#)
  - [alloc\\_allowed, 9-31](#)
  - [allow\\_cross\\_references, 9-28](#)
  - [architecture, 9-15, 9-20](#)
  - [attributes, 9-26, 9-27](#)
  - [blocksize, 9-32](#)
  - [bus, 9-15, 9-18, 9-23](#)
  - [clustered, 9-28](#)
  - [contiguous, 9-28](#)
  - [copy, 9-17, 9-27](#)
  - [copy\\_unit, 9-16](#)
  - [copytable, 9-16, 9-32](#)
  - [core, 9-20](#)
  - [derivative, 9-20, 9-22](#)
  - [dest, 9-16, 9-18](#)
  - [dest\\_dbits, 9-18](#)
  - [dest\\_offset, 9-18](#)
  - [direction, 9-26, 9-28](#)
  - [else, 9-33](#)
  - [extends, 9-15, 9-20](#)
  - [fill, 9-17, 9-21, 9-28, 9-31](#)
  - [fixed, 9-16, 9-30](#)
  - [group, 9-26, 9-27](#)
  - [grows, 9-16](#)
  - [heap, 9-16, 9-30](#)
  - [high\\_to\\_low, 9-16, 9-26](#)
  - [id, 9-15, 9-16](#)
  - [id\\_symbol\\_prefix, 9-17](#)
  - [if, 9-33](#)
  - [load\\_addr, 9-29](#)
  - [low\\_to\\_high, 9-16, 9-26](#)
  - [map, 9-15, 9-16, 9-18, 9-21](#)
  - [mau, 9-15, 9-16, 9-21, 9-23](#)
  - [mem, 9-29](#)
  - [memory, 9-21, 9-23](#)
  - [min\\_size, 9-16, 9-30](#)
  - [no\\_inline, 9-17](#)
  - [nocopy, 9-28](#)
  - [nvram, 9-21](#)
  - [ordered, 9-28](#)
  - [overflow, 9-32](#)
  - [overlay, 9-28](#)
  - [page, 9-16, 9-29](#)
  - [page\\_size, 9-16, 9-29](#)
  - [priority, 9-30](#)
  - [processor, 9-22](#)
  - [ram, 9-21](#)
  - [ref\\_tree, 9-27](#)
  - [reserved, 9-18, 9-21, 9-30](#)

rom, 9-21  
 run\_addr, 9-17, 9-18, 9-29  
 section, 9-31  
 section\_layout, 9-25  
 section\_setup, 9-24  
 select, 9-26  
 size, 9-17, 9-18, 9-21, 9-23, 9-30, 9-32  
 space, 9-15, 9-18  
 speed, 9-21, 9-23  
 src\_dbits, 9-18  
 src\_offset, 9-18  
 stack, 9-16, 9-30  
 start\_address, 9-18  
 symbol, 9-18  
 template, 9-17  
 template\_symbol, 9-17  
 type, 9-21, 9-23  
 vector, 9-17  
 vector\_prefix, 9-17  
 vector\_size, 9-17  
 vector\_table, 9-17  
 width, 9-15  
 LSL syntax, 9-3  
   architecture definition, 9-7  
   board specification, 9-10  
   bus definition, 9-6  
   derivative definition, 9-9  
   memory definition, 9-6  
   processor definition, 9-10  
   section layout definition, 9-10  
 lstat, 3-24  
 lsw, 4-7  
 ltorg, 4-31

## M

macro / nomacro, 1-9  
 Macro argument string, 4-48  
 Macro call, 4-1  
 Macro definition, 6-9, 6-68, 6-167, 6-245  
 Macro operations, 4-46  
 macro/endm, 4-32  
 Macros, 4-46  
   .for directive, 4-49  
   .repeat directive, 4-49  
   argument concatenation, 4-47  
   argument operator, 4-47  
   argument string, 4-48  
   calling, 4-46  
   conditional assembly, 4-49  
   defining, 4-46  
   local label override, 4-49  
   make utility, 6-292  
   predefined, 2-22  
   return decimal value operator, 4-48  
   return hex value operator, 4-48  
   variable arguments list, 6-77, 6-157  
 Macros (preprocessor), 1-12, 4-3  
 Magnitude, 3-4

