TASKING

TASKING SmartCode - ARC/PPU User Guide

Table of Contents

Chapter 1. C Language

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

The TASKING C compiler fully supports the ISO C99 standard and supports all mandatory language features of the C11 and C17 standard, and adds extra possibilities to program the special functions of the target. C17 is the default of the C compiler.

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

C11/C17 language features

All mandatory ISO C11/C17 language features are supported (ISO/IEC 9899:2011/9899:2018 section 6.10.8.1 Mandatory macros). Furthermore the C compiler supports the following conditional features (ISO/IEC 9899:2011/9899:2018 section 6.10.8.3 Conditional feature macros):

• variable length arrays and variably modified types

Other conditional language features such as threads, as mentioned in section 6.10.8.3 Conditional feature macros and section 6.10.8.2 Environment macros of the ISO/IEC 9899:2011/9899:2018 standard, are not supported. `__STDC_NO_ATOMICS__` and `__STDC_NO_THREADS__` are defined as 1.

Additional language features

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

• keywords to specify memory types for data and functions

• attribute to specify alignment and absolute addresses

• intrinsic (built-in) functions that result in target specific assembly instructions

• pragmas to control the compiler from within the C source

• predefined macros

• the possibility to use assembly instructions in the C source

• keywords for inlining functions and programming interrupt routines

• libraries

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

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

1.1. Data Types

The C compiler supports the ISO C defined data types. The sizes of these types are shown in the following table.

C Type	Size	Align	Limits
_Bool	1	8	0 or 1
signed char	8	8	$[-2^7, 2^7-1]$
unsigned char	8	8	$[0, 2^8-1]$
short	16	16	$[-2^{15}, 2^{15}-1]$
unsigned short	16	16	$[0, 2^{16}-1]$
int	32	32	$[-2^{31}, 2^{31}-1]$
unsigned int	32	32	$[0, 2^{32}-1]$
enum [1]	8 16 32	8 16 32	$[-2^7, 2^7-1]$ or $[0, 2^8-1]$ $[-2^{15}, 2^{15}-1]$ or $[0, 2^{16}-1]$ $[-2^{31}, 2^{31}-1]$
long	32	32	$[-2^{31}, 2^{31}-1]$
unsigned long	32	32	$[0, 2^{32}-1]$
long long	64	32	$[-2^{63}, 2^{63}-1]$
unsigned long long	64	32	$[0, 2^{64}-1]$
_Float16 (10-bit significand) [2]	16	16	[-65504.0F,-6.103515625E-05] [+6.103515625E-05,+65504.0F]
float (23-bit significand)	32	32	[−3.402E+38, −1.175E-38] [+1.175E-38, +3.402E+38]
double long double (52-bit significand)	64	32	[-1.797E+308, -2.225E-308] [+2.225E-308, +1.797E+308]
_Imaginary float	32	32	[−3.402E+38i, −1.175E-38i] [+1.175E-38i, +3.402E+38i]
_Imaginary double _Imaginary long double	64	32	[-1.797E+308i, -2.225E-308i] [+2.225E-308i, +1.797E+308i]
_Complex float	64	32	real part + imaginary part
_Complex double _Complex long double	128	32	real part + imaginary part
pointer to data or function	32	32	$[0, 2^{32}-1]$

[1] When you use the `enum` type, the compiler will use the smallest suitable integer type (`char`, `unsigned char`, `short`, `unsigned short` or `int`).

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

__bitsizeof() operator

The `sizeof` operator always returns the size in bytes. Use the `__bitsizeof` operator in a similar way to return the size of an object or type in bits.

```
__bitsizeof( object | type )
```

Aggregate and Union Types

Aggregates (structures, classes, and arrays) and unions assume the alignment of their most strictly aligned component, that is, the component with the largest alignment. All members of the aggregate types are aligned as required by their individual types as listed in the table above. The struct/union data types may contain bit-fields. The allowed bit-field fundamental data types are `_Bool`, `(un)signed char`, `(un)signed short` and `(un)signed int`. The maximum bit-field size is equal to that of the type's size. For the bit-field types the same rules regarding to alignment and signed-ness apply as specified for the fundamental data types. In addition, the following rules apply:

• The first bit-field is stored at the least significant bits. Subsequent bit-fields fill the higher significant bits.

• A bit-field of a particular type cannot cross a boundary as is specified by its maximum width. For example, a bit-field of type `int` cannot cross a 32-bit boundary.

• Bit-fields share a storage unit with other bit-field members if and only if there is sufficient space in the storage unit.

• An unnamed bit-field creates a gap that has the size of the specified width. As a special case, an unnamed bit-field having width 0 (zero) prevents any further bit-field from residing in the storage unit corresponding to the type of the zero-width bit-field.

1.1.1. Half Precision Floating-Point

The TASKING C compiler supports half precision (16-bit) floating-point via the `_Float16` type using the binary16 interchange format. The binary16 interchange format is defined in IEEE Std 754-2008 IEEE Standard for Floating-Point Arithmetic. The `_Float16` type is defined in *ISO/IEC TS 18661-3 Draft Technical Specification – December 4, 2014 WG14 N1896*.

The `_Float16` type with binary16 format can represent normalized values in the range of 2^{-14} to 65504. There are 11 bits of significant precision, approximately 3 decimal digits. Also subnormal values are supported, as defined by `FLT16_HAS_SUBNORM` in `float.h`.

The `_Float16` type is a storage format only. For purposes of arithmetic and other operations, `_Float16` values in C expressions are automatically promoted to `float`.

Note that all conversions from and to `_Float16` involve an intermediate conversion to `float`. Because of rounding, this can sometimes produce a different result than a direct conversion.

1.1.2. Vector Data Types

The C compiler supports the following native width vector data types:

Vector Data Type	Description
vNint_t	Vector of 8 (ppu_tc45x, ppu_tc49xn, ppu_tc4dx) or 16 (ppu_tc49x) signed 32-bit integers
vNuint_t	Vector of 8 (ppu_tc45x, ppu_tc49xn, ppu_tc4dx) or 16 (ppu_tc49x) unsigned 32-bit integers
vNfloat_t	Vector of 8 (ppu_tc45x, ppu_tc49xn, ppu_tc4dx) or 16 (ppu_tc49x) single-precision 32-bit floats
vNx2short_t	Vector of 16 (ppu_tc45x, ppu_tc49xn, ppu_tc4dx) or 32 (ppu_tc49x) signed 16-bit integers
vNx2ushort_t	Vector of 16 (ppu_tc45x, ppu_tc49xn, ppu_tc4dx) or 32 (ppu_tc49x) unsigned 16-bit integers
vNx2half_t	Vector of 16 (ppu_tc45x, ppu_tc49xn, ppu_tc4dx) or 32 (ppu_tc49x) half-precision 16-bit floats
vNx4char_t	Vector of 32 (ppu_tc45x, ppu_tc49xn, ppu_tc4dx) or 64 (ppu_tc49x) signed 8-bit integers
vNx4uchar_t	Vector of 32 (ppu_tc45x, ppu_tc49xn, ppu_tc4dx) or 64 (ppu_tc49x) unsigned 8-bit integers

The C compiler supports the following double native width vector data types:

Vector Data Type	Description
vNx2int_t	Vector of 16 (ppu_tc45x, ppu_tc49xn, ppu_tc4dx) or 32 (ppu_tc49x) signed 32-bit integers
vNx2uint_t	Vector of 16 (ppu_tc45x, ppu_tc49xn, ppu_tc4dx) or 32 (ppu_tc49x) unsigned 32-bit integers
vNx2float_t	Vector of 16 (ppu_tc45x, ppu_tc49xn, ppu_tc4dx) or 32 (ppu_tc49x) single-precision 32-bit floats
vNx4short_t	Vector of 32 (ppu_tc45x, ppu_tc49xn, ppu_tc4dx) or 64 (ppu_tc49x) signed 16-bit integers
vNx4ushort_t	Vector of 32 (ppu_tc45x, ppu_tc49xn, ppu_tc4dx) or 64 (ppu_tc49x) unsigned 16-bit integers
vNx4half_t	Vector of 32 (ppu_tc45x, ppu_tc49xn, ppu_tc4dx) or 64 (ppu_tc49x) half-precision 16-bit floats

The C compiler supports the following quad native width vector data types:

Vector Data Type	Description
vNx4int_t	Vector of 32 (ppu_tc45x, ppu_tc49xn, ppu_tc4dx) or 64 (ppu_tc49x) signed 32-bit integers

Vector Data Type	Description
vNx4uint_t	Vector of 32 (ppu_tc45x, ppu_tc49xn, ppu_tc4dx) or 64 (ppu_tc49x) unsigned 32-bit integers
vNx4float_t	Vector of 32 (ppu_tc45x, ppu_tc49xn, ppu_tc4dx) or 64 (ppu_tc49x) single-precision 32-bit floats

Automatic variables of these native width, double native width, and quad native width types are allocated to the vector registers (vr0 .. vr39), unless they must be allocated to memory (e.g. because address is taken). Double native width types take up two vector registers (register pair) and quad native width types take up four vector registers. Note that vector register pairs start on an even register (vr0, vr2, etc.) and quad registers start on a number divisible by 4.

Furthermore the C compiler supports the following predicate vector types:

Predicate Vector Type	Description
pvN_t	Predicate vector of 8 (ppu_tc45x, ppu_tc49xn, ppu_tc4dx) or 16 (ppu_tc49x) _Bool values
pvNx2_t	Predicate vector of 16 (ppu_tc45x, ppu_tc49xn, ppu_tc4dx) or 32 (ppu_tc49x) _Bool values
pvNx4_t	Predicate vector of 32 (ppu_tc45x, ppu_tc49xn, ppu_tc4dx) or 64 (ppu_tc49x) _Bool values

Automatic variables of these types are allocated to the vector predicate registers (p1 .. p7).

You should use vector variables either as automatic variables (allocated to registers), static variables with the qualifier __vccm (allocated to .vdata or .vbss section), or pointers dynamically allocated with the intrinsic function __vccm_alloca() (allocated on vector stack, freed automatically upon function exit). Other uses of vectors (function parameters, struct fields, non-vccm memory) are supported, but inefficient.

Vector operations

All the basic arithmetic operations are supported natively:

- The following element-wise operations are supported for integer vector types: +, -, *, /, %, |, &, ^, >>, <<, unary -, unary ~, producing the same resulting type.

- The following element-wise operations are supported for native width floating-point vector types: +, -, *, /, unary -, producing the same resulting type.

- The following element-wise comparison operations are supported for native width integer and floating-point vector types: ==, !=, <, >, <=, >=.

  The resulting type of these operations is a native signed integer vector of the same number of elements, i.e.:

  - Comparison of vectors with 32-bit elements (vNint_t, vNuint_t, vNfloat_t) results in vNint_t;

5

- Comparison of vectors with 16-bit elements (`vNx2short_t`, `vNx2ushort_t`, `vNx2half_t`) results in `vNx2short_t`;

- Comparison of vectors with 8-bit elements (`vNx4char_t`, `vNx4uchar_t`) results in `vNx4char_t`.

  The value of the comparison result vector element is either 0 (corresponding to false), or -1 (all bits set to 1, corresponding to true).

  To convert the comparison result vector to a predicate type, use either an explicit cast (e.g. `(pvN_t)(v1 < v2)`), or one of the helper macros from `arc_vector.h` (e.g. `to_pvNx2(v1 < v2)`).

- The following element-wise operations are supported for vector predicate types: &, |, ^, ==, !=, unary ~, producing the same resulting type.

More complicated operations are supported through intrinsics. Instead of using the raw intrinsics it is recommended to use the overloaded wrappers defined in the `arc_vector.h` header file.

Vector initialization with the '{}' notation is supported. Vector element access (using the subscript operator '[]') is supported for all vector types, producing corresponding scalar resulting type. For predicate vectors there are limitations: non-constant index and pointer casts are not supported.

Loads and stores from `__vccm` qualified pointers can be done either directly (with the dereference operator '*'), or with `vvld*`/`vvst*` wrapped intrinsic functions (available in `arc_vector.h`). These intrinsics can also be used for predicated loads and stores, partial loads and stores, and bit-reverse addressing.

The `arc_vector.h` header file also defines macros for dereference operators vloadN, vloadNx2, vloadNx4, vstoreN, vstoreNx2, vstoreNx4. These macros differ from dereference operators in that they accept a pointer to an element type rather than to vector type. For example:

```
short __vccm *p;
vNx2short_t v;
vstoreNx2(v, p); // the same as *(vNx2short_t __vccm*)p = v;
```

Casts between vectors of different types are not supported, except for casts between signed and unsigned integer types of the same size, casts between `float` and `int` vectors of the same size (`vNint_t` **<->** `vNfloat_t` and `vNx2short_t` **<->** `vNx2half_t`), and casts from integer vectors to predicate vectors, which is done using the convenience routines (`to_vNint_t`, `to_pvN`, etc.).

1.2. Changing the Alignment: __aligned__()

By default the compiler aligns variables, functions and structure members to the minimum alignment required by the architecture. See Section 1.1, *Data Types*. With the attribute `__aligned__(`*n*`)` or `__attribute__((aligned(`*n*`)))` you can increase the default alignment of variables, functions or structure members to *n* bytes. If you apply an alignment with a value lower than the default alignment of the variable, function or structure member, this has no effect on the alignment of the variable, function or structure member. The C compiler issues a warning in that case. The alignment must be a power of two. The compiler issues an error message otherwise.

When a function is inlined the attribute `__aligned__()` has no effect on the inlined code, the alignment attribute is ignored.

Example:

```
__aligned__(8) int globalvar; /* changed to 8 bytes alignment
                                 instead of default 4 bytes  */
```

1.3. Accessing Memory

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

In addition, you can place variables at absolute addresses with the keyword __at().

1.3.1. Memory Type Qualifiers

In the C language you can specify that a variable must lie in a specific part of memory. You can do this with a *memory type qualifier*. If you do not specify a memory type qualifier, data objects get a default memory type.

You can specify the following memory type qualifiers:

Qualifier	Description	Location	Maximum object size	Pointer size	Section type
__sda	Small data area (SDA)	2 KiB around GP pointer	2 KiB	32-bit	sdata, sbss
__no_sda	Direct addressable data	anywhere	no limit	32-bit	data, bss
__vccm	Vector closely coupled memory (VCCM)	address AUX_VECMEM_REGION with size VEC_MEM_SIZE	no limit	32-bit	data

The qualifiers are described in more detail in the Section 1.3.3, *Small Data Area (SDA)* and Section 1.3.4, *Vector Closely Coupled Memory (VCCM)* .

1.3.2. Placing an Object at an Absolute Address: __at()

Just like you can declare a variable in a specific part of memory (using memory qualifiers), you can also place an object or a function at an absolute address in memory.

With the attribute __at() you can specify an absolute address. The address is a 32-bit linear address.

Instead of the attribute __at(*n*) you can also use __at__(*n*) or __attribute__((at(*n*))).

Examples

```
unsigned char Display[80*24] __at( 0x93000000 );
```

The array Display is placed at address 0x93000000. In the generated assembly, an absolute section is created. On this position space is reserved for the variable Display.

```
int i __at(0x93000000) = 1;
int j __at__(0x93000004);
int k __attribute__((at(0x93000008)));

void f(void) __at( 0xa0001000 ) { __nop(); }
```

The function `f` is placed at address 0xa0001000.

Restrictions

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

- The argument of the `__at()` attribute must be a constant address expression. Otherwise the compiler generates an error.

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

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

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

- Absolute variables cannot overlap each other. If you declare two absolute variables at the same address, the assembler and/or linker issues an error. The compiler does not check this.

1.3.3. Small Data Area (SDA)

By default, data consisting of 4 bytes or less will be placed in the SDA. You can change this default limit of 4 bytes with C compiler option **--sda-max-data-size**. Instead of this option you can also use pragma `sda_max_data_size` around an object declaration. For example,

```
#pragma sda_max_data_size 16
int arr[4];
#pragma sda_max_data_size restore
```

You have to compile the entire program with the same **--sda-max-data-size** option value. More precisely, for every object all of its declarations have to be consistent with its definition with respect to the **--sda-max-data-size** option value (specified either as a compiler option, or as a pragma). So, if for example you override the option at a variable definition in some file with a pragma, you have to use the same pragma around all its `extern` declarations in other files.

With `__sda` you can indicate that a variable should be placed in the SDA, irrespective of its size.

Initialized data is placed in a `.sdata` section (which is similar to the `.data` section), while uninitialized or zero-initialized data is placed in a `.sbss` section (which is similar to the `.bss` section). The compiler allocates each symbol either completely in or completely outside of the SDA.

With `__no_sda` you can explicitly indicate that a variable should be placed in normal memory (`.data` or `.bss`).

The instruction set supports only a limited addressing range for SDA objects, and it's your responsibility to make sure all program objects fit into it. Objects accessed as bytes and half-words have even a more narrow range around the GP pointer: 512 bytes for single bytes and 1 KiB for half-words. If any access does not fit in the range the linker issues an error like:

```
larc E121: relocation error in "task1": relocation value 0x103680,
type R_ARC_SDA16_LD2, offset 0x222, section ".text" at address 0x86d4
is not within a 11-bit signed range from the value of gp as defined
by the symbol _SDA_BASE_
```

In this case you should mark some of the excessive variables with the `__no_sda` qualifier, reduce the value of the **--sda-max-data-size** option, or disable automatic SDA allocation completely by using **--sda-max-data-size=0**.

Examples

```
        char  c;    // 8-bit object in .bss
__no_sda char  d;   // 8-bit object in .bss (forced, overrules the option)
        short s;    // 16-bit object in .bss
        int   i;    // 32-bit object in .bss
        char  text[] = "No smoking"; // 11 bytes in .data

__sda char  c;    // 8-bit object in .sbss
__sda short s;    // 16-bit object in .sbss
__sda int   i;    // 32-bit object in .sbss
__sda char  text[] = "No smoking"; // 11 bytes in .sdata
```

1.3.4. Vector Closely Coupled Memory (VCCM)

You can use the `__vccm` memory qualifier to place a variable into a vector memory area.

It is allowed for:

• Static object declarations

• Pointer declarations to type-qualified objects

Non-automatic variables with the `__vccm` memory qualifier are located in the vector memory (VCCM). Function automatic variables can only be declared as pointers to dynamically allocated uninitialized `__vccm` qualified data. You can use the intrinsic function `__vccm_alloca()` for this dynamic allocation. By default, dynamic allocation is restricted by the vector stack size as defined in `vppu*.lsl` with `__VSTACK_SIZE`.

Non-automatic variables with the `__vccm` memory qualifier are located in the vector memory (VCCM). Function automatic variables can only be declared as pointers to dynamically allocated uninitialized `__vccm` qualified data. You can use the intrinsic function `__vccm_alloca()` for this dynamic allocation. By default, dynamic allocation is restricted by the vector stack size as defined in `vppu*.lsl` with `__VSTACK_SIZE`.

Initialized data is placed in a `.vdata` section (which is similar to the `.data` section), while uninitialized data is placed in a `.vbss` section (which is similar to the `.bss` section). By default uninitialized data is cleared with zeroes unless you specify C compiler option **--vccm-no-clear**. The compiler allocates each symbol either completely in or completely outside of the VCCM.

Example

```
__vccm int vccm_block[1024]; // Allocate 1024*4 == 4K bytes statically
                             // to VCCM

void func1()
{
    int __vccm* p = __vccm_alloca(1024*sizeof(int));
    // Allocate 1024*4 == 4K bytes for func1's lifetime
    // on VCCM vector stack
    ...
} // 4K bytes released from VCCM vector stack
```

1.3.5. Uncached Memory Access

You can use the `_Uncached` qualifier, or its alias `__uncached` to ensure that access to a variable bypasses the cache.

It is allowed for:

• Static object declarations

• Pointer declarations to type-qualified objects

It is ignored for automatic variables.

Example

```
_Uncached int u;      // uncached global

void func1(int v)
{
    u = v;             // uncached store to u
    int w = u + 1;     // uncached load of u
    ...
}
```

1.3.6. Accessing Hardware from C

Using Special Function Registers

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

Example use in C (SFRs from `regppu.sfr`):

```
void access_sfr(void)
{
    int chipid;

    JLI_BASE = 0;              /* access Jump and Link Indexed Base Address
                                  register as a whole      */

    chipid = IDENTITY.CHIPID; /* read CHIPID bit-field of IDENTITY
                                 Service Request Control register */
}
```

You can find a list of defined SFRs and defined bits by inspecting the SFR file for a specific processor. The files are located in the `sfr` subdirectory of the standard `include` directory. The file is named `regppu.sfr`. The compiler includes this register file if you specify option **--include-file=sfr/regppu.sfr**, or you can use control program option **--tasking-sfr**.

The names in `regppu.sfr` are the same as in chapters 5.3, 45.4 and 45.5 from the *DesignWare ARCv2 ISA Programmer's Reference Manual for DW EV7x Processors* [Version 6367-001 April 2020, Synopsys, Inc.], and there are bit-fields defined for SFRs with multiple fields.

Defining Special Function Registers: __sfr

SFRs are defined in SFR files and are written in C. To define that a pointer points to a value in the SFR memory space, you can use the qualifier `__sfr` (only valid for pointers).

When the SFR contains fields, the layout of the SFR is defined by a typedef-ed struct. The next example is part of an SFR file and illustrates the declaration of a special function register:

```
typedef struct {
    unsigned int CHIPID : 16;
    unsigned int ARCNUM :  8;
    unsigned int ARCVER :  8;
} identity_t;
```

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

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

```
#define IDENTITY (*((__sfr volatile const identity_t *) 0x004 ))
#define JLI_BASE (*((__sfr volatile uint32_t *)         0x290 ))
```

1.4. Shift JIS Kanji Support

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

• First (high) byte is in the range 0x81-0x9f or 0xe0-0xef.

• Second (low) byte is in the range 0x40-0x7e or 0x80-0xfc

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

Note that Shift JIS also includes Katakana and Hiragana.

Example:

```
// Example usage of Shift JIS Kanji
// Do not switch off option -Ak
// At the position of the italic text you can
// put your Shift JIS Kanji code
int  i;  // put Shift JIS Kanji here
char c1;
char c2;
unsigned int ui;
const char mes[]="put Shift JIS Kanji here";
const unsigned int ar[5]={'K','a','n',
                    'j','i'};
                    // 5 Japanese array
void main(void)
{
    i=(int)c1;
    i++; /* put Shift JIS Kanji here\
            continuous comment */
    c2=mes[9];
    ui=ar[0];
}
```

1.5. Using Assembly in the C Source: __asm()

With the keyword `__asm()` you can use assembly instructions in the C source and pass C variables as operands to the assembly code.

> It is recommended to use constructs in C or use intrinsic functions instead of `__asm()`. Be aware that C modules that contain assembly are not portable and harder to compile in other environments.

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

You need to tell the compiler exactly what happens in the inline assembly code because it uses that for code generation and optimization. The compiler needs to know exactly which registers are written and which registers are only read. For example, if the inline assembly writes to a register from which the compiler assumes that it is only read, the generated code after the inline assembly is based on the fact that the register still contains the same value as before the inline assembly. If that is not the case the

results may be unexpected. Also, an inline assembly statement using multiple input parameters may be assigned the same register if the compiler finds that the input parameters contain the same value. As long as this register is only read this is not a problem.

General syntax of the __asm keyword

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

or in a C variable declaration to replace a C variable name with another name in assembly:

```
type c_variable_name __asm( "assembly_variable_name" );
```

Instead of __asm(), you can also use the GNU constructs:

```
__asm__ [volatile] ( );  /* always allowed     */
asm [volatile] ( );      /* only when -Ag is used */
```

instruction_template	Assembly instructions may contain parameters from the input list or output list in the form: *instruction* **%**[**p**]*parm_nr*,.. or *instruction* **%**[**p**][*parm_name*]**,**... If an instruction template references registers explicitly, they must be spelled with a double percent sign (e.g. %%r0). For predicate register operands you can use the 'p' modifier character to prevent that the '%' part of the register name is being emitted. This is useful when the predicate register is part of an instruction name (see Example 7). To include the % character in the assembly output, it is necessary to specify %%.
instruction	Instruction mnemonic.
%*parm_nr*	Parameter number in the range 0 .. 9. The number is treated as an index to the combined output and input parameter lists.
%[*parm_name***]**	Reference to a parameter name used in the input or output parameter list.
output_param_list	[[[[*parm_name*]] "{=|+}[**&**]*constraint_char*"(*C_expression*)],...]
input_param_list	[[[[*parm_name*]] "*constraint_char*"(*C_expression*)],...]
=	**+**	When you use the **+** instead of **=** this informs the compiler that the operand is both read and written.
&	Says that an output operand is written to before the inputs are read, so this output must not be the same register as any input.
constraint _char	Constraint character: the type of register to be used for the *C_expression*. See the table below.
C_expression	Any C expression. For output parameters it must be an lvalue, that is, something that is legal to have on the left side of an assignment.

register_reserve_list	[["*register_name*"],...]
register_name	Name of the register you want to reserve. For example because this register gets clobbered by the assembly code. The compiler will not use this register for inputs or outputs. Note that reserving too many registers can make register allocation impossible. The register name must be prefixed with a % (e.g. %r0).

Specifying registers for C variables

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

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

Constraint character	Type	Operand	Remark
i	immediate value	*value*	
m	memory	*variable*	memory operand
r	general purpose register vector register predicate register	r0 .. r31, r58, r59, r60, r61, r63 vr0 .. vr31 (for ppu_tc49x and ppu_tc49xn), vr0 .. vr39 (for ppu_tc45x and ppu_tc4dx) p0 .. p7	
number	type of operand it is associated with	same as **%***number*	Input constraint only. The *number* must refer to an output parameter. Indicates that **%***number* and *number* are the same register.

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

Loops and conditional jumps

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

If you want to create a loop with `__asm()`, the whole loop must be contained in a single `__asm()` statement. The same counts for (conditional) jumps. As a rule of thumb, all references to a label in an `__asm()` statement must be in that same statement. You can use numeric labels for these purposes.

Example 1: no input or output

A simple example without input or output parameters. You can use any instruction or label. When it is required that a sequence of __asm() statements generates a contiguous sequence of instructions, then they can be best combined to a single __asm() statement. Compiler optimizations can insert instruction(s) in between __asm() statements. Note that you can use standard C escape sequences. Use newline characters '\n' to continue on a new line in a __asm() statement. For multi-line output, use tab characters '\t' to indent instructions.

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

Example 2: using output parameters

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

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

Generated assembly code:

```
    mov %r0,0xff
    st  %r0,[out]
```

Example 3: using input parameters

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

```
int in;
void initreg( void )
{
    __asm( "MOV  %%R0,%0"
           :
           : "r" (in) );
}
```

Generated assembly code:

```
    ld  %r0,[in]
    MOV %R0,%r0
```

Example 4: using input and output parameters

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

```
int in1, in2, out;

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

Generated assembly code:

```
        ld      %r0,[in1]
        ld      %r1,[in2]
        add %r0, %r0, %r1
        st      %r0,[out]
```

Example 5: reserving registers

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

Same as *Example 4*, but now register `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 `r2`).

```
int in1, in2, out;

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

Generated assembly code:

```
        ld      %r2,[in1]
        ld      %r3,[in2]
        add %r2, %r2, %r3
        st      %r2,[out]
```

Example 6: use the same register for input and output

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

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

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

int result,parm;

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

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

This results in the following generated assembly code:

```
    mov %r0,1000
    mov %r1,%r0
    mov %r2,parm

    asr %r0,%r0,2
    add %r1,%r1,%r0
    mov %r2,%r1
    mov %r3,%r1
    st %r3,[result]
```

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

```
__asm(
    "asr %1,%1,2\n\t"
    "add %2,%2,%1\n\t"
    "mov %3,%2\n\t"
    "mov %0,%2"
    : "=&r" (retvalue)
    : "r" (par1), "r" (par2), "r" (par3)
  );
  return retvalue;
}
```

Generated assembly code:

```
    mov %r0,1000
    mov %r1,parm

    asr %r0,%r0,2          ; same register, but is expected read-only
    add %r0,%r0,%r0
    mov %r1,%r0
    mov %r2,%r0
    st %r2,[result]        ; contains unexpected result
```

Example 7: use the p modifier

For predicate register operands you can use the 'p' modifier character in the instruction template to prevent that the '%' part of the register name is being emitted. This is useful when the predicate register is part of an instruction name. For example:

```
void ex7( void )
{
    pvN_t pv;
    v16int_t vec;

    __asm("vvpinit.w %0, 0, 0xFF00" : "=r" (pv) );
    __asm("vvmov.w.%p1 %0, 5" : "=r" (vec) : "r" (pv) );
}
```

Generated assembly code:

```
    vvpinit.w %p1, 0, 0xFF00
    vvmov.w.p1 %vr0, 5
```

Example 8: using + prefix in output operand

Perform a bitwise exclusive OR (XOR) operation. Registers are used for the input and output parameters (constraint `r`, `%0` for b, `%1` for a in the instruction template). An output operand usually starts with `=`, but it can also start with `+` to signify that the operand is both read and written by the instruction.

```
int a, b;
void initreg( void )
{
```

```
    __asm( "xor\t%0,%0,%1"
            : "+r"(b)
            : "r"(a));
}
```

Generated assembly code:

```
    ld_s    %r0,[%gp,@sda(a)]
    ld_s    %r1,[%gp,@sda(b)]
    xor     %r1,%r1,%r0
    st.as   %r1,[%gp,@sda(b)]
```

Example 9: using symbolic name for operands

Instead of numbers, you can also use symbolic names for input and output operands by specifying it inside the square brackets (**[ ]**). For example, [output] is the symbolic name for an operand.

```
int a, b;
void initreg( void )
{
    __asm("xor\t%[output],%[output],%2"
        : [output] "=r"(b)
        : "0"(a), "r"(b));
}
```

Generated assembly code:

```
    ld_s    %r0,[%gp,@sda(a)]
    ld_s    %r1,[%gp,@sda(b)]
    xor     %r0,%r0,%r1
    st_s    %r0,[%gp,@sda(b)]
```

Example 10: replacing a C variable name with another name in assembly

You can replace a C variable name with another name in assembly. Here, name_should_not_appear is replaced with _test_symname at assembly level.

```
char name_should_not_appear __asm("_test_symname") = 2;

__attribute__((noinline)) void initreg (void)
{
    __asm ("stb %0,[%%gp,@sda(_test_symname)]"
            : : "r" (5)
            : "memory");
}
```

Generated assembly code:

```
    .section    .text.test.initreg
    .global     initreg
; Function initreg
```

```
initreg:     .type     func
    mov_s     %r0,5
    stb %r0,[%gp,@sda(_test_symname)]
    j_s    [%blink]
    .size     initreg,*-initreg
    ; End of function
    .endsec    ; End of section

    .section    .sdata.test._test_symname
    .global    _test_symname
_test_symname:     .type    object
    .size    _test_symname,1
    .db    2
    .endsec    ; End of section
```

1.6. GNU C Language Extensions

The C compiler supports the following GNU C language extensions:

- `__builtin` intrinsics of specific C library functions. See Section 1.10.4.6, *Built-in Library Functions*

- Intrinsic functions `__builtin_choose_expr`, and `__builtin_types_compatible_p`. See Section 1.10.4, *Intrinsic Functions*.

- Token `__builtin_offsetof` that defines the macro `offsetof`.

- Keyword variants:

  - Instead of `__alignof`, you can also use `__alignof__` or alternatively `alignof` with C compiler option **--language=+gcc** (**-Ag**).

  - Instead of `__asm`, you can also use `__asm__` or alternatively `asm` with C compiler option **--language=+gcc** (**-Ag**).

  - Instead of `__typeof`, you can also use `__typeof__` or alternatively `typeof` with C compiler option **--language=+gcc** (**-Ag**).

  - Instead of `volatile`, you can also use `__volatile__` or `__volatile`.

1.7. Attributes

You can use the keyword `__attribute__` to specify special attributes on declarations of variables, functions, types, and fields.

Syntax:

`__attribute__((`*name*`,...))`

or:

__name__

The second syntax allows you to use attributes in header files without being concerned about a possible macro of the same name. This second syntax is only possible on attributes that do not already start with an underscore. For example, you may use __noreturn__ instead of __attribute__((noreturn)).

The following attributes are supported:

alias("*symbol*")

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

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

declares 'f' to be a weak alias for '__f'.

aligned(*value*)

You can use __attribute__((aligned(n))) to increase the alignment of variables or functions. If you apply an alignment with a value lower than the default alignment of the variable or function, this has no effect on the alignment of the variable or function. The C compiler issues a warning in that case. The alignment must be a power of two. The compiler issues an error message otherwise. When a function is inlined the attribute has no effect on the inlined code, the attribute is ignored. See also Section 1.2, *Changing the Alignment: __aligned__()*.

at(*value*)

You can use __attribute__((at(n))) to place an object or a function at an absolute address in memory. The address is a 32-bit linear address.

Instead of the attribute __attribute__((at(*n*))) you can also use __at(*n*) or __at__(*n*). See also Section 1.3.2, *Placing an Object at an Absolute Address: __at()*.

```
int i __attribute__((at(0x93000008)));
```

const

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

The following kinds of functions should not be declared __const__:

• A function with pointer arguments which examines the data pointed to.

• A function that calls a non-const function.

export

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

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

flatten

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

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

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

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

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

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

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

Example:

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

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

leaf

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

malloc

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

• The return value of a call to such a function points to a memory location or can be a null pointer.

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

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

noinline

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

always_inline

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

noreturn

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

```
void fatal() __attribute__((noreturn));

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

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

overloadable

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

It is not possible to have both a normal and an `overloadable` function of the same name. In that case, the normal function takes precedence. The `overloadable` attribute is ignored for functions without a prototype.

When calling a function for which only overloadable definitions are visible, the function with the best match is selected. The best match is the function with the correct number of arguments, requiring the least amount of argument conversions. When there are no matches, or when there are multiple ambiguous matches, an error is generated.

protect

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

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

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

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

pure

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

section("*section_name*")

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

```
void foobar(void) __attribute__((section(".text.foobar")));
int baz __attribute__((section(".bss.baz")));
```

puts the function `foobar` in the section named `.text.foobar`, and puts variable `baz` in the section named `.bss.baz`.

used

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

```
static const char copyright[] __attribute__((used)) = "Copyright 2020 TASKING BV";
```

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

unused

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

weak

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

See also `#pragma weak`.

1.8. Pragmas to Control the Compiler

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

The syntax is:

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

or:

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

Some pragmas can accept the following special arguments:

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

Examples:

```
// by default all warnings are shown

#pragma warning 535        // disable W535
#pragma warning 530        // also disable W530
const char var_1 = 0x5678;  // W530 is not shown
```

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

Label pragmas

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

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

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

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

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

Example:

```
#pragma lab1:optimize P

volatile int v;

void f( void )
{
      int i, a;

      a = 42;

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

Supported pragmas

The compiler recognizes the following pragmas, other pragmas are ignored. On the command line you can use **carc --help=pragmas** to get a list of all supported pragmas. Pragmas marked with (*) support a label prefix.

STDC FP_CONTRACT [on | off | default | restore] (*)

This pragma is defined in ISO C99/C11/C17. With this pragma you can control the **+contract** flag of C compiler option **--fp-model**.

alias *symbol=defined_symbol*

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

align {*value* | default | restore} (*)

Increase the alignment of variables or functions. If you apply an alignment with a value lower than the default alignment of the variable or function, this has no effect on the alignment of the variable or function. The C compiler issues a warning in that case. When a function is inlined the pragma has no effect on the inlined code, the pragma is ignored. The alignment value must be a power of two or 0. Value 0 defaults to the compiler natural object alignment.

See Section 1.2, *Changing the Alignment: __aligned__()*.

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

This pragma is used to mark the macros "false" and "true" from the library header file stdbool.h as "essentially BOOLEAN", which is a concept from the MISRA C:2012 standard.

compactmaxmatch {*value* | default | restore} (*)

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

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

extension isuffix [on | off | default | restore] (*)

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

```
float 0.5i
```

extern *symbol*

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

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

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

With this pragma you can control the **+negzero** flag of C compiler option **--fp-model**.

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

With this pragma you can control the **+nonan** flag of C compiler option **--fp-model**.

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

With this pragma you can control the **+rewrite** flag of C compiler option **--fp-model**.

inline / noinline / smartinline [default | restore] (*)

See Section 1.10.2, *Inlining Functions: inline*.

inline_max_incr {*value* | default | restore} (*)
inline_max_size {*value* | default | restore} (*)

With these pragmas you can control the automatic function inlining optimization process of the compiler. It has effect only when you have enabled the inlining optimization (C compiler option **--optimize=+inline**).

See C compiler options **--inline-max-incr / --inline-max-size**.

macro / nomacro [on | off | default | restore] (*)

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

maxcalldepth {*value* | default | restore} (*)

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

See C compiler option **--max-call-depth**.

message "*message*" ...

Print the message string(s) on standard output.

modulo_schedule *value* / end_modulo_schedule [default | restore] (*)

Enable modulo scheduling and specify the maximum number of overlapping iterations allowed. At the end of the loop use end_modulo_schedule.

See also Section 4.6.3, *Modulo Scheduling*.

nomisrac [*nr,...*] [default | restore] (*)

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

See C compiler option **--misrac** and Section 4.7.2, *C Code Checking: MISRA C*.

optimize [*flags*] / endoptimize [default | restore] (*)

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

See Section 4.6, *Compiler Optimizations*.

protect / endprotect [on | off | default | restore] (*)

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

runtime [*flags* | default | restore] (*)

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

section all ["*name*" | default | restore] (*)
section *type* ["*name*" | default | restore] (*)

Changes section names. See Section 1.11, *Compiler Generated Sections* and C compiler option **--rename-sections** for more information.

sda_max_data_size {*size* | default | restore} (*)

By default, data consisting of 4 bytes or less will be placed in the Small Data Area (SDA). With this pragma you can change this default limit of 4 bytes. The pragma works the same as C compiler option **--sda-max-data-size**.

source / nosource [on | off | default | restore] (*)

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

See C compiler option **--source**.

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

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

tradeoff {*level* | default | restore} (*)

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

unroll_factor *value* / endunroll_factor [default | restore] (*)

Specify how many extra iterations should be added for the following loop if and when it is unrolled. At the end of the loop use endunroll_factor, leaving the unroller to determine the best factor automatically.

To achieve that a loop is unrolled four times, enter:

```
#pragma unroll_factor 3
  ...
#pragma endunroll_factor
```

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

By default, uninitialized vector data is cleared to zero on startup. With pragma vccm_noclear this step is skipped.

See C compiler option **--vccm-no-clear**.

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

By default, any possible aliases will disable auto-vectorization for a loop. With pragma vectorize_noalias you can selectively disable alias checking for specific loops.

See C compiler option **--vectorize-noalias**.

warning [*number*,...] [default | restore] (*)

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

weak *symbol*

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

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

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

1.9. Predefined Preprocessor Macros

You can use the following predefined macros in your C source. The macros are useful to create conditional C code.