Make utility options  
   -?, 6-293  
   -a, 6-294  
   -c, 6-295  
   -D, 6-296  
   -d, 6-297  
   -DD, 6-296  
   -dd, 6-297  
   -e, 6-298  
   -err, 6-299  
   -f, 6-300  
   -G, 6-301  
   -i, 6-302  
   -K, 6-303  
   -k, 6-304  
   -m, 6-305, 6-310  
   -n, 6-306  
   -p, 6-307  
   -q, 6-308  
   -r, 6-309  
   -s, 6-311  
   -t, 6-312  
   -time, 6-313  
   -V, 6-314  
   -W, 6-315  
   -x, 6-316  
   defining a macro, 6-292  
 malloc, 3-8, 3-19  
 Map file generation, 6-222  
 Mappings, 9-18  
 MB\_CUR\_MAX, 3-18, 3-24  
 MB\_LEN\_MAX, 3-24  
 mblen, 3-20  
 mbrlen, 3-24  
 mbrtowc, 3-24  
 mbsinit, 3-24  
 mbsrtowcs, 3-24  
 mbstate\_t, 3-24  
 mbstowcs, 3-20  
 mbtowc, 3-20  
 memchr, 3-21  
 memcmp, 3-21  
 memcpy, 3-20  
 memmove, 3-20  
 Memory definition, 9-2  
 memset, 3-21  
 Merging source code, 6-48  
 message, 1-9, 4-34  
 MISRA-C, 6-30, 6-31  
   MISRA-C report, 6-224  
   supported rules 1998, 11-1  
   supported rules 2004, 11-5  
   version, 6-32  
 mktime, 3-22  
 modf functions, 3-9  
 Modulus, 3-4  
 Motorola S-record format, 8-2  
 msb, 4-7

msh, [4-7](#)  
 msw, [4-8](#)  
 Multibyte characters, [6-99](#)

## N

Namespace, [2-13](#), [6-100](#)  
     *std*, [6-156](#)  
 nan functions, [3-10](#)  
 nearbyint functions, [3-9](#)  
 nextafter functions, [3-10](#)  
 nexttoward functions, [3-10](#)  
 no\_pch, [2-26](#)  
 no\_tbh\_switch, [1-10](#)  
 Functions, [1-14](#)  
     *intrinsic*, [1-16](#)  
     *parameter passing*, [1-14](#)  
     *return types*, [1-14](#)  
 Norm, [3-4](#)  
 NULL, [3-13](#)

## O

offset, [4-35](#)  
 offsetof, [3-13](#)  
 open, [3-6](#)  
 Operands, [4-2](#)  
 Operator keywords, [6-59](#)  
 Optimization, [6-36](#), [6-182](#), [6-230](#)  
     *code compaction*, [6-20](#)  
     *inlining*, [6-21](#)  
     *optimize for speed/size*, [6-52](#)  
 optimize / endoptimize, [1-9](#)  
 Option file, [6-38](#), [6-130](#), [6-231](#), [6-276](#), [6-305](#)  
 Options, saving / restoring, [6-38](#), [6-130](#), [6-231](#)  
 Output file, [6-132](#)

## P

page, [4-36](#)  
 Parameter passing, [1-14](#)  
 Passing options, [6-278](#)  
 PCH mode  
     *automatic*, [2-24](#), [6-133](#)  
     *manual*, [2-26](#), [6-67](#), [6-155](#)  
 perror, [3-18](#)  
 Phase angle, [3-4](#)  
 pow functions, [3-10](#)  
 Pragma  
     *can\_instantiate*, [2-16](#)  
     *do\_not\_instantiate*, [2-16](#)  
     *hdrstop*, [2-24](#), [2-26](#)  
     *instantiate*, [2-16](#)  
     *no\_pch*, [2-26](#)  
 Pragmas, [1-8](#)  
     *in C++*, [2-21](#)  
 Precompiled header, [2-24](#)  
     *automatic*, [2-24](#), [6-133](#)  
     *create*, [2-26](#), [6-67](#)  
     *directory*, [2-26](#), [6-134](#)  
     *file cannot be used*, [6-135](#)