Macro	Description
__BIG_ENDIAN__	Expands to 0. The processor accesses data in little-endian.
__BUILD__	Identifies the build number of the compiler in the format yymmddqq (year, month, day and quarter in UTC). For example: 20051340 means May 13, 2020 between 10:00 and 10:15.
__CARC__	Expands to 1 for the TASKING ARC C compiler, otherwise unrecognized as macro.
__DATE__	Expands to the compilation date: "mmm dd yyyy".
__DOUBLE_FP__	Expands to 1 for double-precision FPU ('double' is always fully featured for the ppu_tc45x, ppu_tc49x, ppu_tc49xn, and ppu_tc4dx).
__FILE__	Expands to the current source file name.
__LINE__	Expands to the line number of the line where this macro is called.
__MISRAC_VERSION__	Expands to the MISRA C version used 1998, 2004 or 2012 (option **--misrac-version**). The default is 2012.
__REVISION__	Expands to the revision number of the compiler. Digits are represented as they are; characters (for prototypes, alphas, betas) are represented by -1. Examples: v1.0r1 -> 1, v1.0rb -> -1
__STDC__	Identifies the level of ANSI standard. The macro expands to 1 if you set option **--language** (Control language extensions), otherwise expands to 0.
__STDC_HOSTED__	Always expands to 0, indicating the implementation is not a hosted implementation.
__STDC_NO_ATOMICS__	(C11/C17 only) Expands to 1 to indicate that this implementation does not support atomic types and the stdatomic.h header file.
__STDC_NO_THREADS__	(C11/C17 only) Expands to 1 to indicate that this implementation does not support the threads.h header file.
__STDC_VERSION__	Identifies the ISO C version number. Expands to 201710L for ISO C17, 201112L for ISO C11, 199901L for ISO C99 or 199409L for ISO C90.
__TASKING__	Identifies the compiler as a TASKING compiler. Expands to 1 if a TASKING compiler is used.
__TIME__	Expands to the compilation time: "hh:mm:ss"
__VDSP_VEC_WIDTH__	Expands to the native vector width in bits. 512 for the ppu_tc49x, or 256 for the ppu_tc45x, ppu_tc49xn, and ppu_tc4dx.
__VERSION__	Identifies the version number of the compiler. For example, if you use version 1.2r3 of the compiler, __VERSION__ expands to 1002 (dot and revision number are omitted, the minor version number is in 3 digits).

Example

```
#ifdef __CARC__
/* this part is only valid for the ARC C compiler */
...
#endif
```

1.10. Functions

1.10.1. Calling Convention and Register Usage

The compiler follows the calling convention and register usage as described in *section 2.2 Function Calling Sequence* of the *DesignWare ARCv2 System V ABI Supplement* [Version 4092-007, Nov 01, 2019, Synopsys, Inc.].

1.10.2. Inlining Functions: inline

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

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

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

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

Combining inline with __asm to create intrinsic functions

With the keyword `__asm` it is possible to use assembly instructions in the body of an inline function. Because the compiler inserts the (assembly) body at the place the function is called, you can create your own intrinsic function. See Section 1.10.4, *Intrinsic Functions*.

1.10.3. Interrupt Functions / Exception Handling

The compiler supports user-designated interrupt functions and exception handling functions. The ARC/PPU interrupt unit has 16 allocated exceptions associated with vectors 0 to 15 and 240 interrupts associated with vectors 16 to 255. For an extensive description see chapter 6 *Interrupts and Exceptions* in the *DesignWare ARCv2 ISA Programmer's Reference Manual for DW EV7x Processors* [Version 6367-001 April 2020, Synopsys, Inc.].

1.10.3.1. Defining an Interrupt Service Routine: __interrupt()

With the function type qualifier `__interrupt()` you can declare a function as an interrupt function or an exception handler. The function type qualifier `__interrupt()` takes one vector number (0..255) as argument.

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

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

The argument *n* is the vector number. The vector number must be in range [0..255]. Vectors 0 to 15 are assigned to internal exceptions and vectors 16 to 255 are assigned to external interrupts.

Example

```
void __interrupt( 1 ) isr( void )
{
      ...
}
```

1.10.4. 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 (__).

1.10.4.1. Intrinsics Used By Compiler and Libraries

Intrinsic Function	Description
void * volatile __alloc(__size_t *size*);	Allocate memory. Returns a pointer to memory of *size* bytes length. Returns NULL if there is not enough space left. This function is used internally for variable length arrays, it is not to be used by end users.
void * __va_start(void);	Variable argument '...' operator. Used in library function `va_start()`. Returns the stack offset to the variable argument list.
volatile void __free(void *p);	Deallocates the memory pointed to by `p`. `p` must point to memory earlier allocated by a call to `__alloc()`.
volatile void __nop(void);	Generate a NOP instruction.

1.10.4.2. SIMD Intrinsics

Intrinsic Function	Mapped Instruction
unsigned long long __vadd2(unsigned long long x, unsigned long long y);	VADD2
unsigned long __vadd2h(unsigned long x, unsigned long y);	VADD2H
unsigned long long __vadd4h(unsigned long long x, unsigned long long y);	VADD4H
unsigned long long __vsub2(unsigned long long x, unsigned long long y);	VSUB2
unsigned long __vsub2h(unsigned long x, unsigned long y);	VSUB2H
unsigned long long __vsub4h(unsigned long long x, unsigned long long y);	VSUB4H
unsigned long long __vaddsub(unsigned long long x, unsigned long long y);	VADDSUB
unsigned long __vaddsub2h(unsigned long x, unsigned long y);	VADDSUB2H
unsigned long long __vaddsub4h(unsigned long long x, unsigned long long y);	VADDSUB4H
unsigned long long __vsubadd(unsigned long long x, unsigned long long y);	VSUBADD
unsigned long __vsubadd2h(unsigned long x, unsigned long y);	VSUBADD2H
unsigned long long __vsubadd4h(unsigned long long x, unsigned long long y);	VSUBADD4H
long long __vmpy2h(unsigned long x, unsigned long y);	VMPY2H
unsigned long long __vmpy2hu(unsigned long x, unsigned long y);	VMPY2HU
long long __vmac2h(unsigned long x , unsigned long y);	VMAC2H
unsigned long long __vmac2hu(unsigned long x, unsigned long y);	VMAC2HU

1.10.4.3. DSP Intrinsics

Intrinsic Function	Mapped Instruction
long long __mpyd(long x, long y);	MPYD
unsigned long long __mpydu(unsigned long x, unsigned long y);	MPYDU

Intrinsic Function	Mapped Instruction
long __dmpyh(unsigned long x, unsigned long y);	DMPYH
unsigned long __dmpyhu(unsigned long x, unsigned long y);	DMPYHU
long long __dmpywh(unsigned long long x, unsigned long y);	DMPYWH
unsigned long long __dmpywhu(unsigned long long x, unsigned long y);	DMPYWHU
long long __qmpyh(unsigned long long x, unsigned long long y);	QMPYH
unsigned long long __qmpyhu(unsigned long long x, unsigned long long y);	QMPYHU
long __mac(long x, long y);	MAC
unsigned long __macu(unsigned long x, unsigned long y);	MACU
long long __macd(long x, long y);	MACD
unsigned long long __macdu(unsigned long x, unsigned long y);	MACDU
long __dmach(unsigned long x, unsigned long y);	DMACH
unsigned long __dmachu(unsigned long x, unsigned long y);	DMACHU
long long __dmacwh(unsigned long long x, unsigned long y);	DMACWH
unsigned long long __dmacwhu(unsigned long long x, unsigned long y);	DMACWHU
long long __qmach(unsigned long long x, unsigned long long y);	QMACH
unsigned long long __qmachu(unsigned long long x, unsigned long long y);	QMACHU

1.10.4.4. Miscellaneous Intrinsics

Intrinsic Function	Mapped Instruction	Operand Limitations
void __brk(void);	BRK	
unsigned __clri(void);	CLRI	
void __dmb(unsigned x);	DMB	x must be a constant ($0 <= x <= 7$).
void __dsync(void);	DSYNC	
void __flag(unsigned x);	FLAG	
void __kflag(unsigned x);	KFLAG	

Intrinsic Function	Mapped Instruction	Operand Limitations
void __prealloc(void * p);	PREALLOC	
void __prefetch(void * p);	PREFETCH	
void __prefetchw(void * p);	PREFETCHW	
void __seti(unsigned x);	SETI	
void __sleep(int t);	SLEEP	
void __sync(void);	SYNC	
void __swi(void);	SWI	
void __wevt(unsigned x);	WEVT	
void __wlfc(unsigned x);	WLFC	

Intrinsic Function	Description
type __builtin_choose_expr(*const_exp*, *exp1*, *exp2*);	Evaluate code depending on the value of a constant expression. Returns *exp1* if *const_exp*, which is an integer constant expression, is nonzero. Otherwise it returns *exp2*.
int __builtin_constant_p(*type exp*);	During compilation, constant propagation returns 1 if *exp* is a constant expression. Otherwise, it returns 0. Note that *type* can be int, long, long long, double or long double.
long __builtin_expect(long *exp*, long *c*);	Indicates that *exp* is likely to evaluate to a specified value *c*.
int __builtin_types_compatible_p(*type1*, *type2*);	Determine whether two types are the same. Returns 1 if *type1* and *type2* are compatible, 0 otherwise. You can use the result in integer constant expressions.

1.10.4.5. Vector Support Intrinsics

Intrinsic Function	Description
void __vccm * volatile __vccm_alloca(__size_t *size*);	Allocate memory on VCCM vector stack. Returns a pointer to VCCM memory of *size* bytes length.

Wrapped vector intrinsics

The vector intrinsic names in the following sections are wrapped. These intrinsic names do not begin with a double underscore character. The intrinsic wrappers are defined in file `arc_vector.h`.

Vector support load and store intrinsics

The header file `arc_vector.h` declares several functions (as macro wrappers over intrinsics) for vector load/store operations.

Intrinsic Function	Mapped Instruction	Remark
vNx4char_t vvld(const signed char __vccm * addr);	vvld.b	
vNx4char_t vvld(pvNx4 pred, const signed char __vccm * addr);	vvld.b.p	
vNx4uchar_t vvld(const unsigned char __vccm * addr);	vvld.b	
vNx4uchar_t vvld(pvNx4 pred, const unsigned char __vccm * addr);	vvld.b.p	
vNx2short_t vvld(const short __vccm * addr);	vvld.h	
vNx2short_t vvld(pvNx2 pred, const short __vccm * addr);	vvld.h.p	
vNx2ushort_t vvld(const unsigned short __vccm * addr);	vvld.h	
vNx2ushort_t vvld(pvNx2 pred, const unsigned short __vccm * addr);	vvld.h.p	
vNint_t vvld(const int __vccm * addr);	vvld.w	
vNint_t vvld(pvN pred, const int __vccm * addr);	vvld.w.p	
vNuint_t vvld(const unsigned int __vccm * addr);	vvld.w	
vNuint_t vvld(pvN pred, const unsigned int __vccm * addr);	vvld.w.p	
vNx2half_t vvld(const _Float16 __vccm * addr);	vvld.h	
vNx2half_t vvld(pvNx2 pred, const _Float16 __vccm * addr);	vvld.h.p	
vNfloat_t vvld(const float __vccm * addr);	vvld.w	
vNfloat_t vvld(pvN pred, const float __vccm * addr);	vvld.w.p	
vNuint_t vvld_ub_w(const unsigned char __vccm * addr);	vvld.ub.w	
vNuint_t vvld_ub_w(pvN pred, const unsigned char __vccm * addr);	vvld.ub.w.p	
vNuint_t vvld_uh_w(const unsigned short __vccm * addr);	vvld.uh.w	
vNuint_t vvld_uh_w(pvN pred, const unsigned short __vccm * addr);	vvld.uh.w.pv	
vNx2ushort_t vvld_ub_h(const unsigned char __vccm * addr);	vvld.ub.hv	
vNx2ushort_t vvld_ub_h(pvNx2 pred, const unsigned char __vccm * addr);	vvld.ub.h.p	
void vvst(vNx4char_t val, signed char __vccm * addr);	vvst.b	

Intrinsic Function	Mapped Instruction	Remark
void vvst(vNx4char_t val, pvNx4 pred, signed char __vccm * addr);	vvst.b.p	
void vvst(vNx4uchar_t val, unsigned char __vccm * addr);	vvst.b	
void vvst(vNx4uchar_t val, pvNx4 pred, unsigned char __vccm * addr);	vvst.b.p	
void vvst(vNx2short_t val, short __vccm * addr);	vvst.h	
void vvst(vNx2short_t val, pvNx2 pred, short __vccm * addr);	vvst.h.p	
void vvst(vNx2ushort_t val, unsigned short __vccm * addr);	vvst.h	
void vvst(vNx2ushort_t val, pvNx2 pred, unsigned short __vccm * addr);	vvst.h.p	
void vvst(vNint_t val, int __vccm * addr);	vvst.w	
void vvst(vNint_t val, pvN pred, int __vccm * addr);	vvst.w.p	
void vvst(vNuint_t val, unsigned int __vccm * addr);	vvst.w	
void vvst(vNuint_t val, pvN pred, unsigned int __vccm * addr);	vvst.w.p	
void vvst(vNx2half_t val, _Float16 __vccm * addr);	vvst.h	
void vvst(vNx2half_t val, pvNx2 pred, _Float16 __vccm * addr);	vvst.h.p	
void vvst(vNfloat_t val, float __vccm * addr);	vvst.w	
void vvst(vNfloat_t val, pvN pred, float __vccm * addr);	vvst.w.p	
void vvst_db(vNint_t val, signed char __vccm * addr);	vvst.db.w	
void vvst_db(vNint_t val, pvN pred, signed char __vccm * addr);	vvst.db.w.p	
void vvst_db(vNx2short_t val, signed char __vccm * addr);	vvst.db.h	
void vvst_db(vNx2short_t val, pvNx2 pred, signed char __vccm * addr);	vvst.db.h.p	
void vvst_dh(vNint_t val, short __vccm * addr);	vvst.dh.w	
void vvst_dh(vNint_t val, pvN pred, short __vccm * addr);	vvst.dh.w.p	
vNx4char_t vvld_bitr(vNx4char_t __vccm **addr, int c);	vvld.b.bitr / vvld.b.bitrpow2	The address addr is passed as a double pointer and updated.
vNx4uchar_t vvld_bitr(vNx4uchar_t __vccm **addr, int c);	vvld.b.bitr / vvld.b.bitrpow2	The address addr is passed as a double pointer and updated.

Intrinsic Function	Mapped Instruction	Remark
vNx2short_t vvld_bitr(vNx2short_t __vccm **addr, int c);	vvld.h.bitr / vvld.h.bitrpow2	The address `addr` is passed as a double pointer and updated.
vNx2ushort_t vvld_bitr(vNx2ushort_t __vccm **addr, int c);	vvld.h.bitr / vvld.h.bitrpow2	The address `addr` is passed as a double pointer and updated.
vNint_t vvld_bitr(vNint_t __vccm **addr, int c);	vvld.w.bitr / vvld.w.bitrpow2	The address `addr` is passed as a double pointer and updated.
vNuint_t vvld_bitr(vNuint_t __vccm **addr, int c);	vvld.w.bitr / vvld.w.bitrpow2	The address `addr` is passed as a double pointer and updated.
vNx2half_t vvld_bitr(vNx2half_t __vccm **addr, int c);	vvld.h.bitr / vvld.h.bitrpow2	The address `addr` is passed as a double pointer and updated.
vNfloat_t vvld_bitr(vNfloat_t __vccm **addr, int c);	vvld.w.bitr / vvld.w.bitrpow2	The address `addr` is passed as a double pointer and updated.
void vvst_bitr(vNx4char_t val, V __vccm **addr, int c);	vvst.b.bitr / vvst.b.bitrpow2	The address `addr` is passed as a double pointer and updated.
void vvst_bitr(vNx4uchar_t val, V __vccm **addr, int c);	vvst.b.bitr / vvst.b.bitrpow2	The address `addr` is passed as a double pointer and updated.
void vvst_bitr(vNx2short_t val, V __vccm **addr, int c);	vvst.h.bitr / vvst.h.bitrpow2	The address `addr` is passed as a double pointer and updated.
void vvst_bitr(vNx2ushort_t val, V __vccm **addr, int c);	vvst.h.bitr / vvst.h.bitrpow2	The address `addr` is passed as a double pointer and updated.
void vvst_bitr(vNint_t val, V __vccm **addr, int c);	vvst.w.bitr / vvst.w.bitrpow2	The address `addr` is passed as a double pointer and updated.
void vvst_bitr(vNuint_t val, V __vccm **addr, int c);	vvst.w.bitr / vvst.w.bitrpow2	The address `addr` is passed as a double pointer and updated.
void vvst_bitr(vNx2half_t val, V __vccm **addr, int c);	vvst.h.bitr / vvst.w.bitrpow2	The address `addr` is passed as a double pointer and updated.
void vvst_bitr(vNfloat_t val, V __vccm **addr, int c);	vvst.w.bitr / vvst.w.bitrpow2	The address `addr` is passed as a double pointer and updated.

Vector support gather and scatter intrinsics

The compiler supports intrinsics implementing vector gather load and scatter store operations for vectors of native width with element types of size 8, 16 and 32-bit. The header file `arc_vector.h` declares several functions as macro wrappers over these intrinsics.

The following `vgather` functions return values loaded from `addr + offsets`. For 8-bit element types, there are additional variants called `vgather_lo` and `vgather_hi` which load only the elements in the upper or lower half of the vector. Here

- `addr` is a pointer parameter to `__vccm`.

- `offsets` is a vector of displacements added to `addr` pointer to create a vector of addresses for the memory operation.

- `pred` is a predicate used to mask the load. If the `dflt` parameter is not specified, lane `i` is undefined if `pred[i]` is false.

- `dflt` is a vector, lane `i` is `dflt[i]` if `pred[i]` is false.

Intrinsic Function	Mapped Instruction
vNint_t vgather(const int __vccm * addr, vNint_t offsets);	vvld.fa.w
vNint_t vgather(const int __vccm * addr, vNint_t offsets, pvN pred);	vvld.fa.w.p
vNint_t vgather(const int __vccm * addr, vNint_t offsets, vNint_t dflt, pvN pred);	vvld.fa.w.p
vNuint_t vgather(const unsigned int __vccm * addr, vNint_t offsets);	vvld.fa.w
vNuint_t vgather(const unsigned int __vccm * addr, vNint_t offsets, pvN pred);	vvld.fa.w.p
vNuint_t vgather(const unsigned int __vccm * addr, vNint_t offsets, vNuint_t dflt, pvN pred);	vvld.fa.w.p
vNfloat_t vgather(const float __vccm * addr, vNint_t offsets);	vvld.fa.w
vNfloat_t vgather(const float __vccm * addr, vNint_t offsets, pvN pred);	vvld.fa.w.p
vNfloat_t vgather(const float __vccm * addr, vNint_t offsets, vNfloat_t dflt, pvN pred);	vvld.fa.w.p
vNx2short_t vgather(const short __vccm * addr, vNx2int_t offsets);	vvld.fa.h
vNx2short_t vgather(const short __vccm * addr, vNx2int_t offsets, pvNx2 pred);	vvld.fa.h.p
vNx2short_t vgather(const short __vccm * addr, vNx2int_t offsets, vNx2short_t dflt, pvNx2 pred);	vvld.fa.h.p
vNx2ushort_t vgather(const unsigned short __vccm * addr, vNx2int_t offsets);	vvld.fa.h
vNx2ushort_t vgather(const unsigned short __vccm * addr, vNx2int_t offsets, pvNx2 pred);	vvld.fa.h.p
vNx2ushort_t vgather(const unsigned short __vccm * addr, vNx2int_t offsets, vNx2ushort_t dflt, pvNx2 pred);	vvld.fa.h.p
vNx2half_t vgather(const _Float16 __vccm * addr, vNx2int_t offsets);	vvld.fa.h
vNx2half_t vgather(const _Float16 __vccm * addr, vNx2int_t offsets, pvNx2 pred);	vvld.fa.h.p
vNx2half_t vgather(const _Float16 __vccm * addr, vNx2int_t offsets, vNx2half_t dflt, pvNx2 pred);	vvld.fa.h.p

Intrinsic Function	Mapped Instruction
vNx4char_t vgather(const signed char __vccm * addr, vNx4int_t offsets);	vvld.fa.b
vNx4char_t vgather(const signed char __vccm * addr, vNx4int_t offsets, pvNx4 pred);	vvld.fa.b.p
vNx4char_t vgather(const signed char __vccm * addr, vNx4int_t offsets, vNx4char_t dflt, pvNx4 pred);	vvld.fa.b.p
vNx4uchar_t vgather(const unsigned char __vccm * addr, vNx4int_t offsets);	vvld.fa.b
vNx4uchar_t vgather(const unsigned char __vccm * addr, vNx4int_t offsets, pvNx4 pred);	vvld.fa.b.p
vNx4uchar_t vgather(const unsigned char __vccm * addr, vNx4int_t offsets, vNx4uchar_t dflt, pvNx4 pred);	vvld.fa.b.p
vNx4char_t vgather_lo(const signed char __vccm * addr, vNx2int_t offsets, vNx4char_t dflt, pvNx4 pred);	vvld.fa.b.p
vNx4char_t vgather_lo(const signed char __vccm * addr, vNx2int_t offsets, vNx4char_t dflt);	vvld.fa.b.p
vNx4uchar_t vgather_lo(const unsigned char __vccm * addr, vNx2int_t offsets, pvNx4 pred);	vvld.fa.b.p
vNx4uchar_t vgather_lo(const unsigned char __vccm * addr, vNx2int_t offsets, vNx4uchar_t dflt, pvNx4 pred);	vvld.fa.b.p
vNx4char_t vgather_hi(const signed char __vccm * addr, vNx2int_t offsets, vNx4char_t dflt, pvNx4 pred);	vvld.fa.b.p
vNx4char_t vgather_hi(const signed char __vccm * addr, vNx2int_t offsets, vNx4char_t dflt);	vvld.fa.b.p
vNx4uchar_t vgather_hi(const unsigned char __vccm * addr, vNx2int_t offsets, pvNx4 pred);	vvld.fa.b.p
vNx4uchar_t vgather_hi(const unsigned char __vccm * addr, vNx2int_t offsets, vNx4uchar_t dflt, pvNx4 pred);	vvld.fa.b.p

The following `vscatter` functions store values from vector `val` to `addr + offsets`. For 8-bit element types, there are additional variants called `vscatter_lo` and `vscatter_hi` which store only the elements in the upper or lower half of the input vector. If the `pred` parameter is specified, `val[i]` will be stored only if `pred[i]` is true.

Intrinsic Function	Mapped Instruction
void vscatter(vNint_t val, int __vccm * addr, vNint_t offsets, pvN pred);	vvst.fa.w.p

Intrinsic Function	Mapped Instruction
void vscatter(vNint_t val, int __vccm * addr, vNint_t offsets);	vvst.fa.w
void vscatter(vNuint_t val, unsigned int __vccm * addr, vNint_t offsets, pvN pred);	vvst.fa.w.p
void vscatter(vNuint_t val, unsigned int __vccm * addr, vNint_t offsets);	vvst.fa.w
void vscatter(vNfloat_t val, float __vccm * addr, vNint_t offsets, pvN pred);	vvst.fa.w.p
void vscatter(vNfloat_t val, float __vccm * addr, vNint_t offsets);	vvst.fa.w
void vscatter(vNx2short_t val, short __vccm * addr, vNx2int_t offsets, pvNx2 pred);	vvst.fa.h.p
void vscatter(vNx2short_t val, short __vccm * addr, vNx2int_t offsets);	vvst.fa.h
void vscatter(vNx2ushort_t val, unsigned short __vccm * addr, vNx2int_t offsets, pvNx2 pred);	vvst.fa.h.p
void vscatter(vNx2ushort_t val, unsigned short __vccm * addr, vNx2int_t offsets);	vvst.fa.h
void vscatter(vNx2half_t val, _Float16 __vccm * addr, vNx2int_t offsets, pvNx2 pred);	vvst.fa.h.p
void vscatter(vNx2half_t val, _Float16 __vccm * addr, vNx2int_t offsets);	vvst.fa.h
void vscatter(vNx4char_t val, signed char __vccm * addr, vNx4int_t offsets, pvNx4 pred);	vvst.fa.b.p
void vscatter(vNx4char_t val, signed char __vccm * addr, vNx4int_t offsets);	vvst.fa.b
void vscatter(vNx4uchar_t val, unsigned char __vccm * addr, vNx4int_t offsets, pvNx4 pred);	vvst.fa.b.p
void vscatter(vNx4uchar_t val, unsigned char __vccm * addr, vNx4int_t offsets);	vvst.fa.b
void vscatter_lo(vNx4char_t val, signed char __vccm * addr, vNx2int_t offsets, pvNx4 pred);	vvst.fa.b.p
void vscatter_lo(vNx4char_t val, signed char __vccm * addr, vNx2int_t offsets);	vvst.fa.b
void vscatter_lo(vNx4uchar_t val, unsigned char __vccm * addr, vNx2int_t offsets, pvNx4 pred);	vvst.fa.b.p
void vscatter_lo(vNx4uchar_t val, unsigned char __vccm * addr, vNx2int_t offsets);	vvst.fa.b
void vscatter_hi(vNx4char_t val, signed char __vccm * addr, vNx2int_t offsets, pvNx4 pred);	vvst.fa.b.p

Intrinsic Function	Mapped Instruction
void vscatter_hi(vNx4char_t val, signed char __vccm * addr, vNx2int_t offsets);	vvst.fa.b
void vscatter_hi(vNx4uchar_t val, unsigned char __vccm * addr, vNx2int_t offsets, pvNx4 pred);	vvst.fa.b.p
void vscatter_hi(vNx4uchar_t val, unsigned char __vccm * addr, vNx2int_t offsets);	vvst.fa.b

Vector support reinterpreting intrinsics

The compiler supports intrinsics for reinterpreting a vector variable as a different vector data type. The header file `arc_vector.h` declares several functions as macro wrappers over these intrinsics.

The original type and the resulting type of the vector variable after reinterpretation must have the same size and number of elements.

Intrinsic Function	Description
vNint_t reinterpret_as_vNint_t(vNuint_t x);	Reinterpret a vector of type vNuint_t as vNint_t
vNint_t reinterpret_as_vNint_t(vNfloat_t x);	Reinterpret a vector of type vNfloat_t as vNint_t
vNint_t reinterpret_as_vNint_t(vNint_t x);	Reinterpret a vector of type vNint_t as vNint_t
vNuint_t reinterpret_as_vNuint_t(vNint_t x);	Reinterpret a vector of type vNint_t as vNuint_t
vNuint_t reinterpret_as_vNuint_t(vNfloat_t x);	Reinterpret a vector of type vNfloat_t as vNuint_t
vNuint_t reinterpret_as_vNuint_t(vNuint_t x);	Reinterpret a vector of type vNuint_t as vNuint_t
vNfloat_t reinterpret_as_vNfloat_t(vNint_t x);	Reinterpret a vector of type vNint_t as vNfloat_t
vNfloat_t reinterpret_as_vNfloat_t(vNuint_t x);	Reinterpret a vector of type vNuint_t as vNfloat_t
vNfloat_t reinterpret_as_vNfloat_t(vNfloat_t x);	Reinterpret a vector of type vNfloat_t as vNfloat_t
vNx2int_t reinterpret_as_vNx2int_t(vNx2uint_t x);	Reinterpret a vector of type vNx2uint_t as vNx2int_t
vNx2int_t reinterpret_as_vNx2int_t(vNx2float_t x);	Reinterpret a vector of type vNx2float_t as vNx2int_t
vNx2int_t reinterpret_as_vNx2int_t(vNx2int_t x);	Reinterpret a vector of type vNx2int_t as vNx2int_t
vNx2uint_t reinterpret_as_vNx2uint_t(vNx2int_t x);	Reinterpret a vector of type vNx2int_t as vNx2uint_t
vNx2uint_t reinterpret_as_vNx2uint_t(vNx2float_t x);	Reinterpret a vector of type vNx2float_t as vNx2uint_t
vNx2uint_t reinterpret_as_vNx2uint_t(vNx2uint_t x);	Reinterpret a vector of type vNx2uint_t as vNx2uint_t
vNx2float_t reinterpret_as_vNx2float_t(vNx2int_t x);	Reinterpret a vector of type vNx2int_t as vNx2float_t
vNx2float_t reinterpret_as_vNx2float_t(vNx2uint_t x);	Reinterpret a vector of type vNx2uint_t as vNx2float_t
vNx2float_t reinterpret_as_vNx2float_t(vNx2float_t x);	Reinterpret a vector of type vNx2float_t as vNx2float_t
vNx4int_t reinterpret_as_vNx4int_t(vNx4uint_t x);	Reinterpret a vector of type vNx4uint_t as vNx4int_t
vNx4int_t reinterpret_as_vNx4int_t(vNx4float_t x);	Reinterpret a vector of type vNx4float_t as vNx4int_t

Intrinsic Function	Description
vNx4int_t reinterpret_as_vNx4int_t(vNx4int_t x);	Reinterpret a vector of type vNx4int_t as vNx4int_t
vNx4uint_t reinterpret_as_vNx4uint_t(vNx4int_t x);	Reinterpret a vector of type vNx4int_t as vNx4uint_t
vNx4uint_t reinterpret_as_vNx4uint_t(vNx4float_t x);	Reinterpret a vector of type vNx4float_t as vNx4uint_t
vNx4uint_t reinterpret_as_vNx4uint_t(vNx4uint_t x);	Reinterpret a vector of type vNx4uint_t as vNx4uint_t
vNx4float_t reinterpret_as_vNx4float_t(vNx4int_t x);	Reinterpret a vector of type vNx4int_t as vNx4float_t
vNx4float_t reinterpret_as_vNx4float_t(vNx4uint_t x);	Reinterpret a vector of type vNx4uint_t as vNx4float_t
vNx4float_t reinterpret_as_vNx4float_t(vNx4float_t x);	Reinterpret a vector of type vNx4float_t as vNx4float_t
vNx2short_t reinterpret_as_vNx2short_t(vNx2ushort_t x);	Reinterpret a vector of type vNx2ushort_t as vNx2short_t
vNx2short_t reinterpret_as_vNx2short_t(vNx2half_t x);	Reinterpret a vector of type vNx2half_t as vNx2short_t
vNx2short_t reinterpret_as_vNx2short_t(vNx2short_t x);	Reinterpret a vector of type vNx2short_t as vNx2short_t
vNx2ushort_t reinterpret_as_vNx2ushort_t(vNx2short_t x);	Reinterpret a vector of type vNx2short_t as vNx2ushort_t
vNx2ushort_t reinterpret_as_vNx2ushort_t(vNx2half_t x);	Reinterpret a vector of type vNx2half_t as vNx2ushort_t
vNx2ushort_t reinterpret_as_vNx2ushort_t(vNx2ushort_t x);	Reinterpret a vector of type vNx2ushort_t as vNx2ushort_t
vNx2half_t reinterpret_as_vNx2half_t(vNx2short_t x);	Reinterpret a vector of type vNx2short_t as vNx2half_t
vNx2half_t reinterpret_as_vNx2half_t(vNx2ushort_t x);	Reinterpret a vector of type vNx2ushort_t as vNx2half_t
vNx2half_t reinterpret_as_vNx2half_t(vNx2half_t x);	Reinterpret a vector of type vNx2half_t as vNx2half_t
vNx4short_t reinterpret_as_vNx4short_t(vNx4ushort_t x);	Reinterpret a vector of type vNx4ushort_t as vNx4short_t
vNx4short_t reinterpret_as_vNx4short_t(vNx4half_t x);	Reinterpret a vector of type vNx4half_t as vNx4short_t
vNx4short_t reinterpret_as_vNx4short_t(vNx4short_t x);	Reinterpret a vector of type vNx4short_t as vNx4short_t
vNx4ushort_t reinterpret_as_vNx4ushort_t(vNx4short_t x);	Reinterpret a vector of type vNx4short_t as vNx4ushort_t
vNx4ushort_t reinterpret_as_vNx4ushort_t(vNx4half_t x);	Reinterpret a vector of type vNx4half_t as vNx4ushort_t
vNx4ushort_t reinterpret_as_vNx4ushort_t(vNx4ushort_t x);	Reinterpret a vector of type vNx4ushort_t as vNx4ushort_t

Intrinsic Function	Description
vNx4half_t reinterpret_as_vNx4half_t(vNx4short_t x);	Reinterpret a vector of type vNx4short_t as vNx4half_t
vNx4half_t reinterpret_as_vNx4half_t(vNx4ushort_t x);	Reinterpret a vector of type vNx4ushort_t as vNx4half_t
vNx4half_t reinterpret_as_vNx4half_t(vNx4half_t x);	Reinterpret a vector of type vNx4half_t as vNx4half_t
vNx4char_t reinterpret_as_vNx4char_t(vNx4uchar_t x);	Reinterpret a vector of type vNx4uchar_t as vNx4char_t
vNx4char_t reinterpret_as_vNx4char_t(vNx4char_t x);	Reinterpret a vector of type vNx4char_t as vNx4char_t
vNx4uchar_t reinterpret_as_vNx4uchar_t(vNx4char_t x);	Reinterpret a vector of type vNx4char_t as vNx4uchar_t
vNx4uchar_t reinterpret_as_vNx4uchar_t(vNx4uchar_t x);	Reinterpret a vector of type vNx4uchar_t as vNx4uchar_t

Vector support intrinsics with predicates

Intrinsic Function	Mapped Instruction	Remarks
vNuint_t vvscan_excl_add(pvN_t p);	vscan_excl_add.w.p	
vNx2ushort_t vvscan_excl_add(pvNx2_t p);	vscan_excl_add.h.p	
vNx4uchar_t vvscan_excl_add(pvNx4_t p);	vscan_excl_add.b.p	
vNint_t vvsel(pvN_t p, vNint_t x, vNint_t y);	vvsel.w.p	
vNuint_t vvsel(pvN_t p, vNuint_t x, vNuint_t y);	vvsel.w.p	
vNfloat_t vvsel (pvN_t p, vNfloat_t x, vNfloat_t y);	vvsel.w.p	
vNx2short_t vvsel(pvNx2_t p, vNx2short_t x, vNx2short_t y);	vvsel.h.p	
vNx2ushort_t vvsel(pvNx2_t p, vNx2ushort_t x, vNx2ushort_t y);	vvsel.h.p	
vNx2half_t vvsel (pvNx2_t p, vNx2half_t x, vNx2half_t y);	vvsel.h.p	
vNx4char_t vvsel(pvNx4_t p, vNx4char_t x, vNx4char_t y);	vvsel.b.p	
vNx4uchar_t vvsel(pvNx4_t p, vNx4uchar_t x, vNx4uchar_t y);	vvsel.b.p	
vNint_t vvshfl_p(pvN_t p, vNint_t t, vNint_t x, vNint_t y);	vvshfl.w.p	
vNuint_t vvshfl_p(pvN_t p, vNuint_t t, vNuint_t x, vNuint_t y);	vvshfl.w.p	
vNx2short_t vvshfl_p(pvNx2_t p, vNx2short_t t, vNx2short_t x, vNx2short_t y);	vvshfl.h.p	

Intrinsic Function	Mapped Instruction	Remarks
vNx2ushort_t vvshfl_p(pvNx2_t p, vNx2ushort_t t, vNx2ushort_t x, vNx2ushort_t y);	vvshfl.h.p	
vNx4char_t vvshfl_p(pvNx4_t p, vNx4char_t t, vNx4char_t x, vNx4char_t y);	vvshfl.b.p	
vNx4uchar_t vvshfl_p(pvNx4_t p, vNx4uchar_t t, vNx4uchar_t x, vNx4uchar_t y);	vvshfl.b.p	
vNint_t vvshfleven(pvN_t p, vNint_t x);	vvshfleven.w.p	Non-predicated result vector items are undefined.
vNuint_t vvshfleven(pvN_t p, vNuint_t x);	vvshfleven.w.p	Non-predicated result vector items are undefined.
vNfloat_t vvshfleven(pvN_t p, vNfloat_t x);	vvshfleven.w.p	Non-predicated result vector items are undefined.
vNx2short_t vvshfleven(pvNx2_t p, vNx2short_t x);	vvshfleven.h.p	Non-predicated result vector items are undefined.
vNx2ushort_t vvshfleven(pvNx2_t p, vNx2ushort_t x);	vvshfleven.h.p	Non-predicated result vector items are undefined.
vNx2half_t vvshfleven(pvNx2_t p, vNx2half_t x);	vvshfleven.h.p	Non-predicated result vector items are undefined.
vNx4char_t vvshfleven(pvNx4_t p, vNx4char_t x);	vvshfleven.b.p	Non-predicated result vector items are undefined.
vNx4uchar_t vvshfleven(pvNx4_t p, vNx4uchar_t x);	vvshfleven.b.p	Non-predicated result vector items are undefined.
vNint_t vvshflodd(pvN_t p, vNint_t x);	vvshflodd.w.p	Non-predicated result vector items are undefined.
vNuint_t vvshflodd(pvN_t p, vNuint_t x);	vvshflodd.w.p	Non-predicated result vector items are undefined.
vNfloat_t vvshflodd(pvN_t p, vNfloat_t x);	vvshflodd.w.p	Non-predicated result vector items are undefined.
vNx2short_t vvshflodd(pvNx2_t p, vNx2short_t x);	vvshflodd.h.p	Non-predicated result vector items are undefined.
vNx2ushort_t vvshflodd(pvNx2_t p, vNx2ushort_t x);	vvshflodd.h.p	Non-predicated result vector items are undefined.
vNx2half_t vvshflodd(pvNx2_t p, vNx2half_t x);	vvshflodd.h.p	Non-predicated result vector items are undefined.
vNx4char_t vvshflodd(pvNx4_t p, vNx4char_t x);	vvshflodd.b.p	Non-predicated result vector items are undefined.
vNx4uchar_t vvshflodd(pvNx4_t p, vNx4uchar_t x);	vvshflodd.b.p	Non-predicated result vector items are undefined.
pvN_t vvp2reduce(pvN_t x);	vvp2reduce.w	

Intrinsic Function	Mapped Instruction	Remarks
pvNx2_t vvp2reduce(pvNx2_t x);	vvp2reduce.h	
pvNx4_t vvp2reduce(pvNx4_t x);	vvp2reduce.b	
pvN_t vvp4pack(pvN_t x);	vvp4pack.w	
pvNx2_t vvp4pack(pvNx2_t x);	vvp4pack.h	
pvNx4_t vvp4pack(pvNx4_t x);	vvp4pack.b	
pvN_t vvp4reduce(pvN_t x);	vvp4reduce.w	
pvNx2_t vvp4reduce(pvNx2_t x);	vvp4reduce.h	
pvNx4_t vvp4reduce(pvNx4_t x);	vvp4reduce.b	
pvN_t vvpand(pvN_t x, pvN_t y);	vvpand	
pvNx2_t vvpand(pvNx2_t x, pvNx2_t y);	vvpand	
pvNx4_t vvpand(pvNx4_t x, pvNx4_t y);	vvpand	
pvN_t vvpclrallN(void);	vvpclrall	
pvNx2_t vvpclrallNx2(void);	vvpclrall	
pvNx4_t vvpclrallNx4(void);	vvpclrall	
pvN_t vvpconcat(pvN_t x, pvN_t y);	vvpconcat	
pvNx2_t vvpconcat(pvNx2_t x, pvNx2_t y);	vvpconcat	
pvNx4_t vvpconcat(pvNx4_t x, pvNx4_t y);	vvpconcat	
pvN_t vvpeven(pvN_t x);	vvpeven.w	
pvNx2_t vvpeven(pvNx2_t x);	vvpeven.h	
pvNx4_t vvpeven(pvNx4_t x);	vvpeven.b	
pvN_t vvphi(pvN_t x);	vvphi	
pvNx2_t vvphi(pvNx2_t x);	vvphi	
pvNx4_t vvphi(pvNx4_t x);	vvphi	
pvN_t vvplo(pvN_t x);	vvplo	
pvNx2_t vvplo(pvNx2_t x);	vvplo	
pvNx4_t vvplo(pvNx4_t x);	vvplo	
pvN_t vvpnot(pvN_t x);	vvpnot	
pvNx2_t vvpnot(pvNx2_t x);	vvpnot	
pvNx4_t vvpnot(pvNx4_t x);	vvpnot	
int vvpnumset(pvN_t x);	vvpnumset.w	
int vvpnumset(pvNx2_t x);	vvpnumset.h	
int vvpnumset(pvNx4_t x);	vvpnumset.b	
pvN_t vvpor(pvN_t x, pvN_t y);	vvpor	
pvNx2_t vvpor(pvNx2_t x, pvNx2_t y);	vvpor	

Intrinsic Function	Mapped Instruction	Remarks
pvNx4_t vvpor(pvNx4_t x, pvNx4_t y);	vvpor	
pvN_t vvpsetallN(void);	vvpsetall	
pvNx2_t vvpsetallNx2(void);	vvpsetall	
pvNx4_t vvpsetallNx4(void);	vvpsetall	
pvNx2_t vvpwiden2(pvNx2_t x);	vvpwiden2.h	
pvNx4_t vvpwiden2(pvNx4_t x);	vvpwiden2.b	
pvNx4_t vvpwiden4(pvNx4_t x);	vvpwiden4.b	
pvN_t vvpxor(pvN_t x, pvN_t y);	vvpor	
pvNx2_t vvpxor(pvNx2_t x, pvNx2_t y);	vvpxor	
pvNx4_t vvpxor(pvNx4_t x, pvNx4_t y);	vvpxor	
pvNx4_t vvpinit_b (long long init);	vvpinit.b	Uses 2 vvpinit.b intructions when __VDSP_VEC_WIDTH__ is 512 and 1 otherwise.
pvNx4_t vvpinit_0_b (int init, pvNx4_t p);	vvpinit.b	Writes to low part of the given predicate.
pvNx4_t vvpinit_1_b (int init, pvNx4_t p);	vvpinit.b	Writes to high part of the given predicate if __VDSP_VEC_WIDTH__ is 512. Otherwise, it returns the given predicate unchanged.
pvNx2_t vvpinit_h (int init);	vvpinit.h	
pvN_t vvpinit_w (int init);	vvpinit.w	
int vvpmovps_0 (pvNx4_t p);	vvpmovps	Returns the first 32 bits of the given predicate as an integer.
int vvpmovps_0 (pvNx2_t p);	vvpmovps	Returns the first 32 bits of the given predicate as an integer.
int vvpmovps_0 (pvN_t p);	vvpmovps	Returns the first 32 bits of the given predicate as an integer.
int vvpmovps_1 (pvNx4_t p);	vvpmovps	Returns the last 32 bits of the given predicate as an integer.
int vvpmovps_1 (pvNx2_t p);	vvpmovps	Returns the last 32 bits of the given predicate as an integer.
int vvpmovps_1 (pvN_t p);	vvpmovps	Returns the last 32 bits of the given predicate as an integer.
int vvpall (pvNx4_t p);	vvpnumset.b	Returns 1 if all (_VDSP_NUM_8BIT_LANES) bits of the given predicate are set.

Intrinsic Function	Mapped Instruction	Remarks
int vvpall (pvNx2_t p);	vvpnumset.h	Returns 1 if all (_VDSP_NUM_16BIT_LANES) bits of the given predicate are set.
int vvpall (pvN_t p) ;	vvpnumset.w	Returns 1 if all (_VDSP_NUM_32BIT_LANES) bits of the given predicate are set.
int vvpany (pvNx4_t p);	vvpnumset.b	Returns 1 if any bit of the given predicate is set.
int vvpany (pvNx2_t p);	vvpnumset.h	Returns 1 if any bit of the given predicate is set.
int vvpany (pvN_t p);	vvpnumset.w	Returns 1 if any bit of the given predicate is set.

Vector support floating-point intrinsics

Intrinsic Function	Mapped Instruction
vNfloat_t vvfadd(vNfloat_t x, vNfloat_t y);	vvfadd.w
vNx2half_t vvfadd(vNx2half_t x, vNx2half_t y);	vvfadd.h
float vvffadd(vNfloat_t x);	vvffadd.w
_Float16 vvffadd(vNx2half_t x);	vvffadd.h
vNfloat_t vvf2add(vNfloat_t x);	vvf2add.w
vNx2half_t vvf2add(vNx2half_t x);	vvf2add.h
vNfloat_t vvfcopysign(vNfloat_t x, vNfloat_t y);	vvfcopysign.w
vNx2half_t vvfcopysign(vNx2half_t x, vNx2half_t y);	vvfcopysign.h
vNfloat_t vvfcos(vNfloat_t x);	vvfcos.w
vNx2half_t vvfcos(vNx2half_t x);	vvfcos.h
vNfloat_t vvfexp(vNfloat_t x);	vvfexp.w
vNx2half_t vvfexp(vNx2half_t x);	vvfexp.h
vNfloat_t vvfexp2(vNfloat_t x);	vvfexp2.w
vNx2half_t vvfexp2(vNx2half_t x);	vvfexp2.h
vNfloat_t vvflog2(vNfloat_t x);	vvflog2.w
vNx2half_t vvflog2(vNx2half_t x);	vvflog2.h
vNfloat_t vvfmadd(vNfloat_t x, vNfloat_t y, vNfloat_t z);	vvfmadd.w
vNx2half_t vvfmadd(vNx2half_t x, vNx2half_t y, vNx2half_t z);	vvfmadd.h
vNfloat_t vvfmsub(vNfloat_t x, vNfloat_t y, vNfloat_t z);	vvfmsub.w

Intrinsic Function	Mapped Instruction
vNx2half_t vvfmsub(vNx2half_t x, vNx2half_t y, vNx2half_t z);	vvfmsub.h
vNfloat_t vvfmax(vNfloat_t x, vNfloat_t y);	vvfmax.w
vNx2half_t vvfmax(vNx2half_t x, vNx2half_t y);	vvfmax.h
vNfloat_t vvf2max(vNfloat_t x);	vvf2max.w
vNx2half_t vvf2max(vNx2half_t x);	vvf2max.h
vNfloat_t vvfmin(vNfloat_t x, vNfloat_t y);	vvfmin.w
vNx2half_t vvfmin(vNx2half_t x, vNx2half_t y);	vvfmin.h
vNfloat_t vvf2min(vNfloat_t x);	vvf2min.w
vNx2half_t vvf2min(vNx2half_t x);	vvf2min.h
vNfloat_t vvfmul(vNfloat_t x, vNfloat_t y);	vvfmul.w
vNx2half_t vvfmul(vNx2half_t x, vNx2half_t y);	vvfmul.h
vNfloat_t vvfrdiv(vNfloat_t x, vNfloat_t y);	vvfrdiv.w
vNx2half_t vvfrdiv(vNx2half_t x, vNx2half_t y);	vvfrdiv.h
vNfloat_t vvfrsqrt(vNfloat_t x);	vvfrsqrt.w
vNx2half_t vvfrsqrt(vNx2half_t x);	vvfrsqrt.h
vNfloat_t vvfsin(vNfloat_t x);	vvfsin.w
vNx2half_t vvfsin(vNx2half_t x);	vvfsin.h
vNfloat_t vvfsqrt(vNfloat_t x);	vvfsqrt.w
vNx2half_t vvfsqrt(vNx2half_t x);	vvfsqrt.h
vNfloat_t vvfsub(vNfloat_t x, vNfloat_t y);	vvfsub.w
vNx2half_t vvfsub(vNx2half_t x, vNx2half_t y);	vvfsub.h

Vector support miscellaneous intrinsics

Intrinsic Function	Mapped Instruction
vNint_t vvabs(vNint_t x);	vvabs.w
vNx2short_t vvabs(vNx2short_t x);	vvabs.h
vNx4char_t vvabs(vNx4char_t x);	vvabs.b
vNint_t vvabs_sat(vNint_t x);	vvabs.sat.w
vNx2short_t vvabs_sat(vNx2short_t x);	vvabs.sat.h
vNx4char_t vvabs_sat(vNx4char_t x);	vvabs.sat.b
vNint_t vvcabs(vNint_t x);	vvcabs.w
vNx2short_t vvcabs(vNx2short_t x);	vvcabs.h
vNx4char_t vvcabs(vNx4char_t x);	vvcabs.b
vNint_t vvadd_sat(vNint_t x, vNint_t y);	vvadd.sat.w

Intrinsic Function	Mapped Instruction
vNx2short_t vvadd_sat(vNx2short_t x, vNx2short_t y);	vvadd.sat.h
vNx4char_t vvadd_sat(vNx4char_t x, vNx4char_t y);	vvadd.sat.b
vNuint_t vvadd_satu(vNuint_t x, vNuint_t y);	vvadd.satu.w
vNx2ushort_t vvadd_satu(vNx2ushort_t x, vNx2ushort_t y);	vvadd.satu.h
vNx4uchar_t vvadd_satu(vNx4uchar_t x, vNx4uchar_t y);	vvadd.satu.b
vNint_t vvcadd(vNint_t x, vNint_t y, vNint_t z);	vvcadd.w
vNx2short_t vvcadd(vNx2short_t x, vNx2short_t y, vNx2short_t z);	vvcadd.h
vNx4char_t vvcadd(vNx4char_t x, vNx4char_t y, vNx4char_t z);	vvcadd.b
vNuint_t vvcadd_uu(vNuint_t x, vNuint_t y, vNuint_t z);	vvcadd.uu.w
vNx2ushort_t vvcadd_uu(vNx2ushort_t x, vNx2ushort_t y, vNx2ushort_t z);	vvcadd.uu.h
vNx4uchar_t vvcadd_uu(vNx4uchar_t x, vNx4uchar_t y, vNx4uchar_t z);	vvcadd.uu.b
vNint_t vvcadd1(vNint_t x, vNint_t y, vNint_t z);	vvcadd1.w
vNx2short_t vvcadd1(vNx2short_t x, vNx2short_t y, vNx2short_t z);	vvcadd1.h
vNx4char_t vvcadd1(vNx4char_t x, vNx4char_t y, vNx4char_t z);	vvcadd1.b
vNuint_t vvcadd1_uu(vNuint_t x, vNuint_t y, vNuint_t z);	vvcadd1.uu.w
vNx2ushort_t vvcadd1_uu(vNx2ushort_t x, vNx2ushort_t y, vNx2ushort_t z);	vvcadd1.uu.h
vNx4uchar_t vvcadd1_uu(vNx4uchar_t x, vNx4uchar_t y, vNx4uchar_t z);	vvcadd1.uu.b
vNint_t vvcadd1b(vNint_t x, vNint_t y, vNint_t z);	vvcadd1b.w
vNx2short_t vvcadd1b(vNx2short_t x, vNx2short_t y, vNx2short_t z);	vvcadd1b.h
vNx4char_t vvcadd1b(vNx4char_t x, vNx4char_t y, vNx4char_t z);	vvcadd1b.b
vNuint_t vvcadd1b_uu(vNuint_t x, vNuint_t y, vNuint_t z);	vvcadd1b.uu.w
vNx2ushort_t vvcadd1b_uu(vNx2ushort_t x, vNx2ushort_t y, vNx2ushort_t z);	vvcadd1b.uu.h

Intrinsic Function	Mapped Instruction
vNx4uchar_t vvcadd1b_uu(vNx4uchar_t x, vNx4uchar_t y, vNx4uchar_t z);	vvcadd1b.uu.b
vNint_t vvcaddsub(vNint_t x, vNint_t y, vNint_t z);	vvcaddsub.w
vNx2short_t vvcaddsub(vNx2short_t x, vNx2short_t y, vNx2short_t z);	vvcaddsub.h
vNx4char_t vvcaddsub(vNx4char_t x, vNx4char_t y, vNx4char_t z);	vvcaddsub.b
vNuint_t vvcaddsub_uu(vNuint_t x, vNuint_t y, vNuint_t z);	vvcaddsub.uu.w
vNx2ushort_t vvcaddsub_uu(vNx2ushort_t x, vNx2ushort_t y, vNx2ushort_t z);	vvcaddsub.uu.h
vNx4uchar_t vvcaddsub_uu(vNx4uchar_t x, vNx4uchar_t y, vNx4uchar_t z);	vvcaddsub.uu.b
vNint_t vvcaddsub1(vNint_t x, vNint_t y, vNint_t z);	vvcaddsub1.w
vNx2short_t vvcaddsub1(vNx2short_t x, vNx2short_t y, vNx2short_t z);	vvcaddsub1.h
vNx4char_t vvcaddsub1(vNx4char_t x, vNx4char_t y, vNx4char_t z);	vvcaddsub1.b
vNuint_t vvcaddsub1_uu(vNuint_t x, vNuint_t y, vNuint_t z);	vvcaddsub1.uu.w
vNx2ushort_t vvcaddsub1_uu(vNx2ushort_t x, vNx2ushort_t y, vNx2ushort_t z);	vvcaddsub1.uu.h
vNx4uchar_t vvcaddsub1_uu(vNx4uchar_t x, vNx4uchar_t y, vNx4uchar_t z);	vvcaddsub1.uu.b
vNint_t vvcaddsub1b(vNint_t x, vNint_t y, vNint_t z);	vvcaddsub1b.w
vNx2short_t vvcaddsub1b(vNx2short_t x, vNx2short_t y, vNx2short_t z);	vvcaddsub1b.h
vNx4char_t vvcaddsub1b(vNx4char_t x, vNx4char_t y, vNx4char_t z);	vvcaddsub1b.b
vNuint_t vvcaddsub1b_uu(vNuint_t x, vNuint_t y, vNuint_t z);	vvcaddsub1b.uu.w
vNx2ushort_t vvcaddsub1b_uu(vNx2ushort_t x, vNx2ushort_t y, vNx2ushort_t z);	vvcaddsub1b.uu.h
vNx4uchar_t vvcaddsub1b_uu(vNx4uchar_t x, vNx4uchar_t y, vNx4uchar_t z);	vvcaddsub1b.uu.b
vNint_t vvcadd_init(vNint_t x, vNint_t y);	vvcadd.init.w
vNx2short_t vvcadd_init(vNx2short_t x, vNx2short_t y);	vvcadd.init.h

Intrinsic Function	Mapped Instruction
vNx4char_t vvcadd_init(vNx4char_t x, vNx4char_t y);	vvcadd.init.b
vNuint_t vvcadd_init_uu(vNuint_t x, vNuint_t y);	vvcadd.init.uu.w
vNx2ushort_t vvcadd_init_uu(vNx2ushort_t x, vNx2ushort_t y);	vvcadd.init.uu.h
vNx4uchar_t vvcadd_init_uu(vNx4uchar_t x, vNx4uchar_t y);	vvcadd.init.uu.b
vNint_t vvc2add(vNint_t x);	vvc2add.w
vNx2short_t vvc2add(vNx2short_t x);	vvc2add.h
vNx4char_t vvc2add(vNx4char_t x);	vvc2add.b
vNint_t vvc4add(vNint_t x);	vvc4add.w
vNx2short_t vvc4add(vNx2short_t x);	vvc4add.h
vNx4char_t vvc4add(vNx4char_t x);	vvc4add.b
vNint_t vvavg(vNint_t x, vNint_t y);	vvavg.w
vNx2short_t vvavg(vNx2short_t x, vNx2short_t y);	vvavg.h
vNx4char_t vvavg(vNx4char_t x, vNx4char_t y);	vvavg.b
vNuint_t vvavg_uu(vNuint_t x, vNuint_t y);	vvavg.uu.w
vNx2ushort_t vvavg_uu(vNx2ushort_t x, vNx2ushort_t y);	vvavg.uu.h
vNx4uchar_t vvavg_uu(vNx4uchar_t x, vNx4uchar_t y);	vvavg.uu.b
vNint_t vvavgr(vNint_t x, vNint_t y);	vvavgr.w
vNx2short_t vvavgr(vNx2short_t x, vNx2short_t y);	vvavgr.h
vNx4char_t vvavgr(vNx4char_t x, vNx4char_t y);	vvavgr.b
vNuint_t vvavgr_uu(vNuint_t x, vNuint_t y);	vvavgr.uu.w
vNx2ushort_t vvavgr_uu(vNx2ushort_t x, vNx2ushort_t y);	vvavgr.uu.h
vNx4uchar_t vvavgr_uu(vNx4uchar_t x, vNx4uchar_t y);	vvavgr.uu.b
vNint_t vvbclr(vNint_t x, vNint_t y);	vvbclr.w
vNuint_t vvbclr(vNuint_t x, vNuint_t y);	vvbclr.w
vNx2short_t vvbclr(vNx2short_t x, vNx2short_t y);	vvbclr.h
vNx2ushort_t vvbclr(vNx2ushort_t x, vNx2ushort_t y);	vvbclr.h
vNx4char_t vvbclr(vNx4char_t x, vNx4char_t y);	vvbclr.b

Intrinsic Function	Mapped Instruction
vNx4uchar_t vvbclr(vNx4uchar_t x, vNx4uchar_t y);	vvbclr.b
vNint_t vvbic(vNint_t x, vNint_t y);	vvbic.w
vNuint_t vvbic(vNuint_t x, vNuint_t y);	vvbic.w
vNx2short_t vvbic(vNx2short_t x, vNx2short_t y);	vvbic.h
vNx2ushort_t vvbic(vNx2ushort_t x, vNx2ushort_t y);	vvbic.h
vNx4char_t vvbic(vNx4char_t x, vNx4char_t y);	vvbic.b
vNx4uchar_t vvbic(vNx4uchar_t x, vNx4uchar_t y);	vvbic.b
vNint_t vvbmsk(vNint_t x, vNint_t y);	vvbmsk.w
vNuint_t vvbmsk(vNuint_t x, vNuint_t y);	vvbmsk.w
vNx2short_t vvbmsk(vNx2short_t x, vNx2short_t y);	vvbmsk.h
vNx2ushort_t vvbmsk(vNx2ushort_t x, vNx2ushort_t y);	vvbmsk.h
vNx4char_t vvbmsk(vNx4char_t x, vNx4char_t y);	vvbmsk.b
vNx4uchar_t vvbmsk(vNx4uchar_t x, vNx4uchar_t y);	vvbmsk.b
vNint_t vvbmskn(vNint_t x, vNint_t y);	vvbmskn.w
vNuint_t vvbmskn(vNuint_t x, vNuint_t y);	vvbmskn.w
vNx2short_t vvbmskn(vNx2short_t x, vNx2short_t y);	vvbmskn.h
vNx2ushort_t vvbmskn(vNx2ushort_t x, vNx2ushort_t y);	vvbmskn.h
vNx4char_t vvbmskn(vNx4char_t x, vNx4char_t y);	vvbmskn.b
vNx4uchar_t vvbmskn(vNx4uchar_t x, vNx4uchar_t y);	vvbmskn.b
vNint_t vvbset(vNint_t x, vNint_t y);	vvbset.w
vNuint_t vvbset(vNuint_t x, vNuint_t y);	vvbset.w
vNx2short_t vvbset(vNx2short_t x, vNx2short_t y);	vvbset.h
vNx2ushort_t vvbset(vNx2ushort_t x, vNx2ushort_t y);	vvbset.h
vNx4char_t vvbset(vNx4char_t x, vNx4char_t y);	vvbset.b
vNx4uchar_t vvbset(vNx4uchar_t x, vNx4uchar_t y);	vvbset.b

Intrinsic Function	Mapped Instruction
vNint_t vvbxor(vNint_t x, vNint_t y);	vvbxor.w
vNuint_t vvbxor(vNuint_t x, vNuint_t y);	vvbxor.w
vNx2short_t vvbxor(vNx2short_t x, vNx2short_t y);	vvbxor.h
vNx2ushort_t vvbxor(vNx2ushort_t x, vNx2ushort_t y);	vvbxor.h
vNx4char_t vvbxor(vNx4char_t x, vNx4char_t y);	vvbxor.b
vNx4uchar_t vvbxor(vNx4uchar_t x, vNx4uchar_t y);	vvbxor.b
vNint_t vvcdiv(vNint_t x, vNint_t y);	vvcdiv.w
vNx2short_t vvcdiv(vNx2short_t x, vNx2short_t y);	vvcdiv.h
vNx4char_t vvcdiv(vNx4char_t x, vNx4char_t y);	vvcdiv.b
vNuint_t vvcdiv_uu(vNuint_t x, vNuint_t y);	vvcdiv.uu.w
vNx2ushort_t vvcdiv_uu(vNx2ushort_t x, vNx2ushort_t y);	vvcdiv.uu.h
vNx4uchar_t vvcdiv_uu(vNx4uchar_t x, vNx4uchar_t y);	vvcdiv.uu.b
vNint_t vvceven(vNint_t x);	vvceven.w
vNx2short_t vvceven(vNx2short_t x);	vvceven.h
vNx4char_t vvceven(vNx4char_t x);	vvceven.b
vNint_t vvci_w(void);	vvci.w
vNx2short_t vvci_h(void);	vvci.h
vNx4char_t vvci_b(void);	vvci.b
vNint_t vvci_stride_w(int x);	vvci.stridw.w
vNx2short_t vvci_stride_h(int x);	vvci.stride.h
vNint_t vvcmac_hi(vNint_t x, vNint_t y, vNint_t z);	vvcmac.hi.w
vNuint_t vvcmac_hi_uu(vNuint_t x, vNuint_t y, vNuint_t z);	vvcmac.hi.uu.w
vNint_t vvcmac_lo(vNint_t x, vNint_t y, vNint_t z);	vvcmac.lo.w
vNuint_t vvcmac_lo_uu(vNuint_t x, vNuint_t y, vNuint_t z);	vvcmac.lo.uu.w
vNx2int_t vvcmac_h(vNx2int_t, vNx2short_t x, vNx2short_t y);	vvcmac.h
vNx2uint_t vvcmac_uu_h(vNx2uint_t, vNx2ushort_t x, vNx2ushort_t y);	vvcmac.uu.h
vNx4short_t vvcmac_b(vNx4short_t, vNx4char_t x, vNx4char_t y);	vvcmac.b

Intrinsic Function	Mapped Instruction
vNx4ushort_t vvcmac_uu_b(vNx4ushort_t, vNx4uchar_t x, vNx4uchar_t y);	vvcmac.uu.b
vNint_t vvmax(vNint_t x, vNint_t y);	vvmax.w
vNx2short_t vvmax(vNx2short_t x, vNx2short_t y);	vvmax.h
vNx4char_t vvmax(vNx4char_t x, vNx4char_t y);	vvmax.b
vNuint_t vvmax_uu(vNuint_t x, vNuint_t y);	vvmax.uu.w
vNx2ushort_t vvmax_uu(vNx2ushort_t x, vNx2ushort_t y);	vvmax.uu.h
vNx4uchar_t vvmax_uu(vNx4uchar_t x, vNx4uchar_t y);	vvmax.uu.b
vNint_t vvcmax(vNint_t x, vNint_t y);	vvcmax.w
vNx2short_t vvcmax(vNx2short_t x, vNx2short_t y);	vvcmax.h
vNx4char_t vvcmax(vNx4char_t x, vNx4char_t y);	vvcmax.b
vNuint_t vvcmax_uu(vNuint_t x, vNuint_t y);	vvcmax.uu.w
vNx2ushort_t vvcmax_uu(vNx2ushort_t x, vNx2ushort_t y);	vvcmax.uu.h
vNx4uchar_t vvcmax_uu(vNx4uchar_t x, vNx4uchar_t y);	vvcmax.uu.b
vNint_t vvc2max(vNint_t x);	vvc2max.w
vNx2short_t vvc2max(vNx2short_t x);	vvc2max.h
vNx4char_t vvc2max(vNx4char_t x);	vvc2max.b
vNuint_t vvc2max_uu(vNuint_t x);	vvc2max.uu.w
vNx2ushort_t vvc2max_uu(vNx2ushort_t x);	vvc2max.uu.h
vNx4uchar_t vvc2max_uu(vNx4uchar_t x);	vvc2max.uu.b
vNint_t vvmin(vNint_t x, vNint_t y);	vvmin.w
vNx2short_t vvmin(vNx2short_t x, vNx2short_t y);	vvmin.h
vNx4char_t vvmin(vNx4char_t x, vNx4char_t y);	vvmin.b
vNuint_t vvmin_uu(vNuint_t x, vNuint_t y);	vvmin.uu.w
vNx2ushort_t vvmin_uu(vNx2ushort_t x, vNx2ushort_t y) ;	vvmin.uu.h
vNx4uchar_t vvmin_uu(vNx4uchar_t x, vNx4uchar_t y);	vvmin.uu.b
vNint_t vvcmin(vNint_t x, vNint_t y);	vvcmin.w
vNx2short_t vvcmin(vNx2short_t x, vNx2short_t y);	vvcmin.h
vNx4char_t vvcmin(vNx4char_t x, vNx4char_t y);	vvcmin.b

Intrinsic Function	Mapped Instruction
vNuint_t vvcmin_uu(vNuint_t x, vNuint_t y);	vvcmin.uu.w
vNx2ushort_t vvcmin_uu(vNx2ushort_t x, vNx2ushort_t y);	vvcmin.uu.h
vNx4uchar_t vvcmin_uu(vNx4uchar_t x, vNx4uchar_t y);	vvcmin.uu.b
vNint_t vvc2min(vNint_t x);	vvc2min.w
vNx2short_t vvc2min(vNx2short_t x);	vvc2min.h
vNx4char_t vvc2min(vNx4char_t x);	vvc2min.b
vNuint_t vvc2min_uu(vNuint_t x);	vvc2min.uu.w
vNx2ushort_t vvc2min_uu(vNx2ushort_t x);	vvc2min.uu.h
vNx4uchar_t vvc2min_uu(vNx4uchar_t x);	vvc2min.uu.b
vNint_t vvmpy_hi(vNint_t x, vNint_t y);	vvmpy.hi.w
vNx2short_t vvmpy_hi(vNx2short_t x, vNx2short_t y);	vvmpy.hi.h
vNx4char_t vvmpy_hi(vNx4char_t x, vNx4char_t y);	vvmpy.hi.b
vNuint_t vvmpy_hi_uu(vNuint_t x, vNuint_t y);	vvmpy.hi.uu.w
vNx2ushort_t vvmpy_hi_uu(vNx2ushort_t x, vNx2ushort_t y);	vvmpy.hi.uu.h
vNx4uchar_t vvmpy_hi_uu(vNx4uchar_t x, vNx4uchar_t y);	vvmpy.hi.uu.b
vNint_t vvcmpy_hi(vNint_t x, vNint_t y);	vvcmpy.hi.w
vNuint_t vvcmpy_hi_uu(vNuint_t x, vNuint_t y);	vvcmpy.hi.uu.w
vNint_t vvcmpy_lo(vNint_t x, vNint_t y);	vvcmpy.lo
vNuint_t vvcmpy_lo_uu(vNuint_t x, vNuint_t y);	vvcmpy.lo.uu
vNx2int_t vvcmpy_h(vNx2short_t x, vNx2short_t y);	vvcmpy.h
vNx2uint_t vvcmpy_uu_h(vNx2ushort_t x, vNx2ushort_t y);	vvcmpy.uu.h
vNx4short_t vvcmpy_b(vNx4char_t x, vNx4char_t y);	vvcmpy.b
vNx4ushort_t vvcmpy_uu_b(vNx4uchar_t x, vNx4uchar_t y);	vvcmpy.uu.b
vNint_t vvcmsub_hi(vNint_t x, vNint_t y, vNint_t z);	vvcmsub.hi.w
vNuint_t vvcmsub_hi_uu(vNuint_t x, vNuint_t y, vNuint_t z);	vvcmsub.hi.uu.w

Intrinsic Function	Mapped Instruction
vNint_t vvcmsub_lo(vNint_t x, vNint_t y, vNint_t z);	vvcmsub.lo.w
vNuint_t vvcmsub_lo_uu(vNuint_t x, vNuint_t y, vNuint_t z);	vvcmsub.lo.uu.w
vNx2int_t vvcmsub_h(vNx2int_t, vNx2short_t x, vNx2short_t y);	vvcmsub.h
vNx2uint_t vvcmsub_uu_h(vNx2uint_t, vNx2ushort_t x, vNx2ushort_t y);	vvcmsub.uu.h
vNx4short_t vvcmsub_b(vNx4short_t, vNx4char_t x, vNx4char_t y);	vvcmsub.b
vNx4ushort_t vvcmsub_uu_b(vNx4ushort_t, vNx4uchar_t x, vNx4uchar_t y);	vvcmsub.uu.b
vNint_t vvneg_sat(vNint_t x);	vvneg.sat.w
vNx2short_t vvneg_sat(vNx2short_t x);	vvneg.sat.h
vNx4char_t vvneg_sat(vNx4char_t x);	vvneg.sat.b
vNint_t vvcneg(vNint_t x);	vvcneg.w
vNx2short_t vvcneg(vNx2short_t x);	vvcneg.h
vNx4char_t vvcneg(vNx4char_t x);	vvcneg.b
vNint_t vvcnorm(vNint_t x);	vvcnorm.w
vNx2short_t vvcnorm(vNx2short_t x);	vvcnorm.h
vNx4char_t vvcnorm(vNx4char_t x);	vvcnorm.b
vNuint_t vvnorm(vNuint_t x);	vvnorm.w
vNx2ushort_t vvnorm(vNx2ushort_t x);	vvnorm.h
vNx4uchar_t vvnorm(vNx4uchar_t x);	vvnorm.b
vNint_t vvnorm(vNint_t x);	vvnorm.w
vNx2short_t vvnorm(vNx2short_t x);	vvnorm.h
vNx4char_t vvnorm(vNx4char_t x);	vvnorm.b
vNint_t vvnumlz(vNint_t x);	vvnumlz.w
vNuint_t vvnumlz(vNuint_t x);	vvnumlz.w
vNx2short_t vvnumlz(vNx2short_t x);	vvnumlz.h
vNx2ushort_t vvnumlz(vNx2ushort_t x);	vvnumlz.h
vNx4char_t vvnumlz(vNx4char_t x);	vvnumlz.b
vNx4uchar_t vvnumlz(vNx4uchar_t x);	vvnumlz.b
vNint_t vvnumset(vNint_t x);	vvnumset.w
vNuint_t vvnumset(vNuint_t x);	vvnumset.w
vNx2ushort_t vvnumset(vNx2ushort_t x);	vvnumset.h
vNx2short_t vvnumset(vNx2short_t x);	vvnumset.h

Intrinsic Function	Mapped Instruction
vNx4char_t vvnumset(vNx4char_t x);	vvnumset.b
vNx4uchar_t vvnumset(vNx4uchar_t x);	vvnumset.b
vNint_t vvnumtz(vNint_t x);	vvnumtz.w
vNuint_t vvnumtz(vNuint_t x);	vvnumtz.w
vNx2short_t vvnumtz(vNx2short_t x);	vvnumtz.h
vNx2ushort_t vvnumtz(vNx2ushort_t x);	vvnumtz.h
vNx4char_t vvnumtz(vNx4char_t x);	vvnumtz.b
vNx4uchar_t vvnumtz(vNx4uchar_t x);	vvnumtz.b
vNint_t vvc4pack(vNint_t x);	vvc4pack.w
vNx2short_t vvc4pack(vNx2short_t x);	vvc4pack.h
vNx4char_t vvc4pack(vNx4char_t x);	vvc4pack.b
vNint_t vvrol(vNint_t x);	vvrol.w
vNuint_t vvrol(vNuint_t x);	vvrol.w
vNx2short_t vvrol(vNx2short_t x);	vvrol.h
vNx2ushort_t vvrol(vNx2ushort_t x);	vvrol.h
vNx4char_t vvrol(vNx4char_t x);	vvrol.b
vNx4uchar_t vvrol(vNx4uchar_t x);	vvrol.b
vNint_t vvrol8(vNint_t x);	vvrol8.w
vNuint_t vvrol8(vNuint_t x);	vvrol8.w
vNx2short_t vvrol8(vNx2short_t x);	vvrol8.h
vNx2ushort_t vvrol8(vNx2ushort_t x);	vvrol8.h
vNuint_t vvror(vNuint_t x);	vvror.w
vNx2ushort_t vvror(vNx2ushort_t x);	vvror.h
vNx4uchar_t vvror(vNx4uchar_t x);	vvror.b
vNint_t vvror(vNint_t x);	vvror.w
vNx2short_t vvror(vNx2short_t x);	vvror.h
vNx4char_t vvror(vNx4char_t x);	vvror.b
vNint_t vvror8(vNint_t x);	vvror8.w
vNuint_t vvror8(vNuint_t x);	vvror8.w
vNx2short_t vvror8(vNx2short_t x);	vvror8.w
vNx2ushort_t vvror8(vNx2ushort_t x);	vvror8.h
vNint_t vvrorm(vNint_t x, vNint_t y);	vvrorm.w
vNuint_t vvrorm(vNuint_t x, vNuint_t y);	vvrorm.w
vNx2short_t vvrorm(vNx2short_t x, vNx2short_t y);	vvrorm.h

Intrinsic Function	Mapped Instruction
vNx2ushort_t vvrorm(vNx2ushort_t x, vNx2ushort_t y);	vvrorm.h
vNx4char_t vvrorm(vNx4char_t x, vNx4char_t y);	vvrorm.b
vNx4uchar_t vvrorm(vNx4uchar_t x, vNx4uchar_t y);	vvrorm.b
vNint_t vvcsad(vNint_t x, vNint_t y, vNint_t z);	vvcsad.w
vNx2short_t vvcsad(vNx2short_t x, vNx2short_t y, vNx2short_t z);	vvcsad.h
vNx4char_t vvcsad(vNx4char_t x, vNx4char_t y, vNx4char_t z);	vvcsad.b
vNuint_t vvcsad_uu(vNuint_t x, vNuint_t y, vNuint_t z);	vvcsad.uu.w
vNx2ushort_t vvcsad_uu(vNx2ushort_t x, vNx2ushort_t y, vNx2ushort_t z);	vvcsad.uu.h
vNx4uchar_t vvcsad_uu(vNx4uchar_t x, vNx4uchar_t y, vNx4uchar_t z);	vvcsad.uu.b
vNint_t vvcsad_init(vNint_t x, vNint_t y);	vvcsad.init.w
vNx2short_t vvcsad_init(vNx2short_t x, vNx2short_t y);	vvcsad.init.h
vNx4char_t vvcsad_init(vNx4char_t x, vNx4char_t y);	vvcsad.init.b
vNuint_t vvcsad_init_uu(vNuint_t x, vNuint_t y);	vvcsad.init.uu.w
vNx2ushort_t vvcsad_init_uu(vNx2ushort_t x, vNx2ushort_t y);	vvcsad.init.uu.h
vNx4uchar_t vvcsad_init_uu(vNx4uchar_t x, vNx4uchar_t y);	vvcsad.init.uu.b
vNint_t vvsexb(vNint_t x);	vvsexb.w
vNuint_t vvsexb(vNuint_t x);	vvsexb.w
vNx2short_t vvsexb(vNx2short_t x);	vvsexb.h
vNx2ushort_t vvsexb(vNx2ushort_t x);	vvsexb.h
vNint_t vvsexh(vNint_t x);	vvsexh.w
vNuint_t vvsexh(vNuint_t x);	vvsexh.w
vNint_t vvshfl(vNint_t x, vNint_t y);	vvshfl.w
vNuint_t vvshfl(vNuint_t x, vNuint_t y);	vvshfl.w
vNx2short_t vvshfl(vNx2short_t x, vNx2short_t y);	vvshfl.h
vNx2ushort_t vvshfl(vNx2ushort_t x, vNx2ushort_t y);	vvshfl.h
vNx4char_t vvshfl(vNx4char_t x, vNx4char_t y);	vvshfl.b

Intrinsic Function	Mapped Instruction
vNx4uchar_t vvshfl(vNx4uchar_t x, vNx4uchar_t y);	vvshfl.b
vNfloat_t vvshfl_float(vNfloat_t x, vNint_t y);	vvshfl.w
vNfloat_t vvshfl_float(vNfloat_t x, vNuint_t y);	vvshfl.w
vNx2half_t vvshfl_half(vNx2half_t x, vNx2short_t y);	vvshfl.h
vNx2half_t vvshfl_half(vNx2half_t x, vNx2ushort_t y);	vvshfl.h
vNint_t vvshft(vNint_t x);	vvshft.w
vNx2short_t vvshft(vNx2short_t x);	vvshft.h
vNx4char_t vvshft(vNx4char_t x);	vvshft.b
vNuint_t vvshft(vNuint_t x);	vvshft.w
vNx2ushort_t vvshft(vNx2ushort_t x);	vvshft.h
vNx4uchar_t vvshft(vNx4uchar_t x);	vvshft.b
vNfloat_t vvshft(vNfloat_t y);	vvshft.w
vNx2half_t vvshft(vNx2half_t y);	vvshft.h
vNuint_t vvcsqrt(vNuint_t x);	vvshft.b
vNint_t vvshiftleft(vNint_t x, unsigned y);	vvshiftleft.w
vNfloat_t vvshiftleft(vNfloat_t x, unsigned y);	vvshiftleft.w
vNx2half_t vvshiftleft(vNx2half_t x, unsigned y);	vvshiftleft.h
vNx2short_t vvshiftleft(vNx2short_t x, unsigned y);	vvshiftleft.h
vNx4char_t vvshiftleft(vNx4char_t x, unsigned y);	vvshiftleft.b
vNuint_t vvshiftleft(vNuint_t x, unsigned y);	vvshiftleft.w
vNx2ushort_t vvshiftleft(vNx2ushort_t x, unsigned y);	vvshiftleft.h
vNx4uchar_t vvshiftleft(vNx4uchar_t x, unsigned y);	vvshiftleft.b
vNint_t vvshiftright(vNint_t x, unsigned y);	vvshiftright.w
vNfloat_t vvshiftright(vNfloat_t x, unsigned y);	vvshiftright.w
vNx2half_t vvshiftright(vNx2half_t x, unsigned y);	vvshiftright.h
vNx2short_t vvshiftright(vNx2short_t x, unsigned y);	vvshiftright.h
vNx4char_t vvshiftright(vNx4char_t x, unsigned y);	vvshiftright.b
vNuint_t vvshiftright(vNuint_t x, unsigned y);	vvshiftright.w