*manual*, [2-26](#)  
     *messages*, [6-121](#)  
     *performance*, [2-26](#)  
     *pragmas*, [2-26](#)  
     *prefix*, [2-25](#)  
     *use*, [2-26](#), [6-155](#)  
 Predefined macros, [2-22](#)  
 Predefined preprocessor macros, [1-12](#), [4-3](#)  
 Predefined preprocessor symbols, [4-3](#)  
 Preprocessing, [6-167](#), [6-187](#), [6-274](#), [6-279](#), [9-3](#)  
     *storing output*, [6-40](#), [6-137](#)  
 printf, [3-14](#), [3-16](#)  
     *conversion characters*, [3-15](#)  
 printf versions, [1-20](#)  
 Priority for global initialization functions, [6-90](#)  
 Processor definition, [9-2](#), [9-22](#)  
 Processor options, processor definition, [6-7](#), [6-66](#), [6-165](#),  
     [6-199](#)  
 Processor type  
     *assembling for*, [6-165](#)  
     *compiling for*, [6-7](#), [6-66](#), [6-242](#)  
     *selecting*, [6-7](#), [6-66](#), [6-165](#), [6-242](#)  
 profile, [1-9](#)  
 Profiling, [6-41](#), [6-280](#)  
 profiling, [1-10](#)  
 protect, [1-10](#)  
 ptrdiff\_t, [3-12](#)  
 putc, [3-17](#)  
 putchar, [3-17](#)  
 puts, [3-17](#)  
 putwc, [3-17](#)  
 putwchar, [3-17](#)

## Q

qsort, [3-20](#)  
 Qualifier deduction, [6-118](#)

## R

raise, [3-12](#)  
 rand, [3-19](#)  
 RAND\_MAX, [3-18](#)  
 read, [3-24](#)  
 realloc, [3-8](#), [3-19](#)  
 Register usage, [1-14](#)  
 remainder functions, [3-10](#)  
 Remarks, [6-138](#)  
 remove, [3-18](#)  
 remquo functions, [3-10](#)  
 rename, [3-18](#)  
 Renaming sections, [6-43](#)  
 repeat/endrep, [4-37](#)  
 Reserved address ranges, [9-18](#)  
 Reset handler, [5-1](#), [5-2](#)  
 Reset vector, [9-18](#)  
 Return decimal value operator, [4-48](#)  
 Return hex value operator, [4-48](#)  
 rewind, [3-18](#)  
 Riemann sphere, [3-4](#)

- scalbn functions, 3-9
- scalbn functions, 3-9
- scanf, 3-15, 3-16
  - conversion characters, 3-16
- scanf versions, 1-20
- Secondary translation unit, 2-19
- Section, 4-38
- section / endsection, 1-10
- Section attributes, 4-38
- Section information, 6-190
- Section layout definition, 9-2, 9-25
- Section renaming, 6-43
- Section setup definition, 9-24
- section\_code\_init, 1-10
- section\_const\_init, 1-10
- section\_no\_code\_init, 1-10
- section\_no\_const\_init, 1-10
- Sections, 4-38
  - grouping, 9-26
- SEEK\_CUR, 3-13, 3-17
- SEEK\_END, 3-13, 3-17
- SEEK\_SET, 3-13, 3-17
- set, 4-39
- setbuf, 3-14
- setjmp, 3-11
- setlocale, 3-7
- setvbuf, 3-14
- SIGABRT, 3-12
- SIGFPE, 3-12
- SIGILL, 3-12
- SIGINT, 3-12
- signal, 3-12
- signbit, 3-11
- SIGSEGV, 3-12
- SIGTERM, 3-12
- Silicon bug workarounds, 6-47
- sin functions, 3-8
- sinh functions, 3-8
- size, 4-40
- size\_t, 3-12
- smart\_switch, 1-10
- smartinline, 1-9
- snprintf, 3-16
- source, 4-41
- source / nosource, 1-10
- sprintf, 3-16
- sqrt functions, 3-10
- srand, 3-19
- sscanf, 3-16
- Stack, 2-14, 9-16

**T**

Index-13



- Template instantiation, [2-14](#)
  - #pragma directives*, [2-16](#)
  - automatic*, [2-14](#), [6-103](#)
  - exported templates*, [2-18](#)
  - implicit inclusion*, [2-17](#), [6-85](#)
  - instantiation modes*, [2-15](#), [6-91](#)
  - pending*, [6-136](#)
- Templates, [6-120](#)
- tgamma functions, [3-11](#)
- thumb, [4-8](#), [4-15](#), [6-22](#)
- Thumb instructions, [6-192](#)
- Tie-breakers, [6-93](#)
- time, [3-22](#)
- time\_t, [3-22](#)
- Timing information, [6-150](#)
- title, [4-42](#)
- tm (struct), [3-22](#)
- TMP\_MAX, [3-13](#)
- tmpfile, [3-18](#)
- tmpnam, [3-18](#)
- tolower, [3-4](#)
- toupper, [3-4](#)
- towctrans, [3-25](#)
- towlower, [3-4](#), [3-25](#)
- towupper, [3-4](#), [3-25](#)
- tradeoff, [1-11](#)
- Transferring parameters between functions, [1-14](#)
- trunc functions, [3-9](#)
- type, [4-43](#)
- Type checking, [6-16](#), [6-207](#)
- Type qualifier, \_\_unaligned, [1-2](#)
- Type traits helpers, [6-152](#)
- typename keyword, [6-124](#)

## U

- UAL syntax, [4-8](#)
- undef, [4-44](#)
- ungetc, [3-17](#)
- ungetwc, [3-17](#)
- unlink, [3-24](#)
- Using assembly in C source, [1-4](#)
- using declaration, allow unqualified name, [6-119](#)

## V

- va\_arg, [3-12](#)
- va\_copy, [3-12](#)
- va\_end, [3-12](#)
- va\_start, [3-12](#)
- Vector table, [5-3](#), [9-17](#)
- Version information, [6-314](#), [6-326](#)
- vfprintf, [3-16](#)
- vfscanf, [3-16](#)
- vfwprintf, [3-16](#)
- vfwscanf, [3-16](#)
- Virtual function table
  - force*, [6-78](#)
  - suppress*, [6-146](#)
- vprintf, [3-16](#)

- vscanf, [3-16](#)
- vsprintf, [3-16](#)
- vsscanf, [3-16](#)
- vswprintf, [3-16](#)
- vswscanf, [3-16](#)
- vwprintf, [3-16](#)
- vwwscanf, [3-16](#)

## W

- warning, [1-11](#)
- Warnings
  - suppressing*, [6-35](#), [6-180](#), [6-229](#), [6-275](#)
  - treat as errors*, [6-195](#)
- wchar\_t, [3-12](#)
- wchar\_t keyword, [6-160](#)
- wcrtomb, [3-24](#)
- wcscat, [3-21](#)
- wcschr, [3-21](#)
- wcscmp, [3-21](#)
- wcscoll, [3-21](#)
- wcscpy, [3-20](#)
- wcscspn, [3-21](#)
- wcsncat, [3-21](#)
- wcsncmp, [3-21](#)
- wcsncpy, [3-20](#)
- wcspbrk, [3-21](#)
- wcsrchr, [3-21](#)
- wcsrtombs, [3-24](#)
- wcsspn, [3-21](#)
- wcssstr, [3-21](#)
- wcstod, [3-19](#)
- wcstof, [3-19](#)
- wcstoimax, [3-7](#)
- wcstok, [3-21](#)
- wcstol, [3-19](#)
- wcstold, [3-19](#)
- wcstoll, [3-19](#)
- wcstombs, [3-20](#)
- wcstoul, [3-19](#)
- wcstoull, [3-19](#)
- wcstoumax, [3-7](#)
- wcsxfrm, [3-21](#)
- wctob, [3-24](#)
- wctomb, [3-20](#)
- wctrans, [3-25](#)
- wctype, [3-25](#)
- weak, [1-11](#), [4-45](#)
- WEOF, [3-13](#)
- wmemchr, [3-21](#)
- wmemcmp, [3-21](#)
- wmemcpy, [3-20](#)
- wmemmove, [3-20](#)
- wmemset, [3-21](#)
- wprintf, [3-16](#)
- write, [3-24](#)
- wscanf, [3-16](#)
- wstrftime, [3-22](#)