Intrinsic Function	Mapped Instruction
vNx2ushort_t vvshiftright(vNx2ushort_t x, unsigned y);	vvshiftright.h
vNx4uchar_t vvshiftright(vNx4uchar_t x, unsigned y);	vvshiftright.b
vNint_t vvslm_sat(vNint_t x, vNint_t y);	vvslm.sat.w
vNx2short_t vvslm_sat(vNx2short_t x, vNx2short_t y);	vvslm.sat.h
vNx4char_t vvslm_sat(vNx4char_t x, vNx4char_t y);	vvslm.sat.b
vNuint_t vvslm_satu(vNuint_t x, vNuint_t y);	vvslm.satu.w
vNx2ushort_t vvslm_satu(vNx2ushort_t x, vNx2ushort_t y);	vvslm.satu.h
vNx4uchar_t vvslm_satu(vNx4uchar_t x, vNx4uchar_t y);	vvslm.satu.b
vNint_t vvcslm(vNint_t x, vNint_t y);	vvcslm.w
vNx2short_t vvcslm(vNx2short_t x, vNx2short_t y);	vvcslm.h
vNx4char_t vvcslm(vNx4char_t x, vNx4char_t y);	vvcslm.b
vNint_t vvcasrm(vNint_t x, vNint_t y);	vvcasrm.w
vNx2short_t vvcasrm(vNx2short_t x, vNx2short_t y);	vvcasrm.h
vNx4char_t vvcasrm(vNx4char_t x, vNx4char_t y);	vvcasrm.b
vNuint_t vvclsrm(vNuint_t x, vNuint_t y);	vvclsrm.w
vNx2ushort_t vvclsrm(vNx2ushort_t x, vNx2ushort_t y);	vvclsrm.h
vNx4uchar_t vvclsrm(vNx4uchar_t x, vNx4uchar_t y);	vvclsrm.b
vNuint_t vvsqrt(vNuint_t x);	vvsqrt.w
vNx2ushort_t vvsqrt(vNx2ushort_t x);	vvsqrt.h
vNx4uchar_t vvsqrt(vNx4uchar_t x);	vvsqrt.b
vNx2ushort_t vvcsqrt(vNx2ushort_t x);	vvcsqrt.w
vNx4uchar_t vvcsqrt(vNx4uchar_t x);	vvcsqrt.h
vNint_t vvsub_sat(vNint_t x, vNint_t y);	vvsub.sat.w
vNx2short_t vvsub_sat(vNx2short_t x, vNx2short_t y);	vvsub.sat.h
vNx4char_t vvsub_sat(vNx4char_t x, vNx4char_t y);	vvsub.sat.b

Intrinsic Function	Mapped Instruction
vNuint_t vvsub_satu(vNuint_t x, vNuint_t y);	vvsub.satu.w
vNx2ushort_t vvsub_satu(vNx2ushort_t x, vNx2ushort_t y);	vvsub.satu.h
vNx4uchar_t vvsub_satu(vNx4uchar_t x, vNx4uchar_t y);	vvsub.satu.b
vNint_t vvcsub(vNint_t x, vNint_t y, vNint_t z);	vvcsub.w
vNx2short_t vvcsub(vNx2short_t x, vNx2short_t y, vNx2short_t z);	vvcsub.h
vNx4char_t vvcsub(vNx4char_t x, vNx4char_t y, vNx4char_t z);	vvcsub.b
vNuint_t vvcsub_uu(vNuint_t x, vNuint_t y, vNuint_t z);	vvcsub.uu.w
vNx2ushort_t vvcsub_uu(vNx2ushort_t x, vNx2ushort_t y, vNx2ushort_t z);	vvcsub.uu.h
vNx4uchar_t vvcsub_uu(vNx4uchar_t x, vNx4uchar_t y, vNx4uchar_t z);	vvcsub.uu.b
vNint_t vvcsub1(vNint_t x, vNint_t y, vNint_t z);	vvcsub1.w
vNx2short_t vvcsub1(vNx2short_t x, vNx2short_t y, vNx2short_t z);	vvcsub1.h
vNx4char_t vvcsub1(vNx4char_t x, vNx4char_t y, vNx4char_t z);	vvcsub1.b
vNuint_t vvcsub1_uu(vNuint_t x, vNuint_t y, vNuint_t z);	vvcsub1.uu.w
vNx2ushort_t vvcsub1_uu(vNx2ushort_t x, vNx2ushort_t y, vNx2ushort_t z);	vvcsub1.uu.h
vNx4uchar_t vvcsub1_uu(vNx4uchar_t x, vNx4uchar_t y, vNx4uchar_t z);	vvcsub1.uu.b
vNint_t vvcsub1b(vNint_t x, vNint_t y, vNint_t z);	vvcsub1b.w
vNx2short_t vvcsub1b(vNx2short_t x, vNx2short_t y, vNx2short_t z);	vvcsub1b.h
vNx4char_t vvcsub1b(vNx4char_t x, vNx4char_t y, vNx4char_t z);	vvcsub1b.b
vNuint_t vvcsub1b_uu(vNuint_t x, vNuint_t y, vNuint_t z);	vvcsub1b.uu.w
vNx2ushort_t vvcsub1b_uu(vNx2ushort_t x, vNx2ushort_t y, vNx2ushort_t z);	vvcsub1b.uu.h
vNx4uchar_t vvcsub1b_uu(vNx4uchar_t x, vNx4uchar_t y, vNx4uchar_t z);	vvcsub1b.uu.b
vNint_t vvcsubsub1(vNint_t x, vNint_t y, vNint_t z);	vvcsubsub1.w

Intrinsic Function	Mapped Instruction
vNx2short_t vvcsubsub1(vNx2short_t x, vNx2short_t y, vNx2short_t z);	vvcsubsub1.h
vNx4char_t vvcsubsub1(vNx4char_t x, vNx4char_t y, vNx4char_t z);	vvcsubsub1.b
vNuint_t vvcsubsub1_uu(vNuint_t x, vNuint_t y, vNuint_t z);	vvcsubsub1.uu.w
vNx2ushort_t vvcsubsub1_uu(vNx2ushort_t x, vNx2ushort_t y, vNx2ushort_t z);	vvcsubsub1.uu.h
vNx4uchar_t vvcsubsub1_uu(vNx4uchar_t x, vNx4uchar_t y, vNx4uchar_t z);	vvcsubsub1.uu.b
vNint_t vvcsubsub1b(vNint_t x, vNint_t y, vNint_t z);	vvcsubsub1b.w
vNx2short_t vvcsubsub1b(vNx2short_t x, vNx2short_t y, vNx2short_t z);	vvcsubsub1b.h
vNx4char_t vvcsubsub1b(vNx4char_t x, vNx4char_t y, vNx4char_t z);	vvcsubsub1b.b
vNuint_t vvcsubsub1b_uu(vNuint_t x, vNuint_t y, vNuint_t z);	vvcsubsub1b.uu.w
vNx2ushort_t vvcsubsub1b_uu(vNx2ushort_t x, vNx2ushort_t y, vNx2ushort_t z);	vvcsubsub1b.uu.h
vNx4uchar_t vvcsubsub1b_uu(vNx4uchar_t x, vNx4uchar_t y, vNx4uchar_t z);	vvcsubsub1b.uu.b
vNint_t vvcsub_init(vNint_t x, vNint_t y);	vvcsub.init.w
vNx2short_t vvcsub_init(vNx2short_t x, vNx2short_t y);	vvcsub.init.h
vNx4char_t vvcsub_init(vNx4char_t x, vNx4char_t y);	vvcsub.init.b
vNuint_t vvcsub_init_uu(vNuint_t x, vNuint_t y);	vvcsub.init.uu.w
vNx2ushort_t vvcsub_init_uu(vNx2ushort_t x, vNx2ushort_t y);	vvcsub.init.uu.h
vNx4uchar_t vvcsub_init_uu(vNx4uchar_t x, vNx4uchar_t y);	vvcsub.init.uu.b
vNint_t vvsl (vNint_t x);	vvsl.w
vNx2short_t vvsl (vNx2short_t x);	vvsl.h
vNx4char_t vvsl (vNx4char_t x);	vvsl.b
vNuint_t vvsl (vNuint_t x);	vvsl.w
vNx2ushort_t vvsl (vNx2ushort_t x);	vvsl.h
vNx4uchar_t vvsl (vNx4uchar_t x);	vvsl.b
vNint_t vvsl8 (vNint_t x);	vvsl8.w

Intrinsic Function	Mapped Instruction
vNx2short_t vvsl8 (vNx2short_t x);	vvsl8.h
vNuint_t vvsl8 (vNuint_t x);	vvsl8.w
vNx2ushort_t vvsl8 (vNx2ushort_t x);	vvsl8.h
vNint_t vvsl16 (vNint_t x);	vvsl16.w
vNuint_t vvsl16 (vNuint_t x);	vvsl16.w
vNint_t vvslm (vNint_t x, vNint_t y);	vvslm.w
vNx2short_t vvslm (vNx2short_t x, vNx2short_t y);	vvslm.h
vNx4char_t vvslm (vNx4char_t x, vNx4char_t y);	vvslm.b
vNuint_t vvslm (vNuint_t x, vNuint_t y);	vvslm.w
vNx2ushort_t vvslm (vNx2ushort_t x, vNx2ushort_t y);	vvslm.h
vNx4uchar_t vvslm (vNx4uchar_t x, vNx4uchar_t y);	vvslm.b
vNint_t vvasrm (vNint_t x, vNint_t y);	vvasrm.w
vNx2short_t vvasrm (vNx2short_t x, vNx2short_t y);	vvasrm.h
vNx4char_t vvasrm (vNx4char_t x, vNx4char_t y);	vvasrm.b
vNint_t vvasr (vNint_t x);	vvasr.w
vNx2short_t vvasr (vNx2short_t x);	vvasr.h
vNx4char_t vvasr (vNx4char_t x);	vvasr.b
vNint_t vvasr8 (vNint_t x);	vvasr8.w
vNx2short_t vvasr8 (vNx2short_t x);	vvasr8.h
vNint_t vvasr16 (vNint_t x);	vvasr16.w
vNuint_t vvlsrm (vNuint_t x, vNuint_t y);	vvlsrm.w
vNx2ushort_t vvlsrm (vNx2ushort_t x, vNx2ushort_t y);	vvlsrm.h
vNx4uchar_t vvlsrm (vNx4uchar_t x, vNx4uchar_t y);	vvlsrm.b
vNuint_t vvlsr (vNuint_t x);	vvlsr.w
vNx2ushort_t vvlsr (vNx2ushort_t x);	vvlsr.h
vNx4uchar_t vvlsr (vNx4uchar_t x);	vvlsr.b
vNuint_t vvlsr8 (vNuint_t x);	vvlsr8.w
vNx2ushort_t vvlsr8 (vNx2ushort_t x);	vvlsr8.h
vNuint_t vvlsr16 (vNuint_t x);	vvlsr16.w
vNint_t vvadd (vNint_t x, vNint_t y);	vvadd.w

Intrinsic Function	Mapped Instruction
vNx2short_t vvadd (vNx2short_t x, vNx2short_t y);	vvadd.h
vNx4char_t vvadd (vNx4char_t x, vNx4char_t y);	vvadd.b
vNuint_t vvadd (vNuint_t x, vNuint_t y);	vvadd.w
vNx2ushort_t vvadd (vNx2ushort_t x, vNx2ushort_t y);	vvadd.h
vNx4uchar_t vvadd (vNx4uchar_t x, vNx4uchar_t y);	vvadd.b
vNint_t vvadd1 (vNint_t x, vNint_t y);	vvadd1.w
vNx2short_t vvadd1 (vNx2short_t x, vNx2short_t y);	vvadd1.h
vNx4char_t vvadd1 (vNx4char_t x, vNx4char_t y);	vvadd1.b
vNuint_t vvadd1 (vNuint_t x, vNuint_t y);	vvadd1.w
vNx2ushort_t vvadd1 (vNx2ushort_t x, vNx2ushort_t y);	vvadd1.h
vNx4uchar_t vvadd1 (vNx4uchar_t x, vNx4uchar_t y);	vvadd1.b
vNint_t vvadd2 (vNint_t x, vNint_t y);	vvadd2.w
vNx2short_t vvadd2 (vNx2short_t x, vNx2short_t y);	vvadd2.h
vNx4char_t vvadd2 (vNx4char_t x, vNx4char_t y);	vvadd2.b
vNuint_t vvadd2 (vNuint_t x, vNuint_t y);	vvadd2.w
vNx2ushort_t vvadd2 (vNx2ushort_t x, vNx2ushort_t y);	vvadd2.h
vNx4uchar_t vvadd2 (vNx4uchar_t x, vNx4uchar_t y);	vvadd2.b
vNint_t vvadd3 (vNint_t x, vNint_t y);	vvadd3.w
vNx2short_t vvadd3 (vNx2short_t x, vNx2short_t y);	vvadd3.h
vNx4char_t vvadd3 (vNx4char_t x, vNx4char_t y);	vvadd3.b
vNuint_t vvadd3 (vNuint_t x, vNuint_t y);	vvadd3.w
vNx2ushort_t vvadd3 (vNx2ushort_t x, vNx2ushort_t y);	vvadd3.h
vNx4uchar_t vvadd3 (vNx4uchar_t x, vNx4uchar_t y);	vvadd3.b
vNint_t vvsub (vNint_t x, vNint_t y);	vvsub.w
vNx2short_t vvsub (vNx2short_t x, vNx2short_t y);	vvsub.h

Intrinsic Function	Mapped Instruction
vNx4char_t vvsub (vNx4char_t x, vNx4char_t y);	vvsub.b
vNuint_t vvsub (vNuint_t x, vNuint_t y);	vvsub.w
vNx2ushort_t vvsub (vNx2ushort_t x, vNx2ushort_t y);	vvsub.h
vNx4uchar_t vvsub (vNx4uchar_t x, vNx4uchar_t y);	vvsub.b
vNint_t vvsub1 (vNint_t x, vNint_t y);	vvsub1.w
vNx2short_t vvsub1 (vNx2short_t x, vNx2short_t y);	vvsub1.h
vNx4char_t vvsub1 (vNx4char_t x, vNx4char_t y);	vvsub1.b
vNuint_t vvsub1 (vNuint_t x, vNuint_t y);	vvsub1.w
vNx2ushort_t vvsub1 (vNx2ushort_t x, vNx2ushort_t y);	vvsub1.h
vNx4uchar_t vvsub1 (vNx4uchar_t x, vNx4uchar_t y);	vvsub1.b
vNint_t vvsub2 (vNint_t x, vNint_t y);	vvsub2.w
vNx2short_t vvsub2 (vNx2short_t x, vNx2short_t y);	vvsub2.h
vNx4char_t vvsub2 (vNx4char_t x, vNx4char_t y);	vvsub2.b
vNuint_t vvsub2 (vNuint_t x, vNuint_t y);	vvsub2.w
vNx2ushort_t vvsub2 (vNx2ushort_t x, vNx2ushort_t y);	vvsub2.h
vNx4uchar_t vvsub2 (vNx4uchar_t x, vNx4uchar_t y);	vvsub2.b
vNint_t vvsub3 (vNint_t x, vNint_t y);	vvsub3.w
vNx2short_t vvsub3 (vNx2short_t x, vNx2short_t y);	vvsub3.h
vNx4char_t vvsub3 (vNx4char_t x, vNx4char_t y);	vvsub3.b
vNuint_t vvsub3 (vNuint_t x, vNuint_t y);	vvsub3.w
vNx2ushort_t vvsub3 (vNx2ushort_t x, vNx2ushort_t y);	vvsub3.h
vNx4uchar_t vvsub3 (vNx4uchar_t x, vNx4uchar_t y);	vvsub3.b
vNint_t vvextb (vNint_t x);	vvextb.w
vNx2short_t vvextb (vNx2short_t x);	vvextb.h
vNuint_t vvextb (vNuint_t x);	vvextb.w
vNx2ushort_t vvextb (vNx2ushort_t x);	vvextb.h
vNint_t vvexth (vNint_t x);	vvexth.w

Intrinsic Function	Mapped Instruction
vNuint_t vvexth (vNuint_t x);	vvexth.w
vNint_t vvnot (vNint_t x);	vvnot.w
vNx2short_t vvnot (vNx2short_t x);	vvnot.h
vNx4char_t vvnot (vNx4char_t x);	vvnot.b
vNuint_t vvnot (vNuint_t x);	vvnot.w
vNx2ushort_t vvnot (vNx2ushort_t x);	vvnot.h
vNx4uchar_t vvnot (vNx4uchar_t x);	vvnot.b
vNint_t vvmpy (vNint_t x, vNint_t y);	vvmpy.w
vNx2short_t vvmpy (vNx2short_t x, vNx2short_t y);	vvmpy.h
vNx4char_t vvmpy (vNx4char_t x, vNx4char_t y);	vvmpy.b
vNuint_t vvmpy (vNuint_t x, vNuint_t y);	vvmpy.w
vNx2ushort_t vvmpy (vNx2ushort_t x, vNx2ushort_t y);	vvmpy.h
vNx4uchar_t vvmpy (vNx4uchar_t x, vNx4uchar_t y);	vvmpy.b
vNint_t vvand (vNint_t x, vNint_t y);	vvand.w
vNx2short_t vvand(vNx2short_t x, vNx2short_t y);	vvand.h
vNx4char_t vvand(vNx4char_t x, vNx4char_t y);	vvand.b
vNuint_t vvand(vNuint_t x, vNuint_t y);	vvand.w
vNx2ushort_t vvand(vNx2ushort_t x, vNx2ushort_t y);	vvand.h
vNx4uchar_t vvand(vNx4uchar_t x, vNx4uchar_t y);	vvand.b
vNint_t vvor (vNint_t x, vNint_t y);	vvor.w
vNx2short_t vvor (vNx2short_t x, vNx2short_t y);	vvor.h
vNx4char_t vvor (vNx4char_t x, vNx4char_t y);	vvor.b
vNuint_t vvor (vNuint_t x, vNuint_t y);	vvor.w
vNx2ushort_t vvor (vNx2ushort_t x, vNx2ushort_t y);	vvor.h
vNx4uchar_t vvor (vNx4uchar_t x, vNx4uchar_t y);	vvor.b
vNint_t vvxor (vNint_t x, vNint_t y);	vvxor.w
vNx2short_t vvxor (vNx2short_t x, vNx2short_t y);	vvxor.h
vNx4char_t vvxor (vNx4char_t x, vNx4char_t y);	vvxor.b
vNuint_t vvxor (vNuint_t x, vNuint_t y);	vvxor.w

Intrinsic Function	Mapped Instruction
vNx2ushort_t vvxor (vNx2ushort_t x, vNx2ushort_t y);	vvxor.h
vNx4uchar_t vvxor (vNx4uchar_t x, vNx4uchar_t y);	vvxor.b
vNint_t vvdiv (vNint_t x, vNint_t y);	vvdiv.w
vNx2short_t vvdiv (vNx2short_t x, vNx2short_t y);	vvdiv.h
vNx4char_t vvdiv (vNx4char_t x, vNx4char_t y);	vvdiv.b
vNuint_t vvdiv_uu (vNuint_t x, vNuint_t y);	vvdiv.uu.w
vNx2ushort_t vvdiv_uu (vNx2ushort_t x, vNx2ushort_t y);	vvdiv.uu.h
vNx4uchar_t vvdiv_uu (vNx4uchar_t x, vNx4uchar_t y);	vvdiv.uu.b
vNint_t vvrem (vNint_t x, vNint_t y);	vvrem.w
vNx2short_t vvrem (vNx2short_t x, vNx2short_t y);	vvrem.h
vNx4char_t vvrem (vNx4char_t x, vNx4char_t y);	vvrem.b
vNuint_t vvrem_uu (vNuint_t x, vNuint_t y);	vvrem.uu.w
vNx2ushort_t vvrem_uu (vNx2ushort_t x, vNx2ushort_t y);	vvrem.uu.h
vNx4uchar_t vvrem_uu (vNx4uchar_t x, vNx4uchar_t y);	vvrem.uu.b

1.10.4.6. Built-in Library Functions

The intrinsics described in this section are built-in versions of a number of library functions. The name of the intrinsic is the name of the library function prefixed by `__builtin_`.

The standard header file where the function is declared, defines a macro that maps the function to the intrinsic. You can force a call to the library function by removing the macro definition with an `#undef` directive, or by enclosing the name of the function in parenthesis, e.g.: (`strcpy`)(*dest*, *src*);

Intrinsic Function	Description
void * __builtin_memchr(const void * *s*, int *c*, __size_t *n*);	Checks the first *n* bytes of memory pointed to by *s*, on the occurrence of character c. A call to the library function `memchr()` is generated.
int __builtin_memcmp(const void * *s1*, const void * *s2*, __size_t *n*);	Compares the first *n* bytes of memory pointed to by *s1*, with the memory pointed to by *s2*. A call to the library function `memcmp()` is generated.
void * __builtin_memcpy(void * *dest*, void * *src*, __size_t *n*);	Copy the first *n* bytes of memory pointed to by *src*, to the memory pointed to by *dest*. A call to the library function `memcpy()` is generated.

Intrinsic Function	Description
void * __builtin_memmove(void * *dest*, void * *src*, __size_t *n*);	Copy the first *n* bytes of memory pointed to by *src*, to the memory pointed to by *dest*, but overlapping strings are handled correctly. A call to the library function memmove() is generated.
void * __builtin_memset(void * *s*, int *c*, __size_t *n*);	Fill the first *n* bytes of the memory pointed to by *s* with the value *c*. A call to the library function memset() is generated.
char * __builtin_strcat(char * *dest*, const char * *src*);	Appends a copy of string pointed to by *src*, to the end of string pointed to by *dest*. A call to the library function strcat() is generated.
char * __builtin_strchr(const char * *s*, int *c*);	Returns a pointer to the first occurrence of character *c* in the string pointed to by *s* or the null pointer if not found. A call to the library function strchr() is generated.
int __builtin_strcmp(const char * *s1*, const char * *s2*);	Compares the string pointed to by *s1* to the string pointed to by *s2*. A call to the library function strcmp() is generated.
char * __builtin_strcpy(char * *dest*, const char * *src*);	Copy the string pointed to by *src*, including the terminating null byte, to the buffer pointed to by *dest*. A call to the library function strcpy() is generated.
__size_t __builtin_strcspn(const char * *s*, const char * *set*);	Searches the string pointed to by *s* for a sequence of characters not specified in the string pointed to by *set*. A call to the library function strcspn() is generated.
__size_t __builtin_strlen(const char * *s*);	Computes the length of the string pointed to by *s*. A call to the library function strlen() is generated.
char * __builtin_strncat(char * *dest*, const char * *src*, __size_t *n*);	Appends not more than *n* characters of string pointed to by *src*, to the end of string pointed to by *dest*. A call to the library function strncat() is generated.
int __builtin_strncmp(const char * *s1*, const char * *s2*, __size_t *n*);	Compares the first *n* characters of string pointed to by *s1* to the first *n* characters of string pointed to by *s2*. A call to the library function strncmp() is generated.
char * __builtin_strncpy(char * *dest*, const char * *src*, __size_t *n*);	Copies not more than *n* characters of string pointed to by *src*, to the string pointed to by *dest*. A call to the library function strncpy() is generated.
char * __builtin_strpbrk(const char * *s1*, const char * *s2*);	Returns the first character in the string pointed to by *s1* that also is specified in the string pointed to by *s2*. A call to the library function strpbrk() is generated.

Intrinsic Function	Description
char * __builtin_strrchr(const char * *s*, int *c*);	Returns a pointer to the last occurrence of character *c* in the string pointed to by *s* or the null pointer if not found. A call to the library function `strrchr()` is generated.
__size_t __builtin_strspn(const char * *s*, const char * *set*);	Searches the string pointed to by *s* for a sequence of characters specified in the string pointed to by *set*. A call to the library function `strspn()` is generated.
char * __builtin_strstr(const char * *s1*, const char * *s2*);	Searches for a substring pointed to by *s2* in the string pointed to by *s1*. A call to the library function `strstr()` is generated.

1.10.4.7. Writing Your Own Intrinsic Function

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

1. First write a function with assembly in the body using the keyword __asm(). See Section 1.5, *Using Assembly in the C Source: __asm()*

2. Next make sure that the function is inlined rather than being called. You can do this with the function qualifier `inline`. This qualifier is discussed in more detail in Section 1.10.2, *Inlining Functions: inline*.

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

    __asm( "mov   %0,1\n"
           "1:\n\t"
           "sub   %2,%2,1\n\t"
           "mpy   %0,%0,%1\n\t"
           "bne   1p\n\t"
           : "=&r"(result)
           : "r"(base), "r"(power) );

    return result;
}

int main(void)
{
    int result;

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

    return result;
}
```

Generated assembly code:

```
main:    .type func
     ; __my_pow code is inlined here
         mov     %r1,3
         mov     %r2,2

         mov %r0,1
1:
         sub    %r2,%r2,1
         mpy    %r0,%r0,%r1
         bne    1p
```

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

1.11. Compiler Generated Sections

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

section_type_prefix.module_name.symbol_name

When you use a section renaming pragma, the compiler uses the following section naming convention:

section_type_prefix[*.pragma_value*]

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

The following table lists the section types and name prefixes.

Section type	Name section type prefix	Description
code	.text	program code
data	.data	initialized data
sdata	.sdata	initialized small data
vdata	.vdata	initialized vector memory data
bss	.bss	uninitialized data (cleared)
sbss	.sbss	uninitialized small data (cleared)
vbss	.vbss	uninitialized vector memory data (cleared)
rodata	.rodata	constant data

1.11.1. Rename Sections

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

```
section_type_prefix[.pragma_value]
```

#pragma section *type* ["*name*" | default | restore]

With this pragma all sections of the specified *type* will be named "*section_type_prefix.name*". For example with,

```
#pragma section data "where"
```

all sections of type `data` will have the name ".`data.where`".

`#pragma section` *type* will set section naming for sections of this type conform its name "*section_type_prefix*".

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

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

#pragma section all ["*name*" | default | restore]

With this pragma all sections will be named "*section_type_prefix.name*", unless you use a type specific renaming pragma. For example,

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

all sections have the syntax "*section_type_prefix*.`here`". For example, sections of type `sdata` will have the name ".`sdata.here`"

`#pragma section all` will set section naming conform its name "*section_type_prefix*".

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

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

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

Note that when you use one of the above section renaming pragmas, the module name and symbol name are no longer part of the section name.

Chapter 2. C++ Language

The TASKING C++ compiler (**cparc**) offers a new approach to high-level language programming for the ARC/PPU. The C++ compiler accepts the C++ language as defined by the ISO/IEC 14882:2014 standard with the deviations mentioned in Section 19.3, *Deviations from C++11/C++14*. It also accepts the language extensions of the C compiler (see Chapter 1, *C Language*).

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

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

- ISO/IEC 14882:2011 C++ standard [ISO/IEC]

- ISO/IEC 14882:2014 C++ standard [ISO/IEC]

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

2.1. C++ Language Extension Keywords

The C++ compiler supports the same language extension keywords (keywords with a double underscore) as the C compiler, with the exception of `__vccm`. Data type `_Float16` is supported. When option **--strict** is used, the extensions will be disabled. Note that attributes are not disabled with this option.

Since the C++ compiler is a front-end for the C compiler, the interrupt functions listed in Section 1.10.3, *Interrupt Functions / Exception Handling* and the intrinsic functions listed in Section 1.10.4, *Intrinsic Functions* can also be inserted in C++ source code, with the exception of the vector support intrinsics.

pragmas

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

attributes

The C++ compiler supports a subset of the attributes supported by the C compiler. Unlike the C compiler, the C++ compiler does not support both syntaxes in all cases:

GNU style variant:

```
__attribute__((name,...))
```

or:

```
__name__
```

The C++ compiler supports the following attributes:

Attribute	Related Information
__attribute__((aligned(*n*))) __aligned__(*n*)	Section 1.2, *Changing the Alignment: __aligned__()*, Section 1.7, *Attributes*
__attribute__((format(*type*,*arg_string_index*,*arg_check_start*))) __format__(*type*,*arg_string_index*,*arg_check_start*)	Section 1.7, *Attributes*
__attribute__((leaf))	Section 1.7, *Attributes*
__attribute__((malloc)) __malloc__	Section 1.7, *Attributes*
__attribute__((noreturn)) __noreturn__	Section 1.7, *Attributes*
__attribute__((protect)) __protect__	Section 1.7, *Attributes*
__attribute__((section("*section_name*")) __section(*section_name*)	Section 1.7, *Attributes*
__attribute__((weak))	Section 1.7, *Attributes*

When you want to use the GNU style `__attribute__((name))` in C++ use C++ compiler option **--g++**.

qualifiers

The C++ compiler supports the following qualifiers:

Qualifier	Description	Related Information
__interrupt(*n*)	Interrupt function	Section 1.10.3, *Interrupt Functions / Exception Handling*
__sda	Small data area (SDA)	Section 1.3.3, *Small Data Area (SDA)*, Section 1.3.1, *Memory Type Qualifiers*
__no_sda	Direct addressable data	Section 1.3.3, *Small Data Area (SDA)*, Section 1.3.1, *Memory Type Qualifiers*
__sfr	Special function register	Section 1.3.6, *Accessing Hardware from C*

2.2. C++ Dialect Accepted

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

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

Selecting C++11 or later also enables exceptions. For C++11 and C++14 the libc++ library is used.

You can combine the C++ dialects with the option for strict standard conformance (option **--strict**.

2.2.1. C++ Extensions

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

- The `restrict` keyword is allowed.

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

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

  Note that no warning is generated when 'j' is not used.

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

```
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.

2.2.2. GNU C++ Mode

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

Because the GNU C++ compiler frequently changes behavior between releases, the C++ compiler provides an option (**--gnu_version**) to specify a specific version of GCC to emulate. Generally speaking, features and bugs are emulated to exactly match each known version of GCC, but occasionally the emulation is approximate and in such cases the C++ compiler is often a little more permissive than GCC on the principle

that it is more important to accept source that GCC accepts than to diagnose every case that GCC diagnoses. The C++ compiler does not, however, attempt to emulate every GCC command line option; in particular, GCC options to be extra-permissive are not emulated (however, the severity of specific error diagnostics can sometimes be decreased to accept constructs that are not by default allowed in GNU emulation mode).

The following GNU extensions are *not* supported:

- The forward declaration of function parameters (so they can participate in variable-length array parameters).

- GNU-style complex integral types (complex floating-point types are supported)

- Nested functions

- Local structs with variable-length array fields.

2.3. Namespace Support

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);
}
```

2.4. Template Instantiation

The C++ language includes the concept of *templates*. A template is a description of a class or function that is a model for a family of related classes or functions.[1] For example, one can write a template for a `Stack` class, and then use a stack of integers, a stack of floats, and a stack of some user-defined type. In the source, these might be written `Stack<int>`, `Stack<float>`, and `Stack<X>`. From a single source description of the template for a stack, the compiler can create *instantiations* of the template for each of the types required.

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

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

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

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

(It should be noted that certain template entities are always instantiated when used, e.g., inline functions.)

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

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

2.4.1. Instantiation Modes

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

---

[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*.

--instantiate=used

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

--instantiate=all

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

--instantiate=local

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

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

```
ccarc test.cc
```

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

2.4.2. Instantiation #pragma Directives

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

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

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

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

The argument to the instantiation pragma may be:

- a template class name A<int>

- a template class declaration class A<int>

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

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

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

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

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

- a template variable name `x<int>`

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

```
#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.

2.5. 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.

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

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

* The function has local static variables.

* The function has local constants.

* The function has local types.

* The function has block scopes.

* The function includes pragmas.

* The function has a variable argument list.

2.6. Extern Inline Functions

Depending on the way in which the C++ compiler is configured, out-of-line copies of `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.

2.7. Pragmas to Control the C++ Compiler

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

The syntax is:

```
#pragma pragma-spec
```

or:

```
_Pragma("pragma-spec")
```

The C++ compiler supports the following pragmas:

instantiate / do_not_instantiate / can_instantiate

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

hdrstop / no_pch

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

once

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

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

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

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

2.7.1. C pragmas Supported by the C++ compiler

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

unroll_factor *value* / endunroll_factor [default | restore] (*)

Specify how many extra iterations should be added for the following loop if and when it is unrolled. At the end of the loop use `endunroll_factor`, leaving the unroller to determine the best factor automatically.

To achieve that a loop is unrolled four times, enter:

```
#pragma unroll_factor 3
  ...
#pragma endunroll_factor
```

2.8. Predefined Macros

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

Macro	Description
__ABI_COMPATIBILITY_VERSION	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.
__ABI_CHANGES_FOR_RTTI	This macro is set to TRUE, meaning that the ABI changes for RTTI are implemented. This macro is used when building the C++ library.
__ABI_CHANGES_FOR_ARRAY_NEW_AND_DELETE	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.
__ABI_CHANGES_FOR_PLACEMENT_DELETE	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.
__ARRAY_OPERATORS	Defined when array `new` and `delete` are enabled. This is the default.
__BASE_FILE__	Similar to __FILE__ but indicates the primary source file rather than the current one (i.e., when the current file is an included file).
_BOOL	Defined when `bool` is a keyword. This is the default.
__BUILD__	Identifies the build number of the C++ compiler in the format yymmddqq (year, month, day and quarter in UTC).

Macro	Description
__CHAR_MIN / __CHAR_MAX	Used in `limits.h` to define the minimum/maximum value of a plain `char` respectively.
__CHAR16_T_AND_CHAR32_T	Defined when `char16_t` and `char32_t` are keywords.
__CHAR16_TYPE__	Defined as the underlying type for `char16_t` when option **--g++** or **--gnu-version** is used. (GNU version 40400 and above, see **--gnu-version**).
__CHAR32_TYPE__	Defined as the underlying type for `char32_t` when option **--g++** or **--gnu-version** is used. (GNU version 40400 and above, see **--gnu-version**).
__CORE_*core*__	A symbol is defined depending on the option **--core=***core*. The *core* is converted to uppercase and '.' is removed. For example, if **--core=ppu_tc4dx** is specified, the symbol `__CORE_PPU_TC4DX__` is defined. When no **--core** is supplied, the C++ compiler defines `__CORE_PPU_TC49X__`.
__CPARC__	Identifies the C++ compiler. You can use this symbol to flag parts of the source which must be recognized by the **cparc** C++ compiler only. It expands to 1.
__cplusplus	Always defined.
__DATE__	Defined to the date of the compilation in the form "Mmm dd yyyy".
__DELTA_TYPE	Defines the type of the offset field in the virtual function table. This macro is used when building the C++ library.
__DOUBLE_FP__	Expands to 1 for double-precision FPU ('double' is always fully featured for the ppu_tc45x, ppu_tc49x, ppu_tc49xn, and ppu_tc4dx).
__embedded_cplusplus	Defined as 1 in Embedded C++ mode.
__EXCEPTIONS	Defined when exception handling is enabled, undefined when **--no-exceptions** is used.
__FILE__	Expands to the current source file name.
__FUNCTION__	Defined to the name of the current function. An error is issued if it is used outside of a function.
__func__	Same as __FUNCTION__ in C++11 mode and GNU C++ mode.
__GNUC__	Defined to the GNU C++ major version when option **--g++** or **--gnu-version** is used. When the GNU C++ version is *x.y.z*, this macro is defined to *x*.

Macro	Description
__GNUC_MINOR__	Defined to the GNU C++ minor version when option **--g++** or **--gnu-version** is used. When the GNU C++ version is *x.y.z*, this macro is defined to *y*.
__GNUC_PATCHLEVEL__	Defined to the GNU C++ patch level version when option **--g++** or **--gnu-version** is used. When the GNU C++ version is *x.y.z*, this macro is defined to *z*.
__GNUC_STDC_INLINE__	Defined when option **--g++** or **--gnu-version** is used and if functions declared inline will be handled according to the ISO C99 or later standards.
__GNUC_VERSION__	Defined to a string with the GNU C++ version when option **--g++** or **--gnu-version** is used. When the GNU C++ version is *x.y.z*, this macro is defined to TASKING g++ *x.y.z* mode.
__GNUG__	Defined to the GNU C++ major version when option **--g++** or **--gnu-version** is used. Testing it is equivalent to testing (__GNUC__ && __cplusplus).
__IMPLICIT_USING_STD	Defined when the standard header files should implicitly do a using-directive on the std namespace (**--using-std**).
__JMP_BUF_ELEMENT_TYPE	Specifies the type of an element of the setjmp buffer. This macro is used when building the C++ library.
__JMP_BUF_NUM_ELEMENTS	Defines the number of elements in the setjmp buffer. This macro is used when building the C++ library.
__LINE__	Expands to the line number of the line where this macro is called.
__NAMESPACES	Defined when namespaces are supported (this is the default).
__NO_LONG_LONG	Defined when the long long type is not supported. This is the default.
__NULL_EH_REGION_NUMBER	Defines the value used as the null region number value in the exception handling tables. This macro is used when building the C++ library.
__PLACEMENT_DELETE	Defined when placement delete is enabled.
__PRETTY_FUNCTION__	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.
__PTRDIFF_MIN / __PTRDIFF_MAX	Used in stdint.h to define the minimum/maximum value of a ptrdiff_t type respectively.
__PTRDIFF_TYPE__	Defined to be the type of ptrdiff_t.

Macro	Description
__REGION_NUMBER_TYPE	Defines the type of a region number field in the exception handling tables. This macro is used when building the C++ library.
__REVISION__	Expands to the revision number of the C++ compiler. Digits are represented as they are; characters (for prototypes, alphas, betas) are represented by -1. Examples: v1.0r1 -> 1, v1.0rb -> -1
__RTTI	Defined when RTTI is enabled (**--rtti**).
__RUNTIME_USES_NAMESPACES	Defined when the run-time uses namespaces.
__SIGNED_CHARS__	Defined when plain char is signed.
__SIZE_MIN / __SIZE_MAX	Used in stdint.h to define the minimum/maximum value of a size_t type respectively.
__SIZE_TYPE__	Defined to be the type of size_t.
__STDC__	Always expands to 0.
__TASKING__	Always defined for the TASKING C++ compiler.
__TIME__	Expands to the compilation time: "hh:mm:ss"
__TYPE_TRAITS_ENABLED	Defined when type traits pseudo-functions (to ease the implementation of ISO/IEC TR 19768; e.g., __is_union) are enabled. This is the default in C++ mode.
__VAR_HANDLE_TYPE	Defines the type of the variable-handle field in the exception handling tables. This macro is used when building the C++ library.
__VARIADIC_TEMPLATES	Defined when C++11 variadic templates are supported.
__VDSP_VEC_WIDTH__	Expands to the native vector width in bits. 512 for the ppu_tc49x, or 256 for the ppu_tc45x, ppu_tc49xn, and ppu_tc4dx.
__VERSION__	Identifies the version number of the C++ compiler. For example, if you use version 2.1r1 of the compiler, __VERSION__ expands to 2001 (dot and revision number are omitted, minor version number in 3 digits).
__VIRTUAL_FUNCTION_INDEX_TYPE	Defines the type of the virtual function index field of the virtual function table. This macro is used when building the C++ library.
__WCHAR_MIN / __WCHAR_MAX	Used in stdint.h to define the minimum/maximum value of a wchar_t type respectively.
_WCHAR_T	Defined when wchar_t is a keyword.

2.9. Precompiled Headers

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

2.9.1. Automatic Precompiled Header Processing

When **--pch** appears on the command line, automatic precompiled header processing is enabled. This means the C++ compiler will automatically look for a qualifying precompiled header file to read in and/or will create one for use on a subsequent compilation.

The PCH file will contain a snapshot of all the code preceding the "header stop" point. The header stop point is typically the first token in the primary source file that does not belong to a preprocessing directive, but it can also be specified directly by **#pragma hdrstop** (see below) if that comes first. For example:

```
#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.

2.9.2. Manual Precompiled Header Processing

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

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

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

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

2.9.3. Other Ways to Control Precompiled Headers

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

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

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

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

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

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

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

2.9.4. Performance Issues

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

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

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

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

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

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

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

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

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

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

Chapter 3. Assembly Language

This chapter describes the most important aspects of the TASKING assembly language for the ARC/PPU and contains a detailed description of all built-in assembly functions, assembler directives and controls. For a complete overview of the ARC/PPU architecture, refer to the *DesignWare ARCv2 ISA Programmer's Reference Manual for DW EV7x Processors* [Version 6367-001 April 2020, Synopsys, Inc.] and the *DesignWare EV7x Processor Databook* [Version 6368-004 April 2020, Synopsys, Inc.]

3.1. Assembly Syntax

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

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

The syntax of an assembly statement is:

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

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

                        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
1: LP 1p   ; 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.
	Operands are described in Section 3.3, *Operands of an Assembly Instruction*. The instructions are described in the target's core Architecture Manual.
	The instruction can also be a so-called 'alias instruction'. Alias instructions are pseudo instructions (no instructions from the instruction set). Depending on the situation in which an alias instruction is used, the assembler replaces the alias instruction with appropriate real assembly instruction(s). For a complete list, see Section 3.11, *Alias 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 3.9, *Assembler Directives and Controls*.
macro_call	A call to a previously defined macro. It must not start in the first column. See Section 3.10, *Macro Operations*.
comment	Comment, preceded by a ; (semicolon).

You can use empty lines or lines with only comments.

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

$*control*

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

3.2. Assembler Significant Characters

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

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

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

Character	Description
@	Start of a built-in assembly function
*	Location counter substitution
++	String concatenation operator
[ ]	Substring delimiter

3.3. Operands of an Assembly Instruction

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

Operand	Description
symbol	A symbolic name as described in Section 3.4, *Symbol Names*. Symbols can also occur in expressions.
register	Any valid register as listed in Section 3.5, *Registers*.
expression	Any valid expression as described in Section 3.6, *Assembly Expressions*.
address	A combination of *expression*, *register* and *symbol*.

Addressing modes

The ARC/PPU assembly language has several addressing modes. These are described in detail in the *DesignWare ARCv2 ISA Programmer's Reference Manual for DW EV7x Processors* [Version 6367-001 April 2020, Synopsys, Inc.].

3.4. Symbol Names

User-defined symbols

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

Predefined preprocessor symbols

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

Labels

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

Reserved symbols

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

Examples

Valid symbol names:

```
loop_1
ENTRY
a_B_c
_aBC
```

Invalid symbol names:

```
1_loop      ; starts with a number
r1          ; reserved register name
.DEFINE     ; reserved directive name
```

3.4.1. Predefined Preprocessor Symbols

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

Symbol	Description
__ASARC__	Identifies the assembler. You can use this symbol to flag parts of the source which must be recognized by the **asarc** assembler only. It expands to 1.
__BUILD__	Identifies the build number of the assembler in the format yymmddqq (year, month, day and quarter in UTC).
__REVISION__	Expands to the revision number of the assembler. Digits are represented as they are; characters (for prototypes, alphas, betas) are represented by -1. Examples: v1.0r1 -> 1, v1.0rb -> -1
__TASKING__	Identifies the assembler as a TASKING assembler. Expands to 1 if a TASKING assembler is used.
__VERSION__	Identifies the version number of the assembler. For example, if you use version 2.1r1 of the assembler, __VERSION__ expands to 2001 (dot and revision number are omitted, minor version number in 3 digits).

Example

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

3.5. Registers

To prevent conflicts with user-defined symbol names in an assembly language source file, the following register names, either uppercase or lowercase, should be prefixed with a percent sign '%', for example %R0.

```
R0  .. R31, R56, R58, R59, R60, R63   (general purpose registers)
VR0 .. VR31 (vector registers for ppu_tc49x and ppu_tc49xn)
VR0 .. VR39 (vector registers for ppu_tc45x and ppu_tc4dx)
P0  .. P7   (predicate registers)
GP          (alias for R26)
FP          (alias for R27)
SP          (alias for R28)
ILINK       (alias for R29)
BLINK       (alias for R31)
ACCL        (alias for R58)
ACCH        (alias for R59)
LP_COUNT    (alias for R60)
PCL         (alias for R63)
```

Among the vector registers, registers VR16 .. VR31 (for ppu_tc49x and ppu_tc49xn) and VR24 .. VR39 (for ppu_tc45x and ppu_tc4dx) are used as single-wide accumulator registers and the eight vector register pairs VR16-VR17 .. VR30-VR31 (for ppu_tc49x and ppu_tc49xn) and VR24-VR25 .. VR38-VR39 (for ppu_tc45x and ppu_tc4dx) are used as double-wide accumulators. The default double-wide accumulator is register pair VR30-VR31 for ppu_tc49x and ppu_tc49xn, and VR38-VR39 for ppu_tc45x and ppu_tc4dx and the default single-wide accumulator is VR30 for ppu_tc49x and ppu_tc49xn, and VR38 for ppu_tc45x and ppu_tc4dx .

If no predicate register is used as suffix in an instruction mnemonic, then P0 is taken as the default predicate register.

3.5.1. Special Function Registers

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

For example (from regppu.def):

```
PC      .equ  0x006
```

Without an SFR file the assembler only knows the general purpose registers as listed in Section 3.5, Registers.

3.6. Assembly Expressions

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

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

Expressions follow the conventional rules of algebra and boolean arithmetic.

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

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

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

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

- *numeric constant*

- *string*

- *symbol*

- *expression binary_operator expression*

- *unary_operator expression*

- **(***expression***)**

- *function call*

All types of expressions are explained in separate sections.

3.6.1. Numeric Constants

Numeric constants can be used in expressions. If there is no prefix or suffix, by default the assembler assumes the number is a decimal number. Prefixes and suffixes can be used in either lowercase or uppercase.

Base	Description	Example
Binary	A **0b** or **0B** prefix followed by binary digits (0,1). Or use a **b** or **B** suffix.	`0B1101` `11001010b`

Hexadecimal	A **0x** or **0X** prefix followed by hexadecimal digits (0-9, A-F, a-f). Or use a **h** or **H** suffix.	`0x12FF`
		`0x45`
		`0fa10h`

Decimal integer	Decimal digits (0-9).	`12` `1245`

Base	Description	Example

Decimal floating-point	Decimal digits (0-9), includes a decimal point, or an '**E**' or '**e**' followed by the exponent.	`6E10`
		`.6`
		`3.14`
		`2.7e10`

3.6.2. Strings

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

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

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

```
[string,offset,length]
```

offset is the start position within string. *length* is the length of the desired substring. Both values may not exceed the size of *string*.

Examples

```
'ABCD'              ; (0x44434241)
'''79'              ; to enclose a quote double it
"A\"BC"             ; or to enclose a quote escape it
'AB'+1              ; (0x4341) string used in expression
''                  ; null string
.DW 'abcdef'        ; (0x64636261) 'ef' are ignored
                    ; warning: string value truncated
'ab'++'cd'          ; you can concatenate
                    ; two strings with the '++' operator.
                    ; This results in 'abcd'
['TASKING',0,4]     ; results in the substring 'TASK'
```

3.6.3. Expression Operators

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

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

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

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.
	~	one's complement	Integer only. Returns the one's complement of its operand. It cannot be used with a floating-point operand.
	!	logical negate	Returns 1 if the operands' value is 0; otherwise 0. For example, if `buf` is 0 then `!buf` is 1. If `buf` has a value of 1000 then `!buf` is 0.
Arithmetic	*	multiplication	Yields the product of its operands.
	/	division	Yields the quotient of the division of the first operand by the second. For integer operands, the divide operation produces a truncated integer result.
	%	modulo	Integer only. This operator yields the remainder from the division of the first operand by the second.
	+	addition	Yields the sum of its operands.
	-	subtraction	Yields the difference of its operands.
Shift	<<	shift left	Integer only. Causes the left operand to be shifted to the left (and zero-filled) by the number of bits specified by the right operand.
	>>	shift right	Integer only. Causes the left operand to be shifted to the right by the number of bits specified by the right operand. The sign bit will be extended.
Relational	<	less than	Returns an integer 1 if the indicated condition is TRUE or an integer 0 if the indicated condition is FALSE.
	<=	less than or equal	
	>	greater than	For example, if D has a value of 3 and E has a value of 5, then the result of the expression `D<E` is 1, and the result of the expression `D>E` is 0.
	>=	greater than or equal	
	==	equal	
	!=	not equal	Use tests for equality involving floating-point values with caution, since rounding errors could cause unexpected results.
Bit and Bitwise	&	AND	Integer only. Yields the bitwise AND function of its operand.
	|	OR	Integer only. Yields the bitwise OR function of its operand.

Type	Operator	Name	Description
	^	exclusive OR	Integer only. Yields the bitwise exclusive OR function of its operands.
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

The relational operators and logical operators are intended primarily for use with the conditional assembly `.if` directive, but can be used in any expression.

3.7. Working with Sections

Sections are absolute or relocatable blocks of contiguous memory that can contain code or data. Some sections contain code or data that your program declared and uses directly, while other sections are created by the compiler or linker and contain debug information or code or data to initialize your application. These sections can be named in such a way that different modules can implement different parts of these sections. These sections are located in memory by the linker (using the linker script language, LSL) so that concerns about memory placement are postponed until after the assembly process.

All instructions and directives which generate data or code must be within an active section. The assembler emits a warning if code or data starts without a section definition. The compiler automatically generates sections. If you program in assembly you have to define sections yourself.

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

Section definition

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

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

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

Examples

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

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

103

```
 ; ...
.ENDSEC
```

3.8. Built-in Assembly Functions

The TASKING assembler has several built-in functions to support data conversion, string comparison, and math computations. You can use functions as terms in any expression.

Syntax of an assembly function

`@function_name([argument[,argument]...])`

Functions start with the '@' character and have zero or more arguments, and are always followed by opening and closing parentheses. White space (a blank or tab) is not allowed between the function name and the opening parenthesis and between the (comma-separated) arguments.

The names of assembly functions are case insensitive.

Overview of assembly functions

Function	Description
@ARG('*symbol*' | *expr*)	Test whether macro argument is present
@CNT()	Return number of macro arguments
@DEFINED('*symbol*' | *symbol*)	Test whether *symbol* exists
@LSB(*expr*)	Least significant byte of the expression
@LSH(*expr*)	Least significant half word of the absolute expression
@LSW(*expr*)	Least significant word of the expression
@MSB(*expr*)	Most significant byte of the expression
@MSH(*expr*)	Most significant half word of the absolute expression
@MSW(*expr*)	Most significant word of the expression
@SDA(*expr*)	Access small data area objects
@STRCAT(*str1*,*str2*)	Concatenate *str1* and *str2*
@STRCMP(*str1*,*str2*)	Compare *str1* with *str2*
@STRLEN(*string*)	Return length of string
@STRPOS(*str1*,*str2*[,*start*])	Return position of *str2* in *str1*
@STRSUB(*str*,*expr1*,*expr2*)	Return substring

Detailed Description of Built-in Assembly Functions

@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?
```

@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
```

@DEFINED('*symbol*' | *symbol*)

Returns 1 if *symbol* has been defined, 0 otherwise. If *symbol* is quoted, it is looked up as a `.DEFINE` symbol; if it is not quoted, it is looked up as an ordinary symbol, macro or label.

Example:

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

@LSB(*expression*)

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

Example:

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

@LSH(*expression*)

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

@LSW(*expression*)

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

Example:

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

@MSB(*expression*)

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

@MSH(*expression*)

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

@MSW(*expression*)

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

@SDA(*expr*)

Designates access to objects in the small data area by means of the global pointer (%gp). In other words, this function is used by instructions along with gp-relative symbols to access small data area objects.

Example:

```
ld %r0,[%gp, @sda(var)]
```

This instruction loads the small data object pointed by the offset var to register r0.

@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:

106

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

@STRLEN(*string*)

Returns the length of *string* as an integer.

Example:

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

@STRPOS(*string1*,*string2*[,*start*])

Returns the position of *string2* in *string1* as an integer. If *string2* does not occur in *string1*, the last string position + 1 is returned.

With *start* you can specify the starting position of the search. If you do not specify start, the search is started from the beginning of *string1*.

Example:

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

@STRSUB(*string*,*expression1*,*expression2*)

Returns the substring from *string* as a string. *expression1* is the starting position within *string*, and *expression2* is the length of the desired string. The assembler issues an error if either *expression1* or *expression2* exceeds the length of string. Note that the first position in a string is position 0.

Example:

```
  .DEFINE  ID  "@STRSUB('TASKING',3,4)"  ;ID = 'KING'
```

3.9. Assembler Directives and Controls

An assembler directive is simply a message to the assembler. Assembler directives are not translated into machine instructions. There are three main groups of assembler directives.

• Assembler directives that tell the assembler how to go about translating instructions into machine code. This is the most typical form of assembly directives. Typically they tell the assembler where to put a program in memory, what space to allocate for variables, and allow you to initialize memory with data. When the assembly source is assembled, a location counter in the assembler keeps track of where the code and data is to go in memory.

   The following directives fall under this group:

   • Assembly control directives

   • Symbol definition and section directives

- Data definition / Storage allocation directives

- High Level Language (HLL) directives

- Directives that are interpreted by the macro preprocessor. These directives tell the macro preprocessor how to manipulate your assembly code before it is actually being assembled. You can use these directives to write macros and to write conditional source code. Parts of the code that do not match the condition, will not be assembled at all.

- Some directives act as assembler options and most of them indeed do have an equivalent assembler (command line) option. The advantage of using a directive is that with such a directive you can overrule the assembler option for a particular part of the code. Directives of this kind are called *controls*. A typical example is to tell the assembler with an option to generate a list file while with the controls $LIST ON and $LIST OFF you overrule this option for a part of the code that you do not want to appear in the list file. Controls always appear on a separate line and start with a '$' sign in the first column.

  The following controls are available:

- Assembly listing controls

- Miscellaneous controls

Each assembler directive or control has its own syntax. You can use assembler directives and controls in the assembly code as pseudo instructions.

Some assembler directives can be preceded with a label. If you do not precede an assembler directive with a label, you must use white space instead (spaces or tabs). The assembler recognizes both uppercase and lowercase for directives.

3.9.1. Assembler Directives

Overview of assembly control directives

Directive	Description
`.{, .}`	Indicates the start and end of a bundle of instructions
`.END`	Indicates the end of an assembly module
`.INCLUDE`	Include file
`.MESSAGE`	Programmer generated message

Overview of symbol definition and section directives

Directive	Description
`.ALIAS`	Create an alias for a symbol
`.EQU`	Set permanent value to a symbol
`.EXTERN`	Import global section symbol
`.GLOBAL`	Declare global section symbol
`.LOCAL`	Declare local section symbol

Directive	Description
`.SECTION`, `.ENDSEC`	Start a new section
`.SET`	Set temporary value to a symbol
`.SIZE`	Set size of symbol in the ELF symbol table
`.SOURCE`	Specify name of original C source file
`.TYPE`	Set symbol type in the ELF symbol table
`.WEAK`	Mark a symbol as 'weak'

Overview of data definition / storage allocation directives

Directive	Description
`.ALIGN`	Align location counter
`.ASCII`, `.ASCIIZ`	Define ASCII string without / with ending NULL byte
`.BS`, `.BSB`, `.BSH`, `.BSW`, `.BSD`	Define block storage (initialized)
`.DB`	Define byte
`.DH`	Define half word (16 bits)
`.DW`	Define word (32 bits)
`.DD`	Define double-word (64 bits)
`.DOUBLE`	Define a 64-bit floating-point constant
`.DS`, `.DSB`, `.DSH`, `.DSW`, `.DSD`	Define storage
`.FLOAT`	Define a 32-bit floating-point constant

Overview of macro preprocessor directives

Directive	Description
`.DEFINE`	Define substitution string
`.FOR`, `.ENDFOR`	Repeat sequence of source lines n times
`.IF`, `.ELIF`, `.ELSE`	Conditional assembly directive
`.ENDIF`	End of conditional assembly directive
`.EXITM`	Exit macro
`.MACRO`, `.ENDM`	Define macro
`.PMACRO`	Undefine (purge) macro
`.REPEAT`, `.ENDREP`	Repeat sequence of source lines
`.UNDEF`	Undefine `.DEFINE` symbol

Overview of HLL directives

Directive	Description
.CALLS	Pass call tree information and/or stack usage information
.COMPILER_ENV	Pass environment variables used for the generation of the assembly source file
.COMPILER_INVOCATION	Pass C compiler invocation
.COMPILER_NAME	Pass C compiler name
.COMPILER_VERSION	Pass C compiler version header
.MISRAC	Pass MISRA C information

.{, .}

Syntax

```
.{
instruction1
instruction2
...
.}
```

Description

With the .{ and .} directives you can indicate the beginning and the end of a bundle of instructions. The assembler assembles the instructions that are given within these directives into a variable length instruction word (VLIW), also called Super instruction. The assembler takes the vector instructions in the same order given to form a valid bundle. If a valid bundle cannot be formed, then the assembler issues an error message.

You can use these directives only in code sections and you can only specify instructions within these directives.

Examples

A bundle of a vector instruction and a scalar instruction:

```
.{
vvadd %vr1, %vr2, %vr3
add %r0, %r2, %r3
.}
```

A bundle of two vector instructions:

```
.{
vvabs.h %vr3, %vr19
vvsub.b %vr1, %vr2, %vr12
.}
```

A bundle of two predicated vector instructions and a scalar instruction:

```
.{
vvsub.h.p2 %vr1, %vr2, 0x232
vvadd.b.p6 %vr4, %vr5, %vr8
add_s %r1, %r2, %r3
.}
```

Related Information

-

.ALIAS

Syntax

```
alias-name .ALIAS symbol-name
```

Description

With the .ALIAS directive you can create an alias of a symbol. The C compiler generates this directive when you use the #pragma alias.

Example

```
exit .ALIAS _Exit
```

Related information

Pragma alias

.ALIGN

Syntax

```
.ALIGN expression
```

Description

With the .ALIGN directive you instruct the assembler to align the location counter. By default the assembler aligns on one byte.

When the assembler encounters the .ALIGN directive, it advances the location counter to an address that is aligned as specified by *expression* and places the next instruction or directive on that address. The alignment is in minimal addressable units (MAUs). The assembler fills the 'gap' with NOP instructions for code sections or with zeros for data sections. If the location counter is already aligned on the specified alignment, it remains unchanged. The location of absolute sections will not be changed.

The *expression* must be a power of two: 2, 4, 8, 16, ... If you specify another value, the assembler changes the alignment to the next higher power of two and issues a warning.

The assembler aligns sections automatically to the largest alignment value occurring in that section.

A label is not allowed before this directive.

Example

Example

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

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

.ASCII, .ASCIIZ

Syntax

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

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

Description

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

The `.ASCII` directive does not add a NULL byte to the end of the string. The `.ASCIIZ` directive does add a NULL byte to the end of the string. The "z" in `.ASCIIZ` stands for "zero". Use commas to separate multiple strings.

Example

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

Note that with the `.DB` directive you can obtain exactly the same effect:

```
STRING:  .DB  "Hello world"    ; without a NULL byte
STRINGZ: .DB  "Hello world",0  ; with a NULL byte
```

Related Information

`.DB` (Define a constant byte)

.BS, .BSB, .BSH, .BSW, .BSD

Syntax

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

Description

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

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

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

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

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

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

Directive	Reserved bits
.BSB	8
.BSH	16
.BSW	32
.BSD	64

Example

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

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

Related Information

`.DB` (Define Memory)

`.DS` (Define Storage)

.CALLS

Syntax

.CALLS *'caller','callee'*

or

.CALLS *'caller','',stack_usage*

Description

The first syntax creates a call graph reference between *caller* and *callee*. The linker needs this information to build a call graph. *caller* and *callee* are names of functions.

The second syntax specifies stack information. When *callee* is an empty name, this means we define the stack usage of the function itself. The value specified is the stack usage in bytes at the time of the call including the return address.

This information is used by the linker to compute the used stack within the application. The information is found in the generated linker map file within the Memory Usage.

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

A label is not allowed before this directive.

Example

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

Indicates that the function main calls the function nfunc.

```
  .CALLS 'main','',8
```

The function main uses 8 bytes on the stack.

.COMPILER_ENV, .COMPILER_INVOCATION, .COMPILER_NAME, .COMPILER_VERSION

Syntax

```
.COMPILER_VERSION "version_header"
.COMPILER_INVOCATION "invocation"
.COMPILER_NAME "name"
.COMPILER_ENV "env_var_name=env_var_value"
```

Description

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

A label is not allowed before these directives.

Example

```
.COMPILER_VERSION "TASKING SmartCode - ARC C compiler vx.yrz Build yymmddqq"
.COMPILER_INVOCATION "carc test.c"
.COMPILER_NAME "carc"
.COMPILER_ENV "CARCINC=C:\\Program Files\\TASKING\\SmartCode vx.yrz\\carc\\include"
```

.DB, .DH, .DW, .DD

Syntax

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

Description

With these directive you can define memory. With each directive the assembler allocates and initializes one or more bytes of memory for each argument.

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

An *argument* can be a single- or multiple-character string constant, an expression or empty. Multiple arguments must be separated by commas with no intervening spaces. Empty arguments are stored as 0 (zero).

The following table shows the number of bits initialized.

Directive	Bits
.DB	8
.DH	16
.DW	32
.DD	64

The value of the arguments must be in range with the size of the directive; floating-point numbers are not allowed. If the evaluated argument is too large to be represented in a half word / word / double-word, the assembler issues a warning and truncates the value.

String constants

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

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

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

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

Example

When a string is supplied as argument of a directive that initializes multiple bytes, each character in the string is stored in consecutive bytes whose lower seven bits represent the ASCII value of the character. For example:

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

Related Information

.BS (Block Storage)

.DS (Define Storage)

.DEFINE

Syntax

```
.DEFINE symbol  string
```

Description

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

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

Macros represent a special case. .DEFINE directive translations will be applied to the macro definition as it is encountered. When the macro is expanded, any active .DEFINE directive translations will again be applied.

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

A label is not allowed before this directive.

Example

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

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

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

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

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

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

Related Information

.UNDEF (Undefine a .DEFINE symbol)

.MACRO, .ENDM (Define a macro)

.DS, .DSB, .DSH, .DSW, .DSD

Syntax

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

Description

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

With the *expression* you specify the number of MAUs (Minimal Addressable Units) that you want to reserve, and how much the location counter will advance. The expression must evaluate to an integer greater than zero and cannot contain any forward references (symbols that have not yet been defined).

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

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

Directive	Reserved bits
.DSB	8
.DSH	16
.DSW	32
.DSD	64

Example

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

Related Information

.BS (Block Storage)

.DB (Define Memory)

.END

Syntax

`.END`

Description

With the mandatory `.END` directive you tell the assembler that the end of the module is reached. If the assembler finds assembly source lines beyond the `.END` directive, it ignores those lines and issues a warning.

You cannot use the `.END` directive in a macro expansion.

The assembler does not allow a label with this directive.

When you use the assembler option **--no-require-end**, the `.END` directive is optional.

Example

```
    ; source lines
.END                ; End of assembly module
```

Related Information

Assembler option **--no-require-end**

.EQU

Syntax

symbol **.EQU** *expression*

Description

With the .EQU directive you assign the value of *expression* to *symbol* permanently. The expression can be relocatable or absolute and forward references are allowed. Once defined, you cannot redefine the symbol. With the .GLOBAL directive you can declare the symbol global.

Example

To assign the value 0x400 permanently to the symbol MYSYMBOL:

MYSYMBOL .EQU 0x4000

You cannot redefine the symbol MYSYMBOL after this.

Related Information

.SET (Set temporary value to a symbol)

.EXITM

Syntax

```
.EXITM
```

Description

With the `.EXITM` directive the assembler will immediately terminate a macro expansion. It is useful when you use it with the conditional assembly directive `.IF` to terminate macro expansion when, for example, error conditions are detected.

A label is not allowed before this directive.

Example

```
CALC   .MACRO  XVAL,YVAL
       .IF     XVAL<0
       .MESSAGE F 'Macro parameter value out of range'
       .EXITM  ;Exit macro
       .ENDIF
         .
         .
         .
       .ENDM
```

Related Information

.MACRO, .ENDM (Define a macro)

.EXTERN

Syntax

```
.EXTERN symbol[,symbol]...
```

Description

With the `.EXTERN` directive you define an *external* symbol. It means that the specified symbol is referenced in the current module, but is not defined within the current module. This symbol must either have been defined outside of any module or declared as globally accessible within another module with the `.GLOBAL` directive.

If you do not use the `.EXTERN` directive and the symbol is not defined within the current module, the assembler issues a warning and inserts the `.EXTERN` directive.

A label is not allowed with this directive.

Example

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

Related Information

`.GLOBAL` (Declare global section symbol)

`.LOCAL` (Declare local section symbol)

.FLOAT, .DOUBLE

Syntax

[*label*:]**.FLOAT** *expression*[*,expression*]...

[*label*:]**.DOUBLE** *expression*[*,expression*]...

Description

With the `.FLOAT` or `.DOUBLE` directive the assembler allocates and initializes a floating-point number (32 bits) or a double (64 bits) in memory for each argument.

An *expression* can be:

• a floating-point expression

• NULL (indicated by two adjacent commas: ,,)

You can represent a constant as a signed whole number with fraction or with the 'e' format as used in the C language. For example, `12.457` and `+0.27E-13` are legal floating-point constants.

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

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

Example

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

Related Information

`.DS` (Define Storage)

.FOR, .ENDFOR

Syntax

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

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

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

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

Example

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

```
.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 **--symbol-scope=global**.

With the .GLOBAL directive you declare one of more symbols as global. It means that the specified symbols are defined within the current section or module, and that those definitions should be accessible by all modules.

To access a symbol, defined with .GLOBAL, from another module, use the .EXTERN directive.

Only program labels and symbols defined with .EQU can be made global.

If the symbols that appear in the operand field are not used in the module, the assembler gives a warning.

The assembler does not allow a label with this directive.

Example

```
LOOPA .EQU 1          ; definition of symbol LOOPA
      .GLOBAL  LOOPA  ; LOOPA will be globally
                      ; accessible by other modules
```

Related Information

.EXTERN (Import global section symbol)

.LOCAL (Declare local section symbol)

.IF, .ELIF, .ELSE, .ENDIF

Syntax

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

Description

With the `.IF`/`.ENDIF` directives you can create a part of conditional assembly code. The assembler assembles only the code that matches a specified condition.

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

If the optional `.ELSE` and/or `.ELIF` directives are not present, then the source statements following the `.IF` directive and up to the next `.ENDIF` directive will be included as part of the source file being assembled only if the *expression* had a non-zero result.

If the *expression* has a value of zero, the source file will be assembled as if those statements between the `.IF` and the `.ENDIF` directives were never encountered.

If the `.ELSE` directive is present and expression has a nonzero result, then the statements between the `.IF` and `.ELSE` directives will be assembled, and the statement between the `.ELSE` and `.ENDIF` directives will be skipped. Alternatively, if expression has a value of zero, then the statements between the `.IF` and `.ELSE` directives will be skipped, and the statements between the `.ELSE` and `.ENDIF` directives will be assembled.

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

A label is not allowed with this directive.

Example

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

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

131

```
  ... ; code for the final version
  .ENDIF
```

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

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

You can also define the symbols on the command line with the assembler option **--define** (**-D**):

```
asarc --define=DEMO --define=TEST=0 test.asm
```

.INCLUDE

Syntax

.INCLUDE "*filename*" | <*filename*>

Description

With the .INCLUDE directive you include another file at the exact location where the .INCLUDE occurs. This happens before the resulting file is assembled. The .INCLUDE directive works similarly to the #include statement in C. The source from the include file is assembled as if it followed the point of the .INCLUDE directive. When the end of the included file is reached, assembly of the original file continues.

The string specifies the filename of the file to be included. The filename must be compatible with the operating system (forward/backward slashes) and can contain a directory specification.

If an absolute pathname is specified, the assembler searches for that file. If a relative path is specified or just a filename, the order in which the assembler searches for include files is:

1. The current directory if you use the "*filename*" construction.

   The current directory is not searched if you use the **<*filename*>** syntax.

2. The path that is specified with the assembler option **--include-directory**.

3. The path that is specified in the environment variable ASARCINC when the product was installed.

4. The default include directory in the installation directory.

The assembler does not allow a label with this directive.

Example

```
.INCLUDE 'storage\mem.asm'    ; include file
.INCLUDE <data.asm>           ; Do not look in
                              ; current directory
```

.LOCAL

Syntax

```
.LOCAL symbol[,symbol]...
```

Description

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

With the .LOCAL directive you declare one of more symbols as local. It means that the specified symbols are explicitly local to the module in which you define them.

If the symbols that appear in the operand field are not used in the module, the assembler gives a warning.

The assembler does not allow a label with this directive.

Example

```
        .SECTION  .data
        .LOCAL    LOOPA   ; LOOPA is local to this section

LOOPA .DH        0x100   ; assigns the value 0x100 to LOOPA
```

Related Information

.EXTERN (Import global section symbol)

.GLOBAL (Declare global section symbol)

.MACRO, .ENDM

Syntax

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

Description

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

The definition of a macro consists of three parts:

- *Header*, which assigns a name to the macro and defines the arguments (`.MACRO` directive).

- *Body*, which contains the code or instructions to be inserted when the macro is called.

- *Terminator*, which indicates the end of the macro definition (`.ENDM` directive).

The arguments are symbolic names that the macro processor replaces with the literal arguments when the macro is expanded (called). Each formal *argument* must follow the same rules as symbol names: the name can consist of letters, digits and underscore characters (_). The first character cannot be a digit. Argument names cannot start with a percent sign (**%**).

Macro definitions can be nested but the nested macro will not be defined until the primary macro is expanded.

You can use the following operators in macro definition statements:

Operator	Name	Description
\	Macro argument concatenation	Concatenates a macro argument with adjacent alphanumeric characters.
?	Return decimal value of symbol	Substitutes the **?***symbol* sequence with a character string that represents the decimal value of the symbol.
%	Return hex value of symbol	Substitutes the **%***symbol* 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:

```
CONST.D .MACRO  reg,value                   ;header
        ldl.iu  reg,@HI(value)        ;body
```

135

```
        ldl.il  reg,@LO(value)
        .ENDM                                   ;terminator
```

The macro call:

```
    .section .text
    CONST.D  r5,0x12345678
```

The macro expands as follows:

```
    ldl.iu  r5,@HI(0x12345678)
    ldl.il  r5,@LO(0x12345678)
```

Related Information

Section 3.10, *Macro Operations*

.PMACRO (Undefine macro)

.DEFINE (Define a substitution string)

.MESSAGE

Syntax

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

Description

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

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

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.

An arbitrary number of strings and expressions, in any order but separated by commas with no intervening white space, can be specified to describe the nature of the generated message. If you use expressions, the assembler outputs the result. The assembler outputs a space between each argument.

The error and warning counts will not be affected. The .MESSAGE directive is for example useful in combination with conditional assembly to indicate which part is assembled. The assembling process proceeds normally after the message has been printed.

This directive has no effect on the exit code of the assembler.

A label is not allowed with this directive.

Example

```
    .MESSAGE I 'Generating tables'

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

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

Within single quotes, the defined symbol LONG is not expanded. Within double quotes the symbol LONG is expanded so the actual message is printed as:

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

137

.MISRAC

Syntax

```
.MISRAC string
```

Description

The C compiler can generate the `.MISRAC` directive to pass the compiler's MISRA C settings to the object file. The linker performs checks on these settings and can generate a report. It is not recommended to use this directive in hand-coded assembly.

Example

```
.MISRAC 'MISRA-C:2004,64,e2,0b,e,e11,27,6,ef83,e1,
        ef,66,cb75,af1,eff,e7,e7f,8d,63,87ff7,6ff3,4'
```

Related Information

Section 4.7.2, *C Code Checking: MISRA C*

C compiler option **--misrac**

.PMACRO

Syntax

```
.PMACRO symbol[,symbol]...
```

Description

With the .PMACRO directive you tell the assembler to undefine the specified macro, so that later uses of the symbol will not be expanded.

The assembler does not allow a label with this directive.

Example

```
.PMACRO MAC1,MAC2
```

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

Related Information

.MACRO, .ENDM (Define a macro)

.REPEAT, .ENDREP

Syntax

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

Description

With the `.REPEAT`/`.ENDREP` directive you can repeat a sequence of assembly source lines. With *expression* you specify the number of times the loop is repeated. If the *expression* evaluates to a number less than or equal to 0, the sequence of lines will not be included in the assembler output. The *expression* result must be an absolute integer and cannot contain any forward references (symbols that have not already been defined). The `.REPEAT` directive may be nested to any level.

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

Example

In this example the loop is repeated 3 times. Effectively, the preprocessor repeats the source lines (`.DB 10`) three times, then the assembler assembles the result:

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

Related Information

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

.SECTION, .ENDSEC

Syntax

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

Description

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

If you define a section, you must always specify the section *name*. The names have a special meaning to the locating process and have to start with a predefined name, optionally extended by a dot '**.**' and a user defined name. For example, `.text.myname` or `.data.mymodule.myobject`. 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	Implied attributes	Forbidden attributes
.text	Code sections	code		init
.data	Initialized data	data	init	noinit, clear, romdata
.sdata	Initialized data in small data area	data	init	noinit, clear, romdata
.vdata	Initialized data in vector memory (VCCM)	data	init	noinit, clear, romdata
.bss	Uninitialized data (cleared)	data	clear	init, romdata, noclear
.sbss	Uninitialized data in small data area (cleared)	data	clear	init, romdata, noclear
.vbss	Uninitialized data in vector memory (VCCM) (cleared)	data	clear	init, romdata
.rodata	ROM data (constants)	data	romdata	init, clear
.debug .debug_	Debug sections. .debug_ is used as a prefix	debug		

The section attributes are case insensitive. The defined *attribute*s are:

Attribute	Description	Allowed on type
align(*value*)	Align the section to the value specified. *value* must be a power of two.	code, data
at(*address*)	Locate the section at the given *address*.	code, data

141

Attribute	Description	Allowed on type
clear	Sections are zeroed at startup. Clear can only be used in combination with noinit.	data
cluster('*name*')	Cluster code sections with companion debug sections. Used by the linker during removal of unreferenced sections. The name must be unique for this module (not for the application).	code, data, debug
concat	Concatenate sections. Used by the linker to merge sections with the same name. The assembler removes the default 'separate' ELF section attribute.	data
group('*group*')	Used to group sections. The assembler appends `@group` to the section name.	data
init	Defines that the section contains initialization data, which is copied from ROM to RAM at program startup.	code, data
linkonce '*tag*'	For internal use only.	
max	When data sections with the same name occur in different object modules with the MAX attribute, the linker generates a section of which the size is the maximum of the sizes in the individual object modules. Only works on sections with both attributes noinit and noclear set.	data
noclear	Sections are not zeroed at startup. This is a default attribute for data sections. This attribute is only useful with .vbss sections, which are cleared at startup by default.	data
noinit	Defines that the section contains no initialization data.	code, data
overlay('*name*')	Static stack overlay. Automatic stack variables, function stack parameters and temporary data are stored here. The assembler appends `name@function` to the section prefix.	data

Attribute	Description	Allowed on type
protect	Tells the linker to exclude a section from unreferenced section removal and duplicate section removal.	code, data
romdata	Section contains data to be placed in ROM. This ROM area is not executable.	data

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

Example

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

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

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

Related Information

Section 3.7, *Working with Sections*

.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 forward references are allowed.

Example

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

Related Information

.EQU (Set permanent value to a symbol)

.SIZE

Syntax

```
.SIZE  symbol,expression
```

Description

With the .SIZE directive you set the size of the specified *symbol* to the value represented by *expression*.

The .SIZE directive may occur anywhere in the source file unless the specified symbol is a function. In this case, the .SIZE directive must occur after the function has been defined.

Example

```
        .section .text.hello.main ,cluster('$group_main')
        .global main
        .align  4
; Function main
main:   .type   func
         ;
        .SIZE   main,*-main
        .endsec
```

Related Information

.TYPE (Set symbol type)

.SOURCE

Syntax

`.SOURCE` *string*

Description

With the `.SOURCE` directive you specify the name of the original C source module. This directive is generated by the C compiler. You do not need this directive in hand-written assembly.

Example

```
.SOURCE "main.c"
```

.TYPE

Syntax

symbol **.TYPE** *typeid*

Description

With the `.TYPE` directive you set a *symbol*'s type to the specified value in the ELF symbol table. Valid symbol types are:

FUNC	The symbol is associated with a function or other executable code.
OBJECT	The symbol is associated with an object such as a variable, an array, or a structure.
FILE	The symbol name represents the filename of the compilation unit.

Labels in code sections have the default type FUNC. Labels in data sections have the default type OBJECT.

Example

```
Afunc:  .type   func
```

Related Information

.SIZE (Set symbol size)

.UNDEF

Syntax

```
.UNDEF symbol
```

Description

With the `.UNDEF` directive you can undefine a substitution string that was previously defined with the `.DEFINE` directive. The substitution string associated with *symbol* is released, and *symbol* will no longer represent a valid `.DEFINE` substitution or macro.

The assembler issues a warning if you undefine a non-existing symbol.

The assembler does not allow a label with this directive.

Example

The following example undefines the `LEN` substitution string that was previously defined with the `.DEFINE` directive:

```
.UNDEF LEN
```

Related Information

`.DEFINE` (Define a substitution string)

.WEAK

Syntax

```
.WEAK symbol[,symbol]...
```

Description

With the `.WEAK` directive you mark one or more symbols as 'weak'. The *symbol* can be defined in the same module with the `.GLOBAL` directive or the `.EXTERN` directive. If the symbol does not already exist, it will be created.

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

You can overrule a weak definition with a `.GLOBAL` definition in another module. The linker will not complain about the duplicate definition, and ignore the weak definition.

Only program labels and symbols defined with `.EQU` can be made weak.

Example

```
LOOPA .EQU 1          ; definition of symbol LOOPA
      .GLOBAL  LOOPA  ; LOOPA will be globally
                      ; accessible by other modules
      .WEAK LOOPA     ; mark symbol LOOPA as weak
```

Related Information

.EXTERN (Import global section symbol)

.GLOBAL (Declare global section symbol)

3.9.2. Assembler Controls

Controls start with a **$** as the first character on the line. Unknown controls are ignored after a warning is issued.

Overview of assembler listing controls

Control	Description
$LIST ON/OFF	Print / do not print source lines to list file
$PAGE	Generate form feed in list file
$PAGE *settings*	Define page layout for assembly list file
$PRCTL	Send control string to printer
$STITLE	Set program subtitle in header of assembly list file
$TITLE	Set program title in header of assembly list file

Overview of miscellaneous assembler controls

Control	Description
$CASE ON/OFF	Case sensitive user names ON/OFF
$DEBUG ON/OFF	Generation of symbolic debug ON/OFF
$DEBUG "*flags*"	Select debug information
$IDENT LOCAL/GLOBAL	Assembler treats labels by default as local or global
$WARNING OFF [*num*]	Suppress all or some warnings

$CASE

Syntax

```
$CASE  ON
$CASE  OFF
```

Default

```
$CASE ON
```

Description

With the $CASE ON and $CASE OFF controls you specify wether the assembler operates in case sensitive mode or not. By default the assembler operates in case sensitive mode. This means that all user-defined symbols and labels are treated case sensitive, so LAB and Lab are distinct.

Note that the instruction mnemonics, register names, directives and controls are always treated case insensitive.

Example

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

Related Information

Assembler option **--case-insensitive**

$DEBUG

Syntax

```
$DEBUG   ON
$DEBUG   OFF
$DEBUG   "flags"
```

Default

```
$DEBUG "AhLS"
```

Description

With the $DEBUG ON and $DEBUG OFF controls you turn the generation of debug information on or off. ($DEBUG ON is similar to the assembler option **--debug-info=+local** (**-gl**).

If you use the $DEBUG control with flags, you can set the following flags:

a/A	Assembly source line information
h/H	Pass high level language debug information (HLL)
l/L	Assembler local symbols debug information
s/S	Smart debug information

You cannot specify $DEBUG "ah". Either the assembler generates assembly source line information, or it passes HLL debug information.

Debug information that is generated by the C compiler, is always passed to the object file.

Example

```
;begin of source
$DEBUG ON   ; generate local symbols debug information
```

Related Information

Assembler option **--debug-info**

$IDENT

Syntax

```
$IDENT LOCAL
$IDENT GLOBAL
```

Default

```
$IDENT LOCAL
```

Description

With the controls `$IDENT LOCAL` and `$IDENT GLOBAL` you tell the assembler how to treat symbols that you have not specified explicitly as local or global with the assembler directives `.LOCAL` or `.GLOBAL`.

By default the assembler treats all symbols as local symbols unless you have defined them to be global explicitly.

Example

```
;begin of source
$IDENT GLOBAL  ; assembly labels are global by default
```

Related Information

Assembler directive **.GLOBAL**

Assembler directive **.LOCAL**

Assembler option **--symbol-scope**

$LIST ON/OFF

Syntax

```
$LIST ON
$LIST OFF
```

Default

```
$LIST ON
```

Description

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

The $LIST ON control actually increments a counter that is checked for a positive value and is symmetrical with respect to the $LIST OFF control. Note the following sequence:

```
; Counter value currently 1
$LIST ON          ; Counter value = 2
$LIST ON          ; Counter value = 3
$LIST OFF         ; Counter value = 2
$LIST OFF         ; Counter value = 1
```

The listing still would not be disabled until another $LIST OFF control was issued.

Example

```
    .section .text
    ... ; source line in list file
$LIST OFF
    ... ; source line not in list file
$LIST ON
    ... ; source line also in list file
```

Related Information

Assembler option **--list-file**

$PAGE

Syntax

$PAGE [*pagewidth*[,*pagelength*[,*blankleft*[,*blanktop*[,*blankbtm*]]]]]

Default

$PAGE 132,72,0,0,0

Description

If you generate a list file with the assembler option **--list-file**, you can use the $PAGE control to format the generated list file.

The arguments may be any positive absolute integer expression, and must be separated by commas.

pagewidth	Number of columns per line. The default is 132, the minimum is 40.
pagelength	Total number of lines per page. The default is 72, the minimum is 10. As a special case, a page length of 0 turns off page breaks.
blankleft	Number of blank columns at the left of the page. The default is 0, the minimum is 0, and the maximum must maintain the relationship: *blankleft* < *pagewidth*.
blanktop	Number of blank lines at the top of the page. The default is 0, the minimum is 0 and the maximum must be a value so that (*blanktop* + *blankbtm*) ≤ (*pagelength* - 10).
blankbtm	Number of blank lines at the bottom of the page. The default is 0, the minimum is 0 and the maximum must be a value so that (*blanktop* + *blankbtm*) ≤ (*pagelength* - 10).

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

Example

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

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

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

Related Information

Assembler option **--list-file**

$PRCTL

Syntax

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

Description

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

The $PRCTL control simply concatenates its arguments and sends them to the listing file (the control line itself is not printed unless there is an error).

You can specify the following arguments:

expr	A byte expression which may be used to encode non-printing control characters, such as ESC.
string	An assembler string, which may be of arbitrary length, up to the maximum assembler-defined limits.

The $PRCTL control can appear anywhere in the source file; the assembler sends out the control string at the corresponding place in the listing file.

If a $PRCTL control is the last line in the last input file to be processed, the assembler insures that all error summaries, symbol tables, and cross-references have been printed before sending out the control string. In this manner, you can use a $PRCTL control to restore a printer to a previous mode after printing is done.

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

Example

```
$PRCTL  $1B,'E'  ; Reset HP LaserJet printer
```

Related Information

Assembler option **--list-file**

$STITLE

Syntax

$STITLE "*string***"**

Default

$STITLE ""

Description

If you generate a list file with the assembler option **--list-file**, you can use the $STITLE control to specify the program subtitle which is printed at the top of all succeeding pages in the assembler list file below the title.

The specified subtitle is valid until the assembler encounters a new $STITLE control. By default, the subtitle is empty.

The $STITLE control itself will not be printed in the source listing.

If the page width is too small for the title to fit in the header, it will be truncated.

Example

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

Related Information

Assembler option **--list-file**

Assembler control **$TITLE**

$TITLE

Syntax

`$TITLE "`*string*`"`

Default

`$TITLE ""`

Description

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

The specified title is valid until the assembler encounters a new $TITLE control. By default, the title is empty.

The $TITLE control itself will not be printed in the source listing.

If the page width is too small for the title to fit in the header, it will be truncated.

Example

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

Related Information

Assembler option **--list-file**

Assembler control **$STITLE**

$WARNING OFF

Syntax

$WARNING OFF [*number*]

Default

All warnings are reported.

Description

This control allows you to disable all or individual warnings. The *number* argument must be a valid warning message number.

Example

```
$WARNING OFF      ; all warning messages are suppressed

$WARNING OFF 135  ; suppress warning message 135
```

Related Information

Assembler option **--no-warnings**

3.10. Macro Operations

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

Some patterns contain variable entries which change for each repetition of the pattern. Others are subject to conditional assembly.

When a macro is called, the assembler executes the macro and replaces the call by the resulting in-line source statements. 'In-line' means that all replacements act as if they are on the same line as the macro call. The generated statements may contain substitutable arguments. The statements produced by a macro can be any processor instruction, almost any assembler directive, or any previously-defined macro. Source statements resulting from a macro call are subject to the same conditions and restrictions as any other statements.

Macros can be nested. The assembler processes nested macros when the outer macro is expanded.

3.10.1. Defining a Macro

The first step in using a macro is to define it.

The definition of a macro consists of three parts:

• *Header*, which assigns a name to the macro and defines the arguments (.MACRO directive).

• *Body*, which contains the code or instructions to be inserted when the macro is called.

• *Terminator*, which indicates the end of the macro definition (.ENDM directive).

A macro definition takes the following form:

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

For more information on the definition see the description of the .MACRO directive.

3.10.2. Calling a Macro

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

```
[label] macro_name [argument[,argument]...]  [; comment]
```

where,

label	An optional label that corresponds to the value of the location counter at the start of the macro expansion.
macro_name	The name of the macro. This may not start in the first column.

argument		One or more optional, substitutable arguments. Multiple arguments must be separated by commas.
comment		An optional comment.

The following applies to macro arguments:

• Each argument must correspond one-to-one with the formal arguments of the macro definition. If the macro call does not contain the same number of arguments as the macro definition, the assembler issues a warning.

• If an argument has an embedded comma or space, you must surround the argument by single quotes (').

• You can declare a macro call argument as null in three ways:

  • enter delimiting commas in succession with no intervening spaces

    ```
    macroname ARG1,,ARG3 ; the second argument is a null argument
    ```

  • terminate the argument list with a comma, the arguments that normally would follow, are now considered null

    ```
    macroname ARG1,      ; the second and all following arguments are null
    ```

  • declare the argument as a null string

• No character is substituted in the generated statements that reference a null argument.

3.10.3. Using Operators for Macro Arguments

The assembler recognizes certain text operators within macro definitions which allow text substitution of arguments during macro expansion. You can use these operators for text concatenation, numeric conversion, and string handling.

Operator	Name	Description
\	Macro argument concatenation	Concatenates a macro argument with adjacent alphanumeric characters.
?	Return decimal value of symbol	Substitutes the **?***symbol* sequence with a character string that represents the decimal value of the symbol.
%	Return hex value of symbol	Substitutes the **%***symbol* 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 single quotes and double quotes to avoid inappropriate expansions. Since `.DEFINE` expansion occurs before macro substitution, any `.DEFINE` symbols are replaced first within a macro argument string:

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

If the macro is called as follows:

```
     STR_MAC  sentence
```

it expands as:

```
     .MESSAGE I 'This is a LONG STRING'
     .MESSAGE I 'This is a short sentence'
```

Macro Local Label Override Operator - ^

If you use labels in macros, the assembler normally generates another unique name for the labels (such as LOCAL__M_L000001).

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

Consider the following macro definition:

```
INIT  .MACRO  addr
LOCAL:  b,^addr
       .ENDM
```

The macro is called as follows:

```
LOCAL:
        INIT LOCAL
```

The macro expands as:

```
LOCAL__M_L000001: b,LOCAL
```

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

```
LOCAL__M_L000001: b,LOCAL__M_L000001
```

3.11. Alias Instructions

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

3.11.1. Branch on Compare Alias Instructions

The assembler supports the following alias instructions of the Branch on Compare (BR) instruction.

Alias Instruction	Replacement
BRGT b,c,s9	BRLT c,b,s9
BRLE b,c,s9	BRGE c,b,s9
BRHI b,c,s9	BRLO c,b,s9
BRLS b,c,s9	BRHS c,b,s9

The following alias instructions have a reduced immediate range of 0 to 62 instead of 0 to 63.

Alias Instruction	Replacement
BRGT b,u6,s9	BRGE b,u6+1,s9
BRLE b,u6,s9	BRLT b,u6+1,s9

Alias Instruction	Replacement
`BRHI b,u6,s9`	`BRHS b,u6+1,s9`
`BRLS b,u6,s9`	`BRLO b,u6+1,s9`

3.11.2. Pop and Push Alias Instructions for Load and Store

The assembler supports the following 32-bit instruction aliases, where `a` refers to the destination register from the core register set.

Alias Instruction	Replacement
`PUSH a`	`ST.AW a, [SP, -4]`
`POP a`	`LD.AB a, [SP,+4]`

3.11.3. Alias Instructions for FCVT32 Encodings

The assembler supports the following instruction aliases for the 32-bit data formats conversion instruction `FCVT32`, based on the encoded values for 6-bit unsigned operand `u6` in the instruction format `FCVT32 a, b, u6`.

Alias Instruction	Replacement	Description
`FS2INT a, b`	`FCVT32 a, b, 0b000011`	Single-precision float to signed integer
`FS2INT_RZ a, b`	`FCVT32 a, b, 0b001011`	Single-precision float to signed integer; round to zero
`FINT2S a, b`	`FCVT32 a, b, 0b000010`	Signed integer to single-precision float
`FS2UINT a, b`	`FCVT32 a, b, 0b000001`	Single-precision float to unsigned integer
`FS2UINT_RZ a, b`	`FCVT32 a, b, 0b001001`	Single-precision float to unsigned integer; round to zero
`FUINT2S a, b`	`FCVT32 a, b, 0b000000`	Unsigned integer to single-precision float
`FH2S a, b`	`FCVT32 a, b, 0b010101`	Half-precision to single-precision conversion
`FS2H a, b`	`FCVT32 a, b, 0b010100`	Single-precision to half-precision conversion

3.11.4. Alias Instructions for FCVT32_64 Encoding

The assembler supports the following instruction aliases for 32-bit data formats to 64-bit data formats conversion instruction `FCVT32_64`, based on the encoded values for 6-bit unsigned operand `u6` in the instruction format `FCVT32_64 a, b, u6`.

Alias Instruction	Replacement	Description
`FS2L a, b`	`FCVT32_64 a, b, 0b000011`	Single-precision float to 64-bit integer
`FS2L_RZ a, b`	`FCVT32_64 a, b, 0b001011`	Single-precision float to 64-bit integer; round to zero
`FS2UL a, b`	`FCVT32_64 a, b, 0b000001`	Single-precision float to 64-bit unsigned integer
`FS2UL_RZ a, b`	`FCVT32_64 a, b, 0b001001`	Single-precision float to 64-bit unsigned integer; round to zero
`FINT2D a, b`	`FCVT32_64 a, b, 0b000010`	32-bit integer to double-precision float
`FUINT2D a, b`	`FCVT32_64 a, b, 0b000000`	32-bit unsigned integer to double-precision float
`FS2D a, b`	`FCVT32_64 a, b, 0b000100`	Single-precision float to double-precision float

3.11.5. Alias Instructions for FCVT64 Encoding

The assembler supports the following instruction aliases for 64-bit data formats to 64-bit data formats conversion instruction `FCVT64`, based on the encoded values for 6-bit unsigned operand `u6` in the instruction format `FCVT64 a, b, u6`.

Alias Instruction	Replacement	Description
`FD2L a, b`	`FCVT64 a, b, 0b000011`	Double-precision float to 64-bit signed integer
`FD2L_RZ a, b`	`FCVT64 a, b, 0b001011`	Double-precision float to 64-bit signed integer; round to zero
`FL2D a, b`	`FCVT64 a, b, 0b000010`	64-bit integer to double-precision float
`FD2UL a, b`	`FCVT64 a, b, 0b000001`	Double-precision float to 64-bit unsigned integer
`FD2UL_RZ a, b`	`FCVT64 a, b, 0b001001`	Double-precision float to 64-bit unsigned integer; round to zero
`FUL2D a, b`	`FCVT64 a, b, 0b000000`	64-bit unsigned integer to double-precision float

3.11.6. Alias Instructions for FCVT64_32 Encoding

The assembler supports the following instruction aliases for 64-bit data formats to 32-bit data formats conversion instruction `FCVT64_32`, based on the encoded values for 6-bit unsigned operand `u6` in the instruction format `FCVT64_32 a, b, u6`.

Alias Instruction	Replacement	Description
FD2INT a, b	FCVT64_32 a, b, 0b000011	Double-precision float to 32-bit integer
FD2INT_RZ a, b	FCVT64_32 a, b, 0b001011	Double-precision float to 32-bit integer; round to zero
FD2UINT a, b	FCVT64_32 a, b, 0b000001	Double-precision float to 32-bit unsigned integer
FD2UINT_RZ a, b	FCVT64_32 a, b, 0b001001	Double-precision float to 32-bit unsigned integer; round to zero
FL2S a, b	FCVT64_32 a, b, 0b000010	64-bit integer to single-precision float
FUL2S a, b	FCVT64_32 a, b, 0b000000	64-bit unsigned integer to double-precision float
FD2S a, b	FCVT64_32 a, b, 0b000100	Double-precision float to single-precision float

3.11.7. Floating-point Absolute Alias Instructions for BCLR Encoding

The assembler supports the following double-precision and single-precision floating-point absolute instruction aliases for BCLR encoding.

Alias Instruction	Replacement	Description
FDABS b, c	BCLR b, c, 0x1F	Double-precision float absolute. b and c refer to register pairs representing 64-bit operands and should be specified as even numbered registers. For example, FDABS r0, r2 and BCLR r0, r2, 0x1F share the same encoding.
FSABS b, c	BCLR b, c, 0x1F	Single-precision float absolute, b and c refer to any two operand registers. For example, FSABS r1, r3 and BCLR r1, r3, 0x1F share the same encoding.

3.11.8. Floating-point Negate Alias Instructions for BXOR Encoding

The assembler supports the following double-precision and single-precision floating-point negate instruction aliases for BXOR encoding.

Alias Instruction	Replacement	Description
`FDNEG b, c`	`BXOR b, c, 0x1F`	Double-precision float absolute. `b` and `c` refer to register pairs representing 64-bit operands and should be specified as even numbered registers. For example, `FDNEG r0, r2` and `BXOR r0, r2, 0x1F` share the same encoding.
`FSNEG b, c`	`BXOR b, c, 0x1F`	Single-precision float absolute, `b` and `c` refer to any two operand registers. For example, `FSNEG r1, r3` and `BXOR r1, r3, 0x1F` share the same encoding.

3.11.9. NOP Alias Instruction for MOV Encoding

The assembler supports the following 32-bit NOP instruction alias for `MOV` encoding.

Alias Instruction	Replacement
`NOP`	`MOV r62, 0`

3.11.10. Vector FPU Alias Instructions

The assembler supports the following vector FPU aliases, where `a` and `b` refer to any two vector register operands:

Alias Instruction	Replacement
`vvfabs.h a, b`	`vvbclr.h a, b, 0xf`
`vvfabs.w a, b`	`vvbclr.w a, b, 0x1f`
`vvfneg.h a, b`	`vvbxor.h a, b, 0xf`
`vvfneg.w a, b`	`vvbxor.w a, b, 0x1f`

Chapter 4. Using the C Compiler

This chapter describes the compilation process and explains how to call the C compiler.

The TASKING toolset for ARC/PPU under Eclipse uses the TASKING makefile generator and make utility to build your entire embedded project, from C source till the final ELF/DWARF object file which serves as input for the debugger.

Although in Eclipse you cannot run the C compiler separately from the other tools, this section discusses the options that you can specify for the C compiler.

On the command line it is possible to call the C compiler separately from the other tools. However, it is recommended to use the control program for command line invocations of the toolset (see Section 8.1, *Control Program*). With the control program it is possible to call the entire toolset with only one command line.

The C compiler takes the following files for input and output:
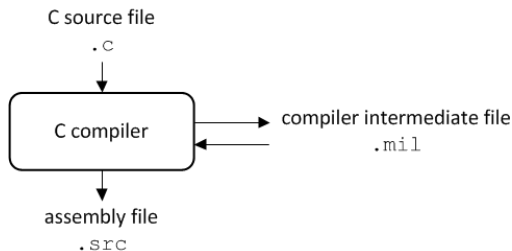
This chapter first describes the compilation process which consists of a *frontend* and a *backend* part. Next it is described how to call the C compiler and how to use its options. An extensive list of all options and their descriptions is included in Section 10.2, *C Compiler Options*. Finally, a few important basic tasks are described, such as including the C startup code and performing various optimizations.

4.1. Compilation Process

During the compilation of a C program, the C compiler runs through a number of phases that are divided into two parts: *frontend* and *backend*.

The backend part is not called for each C statement, but starts after a complete C module or set of modules has been processed by the frontend (in memory). This allows better optimization.

The C compiler requires only one pass over the input file which results in relative fast compilation.

Frontend phases

1. The preprocessor phase:

   The preprocessor includes files and substitutes macros by C source. It uses only string manipulations on the C source. The syntax for the preprocessor is independent of the C syntax but is also described in the ISO C standard.

2. The scanner phase:

The scanner converts the preprocessor output to a stream of tokens.

3. The parser phase:

The tokens are fed to a parser for the C grammar. The parser performs a syntactic and semantic analysis of the program, and generates an intermediate representation of the program. This code is called MIL (Medium level Intermediate Language).

4. The frontend optimization phase:

Target processor independent optimizations are performed by transforming the intermediate code.

Backend phases

1. Instruction selector phase:

This phase reads the MIL input and translates it into Low level Intermediate Language (LIL). The LIL objects correspond to a processor instruction, with an opcode, operands and information used within the C compiler.

2. Peephole optimizer/instruction scheduler/software pipelining phase:

This phase replaces instruction sequences by equivalent but faster and/or shorter sequences, rearranges instructions and deletes unnecessary instructions.

3. Register allocator phase:

This phase chooses a physical register to use for each virtual register. When there are not enough physical registers, virtual registers are spilled to the stack. Intermediate results of any optimization can live, for some time, on the stack or in physical registers.

4. The backend optimization phase:

Performs target processor independent and dependent optimizations which operate on the Low level Intermediate Language.

5. The code generation/formatter phase:

This phase reads through the LIL operations to generate assembly language output.

4.2. Calling the C Compiler

The TASKING toolset for ARC/PPU under Eclipse uses the TASKING makefile generator and make utility to build your entire project. After you have built your project, the output files are available in a subdirectory of your project directory, depending on the active configuration you have set in the **C/C++ Build » Settings** page of the **Project » Properties for** dialog.

Building a project under Eclipse

You have several ways of building your project:

- Build Selected File(s) (). This compiles and assembles the selected file(s) without calling the linker.

  1. In the C/C++ Projects view, select the files you want to compile.

  2. Right-click in the C/C++ Projects view and select **Build Selected File(s)**.

- Build Individual Project ().

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

- Rebuild Project (). This builds every file in the project whether or not a file has been modified since the last build. A rebuild is a clean followed by a build.

  1. Select **Project » Clean...**

  2. Enable the option **Start a build immediately** and click **Clean**.

- Build Automatically. This performs a build of all projects whenever any project file is saved, such as your makefile.

  This way of building is not recommended for C/C++ development, but to enable this feature select **Project » Build Automatically** and ensure there is a check mark beside the **Build Automatically** menu item. In order for this option to work, you must also enable option **Build on resource save (Auto build)** on the **Behavior** tab of the **C/C++ Build** page of the **Project » Properties for** dialog.

See also Chapter 11, *Influencing the Build Time*.

Select a target processor (core)

Processor options affect the invocation of all tools in the toolset. In Eclipse you need to set them for each configuration. Based on the target processor, the compiler includes a special function register file. This is a regular include file which enables you to use virtual registers that are located in memory.

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

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

   The Properties dialog appears.

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

   In the right pane the Processor page appears.

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

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

To access the C compiler options

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

    The Properties dialog appears.

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

    In the right pane the Settings appear.

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

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

5. Select the sub-entries and set the options in the various pages.

> Note that the C compiler options are used to create an object file from a C file. The options you enter
> in the Assembler page are not only used for hand-coded assembly files, but also for intermediate
> assembly files.

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

Invocation syntax on the command line:

carc [[*option*]... [*file*]...]...

You can find a detailed description of all C compiler options in Section 10.2, *C Compiler Options*.

4.3. The C Startup Code

You need the run-time startup code to build an executable application. The startup code consists of the
following components:

- *Initialization code.* This code is executed when the program is initiated and before the function `main()`
  is called.

- *Exit code.* This controls the close down of the application after the program's main function terminates.

The startup code is part of the C library, and the source is present in the file `cstart.c` 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\arch_ppu.lsl`) and you can specify their values on the command line with linker option
--define. If the default run-time startup code does not match your configuration, you need to make a copy
of the startup file, modify it and add it to your project. A typical example for doing this is when `main()`
has arguments, typically argc/argv. In this case cstart needs to be recompiled with the macro
`__USE_ARGC_ARGV` set. When necessary you can use the macro `__ARGCV_BUFSIZE` to define the size
of the buffer used to pass arguments to `main()`.

The entry point of the startup code 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:

- Initialize the stack pointer `sp`. The stack pointer is loaded in memory by the stack address located at linker label `_lc_ub_stack`.

- Initialize the global pointer `gp`. The global pointer is loaded with the address of the Small Data Area (SDA).

- Initialize the vector stack pointer register `r56`. The vector stack pointer is loaded in memory by the vector stack address located at linker label `_lc_ub_vstack`.

- Initialize the registers for vector stack pointer checking. Auxiliary register `VEC_STACK_BASE` points to the first vector memory location above the local memory stack (linker label `_lc_ub_vstack`). `VEC_STACK_TOP` points to the lowest allowed address in the vector memory where the local memory stack pointer can be located (linker label `_lc_ue_vstack`).

- Initialize the vector memory Error Correction Codes by writing the 32-bit value specified by `__ARC_INIT_VAL_VCCM__` to all of the vector memory. When `__ARC_INIT_VAL_VCCM__` is not defined the value 0 is used.

- 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.

- Initialize profiling if profiling is enabled.

- Initialize the `argc` and `argv` arguments.

- 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 a call to the library function `exit()`.

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 compiler option **--define** with the following syntax:

--define=_symbol_[**=**_value_]

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

Define	Description
__PROF_ENABLE__	If defined, initialize profiling.

173

Define	Description
__USE_ARGC_ARGV	If defined, pass arguments to `main`: `int main( int argc, char *argv[] )`.
__ARGCV_BUFSIZE	Define buffer size for `argv`. (default: 256 bytes)
__ARC_DISABLE_VCCM__	If defined, skip vector stack pointer operations.
__ARC_INIT_VAL_VCCM__	If defined, a 32-bit initialization value is used to write to all vector memory, when not defined the value 0 is used.
__ARC_NO_INIT_VCCM__	If defined, skip vector memory Error Correction Code initialization.

4.4. How the Compiler Searches Include Files

When you use include files (with the `#include` statement), you can specify their location in several ways. The compiler searches the specified locations in the following order:

1. If the `#include` statement contains an absolute pathname, the compiler looks for this file. If no path or a relative path is specified, the compiler looks in the same directory as the source file. This is only possible for include files that are enclosed in "".

   This first step is not done for include files enclosed in <>.

2. When the compiler did not find the include file, it looks in the directories that are specified in the **C/C++ Compiler » Include Paths** page in the **C/C++ Build » Settings » Tool Settings** tab of the Project Properties dialog (equivalent to option **--include-directory (-I)**).

3. When the compiler did not find the include file (because it is not in the specified include directory or because no directory is specified), it looks in the path(s) specified in the environment variable `CARCINC`.

4. When the compiler still did not find the include file, it finally tries the default include directory relative to the installation directory (unless you specified option **--no-stdinc**).

Example

Suppose that the C source file `test.c` contains the following lines:

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

You can call the compiler as follows:

```
carc -Imyinclude test.c
```

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

4.5. Compiling for Debugging

Compiling your files is the first step to get your application ready to run on a target. However, during development of your application you first may want to debug your application.

To create an object file that can be used for debugging, you must instruct the compiler to include symbolic debug information in the source file.

To include symbolic debug information

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

    The Properties dialog appears.

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

    In the right pane the Settings appear.

3. On the Tool Settings tab, select **C/C++ Compiler » Debugging**.

4. Select **Default** in the **Generate symbolic debug information** box.

Invocation syntax on the command line

The invocation syntax on the command line is:

```
carc -g file.c
```

Debug and optimizations

Due to different compiler optimizations, it might be possible that certain debug information is optimized away. Therefore, if you encounter strange behavior during debugging it might be necessary to reduce the optimization level, so that the source code is still suitable for debugging. For more information on optimization see Section 4.6, *Compiler Optimizations*.

4.6. Compiler Optimizations

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

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

    The Properties dialog appears.

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

In the right pane the Settings appear.

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

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

5. Select an optimization level in the **Optimization level** box.

    or:

    In the **Optimization level** box select **Custom optimization** and enable the optimizations you want on the Custom optimization page.

Optimization levels

The TASKING C compiler offers four optimization levels and a custom level, at each level a specific set of optimizations is enabled.

*   **Level 0 - No optimization**: No optimizations are performed. The compiler tries to achieve a 1-to-1 resemblance between source code and produced code. Expressions are evaluated in the order written in the source code, associative and commutative properties are not used.

*   **Level 1 - Optimize**: Enables optimizations that do not affect the debug-ability of the source code. Use this level when you encounter problems during debugging your source code with optimization level 2.

*   **Level 2 - Optimize more (default)**: Enables more optimizations to reduce the memory footprint and/or execution time. This is the default optimization level.

*   **Level 3 - Optimize most**: This is the highest optimization level. Use this level when your program/hardware has become too slow to meet your real-time requirements.

*   **Custom optimization**: you can enable/disable specific optimizations on the Custom optimization page.

Optimization pragmas

If you specify a certain optimization, all code in the module is subject to that optimization. Within the C source file you can overrule the C compiler options for optimizations with `#pragma optimize` *flag* and `#pragma endoptimize`. Nesting is allowed:

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

The compiler optimizes the code between the pragma pair as specified.

You can enable or disable the optimizations described in the following subsection. The command line option for each optimization is given in brackets.

4.6.1. Generic Optimizations (frontend)

Common subexpression elimination (CSE) (option -Oc)

The compiler detects repeated use of the same (sub-)expression. Such a "common" expression is replaced by a variable that is initialized with the value of the expression to avoid recomputation. This method is called common subexpression elimination (CSE).

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

Expression simplification (option -Oe)

Multiplication by 0 or 1 and additions or subtractions of 0 are removed. Such useless expressions may be introduced by macros or by the compiler itself (for example, array subscripting).

Constant propagation (option -Op)

A variable with a known value is replaced by that value.

Automatic function inlining (option -Oi)

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

Control flow simplification (option -Of)

A number of techniques to simplify the flow of the program by removing unnecessary code and reducing the number of jumps. For example:

- *Switch optimization*: A number of optimizations of a switch statement are performed, such as removing redundant case labels or even removing an entire switch.

- *Jump chaining*: A (conditional) jump to a label which is immediately followed by an unconditional jump may be replaced by a jump to the destination label of the second jump. This optimization speeds up execution.

- *Conditional jump reversal*: A conditional jump over an unconditional jump is transformed into one conditional jump with the jump condition reversed. This reduces both the code size and the execution time.

- *Dead code elimination*: Code that is never reached, is removed. The compiler generates a warning messages because this may indicate a coding error.

Subscript strength reduction (option -Os)

An array or pointer subscripted with a loop iterator variable (or a simple linear function of the iterator variable), is replaced by the dereference of a pointer that is updated whenever the iterator is updated.

Loop transformations (option -Ol)

Transform a loop with the entry point at the bottom, to a loop with the entry point at the top. This enables constant propagation in the initial loop test and code motion of loop invariant code by the CSE optimization.

Forward store (option -Oo)

A temporary variable is used to cache multiple assignments (stores) to the same non-automatic variable.

Loop auto-vectorization (option -Om)

The auto-vectorization optimization transforms specific types of loops that sequentially process elements of one of more arrays to an equivalent loop that uses vector operations to process multiple array elements per iteration. This optimization is limited to "well-behaved" loops for which the iteration count can be computed beforehand, either at compile-time or at run-time. Additional code is generated to process the final partial vector when the number of iterations of the original loop is not an exact multiple of the vector size. Note that auto-vectorization is restricted to arrays allocated in __vccm memory.

Example:

```
#define N 100

__vccm int a[N], b[N], c[N];

void ex1(void)
{
        for (int i = 0; i < N; i++)
        {
                a[i] = b[i] * c[i];
        }
}
```

The auto-vectorization optimization is disabled by default, so you have to enable it explicitly. The option **--vectorize-info** enables some informational messages about the auto-vectorization:

```
carc -Om --vectorize-info ex1.c
carc I811: ["ex1.c" 7/35] vectorize: rewriting loop with vector size 16,
          6 iteration(s), 4 remaining element(s)
```

When the arrays are accessed via pointers, there may be aliases that would invalidate vectorization. You can use the `restrict` pointer qualifier to specify that there are no aliases:

```
void ex2(__vccm float* restrict a, __vccm float* restrict b,
         __vccm float* restrict c, int n)
{
        for (int i = 0; i < n; i++)
```

```
        {
                a[i] = b[i] + c[i] + 1.0;
        }
}

carc -Om --vectorize-info ex2.c
carc I811: ["ex2.c" 3/37] vectorize: rewriting loop with vector size 16,
          non-constant number of iterations
```

Alternatively, you can disable all alias checking for auto-vectorization with the option **--vectorize-noalias**.

In some situations, two nested loops processing a matrix will be flattened into a single loop before the auto-vectorization transforms the resulting loop to vector operations:

```
#define V 8
#define H 8

__vccm int a[V][H], b[V][H], c[V][H];

void ex3(int x)
{
        for (int v = 0; v < V; v++)
        {
                for (int h = 0; h < H; h++)
                {
                        a[v][h] = b[v][h] * c[v][h];
                }
        }
}

carc -Om --vectorize-info ex3.c
carc I811: ["ex3.c" 8/35] vectorize: flattening two nested loops into
          a single loop
carc I811: ["ex3.c" 10/43] vectorize: rewriting loop with vector size 16,
          4 iteration(s), no remainder
```

Control flow such as an if-statement inside the loop would normally prevent auto-vectorization, but in some cases the loop body can be converted to straight-line code with predicated vector stores, where the condition is used to calculate a predicate vector:

```
#define N 64

__vccm int a[N], b[N];

void ex4(int x)
{
        for (int i = 0; i < N; i++)
        {
                if (b[i] > 0)
                {
                        a[i] /= b[i];
```

```
                }
        }
}

carc -Om --vectorize-info ex4.c
carc I811: ["ex4.c" 7/35] vectorize: rewriting loop with vector size 16,
          4 iteration(s), no remainder
carc I811: ["ex4.c" 9/26] vectorize: conditional code transformed to
          predicated stores
```

When possible, consecutive loops with the same iterator range will be combined into a single loop before vectorization:

```
#define N 64

__vccm int a[N], b[N], q[N], r[N];

void ex5(void)
{
        for (int i = 0; i < N; i++)
        {
                q[i] = a[i] / b[i];
        }
        for (int j = 0; j < N; j++)
        {
                r[j] = a[j] % b[j];
        }
}

carc -Om --vectorize-info ex5.c
carc I811: ["ex5.c" 11/35] vectorize: fusing loop with preceding loop
carc I811: ["ex5.c" 7/35] vectorize: rewriting loop with vector size 16,
          4 iteration(s), no remainder
```

In some cases, a reduction loop may be vectorized by using vector reduction instructions:

```
#define N 64

__vccm float a[N], b[N];

float dot(void)
{
        float sum = 0.0f;

        for (int i = 0; i < N; i++)
        {
                sum += a[i] * b[i];
        }
        return sum;
}
```

```
carc -Om --vectorize-info dot.c
carc I811: ["dot.c" 9/35] vectorize: rewriting loop with vector size 16,
         4 iteration(s), no remainder
```

Vectorization may also be possible for loops containing math functions for which a corresponding vector instruction is available:

```
#include <math.h>

#define N 64

__vccm float a[N], b[N];

void rsqrt(void)
{
        for (int i = 0; i < N; i++)
        {
                a[i] = 1.0f / sqrtf(b[i]);
        }
}
```

```
carc -Om --vectorize-info -Om rsqrt.c -n
carc I811: ["rsqrt.c" 9/28] vectorize: rewriting loop with vector size 16,
         4 iteration(s), no remainder
...
        vvld.av.w %vr0,[%r0,1]
        vvfrsqrt.w %vr0,%vr0
        vvst.av.w %vr0,[%r1,1]
...
```

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

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

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

**181**
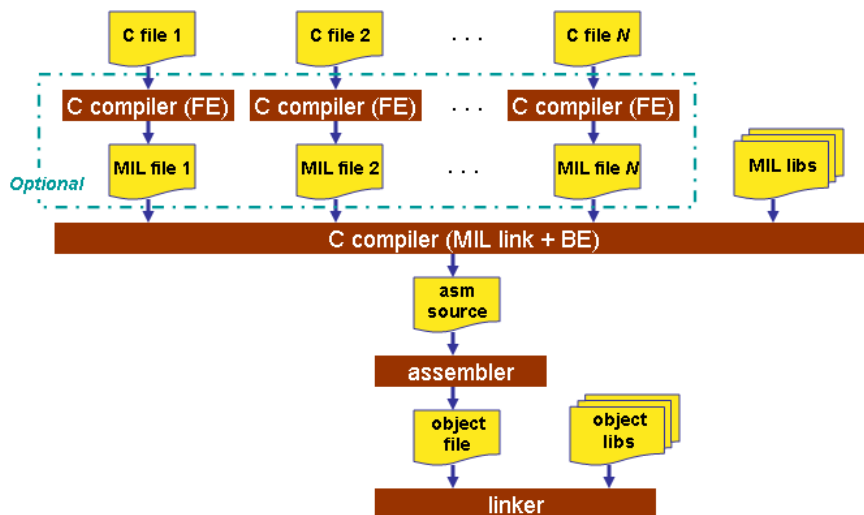
## MIL splitting (option --mil-split)

When you specify that the C compiler has to use MIL splitting, the C compiler will first link the application at MIL level as described above. However, after rerunning the optimizations the MIL code is not passed on to the backend. Instead the frontend writes a `.ms` file for each input file or library. A `.ms` file has the same format as a `.mil` file. Only `.ms` files that really change are updated. The advantage of this approach is that it is possible to use the make utility to translate only those parts of the application to a `.src` file that really have changed. MIL splitting is therefore a more efficient build process than MIL linking. The penalty for this is that the code compaction optimization in the backend does not have application scope. As with MIL linking, it is still required to link with the normal libraries to build an ELF file.
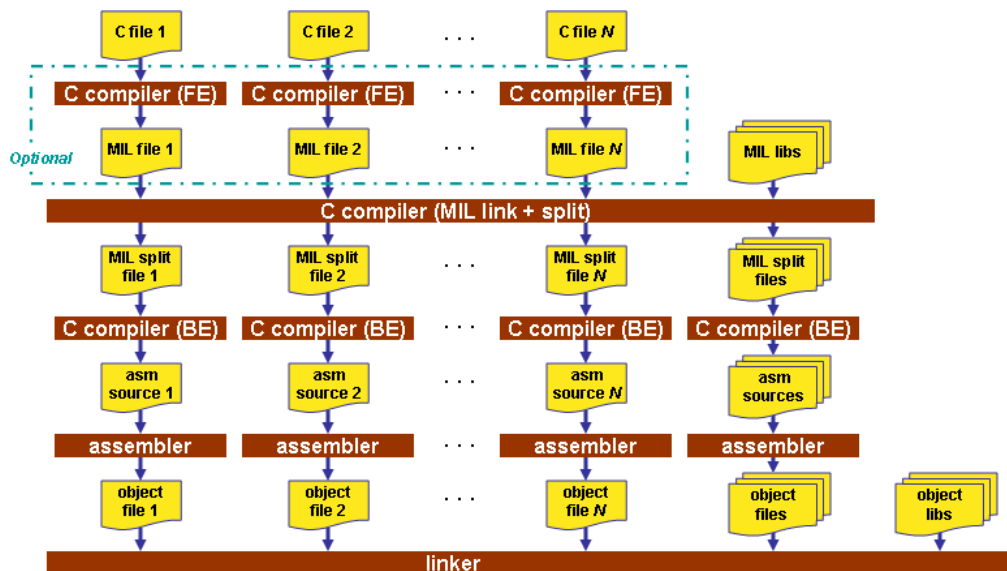
To read more about how MIL linking influences the build process of your application, see Section 11.2, *MIL Linking*.

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

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

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

4.6.2. Core Specific Optimizations (backend)

Coalescer (option -Oa)

The coalescer seeks for possibilities to reduce the number of moves (MOV instruction) by smart use of registers. This optimizes both speed and code size.

Interprocedural register optimization (option -Ob)

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

Peephole optimizations (option -Oy)

The generated assembly code is improved by replacing instruction sequences by equivalent but faster and/or shorter sequences, or by deleting unnecessary instructions.

Instruction Scheduler (option -Ok)

The instruction scheduler is a backend optimization that acts upon the generated instructions. When the processor executes instructions, pipeline hazards might occur which will stall the pipeline. There are different types of pipeline hazards. For example a *data hazard* occurs when an instruction waits for a result of a long-latency instruction, such as a division. When two instructions need the same machine resource - like a bus, register or functional unit - at the same time, they suffer a *structural hazard*. This optimization tries to rearrange instructions to avoid pipeline hazards, for example by inserting another non-related instruction.

Another important job of the instruction scheduler is auto-bundling. The processor can issue several vector instructions simultaneously in groups called bundles, which contain up to three vector instructions and an optional scalar instruction. This optimization tries to form the instruction bundles to minimize the amount of processor cycles required.

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

stalls and combining vector instructions into bundles. In this process data dependencies between instructions are honored.

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

Unroll small loops (option -Ou)

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

IF conversion (option -Ov)

IF - ELSE constructions are transformed into predicated instructions. This avoids unnecessary jumps and allows other optimizations to be applied.

Note that this option is only activated when Generic assembly optimizations are enabled (option **-Og**).

Software pipelining (option -Ow)

A number of techniques to optimize loops. For example, within a loop the most efficient order of instructions is chosen by the *pipeline scheduler* and it is examined what instructions can be executed in parallel.

Please note that this option does not influence modulo scheduling. See Section 4.6.3, *Modulo Scheduling*.

Code compaction (reverse inlining) (option -Or)

Compaction is the opposite of inlining functions: chunks of code that occur more than once, are transformed into a function. This reduces code size at the cost of execution speed. The size of the chunks of code to be inlined depends on the setting of the C compiler option **--tradeoff** (**-t**). See the subsection **Code Compaction** in Section 4.6.4, *Optimize for Code Size or Execution Speed*.

Generic assembly optimizations (option -Og)

A set of optimizations on the generated assembly code that increase speed and decrease code size, similar to peephole optimizations applied within and across basic blocks. The set includes but is not limited to:

- removal of unused code

- removal of superfluous code

- loop optimizations

- flow optimizations

- load/store optimizations

- addressing mode optimizations

4.6.3. Modulo Scheduling

To software pipeline vector loops, you can use modulo scheduling, a software pipelining technique that overlaps successive iterations of the loop, thereby improving resource utilization. The goal is to enhance instruction-level parallelism (ILP) and increase overall throughput. Typically, a higher number of overlapping iterations results in better ILP.

The number of overlapping iterations that can be achieved depends on the loop's data dependencies, as well as the resource utilization of the instructions. The scheduler attempts to find the best schedule within these constraints; however, loop transformation is not always guaranteed. When the loop is transformed, the optimization generates Prologue and Epilogue sections in addition to the loop Kernel. The size of the transformed loop (i.e., Kernel, Prologue, and Epilogue) is proportional to the number of overlapping iterations that the scheduler achieves.

Note that since modulo scheduling is an aggressive software pipelining optimization, a higher number of overlapping iterations can lead to increased code size, register spilling, or cache misses. Therefore, you need to take care when applying modulo scheduling to large loops with high register requirements.

To enable modulo scheduling and control its aggressiveness, you can use `#pragma modulo_schedule` *N*, where *N* is the maximum number of overlapping iterations allowed. You can also use `#pragma end_modulo_schedule` to disable the modulo scheduling.

Please note that modulo scheduling is not influenced by the global software pipelining option (**-Ow**).

Consider the following example where all loops are small, the inner-most vector loops have a constant iteration count, and no control flow.

```
void func()
{
    #pragma modulo_schedule 1
    // loop a{...}
    #pragma end_modulo_schedule

    #pragma modulo_schedule 3
    // loop b{...}
    // loop c{...}
    #pragma end_modulo_schedule

    #pragma modulo_schedule 0
    // loop d{...}
    #pragma end_modulo_schedule

    // loop e{...}
}
```

Here, the modulo scheduler attempts to schedule loop `a` with a number of overlapping iterations not exceeding 1. Meanwhile, it schedules loops `b` and `c` more aggressively. Loop `d` may not be pipelined as its iterations are not allowed to overlap. Loop `e` may not be pipelined by the modulo scheduler, but may be scheduled by the traditional software pipeliner depending on the global options.

Use this optimization with caution. In some cases, the performance may slowdown compared to the traditional software pipelining. When this happens, you can either (a) reduce the aggressiveness of the modulo scheduling by trying a lower parameter value, or (b) exclude the loop from modulo scheduling by removing the pragma.

Note that depending on the slowdown reason, option (a) may still result in better performance than option (b).

4.6.4. Optimize for Code Size or Execution Speed

You can tell the compiler to focus on execution speed or code size during optimizations. You can do this by specifying a size/speed trade-off level from 0 (speed) to 4 (size). This trade-off does not turn optimization phases on or off. Instead, its level is a weight factor that is used in the different optimization phases to influence the heuristics. The higher level, the more the compiler focusses on code size optimization. To choose a trade-off value read the description below about which optimizations are affected and the impact of the different trade-off values.

Note that the trade-off settings are directions and there is no guarantee that these are followed. The compiler may decide to generate different code if it assessed that this would improve the result.

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

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

    The Properties dialog appears.

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

    In the right pane the Settings appear.

3. On the Tool Settings tab, select **C/C++ Compiler » Optimization**.

4. Select a trade-off level in the **Trade-off between speed and size** box.

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

Instruction Selection

Trade-off levels 0, 1 and 2: the compiler selects the instructions with the smallest number of cycles.

Trade-off levels 3 and 4: the compiler selects the instructions with the smallest number of bytes.

Subscript Strength Reduction

Subscript strength reduction is turned off by default, because it is not possible for the ARC/PPU to automatically determine if it improves the generated code.

The total number of additional pointers of a particular type in a particular loop is limited to 4 for the ARC/PPU.

The performance increases when more subscript pointers can be allocated for an ideal situation. Ideal is when no registers are needed for other objects than subscripts. This is rarely the case, therefore the maximum number of registers is set to 4 GPRs.

Automatic Function Inlining

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

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

Trade-off value	inline-max-incr	inline-max-size
0	999	50
1	50	25
2	20	20
3	10	10
4	0	0

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

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

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

Loop Optimization

For a top-loop, the loop is entered at the top of the loop. A bottom-loop is entered at the bottom. Every loop has a test and a jump at the bottom of the loop, otherwise it is not possible to create a loop. Some top-loops also have a conditional jump before the loop. This is only necessary when the number of loop iterations is unknown. The number of iterations might be zero, in this case the conditional jump jumps over the loop.

Bottom loops always have an unconditional jump to the loop test at the bottom of the loop.

Trade-off value	Try to rewrite top-loops to bottom-loops (when peephole optimization is off -OY)	Optimize loops for size/speed
0	no	speed
1	yes	speed
2	yes	speed
3	yes	size

Trade-off value	Try to rewrite top-loops to bottom-loops (when peephole optimization is off -OY)	Optimize loops for size/speed
4	yes	size

Example:

```
int a;

void i( int l, int m )
{
    int i;

    for ( i = m; i < l; i++ )
    {
        a++;
    }
    return;
}
```

Coded as a bottom loop (compiled with **-OIY --tradeoff=4**) is:

```
      ld.as   %r2,[%gp,@sda(a)]
      b       .L2           ;; unconditional jump to loop test at bottom
.L3:
      add     %r2,%r2,1
      add     %r1,%r1,1
.L2:                        ;; loop entry point
      cmp     %r1,%r0
      blt     .L3
      st.as   %r2,[%gp,@sda(a)]
      j_s     [%blink]
```

Coded as a top loop (compiled with **-OIY --tradeoff=0**) is:

```
      ld.as   %r2,[%gp,@sda(a)]
      cmp     %r1,%r0    ;; test for at least one loop iteration
      bge     .L2        ;; can be omitted when number of iterations is known
.L3:                     ;; loop entry point
      add     %r2,%r2,1
      add     %r1,%r1,1
      cmp     %r1,%r0
      blt     .L3
.L2:
      st.as   %r2,[%gp,@sda(a)]
      j_s     [%blink]
```

Code Compaction

Trade-off levels 0 and 1: code compaction is disabled.

Trade-off level 2: only code compaction of matches outside loops.

Trade-off level 3: code compaction of matches outside loops, and matches inside loops of patterns that have an estimate execution frequency lower or equal to 10.

Trade-off level 4: code compaction of matches outside loops, and matches inside loops of patterns that have an estimate execution frequency lower or equal to 100.

For loops where the iteration count is unknown an iteration count of 10 is assumed.

For the execution frequency the compiler also accounts nested loops.

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

4.7. Static Code Analysis

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

SCA Implementation Design Philosophy

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

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

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

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

  For example see the following code fragment:

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

    for (i = 0; i <= 10; i++)
    {
        if (some_condition(i))
        {
            buf[i] = 0; /* subscript may be out of bounds */
        }
    }
}
```

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

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

Effect of optimization level on SCA results

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

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

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

4.7.1. C Code Checking: CERT C

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

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

Versions of the CERT C standard

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

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

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

For a complete overview of the supported CERT C recommendations/rules by the TASKING compiler, see Chapter 16, *CERT C Secure Coding Standard*.

Priority and Levels of CERT C

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

- severity - how serious are the consequences of the rule being ignored

    1. low (denial-of-service attack, abnormal termination)

    2. medium (data integrity violation, unintentional information disclosure)

    3. high (run arbitrary code)

- likelihood - how likely is it that a flaw introduced by ignoring the rule could lead to an exploitable vulnerability

    1. unlikely

    2. probable

    3. likely

- remediation cost - how expensive is it to comply with the rule

    1. high (manual detection and correction)

    2. medium (automatic detection and manual correction)

    3. low (automatic detection and correction)

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

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

For a complete overview of the supported CERT C recommendations/rules by the TASKING compiler, see Chapter 16, *CERT C Secure Coding Standard*.

To apply CERT C code checking to your application

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

    The Properties dialog appears.

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

    In the right pane the Settings appear.

3. On the Tool Settings tab, select **C/C++ Compiler » CERT C Secure Coding**.

4. Make a selection from the **CERT C secure code checking** list.

5.   If you selected **Custom**, expand the **Custom CERT C** entry and enable one or more individual recommendations/rules.

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

```
carc --cert={all | name [-name],...]
```

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

4.7.2. C Code Checking: MISRA C

The C programming language is a standard for high level language programming in embedded systems, yet it is considered somewhat unsuitable for programming safety-related applications. Through enhanced code checking and strict enforcement of best practice programming rules, TASKING MISRA C code checking helps you to produce more robust code.

MISRA C specifies a subset of the C programming language which is intended to be suitable for embedded automotive systems. It consists of a set of rules, defined in *MISRA-C:2012, Guidelines for the Use of the C Language in Critical Systems* (MIRA Limited, 2013).

The compiler also supports MISRA C:1998, the first version of MISRA C and MISRA C: 2004. You can select the version with the following C compiler option:

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

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

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

For a complete overview of all MISRA C rules, see Chapter 17, *MISRA C Rules*.

Implementation issues

The MISRA C implementation in the compiler supports nearly all rules. Only a few rules are not supported because they address documentation, run-time behavior, or other issues that cannot be checked by static source code inspection, or because they require an application-wide overview.

During compilation of the code, violations of the enabled MISRA C rules are indicated with error messages and the build process is halted.

MISRA C rules are divided in mandatory rules, required rules and advisory rules. If rules are violated, errors are generated causing the compiler to stop. With the following options warnings, instead of errors, are generated:

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

Note that not all MISRA C violations will be reported when other errors are detected in the input source. For instance, when there is a syntax error, all semantic checks will be skipped, including some of the MISRA C checks. Also note that some checks cannot be performed when the optimizations are switched off.

Quality Assurance report

To ensure compliance to the MISRA C rules throughout the entire project, the TASKING linker can generate a MISRA C Quality Assurance report. This report lists the various modules in the project with the respective MISRA C settings at the time of compilation. You can use this in your company's quality assurance system to provide proof that company rules for best practice programming have been applied in the particular project.

To apply MISRA C code checking to your application

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

    The Properties dialog appears.

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

    In the right pane the Settings appear.

3. On the Tool Settings tab, select **C/C++ Compiler » MISRA C**.

4. Select the **MISRA C version** (1998, 2004 or 2012).

5. In the **MISRA C checking** box select a MISRA C configuration. Select a predefined configuration for conformance with the required rules in the MISRA C guidelines.

6. (Optional) In the **Custom 1998**, **Custom 2004** or **Custom 2012** entry, specify the individual rules.

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

```
carc --misrac={all | number [-number],...]
```

4.8. C Compiler Error Messages

The C compiler reports the following types of error messages in the Problems view of Eclipse.

193

F (Fatal errors)

After a fatal error the compiler immediately aborts compilation.

E (Errors)

Errors are reported, but the compiler continues compilation. No output files are produced unless you have set the C compiler option **--keep-output-files** (the resulting output file may be incomplete).

W (Warnings)

Warning messages do not result into an erroneous assembly output file. They are meant to draw your attention to assumptions of the compiler for a situation which may not be correct. You can control warnings in the **C/C++ Build » Settings » Tool Settings » C/C++ Compiler » Diagnostics** page of the **Project » Properties for** menu (C compiler option **--no-warnings**).

I (Information)

Information messages are always preceded by an error message. Information messages give extra information about the error.

S (System errors)

System errors occur when internal consistency checks fail and should never occur. When you still receive the system error message

```
S9##: internal consistency check failed - please report
```

please report the error number and as many details as possible about the context in which the error occurred.

Display detailed information on diagnostics

1. From the **Window** menu, select **Show View » Other » TASKING » Problems**.

   The Problems view is added to the current perspective.

2. In the Problems view right-click on a message.

   A popup menu appears.

3. Select **Detailed Diagnostics Info**.

   A dialog box appears with additional information.

On the command line you can use the C compiler option **--diag** to see an explanation of a diagnostic message:

```
carc --diag=[format:]{all | number,...]
```

Chapter 5. Using the C++ Compiler

This chapter describes the compilation process and explains how to call the C++ compiler. You should be familiar with the C++ language and with the ISO C language.

The C++ compiler can be seen as a preprocessor or front end which accepts C++ source files or sources using C++ language features. The output generated by the ARC C++ compiler (**cparc**) is intermediate C, which can be translated with the ARC C compiler (**carc**).

The C++ compiler is part of a complete toolset, the TASKING toolset for ARC/PPU. For details about the C compiler see Chapter 4, *Using the C Compiler*.

The C++ compiler takes the following files for input and output:
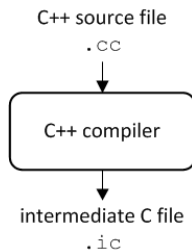
C++ source file
.cc

C++ compiler

intermediate C file
.ic

Although in Eclipse you cannot run the C++ compiler separately from the other tools, this section discusses the options that you can specify for the C++ compiler.

On the command line it is possible to call the C++ compiler separately from the other tools. However, it is recommended to use the control program for command line invocations of the toolset (see Section 8.1, *Control Program*). With the control program it is possible to call the entire toolset with only one command line. Eclipse also uses the control program to call the C++ compiler. Files with the extensions `.C`, `.cc`, `.cpp` or `.cxx` are seen as C++ source files and passed to the C++ compiler.

The C++ compiler accepts the C++ language of the ISO/IEC 14882:2014 C++ standard, with some minor exceptions documented in Chapter 2, *C++ Language*. It also accepts embedded C++ language extensions.

The C++ compiler does no optimization. Its goal is to produce quickly a complete and clean parsed form of the source program, and to diagnose errors. It does complete error checking, produces clear error messages (including the position of the error within the source line), and avoids cascading of errors. It also tries to avoid seeming overly finicky to a knowledgeable C or C++ programmer.

5.1. Calling the C++ Compiler

Under Eclipse you cannot run the C++ compiler separately. However, you can set options specific for the C++ compiler. After you have built your project, the output files are available in a subdirectory of your project directory, depending on the active configuration you have set in the **C/C++ Build » Settings** page of the **Project » Properties for** dialog.

Building a project under Eclipse

You have several ways of building your project:

- Build Selected File(s) (). This compiles and assembles the selected file(s) without calling the linker.

    1. In the C/C++ Projects view, select the files you want to compile.

    2. Right-click in the C/C++ Projects view and select **Build Selected File(s)**.

- Build Individual Project ().

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

- Rebuild Project (). This builds every file in the project whether or not a file has been modified since the last build. A rebuild is a clean followed by a build.

    1. Select **Project » Clean...**

    2. Enable the option **Start a build immediately** and click **Clean**.

- Build Automatically. This performs a build of all projects whenever any project file is saved, such as your makefile.

    This way of building is not recommended for C/C++ development, but to enable this feature select **Project » Build Automatically** and ensure there is a check mark beside the **Build Automatically** menu item. In order for this option to work, you must also enable option **Build on resource save (Auto build)** on the **Behavior** tab of the **C/C++ Build** page of the **Project » Properties for** dialog.

Select a target processor (core)

Processor options affect the invocation of all tools in the toolset. In Eclipse you need to set them for each configuration. Based on the target processor, the compiler includes a special function register file. This is a regular include file which enables you to use virtual registers that are located in memory.

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

    The Properties dialog appears.

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

    In the right pane the Processor page appears.

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

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

To access the C/C++ compiler options

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

    The Properties dialog appears.

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

   In the right pane the Settings appear.

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

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

5. Select the sub-entries and set the options in the various pages.

Note that C++ compiler options are only enabled if you have added a C++ file to your project, a file with the extension `.cc`, `.cpp` or `.cxx`.

Note that the options you enter in the Assembler page are also used for intermediate assembly files.

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

Invocation syntax on the command line:

cparc [[*option*]... [*file*]...]...

You can find a detailed description of all C++ compiler options in Section 10.3, *C++ Compiler Options*.

5.2. How the C++ Compiler Searches Include Files

When you use include files (with the `#include` statement), you can specify their location in several ways. The C++ compiler searches the specified locations in the following order:

1. If the `#include` statement contains an absolute pathname, the C++ compiler looks for this file. If no path or a relative path is specified, the C++ compiler looks in the same directory as the source file. This is only possible for include files that are enclosed in "".

This first step is not done for include files enclosed in <>.

---

2. When the C++ compiler did not find the include file, it looks in the directories that are specified in the **C/C++ Compiler » Include Paths** page in the **C/C++ Build » Settings » Tool Settings** tab of the Project Properties dialog (equivalent to the C++ compiler option **--include-directory (-I)**).

3. When the C++ compiler did not find the include file (because it is not in the specified include directory or because no directory is specified), it looks in the path(s) specified in the environment variable `CPARCINC`.

4. When the C++ compiler still did not find the include file, it finally tries the default `include.cpp` and `include` directory relative to the installation directory.

5. If the include file is still not found, the directories specified in the **--sys-include** option are searched.

If the include directory is specified as **-**, e.g., **-I-**, the option indicates the point in the list of **-I** or **--include-directory** options at which the search for file names enclosed in < . . . > should begin. That is, the search for < . . . > names should only consider directories named in **-I** or **--include-directory** options following the **-I-**, and the directories of items 3 and 4 above. **-I-** also removes the directory containing the current input file (item 1 above) from the search path for file names enclosed in " . . . ".

An include directory specified with the **--sys-include** option is considered a "system" include directory. Warnings are suppressed when processing files found in system include directories.

If the filename has no suffix it will be searched for by appending each of a set of include file suffixes. When searching in a given directory all of the suffixes are tried in that directory before moving on to the next search directory. The default set of suffixes is, no extension and `.stdh`.

Example

Suppose that the C++ source file `test.cc` contains the following lines:

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

You can call the C++ compiler as follows:

```
cparc -Imyinclude test.cc
```

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

The C++ compiler now looks for the file `myinc.h`, in the directory where `test.cc` is located. If the file is not there the C++ compiler searches in the directory `myinclude`. If it was still not found, the C++ compiler searches in the environment variable `CPARCINC` and then in the default `include.cpp` and `include` directories.

5.3. C++ Compiler Error Messages

The C++ compiler reports the following types of error messages in the Problems view of Eclipse.

F (Fatal errors)

Catastrophic errors, also called 'fatal errors', indicate problems of such severity that the compilation cannot continue. For example: command-line errors, internal errors, and missing include files. If multiple source files are being compiled, any source files after the current one will not be compiled.

E (Errors)

Errors indicate violations of the syntax or semantic rules of the C++ language. Compilation continues, but object code is not generated.

W (Warnings)

Warnings indicate something valid but questionable. Compilation continues and object code is generated (if no errors are detected). You can control warnings in the **C/C++ Build » Settings » Tool Settings » C/C++ Compiler » Diagnostics** page of the **Project » Properties for** menu (C++ compiler option **--no-warnings**).

R (Remarks)

Remarks indicate something that is valid and probably intended, but which a careful programmer may want to check. These diagnostics are not issued by default. Compilation continues and object code is generated (if no errors are detected). To enable remarks, enable the option **Issue remarks on C++ code** in the **C/C++ Build » Settings » Tool Settings » C/C++ Compiler » Diagnostics** page of the **Project » Properties for** menu (C++ compiler option **--remarks**).

S (Internal errors)

Internal compiler errors are caused by failed internal consistency checks and should never occur. However, if such a 'SYSTEM' error appears, please report the occurrence to TASKING. Please include a small C++ program causing the error.

Message format

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

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

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

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

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

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

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

```
cparc E0308: ["test.cc" 4] more than one instance of overloaded
        function "f" matches the argument list:
          function "f(int)"
          function "f(float)"
          argument types are: (double)
```

In some cases, some additional context information is provided; specifically, such context information is useful when the C++ compiler issues a diagnostic while doing a template instantiation or while generating a constructor, destructor, or assignment operator function. For example:

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

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

Termination Messages

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

```
n errors detected in the compilation of "file".

1 catastrophic error detected in the compilation of "file".

n errors and 1 catastrophic error detected in the compilation of "file".
```

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

```
Error limit reached.
```

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

```
Compilation terminated.
```

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

```
Compilation aborted
```

is written at the end of a compilation that was prematurely terminated because of an internal error. Such an error indicates an internal problem in the compiler. If such an internal error appears, please report the occurrence to TASKING. Please include a small C++ program causing the error.

Chapter 6. Using the Assembler

This chapter describes the assembly process and explains how to call the assembler.

The assembler converts hand-written or compiler-generated assembly language programs into machine language, resulting in object files in the ELF/DWARF object format.

The assembler takes the following files for input and output:
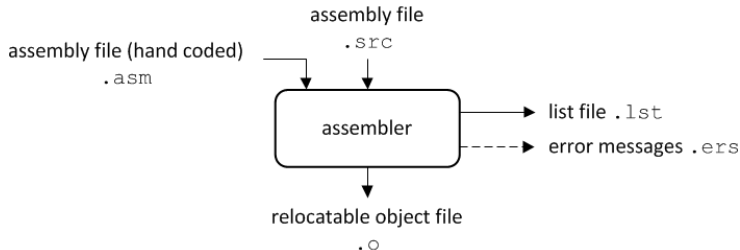
The following information is described:

- The assembly process.

- How to call the assembler and how to use its options. An extensive list of all options and their descriptions is included in Section 10.4, *Assembler Options*.

- How to generate a list file.

- Types of assembler messages.

6.1. Assembly Process

The assembler generates relocatable output files with the extension `.o`. These files serve as input for the linker.

Phases of the assembly process

- Parsing of the source file: preprocessing of assembler directives and checking of the syntax of instructions

- Instruction grouping and reordering

- Generation of the relocatable object file and optionally a list file

The assembler integrates file inclusion and macro facilities. See Section 3.10, *Macro Operations* for more information.

6.2. Calling the Assembler

The TASKING toolset for ARC/PPU under Eclipse uses the TASKING makefile generator and make utility to build your entire project. After you have built your project, the output files are available in a subdirectory of your project directory, depending on the active configuration you have set in the **C/C++ Build » Settings** page of the **Project » Properties for** dialog.

Building a project under Eclipse

You have several ways of building your project:

- Build Selected File(s) (). This compiles and assembles the selected file(s) without calling the linker.

    1. In the C/C++ Projects view, select the files you want to compile.

    2. Right-click in the C/C++ Projects view and select **Build Selected File(s)**.

- Build Individual Project ().

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

- Rebuild Project (). This builds every file in the project whether or not a file has been modified since the last build. A rebuild is a clean followed by a build.

    1. Select **Project » Clean...**

    2. Enable the option **Start a build immediately** and click **Clean**.

- Build Automatically. This performs a build of all projects whenever any project file is saved, such as your makefile.

    This way of building is not recommended for C/C++ development, but to enable this feature select **Project » Build Automatically** and ensure there is a check mark beside the **Build Automatically** menu item. In order for this option to work, you must also enable option **Build on resource save (Auto build)** on the **Behavior** tab of the **C/C++ Build** page of the **Project » Properties for** dialog.

Select a target processor (core)

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

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

    The Properties dialog appears.

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

    In the right pane the Processor page appears.

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

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

To access the assembler options

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

    The Properties dialog appears.

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

    In the right pane the Settings appear.

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

4. On the Tool Settings tab, select **Assembler**.

5. Select the sub-entries and set the options in the various pages.

> Note that the options you enter in the Assembler page are not only used for hand-coded assembly files, but also for the assembly files generated by the compiler.

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

Invocation syntax on the command line:

asarc [[*option*]... [*file*]...]...

The input file must be an assembly source file (.asm or .src).

You can find a detailed description of all assembler options in Section 10.4, *Assembler Options*.

6.3. How the Assembler Searches Include Files

When you use include files (with the .INCLUDE directive), you can specify their location in several ways. The assembler searches the specified locations in the following order:

1. If the .INCLUDE directive contains an absolute path name, the assembler looks for this file. If no path or a relative path is specified, the assembler looks in the same directory as the source file.

2. When the assembler did not find the include file, it looks in the directories that are specified in the **Assembler » Include Paths** page in the **C/C++ Build » Settings » Tool Settings** tab of the Project Properties dialog (equivalent to option **--include-directory (-I)**).

3. When the assembler did not find the include file (because it is not in the specified include directory or because no directory is specified), it looks in the path(s) specified in the environment variable ASARCINC.

4. When the assembler still did not find the include file, it finally tries the default include directory relative to the installation directory.

Example

Suppose that the assembly source file `test.asm` contains the following lines:

```
.INCLUDE 'myinc.inc'
```

You can call the assembler as follows:

```
asarc -Imyinclude test.asm
```

First the assembler looks for the file `myinc.asm`, in the directory where `test.asm` is located. If the file is not there the assembler searches in the directory `myinclude`. If it was still not found, the assembler searches in the environment variable `ASARCINC` and then in the default `include` directory.

6.4. Generating a List File

The list file is an additional output file that contains information about the generated code. You can customize the amount and form of information.

If the assembler generates errors or warnings, these are reported in the list file just below the source line that caused the error or warning.

To generate a list file

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

    The Properties dialog appears.

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

    In the right pane the Settings appear.

3. On the Tool Settings tab, select **Assembler » List File**.

4. Enable the option **Generate list file**.

5. (Optional) Enable the options to include that information in the list file.

Example to generate a list file on the command line

The following command generates the list file `test.lst`:

```
asarc -l test.asm
```

See Section 13.1, *Assembler List File Format*, for an explanation of the format of the list file.

6.5. Assembler Error Messages

The assembler reports the following types of error messages in the Problems view of Eclipse.

F (Fatal errors)

After a fatal error the assembler immediately aborts the assembly process.

E (Errors)

Errors are reported, but the assembler continues assembling. No output files are produced unless you have set the assembler option **--keep-output-files** (the resulting output file may be incomplete).

W (Warnings)

Warning messages do not result into an erroneous assembly output file. They are meant to draw your attention to assumptions of the assembler for a situation which may not be correct. You can control warnings in the **C/C++ Build » Settings » Tool Settings » Assembler » Diagnostics** page of the **Project » Properties for** menu (assembler option **--no-warnings**).

Display detailed information on diagnostics

1. From the **Window** menu, select **Show View » Other » TASKING » Problems**.

    The Problems view is added to the current perspective.

2. In the Problems view right-click on a message.

    A popup menu appears.

3. Select **Detailed Diagnostics Info**.

    A dialog box appears with additional information.

On the command line you can use the assembler option **--diag** to see an explanation of a diagnostic message:

```
asarc --diag=[format:]{all | number,...]
```

Chapter 7. Using the Linker

This chapter describes the linking process, how to call the linker and how to control the linker with a script file.

The TASKING linker is a combined linker/locator. The linker phase combines relocatable object files (`.o` files, generated by the assembler), and libraries into a single relocatable linker object file (`.out`). The locator phase assigns absolute addresses to the linker object file and creates an absolute object file which you can load into a target processor. From this point the term linker is used for the combined linker/locator.

The linker can simultaneously link and locate all programs for all cores available on a target board. The target board may be of arbitrary complexity. A simple target board may contain one standard processor with some external memory that executes one task. A complex target board may contain multiple standard processors and DSPs combined with configurable IP-cores loaded in an FPGA. Each core may execute a different program, and external memory may be shared by multiple cores.

The linker takes the following files for input and output:
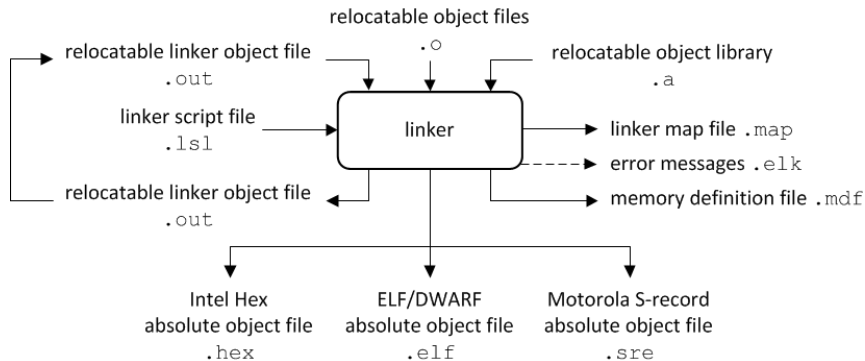
This chapter first describes the linking process. Then it describes how to call the linker and how to use its options. An extensive list of all options and their descriptions is included in Section 10.5, *Linker Options*.

To control the link process, you can write a script for the linker. This chapter shortly describes the purpose and basic principles of the Linker Script Language (LSL) on the basis of an example. A complete description of the LSL is included in Linker Script Language.

7.1. Linking Process

The linker combines and transforms relocatable object files (`.o`) into a single absolute object file. This process consists of two phases: the linking phase and the locating phase.

In the first phase the linker combines the supplied relocatable object files and libraries into a single relocatable object file. In the second phase, the linker assigns absolute addresses to the object file so it can actually be loaded into a target.

Terms used in the linking process

Term	Definition
Absolute object file	Object code in which addresses have fixed absolute values, ready to load into a target.
Address	A specification of a location in an address space.
Address space	The set of possible addresses. A core can support multiple spaces, for example in a Harvard architecture the addresses that identify the location of an instruction refer to *code* space, whereas addresses that identify the location of a data object refer to a *data* space.
Architecture	A description of the characteristics of a core that are of interest for the linker. This encompasses the address space(s) and the internal bus structure. Given this information the linker can convert logical addresses into physical addresses.
Copy table	A section created by the linker. This section contains data that specifies how the startup code initializes the data and BSS sections. For each section the copy table contains the following fields:
	• action: defines whether a section is copied or zeroed
	• destination: defines the section's address in RAM
	• source: defines the sections address in ROM, zero for BSS sections
	• length: defines the size of the section in MAUs of the destination space
Core	An instance of an architecture.
Derivative	The design of a processor. A description of one or more cores including internal memory and any number of buses.
Library	Collection of relocatable object files. Usually each object file in a library contains one symbol definition (for example, a function).
Logical address	An address as encoded in an instruction word, an address generated by a core (CPU).
LSL file	The set of linker script files that are passed to the linker.
MAU	Minimum Addressable Unit. For a given processor the number of bits between an address and the next address. This is not necessarily a byte or a word.
Object code	The binary machine language representation of the C source.
Physical address	An address generated by the memory system.
Processor	An instance of a derivative. Usually implemented as a (custom) chip, but can also be implemented in an FPGA, in which case the derivative can be designed by the developer.
Relocatable object file	Object code in which addresses are represented by symbols and thus relocatable.
Relocation	The process of assigning absolute addresses.

Term	Definition
Relocation information	Information about how the linker must modify the machine code instructions when it relocates addresses.
Section	A group of instructions and/or data objects that occupy a contiguous range of addresses.
Section attributes	Attributes that define how the section should be linked or located.
Target	The hardware board on which an application is executing. A board contains at least one processor. However, a complex target may contain multiple processors and external memory and may be shared between processors.
Unresolved reference	A reference to a symbol for which the linker did not find a definition yet.

7.1.1. Phase 1: Linking

The linker takes one or more relocatable object files and/or libraries as input. A relocatable object file, as generated by the assembler, contains the following information:

- *Header information*: Overall information about the file, such as the code size, name of the source file it was assembled from, and creation date.

- *Object code*: Binary code and data, divided into various named sections. Sections are contiguous chunks of code that have to be placed in specific parts of the memory. The program addresses start at zero for each section in the object file.

- *Symbols*: Some symbols are exported - defined within the file for use in other files. Other symbols are imported - used in the file but not defined (external symbols). Generally these symbols are names of routines or names of data objects.

- *Relocation information*: A list of places with symbolic references that the linker has to replace with actual addresses. When in the code an external symbol (a symbol defined in another file or in a library) is referenced, the assembler does not know the symbol's size and address. Instead, the assembler generates a call to a preliminary relocatable address (usually 0000), while stating the symbol name.

- *Debug information*: Other information about the object code that is used by a debugger. The assembler optionally generates this information and can consist of line numbers, C source code, local symbols and descriptions of data structures.

The linker resolves the external references between the supplied relocatable object files and/or libraries and combines the files into a single relocatable linker object file.

The linker starts its task by scanning all specified relocatable object files and libraries. If the linker encounters an unresolved symbol, it remembers its name and continues scanning. The symbol may be defined elsewhere in the same file, or in one of the other files or libraries that you specified to the linker. If the symbol is defined in a library, the linker extracts the object file with the symbol definition from the library. This way the linker collects all definitions and references of all of the symbols.

Next, the linker combines sections with the same section name and attributes into single sections. The linker also substitutes (external) symbol references by (relocatable) numerical addresses where possible.

At the end of the linking phase, the linker either writes the results to a file (a single relocatable object file) or keeps the results in memory for further processing during the locating phase.

The resulting file of the linking phase is a single relocatable object file (`.out`). If this file contains unresolved references, you can link this file with other relocatable object files (`.o`) or libraries (`.a`) to resolve the remaining unresolved references.

With the linker command line option **--link-only**, you can tell the linker to only perform this linking phase and skip the locating phase. The linker complains if any unresolved references are left.

7.1.2. Phase 2: Locating

In the locating phase, the linker assigns absolute addresses to the object code, placing each section in a specific part of the target memory. The linker also replaces references to symbols by the actual address of those symbols. The resulting file is an absolute object file which you can actually load into a target memory. Optionally, when the resulting file should be loaded into a ROM device the linker creates a so-called copy table section which is used by the startup code to initialize the data and BSS sections.

Code modification

When the linker assigns absolute addresses to the object code, it needs to modify this code according to certain rules or *relocation expressions* to reflect the new addresses. These relocation expressions are stored in the relocatable object file. Consider the following snippet of x86 code that moves the contents of variable a to variable b via the `eax` register:

```
A1 3412 0000 mov a,%eax   (a defined at 0x1234, byte reversed)
A3 0000 0000 mov %eax,b   (b is imported so the instruction refers to
                            0x0000 since its location is unknown)
```

Now assume that the linker links this code so that the section in which a is located is relocated by 0x10000 bytes, and b turns out to be at 0x9A12. The linker modifies the code to be:

```
A1 3412 0100 mov a,%eax   (0x10000 added to the address)
A3 129A 0000 mov %eax,b   (0x9A12 patched in for b)
```

These adjustments affect instructions, but keep in mind that any pointers in the data part of a relocatable object file have to be modified as well.

Locate order

In some cases, the linker sorts sections, and this determines the order in which the sections appear in memory, usually from low to high addresses, but the linker can optionally allocate sections from high to low addresses.

The sort key is composed of (at least) the section alignment, size, name, and contents. The section contents are not compared directly, but a CRC value is computed from the section contents, and this value is compared. This means that if two sections are identical in all but one attribute, and this attribute is changed for one of the sections, then the order of the sections in memory may change.

To see the order in which sections are located, refer to the subsection Locate Rules in Section 13.2, *Linker Map File Format*. The locate rules are processed from top to bottom, while the sections in a locate rule are handled from left to right.

Output formats

The linker can produce its output in different file formats. The default ELF/DWARF format (`.elf`) contains an image of the executable code and data, and can contain additional debug information. The Intel-Hex format (`.hex`) and Motorola S-record format (`.sre`) only contain an image of the executable code and data. You can specify a format with the options **--output** (**-o**) and **--chip-output** (**-c**).

Controlling the linker

Via a so-called *linker script file* you can gain complete control over the linker. The script language is called the *Linker Script Language* (LSL). Using LSL you can define:

• The memory installed in the embedded target system:

To assign locations to code and data sections, the linker must know what memory devices are actually installed in the embedded target system. For each physical memory device the linker must know its start-address, its size, and whether the memory is read-write accessible (RAM) or read-only accessible (ROM).

• How and where code and data should be placed in the physical memory:

Embedded systems can have complex memory systems. If for example on-chip and off-chip memory devices are available, the code and data located in internal memory is typically accessed faster and with dissipating less power. To improve the performance of an application, specific code and data sections should be located in on-chip memory. By writing your own LSL file, you gain full control over the locating process.

• The underlying hardware architecture of the target processor.

To perform its task the linker must have a model of the underlying hardware architecture of the processor you are using. For example the linker must know how to translate an address used within the object file (a logical address) into an offset in a particular memory device (a physical address). In most linkers this model is hard coded in the executable and can not be modified. For the TASKING linker this hardware model is described in the linker script file. This solution is chosen to support configurable cores that are used in system-on-chip designs.

When you want to write your own linker script file, you can use the standard linker script files with architecture descriptions delivered with the product.

See also Section 7.8, *Controlling the Linker with a Script*.

7.2. Calling the Linker

In Eclipse you can set options specific for the linker. After you have built your project, the output files are available in a subdirectory of your project directory, depending on the active configuration you have set in the **C/C++ Build » Settings** page of the **Project » Properties for** dialog.

Building a project under Eclipse

You have several ways of building your project:

* Build Individual Project ().

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

* Rebuild Project (). This builds every file in the project whether or not a file has been modified since the last build. A rebuild is a clean followed by a build.

  1. Select **Project » Clean...**

  2. Enable the option **Start a build immediately** and click **Clean**.

* Build Automatically. This performs a build of all projects whenever any project file is saved, such as your makefile.

  This way of building is not recommended, but to enable this feature select **Project » Build Automatically** and ensure there is a check mark beside the **Build Automatically** menu item. In order for this option to work, you must also enable option **Build on resource save (Auto build)** on the **Behavior** tab of the **C/C++ Build** page of the **Project » Properties for** dialog.

To access the linker options

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

   The Properties dialog appears.

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

   In the right pane the Settings appear.

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

4. On the Tool Settings tab, select **Linker**.

5. Select the sub-entries and set the options in the various pages.

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

Invocation syntax on the command line:

larc [[*option*]... [*file*]...]...

When you are linking multiple files, either relocatable object files (.o) or libraries (.a), it is important to specify the files in the right order. This is explained in Section 7.3, *Linking with Libraries*.

Example:

```
larc -dtc49x.lsl test.o
```

This links and locates the file test.o and generates the file test.elf.

You can find a detailed description of all linker options in Section 10.5, *Linker Options*.

7.3. Linking with Libraries

There are two kinds of libraries: system libraries and user libraries.

System library

System libraries are stored in the directories:

```
<installation path>\carc\lib\tc45x    (ppu_tc45x libraries)
<installation path>\carc\lib\tc49x    (ppu_tc49x libraries)
<installation path>\carc\lib\tc49xn   (ppu_tc49xn libraries)
<installation path>\carc\lib\tc4dx    (ppu_tc4dx libraries)
```

An overview of the system libraries is given in the following table:

Libraries	Description
libc_fpu.a	C library with double-precision FPU instructions for ppu_tc45x, ppu_tc49x, ppu_tc49xn and ppu_tc4dx core architectures.
librt.a	Run-time library (contains other run-time functions needed by the C compiler)
libcp[x]_fpu.a	C++ run-time library with FPU instructions Optional letter: x = exception handling (default, without control program option **--no-exceptions**)
libcxx[x]_fpu.a	libc++ library with FPU instructions Optional letter: x = exception handling (default, without control program option **--no-exceptions**)

To link the default C (system) libraries

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

The Properties dialog appears.

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

   In the right pane the Settings appear.

3. On the Tool Settings tab, select **Linker » Libraries**.

4. Enable the option **Link default libraries**.

When you want to link system libraries from the command line, you must specify this with the option **--library** (**-l**). For example, to specify the system library libc_fpu.a, type:

```
larc --library=c_fpu test.o
```

User library

You can create your own libraries. Section 8.3, *Archiver* describes how you can use the archiver to create your own library with object modules.

To link user libraries

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

   The Properties dialog appears.

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

   In the right pane the Settings appear.

3. On the Tool Settings tab, select **Linker » Libraries**.

4. Add your libraries to the **Libraries** box.

When you want to link user libraries from the command line, you must specify their filenames on the command line:

```
larc cstart.o mylib.a
```

If the library resides in a sub-directory, specify that directory with the library name:

```
larc cstart.o mylibs\mylib.a
```

If you do not specify a directory, the linker searches the library in the current directory only.

Library order

The order in which libraries appear on the command line is important. By default the linker processes object files and libraries in the order in which they appear at the command line. Therefore, when you use a weak symbol construction, like printf, in an object file or your own library, you must position this object/library before the C library.

With the option **--first-library-first** you can tell the linker to scan the libraries from left to right, and extract symbols from the first library where the linker finds it. This can be useful when you want to use newer versions of a library routine:

```
larc --first-library-first a.a test.o b.a
```

If the file `test.o` calls a function which is both present in `a.a` and `b.a`, normally the function in `b.a` would be extracted. With this option the linker first tries to extract the symbol from the first library `a.a`.

Note that routines in `b.a` that call other routines that are present in both `a.a` and `b.a` are now also resolved from `a.a`.

7.3.1. How the Linker Searches Libraries

System libraries

You can specify the location of system libraries in several ways. The linker searches the specified locations in the following order:

1. The linker first looks in the **Library search path** that are specified in the **Linker » Libraries** page in the **C/C++ Build » Settings » Tool Settings** tab of the Project Properties dialog (equivalent to the option **--library-directory (-L)**). If you specify the **-L** option without a pathname, the linker stops searching after this step.

2. When the linker did not find the library (because it is not in the specified library directory or because no directory is specified), it looks in the path(s) specified in the environment variables `LIBPPU_TC45X` / `LIBPPU_TC49X` / `LIBPPU_TC49XN` / `LIBPPU_TC4DX`.

3. When the linker did not find the library, it tries the default `lib` directory relative to the installation directory (or a processor specific sub-directory).

User library

If you use your own library, the linker searches the library in the current directory only.

7.3.2. How the Linker Extracts Objects from Libraries

A library built with the TASKING archiver **ararc** always contains an index part at the beginning of the library. The linker scans this index while searching for unresolved externals. However, to keep the index as small as possible, only the defined symbols of the library members are recorded in this area.

When the linker finds a symbol that matches an unresolved external, the corresponding object file is extracted from the library and is processed. After processing the object file, the remaining library index is searched. If after a complete search of the library unresolved externals are introduced, the library index will be scanned again. After all files and libraries are processed, and there are still unresolved externals and you did not specify the linker option **--no-rescan**, all libraries are rescanned again. This way you do not have to worry about the library order on the command line and the order of the object files in the libraries. However, this rescanning does not work for 'weak symbols'. If you use a weak symbol construction, like `printf`, in an object file or your own library, you must position this object/library before the C library.

The option **--verbose** (**-v**) shows how libraries have been searched and which objects have been extracted.

Resolving symbols

If you are linking from libraries, only the objects that contain symbols to which you refer, are extracted from the library. This implies that if you invoke the linker like:

```
larc mylib.a
```

nothing is linked and no output file will be produced, because there are no unresolved symbols when the linker searches through `mylib.a`.

It is possible to force a symbol as external (unresolved symbol) with the option **--extern** (**-e**):

```
larc --extern=main mylib.a
```

In this case the linker searches for the symbol `main` in the library and (if found) extracts the object that contains `main`.

If this module contains new unresolved symbols, the linker looks again in `mylib.a`. This process repeats until no new unresolved symbols are found.

7.4. Incremental Linking

With the TASKING linker it is possible to link incrementally. Incremental linking means that you link some, but not all `.o` modules to a relocatable object file `.out`. In this case the linker does not perform the locating phase. With the second invocation, you specify both new `.o` files as the `.out` file you had created with the first invocation.

Incremental linking is only possible on the command line.

```
larc --incremental test1.o -otest.out
larc test2.o test.out
```

This links the file `test1.o` and generates the file `test.out`. This file is used again and linked together with `test2.o` to create the file `test.elf` (the default name if no output filename is given in the default ELF/DWARF format).

With incremental linking it is normal to have unresolved references in the output file until all `.o` files are linked and the final `.out` or `.elf` file has been reached. The option **--incremental** (**-r**) for incremental linking also suppresses warnings and errors because of unresolved symbols.

7.5. Importing Binary Files

With the TASKING linker it is possible to add a binary file to your absolute output file. In an embedded application you usually do not have a file system where you can get your data from.

Add a data object in Eclipse

1.   Select **Linker » Data Objects**.

The Data objects box shows the list of object files that are imported.

2.   To add a data object, click on the **Add** button in the **Data objects** box.

3.   Type or select a binary file (including its path).

On the command line you can add raw data to your application with the linker option **--import-object**.

This makes it possible for example to display images on a device or play audio. The linker puts the raw data from the binary file in a section. The section is aligned on a 4-byte boundary. The section name is derived from the filename, in which dots are replaced by an underscore. So, when importing a file called `my.mp3`, a section with the name `my_mp3` is created. In your application you can refer to the created section by using linker labels.

For example:

```
#include <stdio.h>
extern char   _lc_ub_my_mp3; /* linker labels */
extern char   _lc_ue_my_mp3;
char*   mp3 = &_lc_ub_my_mp3;

void main(void)
{
  int size = &_lc_ue_my_mp3 - &_lc_ub_my_mp3;
  int i;
  for (i=0;i<size;i++)
    putchar(mp3[i]);
}
```

If you want to use the export functionality of Eclipse, the binary file has to be part of your project.

7.6. Converting Intel Hex to Binary Format

The linker can convert one or more Intel Hex input files to a single binary output file. This binary output format is only available for "chip" output, not for "space" output. Multiple Intel Hex files may be used as input, as long as there are no address conflicts and as long as there is only one program entry point for a set of multiple Intel Hex files. If more than one entry point is encountered the linker emits an error.

The linker reads the Intel Hex file(s) and stores the contents in an internal database format in as many sections as there are contiguous memory sections within the Intel Hex file(s). All sections are stored within the primary hex file address space. Each section is incrementally named using the following format .

.secN_input_file_name

Conversion from the internal database format to the binary output takes place automatically when the input is detected to be an Intel Hex file and the command line option:

--chip-output=[*basename*]**:***format*[**:***addr_size*],...

is used with the *format* field set to **BIN** and the *addr_size* left empty.

Any memory location included in the binary file that is not occupied by application data can be filled with the value specified by linker option **--binfill=***pattern* (default 0x00).

The resulting binary output file has no knowledge of targets or absolute addresses. It is simply a byte representation of the image data that was read in. The data of a binary output file represents the first MAU in memory (at offset zero) up to the last data MAU of the application in memory. The resulting binary file has no memory holes because they are filled with the fill pattern.

Example:

```
larc myproj_1.hex myproj_2.hex -dtc49x.lsl --core=ppu
    --chip-output=myproj:bin --binfill=0x2D
```

7.7. Linker Optimizations

During the linking and locating phase, the linker looks for opportunities to optimize the object code. Both code size and execution speed can be optimized.

To enable or disable optimizations

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

    The Properties dialog appears.

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

    In the right pane the Settings appear.

3. On the Tool Settings tab, select **Linker » Optimization**.

4. Enable one or more optimizations.

You can enable or disable the optimizations described below. The command line option for each optimization is given in brackets.

Delete unreferenced sections (option -Oc/-OC)

This optimization removes unreferenced sections from the resulting object file.

This optimization considers a section referenced if either of the following two conditions is true:

1. The section is protected from unreferenced section removal, which can be one of:

- the section is assigned an absolute address, either in the object file or in LSL

- the section is selected by exact name in LSL (no wildcard pattern) [*]

- a symbol defined in the section is referenced in LSL

- the section has the 'protected' section flag set, either in the object file or in LSL

2. The section is referenced via a relocation by another section that is considered referenced.

[*] If multiple sections of a specific name are created by using section renaming, all of these sections are protected against unreferenced section removal. With a selection using wildcards, matching sections are selected, but matching sections that are unreferenced may be removed. See Selecting sections for a group in Section 15.8.2, *Creating and Locating Groups of Sections*.

First fit decreasing (option -Ol/-OL)

When the physical memory is fragmented or when address spaces are nested it may be possible that a given application cannot be located although the size of the available physical memory is larger than the sum of the section sizes. Enable the first-fit-decreasing optimization when this occurs and re-link your application.

The linker's default behavior is to place sections in the order that is specified in the LSL file (that is, working from low to high memory addresses or vice versa). This also applies to sections within an unrestricted group. If a memory range is partially filled and a section must be located that is larger than the remainder of this range, then the section and all subsequent sections are placed in a next memory range. As a result of this gaps occur at the end of a memory range.

When the first-fit-decreasing optimization is enabled the linker will first place the largest sections in the smallest memory ranges that can contain the section. Small sections are located last and can likely fit in the remaining gaps.

Compress copy table (option -Ot/-OT)

The startup code initializes the application's data areas. The information about which memory addresses should be zeroed and which memory ranges should be copied from ROM to RAM is stored in the copy table.

When this optimization is enabled the linker will try to locate sections in such a way that the copy table is as small as possible thereby reducing the application's ROM image.

Note that this optimization only affects unrestricted sections that require an initialization action in the copy table. The affected sections get a clustered restriction. Unrestricted sections are sections that do not have their absolute location or their relative location to other sections restricted. See also Define the mutual order of sections in an LSL group in Section 15.8.2, *Creating and Locating Groups of Sections*.

Delete duplicate code (option -Ox/-OX)

Delete duplicate constant data (option -Oy/-OY)

These two optimizations remove code and constant data that is defined more than once, from the resulting object file.

Note that the optimizations perform detailed comparisons between sections with equal contents to check if they are indeed equal. Since these comparisons can be computationally intensive, the optimizations are only performed when the number of sections to be compared is under a limit (default 10000 sections). With option **--duplicate-section-limit** you can specify a custom maximum amount.

7.8. Controlling the Linker with a Script

With the options on the command line you can control the linker's behavior to a certain degree. From Eclipse it is also possible to determine where your sections will be located, how much memory is available, which sorts of memory are available, and so on. Eclipse passes these locating directions to the linker via a script file. If you want even more control over the locating process you can supply your own script.

The language for the script is called the *Linker Script Language*, or shortly LSL. You can specify the script file to the linker, which reads it and locates your application exactly as defined in the script. If you do not specify your own script file, the linker always reads a standard script file which is supplied with the toolset.

7.8.1. Purpose of the Linker Script Language

The Linker Script Language (LSL) serves three purposes:

1. It provides the linker with a definition of the target's core architecture. This definition is supplied with the toolset.

2. It provides the linker with a specification of the memory attached to the target processor.

3. It provides the linker with information on how your application should be located in memory. This gives you, for example, the possibility to create overlaying sections.

The linker accepts multiple LSL files. You can use the specifications of the core architectures that TASKING has supplied in the `include.lsl` directory. Do not change these files.

If you use a different memory layout than described in the LSL file supplied for the target core, you must specify this in a separate LSL file and pass both the LSL file that describes the core architecture and your LSL file that contains the memory specification to the linker. Next you may want to specify how sections should be located and overlaid. You can do this in the same file or in another LSL file.

LSL has its own syntax. In addition, you can use the standard C preprocessor keywords, such as `#include` and `#define`, because the linker sends the script file first to the C preprocessor before it starts interpreting the script.

The complete LSL syntax is described in Chapter 15, *Linker Script Language (LSL)*.

7.8.2. Eclipse and LSL

In Eclipse you can specify the size of the stack and heap; the physical memory attached to the processor; identify that particular address ranges are reserved; and specify which sections are located where in memory. Eclipse translates your input into an LSL file that is stored in the project directory under the name *project_name*.`lsl` and passes this file to the linker. If you want to learn more about LSL you can inspect the generated file *project_name*.`lsl`.

Because an ARC project is part of a TriCore project you only need to specify an LSL file to the TriCore project.

To add a generated Linker Script File to your project

1.   From the **File** menu, select **File » New » TASKING TriCore C/C++ Project**.

      The New C/C++ Project wizard appears.

2.   Fill in the project settings in each dialog and click **Next >** until the following dialog appears.
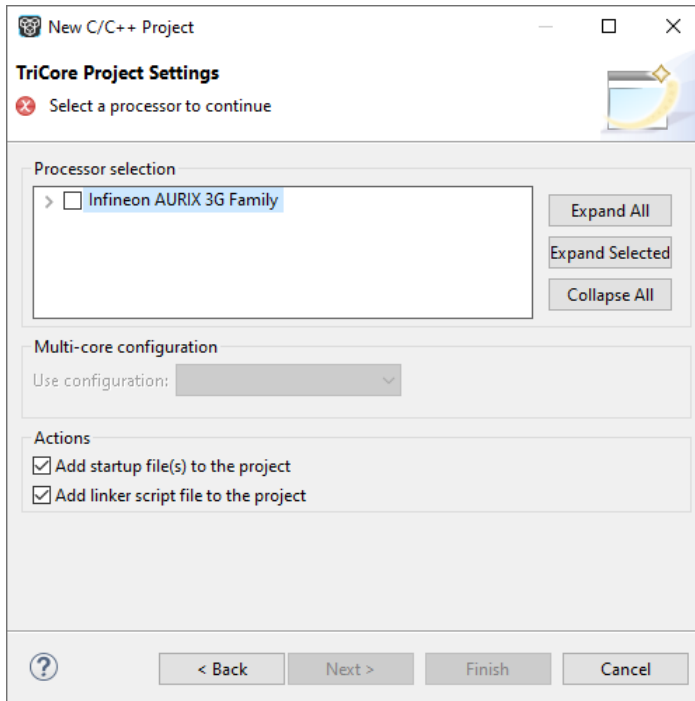
3.   Enable the option **Add linker script file to the project** and click **Finish**.

      Eclipse creates your project and the file "project_name.lsl" in the project directory.

If you do not add the linker script file here, you can always add it later with **File » New » Linker Script File (LSL)**.

To change the Linker Script File in Eclipse

There are two ways of changing the LSL file in Eclipse.

• You can change the LSL file directly in an editor.

    1.   Double-click on the file *project_name*.lsl.

221

The project LSL file opens in the editor area.
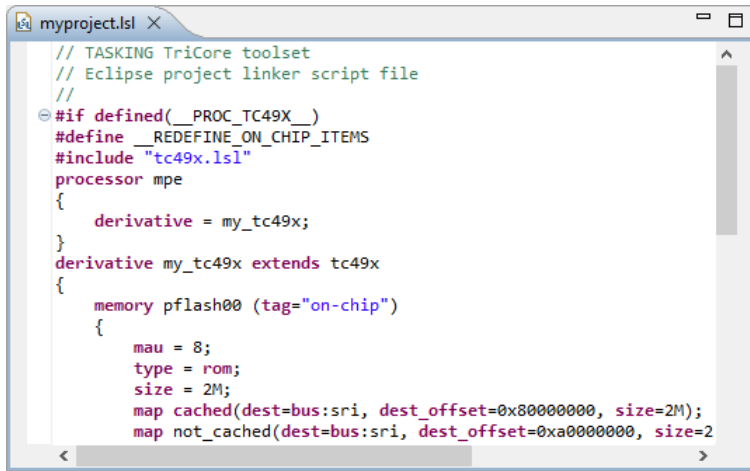
2. You can edit the LSL file directly in the *project_name*.lsl editor.

    *A * appears in front of the name of the LSL file to indicate that the file has changes.*

3. Click 🖫 or select **File » Save** to save the changes.

- You can also make changes to the property pages Memory and Stack/Heap.

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

        The Properties dialog appears.

    2. In the left pane, expand **C/C++ Build** and select **Memory** or **Stack/Heap**.

        In the right pane the corresponding property page appears.

    3. Make changes to memory and/or stack/heap and click **OK**.

        The project LSL file is updated automatically according to the changes you make in the pages.

You can quickly navigate through the LSL file by using the Outline view (**Window » Show View » Outline**).

7.8.3. Structure of a Linker Script File

A script file consists of several definitions. The definitions can appear in any order.

The architecture definition (required)

In essence an *architecture definition* describes how the linker should convert logical addresses into physical addresses for a given type of core. If the core supports multiple address spaces, then for each

space the linker must know how to perform this conversion. In this context a physical address is an offset on a given internal or external bus. Additionally the architecture definition contains information about items such as the (hardware) stack and the interrupt vector table.

This specification is normally written by TASKING. TASKING supplies LSL files in the `include.lsl` directory. The file `arch_ppu.lsl` defines the PPU architecture and defines a vector table.

The architecture definition of the LSL file should not be changed by you unless you also modify the core's hardware architecture. If the LSL file describes a multi-core system an architecture definition must be available for each different type of core.

The linker uses the architecture name in the LSL file to identify the target. For example, the default library search path can be different for each core architecture.

The derivative definition

The *derivative definition* describes the configuration of the internal (on-chip) bus and memory system. Basically it tells the linker how to convert offsets on the buses specified in the architecture definition into offsets in internal memory. Microcontrollers and DSPs often have internal memory and I/O sub-systems apart from one or more cores. The design of such a chip is called a *derivative*.

When you want to use multiple cores of the same type, you must instantiate the cores in a derivative definition, since the linker automatically instantiates only a single core for an unused architecture.

TASKING supplies LSL files for each derivative (`derivative.lsl`), along with "SFR files", which provide easy access to registers in I/O sub-systems from C and assembly programs. When you build an ASIC or use a derivative that is not (yet) supported by the TASKING tools, you may have to write a derivative definition.

The processor definition

The *processor definition* describes an instance of a derivative. A processor definition is only needed in a multi-processor embedded system. It allows you to define multiple processors of the same type.

If for a derivative 'A' no processor is defined in the LSL file, the linker automatically creates a processor named 'A' of derivative 'A'. This is why for single-processor applications it is enough to specify the derivative in the LSL file.

The memory and bus definitions (optional)

Memory and bus definitions are used within the context of a derivative definition to specify internal memory and on-chip buses. In the context of a board specification the memory and bus definitions are used to define external (off-chip) memory and buses. Given the above definitions the linker can convert a logical address into an offset into an on-chip or off-chip memory device.

The board specification

The processor definition and memory and bus definitions together form a board specification. LSL provides language constructs to easily describe single-core and heterogeneous or homogeneous multi-core systems. The board specification describes all characteristics of your target board's system buses, memory

devices, I/O sub-systems, and cores that are of interest to the linker. Based on the information provided in the board specification the linker can for each core:

- convert a logical address to an offset within a memory device

- locate sections in physical memory

- maintain an overall view of the used and free physical memory within the whole system while locating

The section layout definition (optional)

The optional section layout definition enables you to exactly control where input sections are located. Features are provided such as: the ability to place sections at a given address, to place sections in a given order, and to overlay code and/or data sections.

Example: Skeleton of a Linker Script File

A linker script file that defines a derivative "X'" based on the PPU architecture, its external memory and how sections are located in memory, may have the following skeleton:

```
architecture PPU
{
    // Specification of the PPU core architecture.
    // Written by TASKING.
}

derivative X  // derivative name is arbitrary
{
    // Specification of the derivative.
    // Written by TASKING.
    core ppu         // always specify the core
    {
        architecture = PPU;
    }

    bus ppu_bus    // internal bus
    {
        // maps to bus "local_bus" in "ppu" core
    }

    // internal memory
}

memory ext_name
{
    // external memory definition
}

section_layout ppu:linear    // section layout
{
    // section placement statements
```

```
    // sections are located in address space 'linear'
    // of core 'ppu'
}
```

Overview of LSL files delivered by TASKING

TASKING supplies the following LSL files in the directory `include.lsl`.

LSL file	Description
`arch_ppu.lsl`	Defines the PPU architecture based on the ARC_HS architecture. The PPU_TC45x, PPU_TC49x, PPU_TC49xN and PPU_TC4Dx architectures extend the PPU architecture. It also defines the vector table.
`tc45x.lsl` `tc49x.lsl` `tc49xn.lsl` `tc4dx.lsl`	Contains a processor and memory definition. It includes the file `vppu_tc45x.lsl`, `vppu_tc49x.lsl`, `vppu_tc49xn.lsl`, `vppu_tc4dx.lsl` respectively.
`vppu_tc45x.lsl` `vppu_tc49x.lsl` `vppu_tc49xn.lsl` `vppu_tc4dx.lsl`	Contains a derivative definition. It includes the file `vppu.lsl`.
`vppu.lsl`	Contains a derivative definition for the PPU. It includes the file `arch_ppu.lsl`.
`template.lsl`	This file is used by Eclipse as a template for the project LSL file. It includes the file `cpu.lsl`.
`cpu.lsl`	This file includes the file `derivative.lsl` based on your CPU selection. The CPU is specified by the __CPU__ macro.
`default.lsl`	Contains a default memory definition and section layout based on the tc49x derivative. This file is used on a command line invocation of the tools. It includes the file `tc49x.lsl`.

When you select to add a linker script file when you create a project in Eclipse, Eclipse makes a copy of the file `template.lsl` and names it "*project_name*.lsl".

On the command line, the linker uses the file `default.lsl`, unless you specify another file with the linker option **--lsl-file** (**-d**).

7.8.4. The Architecture Definition

Although you will probably not need to write an architecture definition (unless you are building your own processor core) it helps to understand the Linker Script Language and how the definitions are interrelated.

Within an *architecture definition* the characteristics of a target processor core that are important for the linking process are defined. These include:

• space definitions: the logical address spaces and their properties

• bus definitions: the I/O buses of the core architecture

- mappings: the address translations between logical address spaces, the connections between logical address spaces and buses and the address translations between buses

Address spaces

A logical address space is a memory range for which the core has a separate way to encode an address into instructions. Most microcontrollers and DSPs support multiple address spaces. For example, separate spaces for code and data. Normally, the size of an address space is 2^N, with *N* the number of bits used to encode the addresses.

The relation of an address space with another address space can be one of the following:

- one space is a subset of the other. These are often used for "small" absolute or relative addressing.

- the addresses in the two address spaces represent different locations so they do not overlap. This means the core must have separate sets of address lines for the address spaces. For example, in Harvard architectures we can identify at least a code and a data memory space.

Address spaces (even nested) can have different minimal addressable units (MAU), alignment restrictions, and page sizes. All address spaces have a number that identifies the logical space (id). The following table lists the different address spaces for the architecture `PPU` as defined in `arch_ppu.lsl`.

Space	Id	MAU	Description
linear	1	8	Linear address space.

The PPU architecture in LSL notation

The best way to write the architecture definition, is to start with a drawing. The following figure shows a part of the `PPU` architecture:
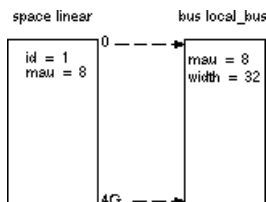
The figure shows one address space called `linear`. The address space has attributes like a number that identifies the logical space (id), a MAU and an alignment. In LSL notation the definition of this address space looks as follows:

```
space linear
{
    id = 1;
    mau = 8;

    map (size=4G, dest=bus:local_bus);
}
```

The keyword `map` corresponds with the arrows in the drawing. You can map:

- address space => address space (not shown in the drawing)

- address space => bus

- memory => bus (not shown in the drawing)

- bus => bus (not shown in the drawing)

Next the internal bus, named `local_bus` must be defined in LSL:

```
bus local_bus
{
     mau = 8;
     width = 32;  // there are 32 data lines on the bus
}
```

This completes the LSL code in the architecture definition. Note that all code above goes into the architecture definition, thus between:

```
architecture PPU
{
    // All code above goes here.
}
```

7.8.5. The Derivative Definition

Although you will probably not need to write a derivative definition (unless you are using multiple cores that both access the same memory device) it helps to understand the Linker Script Language and how the definitions are interrelated.

A *derivative* is the design of a processor, as implemented on a chip (or FPGA). It comprises one or more cores and on-chip memory. The derivative definition includes:

- core definition: an instance of a core architecture

- bus definition: the I/O buses of the core architecture

- memory definitions: internal (or on-chip) memory

Core

Each derivative must have at least one core and each core must have a specification of its core architecture. This core architecture must be defined somewhere in the LSL file(s).

```
core ppu
{
    architecture = PPU;
}
```

Bus

Each derivative can contain a bus definition for connecting external memory. In this example, the bus `ppu_bus` maps to the bus `local_bus` defined in the architecture definition of core `ppu`:

```
bus local_bus
{
   mau = 8;
   width = 32;
   map (size=4G, dest=bus:ppu:local_bus);
}
```

Memory

Memory is usually described in a separate memory definition, but you can specify on-chip memory for a derivative. For example:

```
memory internal_code_rom
{
    mau  = 8;
    type = rom;
    size = 2k;
    map( dest=bus:ppu:local_bus, size = 2k, dest_offset = 0x00100000);
         // src_offset is zero by default
}
```

This completes the LSL code in the derivative definition. Note that all code above goes into the derivative definition, thus between:

```
derivative X    // name of derivative
{
    // All code above goes here
}
```

7.8.6. The Processor Definition

The processor definition is only needed when you write an LSL file for a multi-processor embedded system. The processor definition explicitly instantiates a derivative, allowing multiple processors of the same type.

```
processor name
{
    derivative = derivative_name;
}
```

If no processor definition is available that instantiates a derivative, a processor is created with the same name as the derivative.

7.8.7. The Memory Definition

Once the core architecture is defined in LSL, you may want to extend the processor with external (or off-chip) memory. You need to specify the location and size of the physical external memory devices in the target system.

The principle is the same as defining the core's architecture but now you need to fill the memory definition:

```
memory name
{
    // memory definitions
}
```
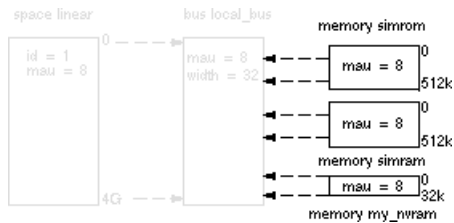

Suppose your embedded system has 512kB of external ROM, named `simrom`, 512kB of external RAM, named `simram` and 32kB of external NVRAM, named `my_nvram` (see figure above.) All memories are connected to the bus `local_bus`. In LSL this looks like follows:

```
memory simrom
{
    mau = 8;
    type = rom;
    size = 512k;
    map ( size = 512k, dest_offset=0, dest=bus:X:local_bus);
}

memory simram
{
    mau = 8;
    type = ram;
    size = 512k;
    map ( size = 512k, dest_offset=512k, dest=bus:X:local_bus);
}

memory my_nvram
{
    mau  = 8;
    size = 32k;
    type = ram;
    map ( size = 32k, dest_offset=1M, dest=bus:X:local_bus);
}
```

If you use a different memory layout than described in the LSL file supplied for the target core, you can specify this in Eclipse or you can specify this in a separate LSL file and pass both the LSL file that describes the core architecture and your LSL file that contains the memory specification to the linker.

To add memory using Eclipse

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

    The Properties dialog appears.

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

    In the right pane the Memory page appears.

3. Open the **Memory** tab and click on the **Add...** button.

    The Add new memory dialog appears.

4. Enter the memory name (for example `my_nvram`), type (for example `nvram`) and size.

5. Click on the **Add...** button.

    The Add new mapping dialog appears.

6. You have to specify at least one mapping. Enter the mapping name (optional), address, size and destination and click **OK**.

    The new mapping is added to the list of mappings.

7. Click **OK**.

    The new memory is added to the list of memories (user memory).

8. Click **Apply and Close** to close the Properties dialog.

    The updated settings are stored in the project LSL file.

If you make changes to the on-chip memory as defined in the architecture LSL file, the memory is copied to your project LSL file and the line `#define __MEMORY` is added. If you remove all the on-chip memory from your project LSL file, also make sure you remove this define.

7.8.8. The Section Layout Definition: Locating Sections

Once you have defined the internal core architecture and optional memory, you can actually define where your application must be located in the physical memory.

During compilation, the compiler divides the application into sections. Sections have a name, an indication (section type) in which address space it should be located and attributes like writable or read-only.

In the section layout definition you can exactly define how input sections are placed in address spaces, relative to each other, and what their absolute run-time and load-time addresses will be.

Example: section propagation through the toolset

To illustrate section placement, the following example of a C program (`bat.c`) is used. The program saves the number of times it has been executed in battery back-upped memory, and prints the number.

```
#define BATTERY_BACKUP_TAG  0xa5f0
#include <stdio.h>

int  uninitialized_data;
int  initialized_data = 1;
#pragma section "non_volatile"
int  battery_backup_tag;
int  battery_backup_invok;
#pragma endsection

void main (void)
{
    if (battery_backup_tag != BATTERY_BACKUP_TAG )
    {
        // battery back-upped memory area contains invalid data
        // initialize the memory
        battery_backup_tag = BATTERY_BACKUP_TAG;
        battery_backup_invok = 0;
    }
    printf( "This application has been invoked %d times\n",
            battery_backup_invok++);
}
```

The compiler assigns names and attributes to sections. With the `#pragma section` and `#pragma endsection` the compiler's default section naming convention is overruled and a section with the name `non_volatile` appended is defined. In this section the battery back-upped data is stored.

As a result of the `#pragma section "non_volatile"`, the data objects between the pragma pair are placed in a section with the name "`.bss.non_volatile`". Note that "`.bss`" sections are cleared at startup. However, battery back-upped sections should not be cleared and therefore we will change this section attribute using the LSL.

Section placement

The number of invocations of the example program should be saved in non-volatile (battery back-upped) memory. This is the memory `my_nvram` from the example in Section 7.8.7, *The Memory Definition*.

To control the locating of sections, you need to write one or more section definitions in the LSL file. At least one for each address space where you want to change the default behavior of the linker. In our example, we need to locate sections in the address space `linear`:

```
section_layout ::linear
{
    // Section placement statements
}
```

231

To locate sections, you must create a group in which you select sections from your program. For the battery back-up example, we need to define one group, which contains the section `.bss.non_volatile`. All other sections are located using the defaults specified in the architecture definition. Section `.bss.non_volatile` should be placed in non-volatile ram. To achieve this, the run address refers to our non-volatile memory called `my_nvram`. Furthermore, the section should not be cleared and therefore the attribute **s** (scratch) is assigned to the group:

```
group ( run_addr = mem:my_nvram, attributes = rws )
{
    select ".bss.non_volatile";
}
```

This completes the LSL file for the sample architecture and sample program. You can now invoke the linker with this file and the sample program to obtain an application that works for this architecture.

For a complete description of the Linker Script Language, refer to Chapter 15, *Linker Script Language (LSL)*.

7.8.9. Locating Data in LMURAM

You can locate data sections in LMURAM as follows:

```
section_layout :ppu:linear
{
    group lmuram (run_addr = mem:mpe:lmuram0)
    {
        select ".data.*.lmu*"; // select section that should be placed in lmuram
    }
}
```

Please note that such variables need to be qualified `__no_sda`, or compiled with C compiler option **--sda-max-data-size=0**.

7.9. Linker Labels

The linker creates labels that you can use to refer to from within the application software. Some of these labels are real labels at the beginning or the end of a section. Other labels have a second function, these labels are used to address generated data in the locating phase. The data is only generated if the label is used.

Linker labels are labels starting with `_lc_`. The linker assigns addresses to the following labels when they are referenced:

Label	Description

`_lc_ub_name`
 	
`_lc_b_name`
	Begin of section *name*. Also used to mark the begin of the stack or heap or copy table.

Label	Description
`_lc_ue_`*name* `_lc_e_`*name*	End of section *name*. Also used to mark the end of the stack or heap. It points to the section address + section size, in other words the first MAU behind the section.
`_lc_cb_`*name*	Start address of an overlay section in ROM.
`_lc_ce_`*name*	End address of an overlay section in ROM.
`_lc_gb_`*name*	Begin of group *name*. This label appears in the output file even if no reference to the label exists in the input file.
`_lc_ge_`*name*	End of group *name*. It points to the first MAU behind the last section in the group. This label appears in the output file even if no reference to the label exists in the input file.

The linker only allocates space for the stack and/or heap when a reference to either of the section labels exists in one of the input object files.

If you want to use linker labels in your C source for sections that have a dot (.) in the name, you have to replace all dots by underscores.

Example: refer to a label with section name with dots from C

Suppose a section has the name `.text`. When you want to refer to the begin of this section you have to replace all dots in the section name by underscores:

```
#include <stdio.h>

extern char _lc_ub__text[];

void main(void)
{
    printf("The function main is located at %p\n",
            _lc_ub__text);
}
```

Example: refer to a TriCore variable from PPU source

From within the PPU source you can access TriCore variables. The same symbol name can be defined in different TriCore cores, if the `vtc` (all cores) application configuration is not used. To uniquely select a variable from a specific core, you prefix the variable name with **`_lc_t_`*core*`_`**. When the linker sees the **`_lc_t_`**, it removes the linker label prefix, and the core name prefix. The remainder is a symbol name, that has to be found in the application for the specified core.

For example, when a symbol `count` is defined in sources of two different TriCore cores, you can access them from a PPU C source file as follows:

```
extern __no_sda __uncached int _lc_t_tc0_count; /* variable count in tc0 */
extern __no_sda __uncached int _lc_t_tc3_count; /* variable count in tc3 */
```

```
void main(int argc, char **argv)
{
    _lc_t_tc0_count = 0;
    _lc_t_tc3_count = 1;
}
```

Example: refer to the stack

Suppose in an LSL file a stack section is defined with the name "stack" (with the keyword stack). You can refer to the begin and end of the stack from your C source as follows:

```
#include <stdio.h>

extern char _lc_ub_stack[];
extern char _lc_ue_stack[];

void main()
{
  printf( "Size of stack is %d\n",
          _lc_ub_stack - _lc_ue_stack );
          /* stack grows from high to low */
}
```

From assembly you can refer to the end of the stack with:

```
   .extern _lc_ue_stack   ; end of user stack
```

7.10. Generating a Map File

The map file is an additional output file that contains information about the location of sections and symbols. You can customize the type of information that should be included in the map file.

When the linker works on more than one task, a map file can be created for each of the tasks. There is also an option to create one global map file that includes information for all tasks involved. Use linker option **--global-map-file** to generate the global map file. This map file format is very similar to that of the map file for a single task.

To generate a map file

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

    The Properties dialog appears.

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

    In the right pane the Settings appear.

3. On the Tool Settings tab, select **Linker » Map File**.

4. Enable the option **Generate XML map file format (.mapxml) for map file viewer**.

5. (Optional) Enable the option **Generate map file (.map)**.

6. (Optional) Enable the options to include that information in the map file.

Example on the command line

The following command generates the map file `test.map`:

```
larc --map-file test.o
```

With this command the map file `test.map` is created.

See Section 13.2, *Linker Map File Format* for an explanation of the format of the map file.

7.11. Linker ELF Note Sections

The linker passes ELF Note sections from input files to ELF output files and adds the following notes:

• linker name

• linker version

• linker invocation

• for each defined environment variable that could influence the output, the name and value of the environment variable

You can inspect ELF note sections with **hldumparc -F0n** or with **elfdump**.

7.12. Linker Error Messages

The linker reports the following types of error messages in the Problems view of Eclipse.

F (Fatal errors)

After a fatal error the linker immediately aborts the link/locate process.

E (Errors)

Errors are reported, but the linker continues linking and locating. No output files are produced unless you have set the linker option **--keep-output-files**.

W (Warnings)

Warning messages do not result into an erroneous output file. They are meant to draw your attention to assumptions of the linker for a situation which may not be correct. You can control warnings in the **C/C++ Build » Settings » Tool Settings » Linker » Diagnostics** page of the **Project » Properties for** menu (linker option **--no-warnings**).

I (Information)

Verbose information messages do not indicate an error but tell something about a process or the state of the linker. To see verbose information, use the linker option **--verbose**.

S (System errors)

System errors occur when internal consistency checks fail and should never occur. When you still receive the system error message

```
S6##: message
```

please report the error number and as many details as possible about the context in which the error occurred.

Display detailed information on diagnostics

1. From the **Window** menu, select **Show View » Other » TASKING » Problems**.

    The Problems view is added to the current perspective.

2. In the Problems view right-click on a message.

    A popup menu appears.

3. Select **Detailed Diagnostics Info**.

    A dialog box appears with additional information.

On the command line you can use the linker option **--diag** to see an explanation of a diagnostic message:

```
larc --diag=[format:]{all | number,...]
```

Chapter 8. Using the Utilities

The TASKING toolset for ARC/PPU comes with a number of utilities:

ccarc   A control program. The control program invokes all tools in the toolset and lets you quickly generate an absolute object file from C/C++ and/or assembly source input files. Eclipse uses the control program to call the C++ compiler, C compiler, assembler and linker.

amk   A make utility to maintain, update, and reconstruct groups of programs. The make utility looks whether files are out of date, rebuilds them and determines which other files as a consequence also need to be rebuilt. It supports parallelism which utilizes the multiple cores found on modern host hardware.

ararc   An archiver. With this utility you create and maintain library files with relocatable object modules (`.o`) generated by the assembler.

hldumparc A high level language (HLL) object dumper. With this utility you can dump information about an absolute object file (`.elf`). Key features are a disassembler with HLL source intermixing and HLL symbol display and a HLL symbol listing of static and global symbols.

8.1. Control Program

The control program is a tool that invokes all tools in the toolset for you. It provides a quick and easy way to generate the final absolute object file out of your C/C++ sources without the need to invoke the C++ compiler, C compiler, assembler and linker manually.

Eclipse uses the control program to call the C++ compiler, C compiler, assembler and linker, but you can call the control program from the command line. The invocation syntax is:

ccarc [[*option*]... [*file*]...]...

Recognized input files

- Files with a `.C`, `.cc`, `.cxx` or `.cpp` suffix are interpreted as C++ source programs and are passed to the C++ compiler.

- Files with a `.c` suffix are interpreted as C source programs and are passed to the compiler.

- Files with a `.asm` suffix are interpreted as hand-written assembly source files which have to be passed to the assembler.

- Files with a `.src` suffix are interpreted as compiled assembly source files. They are directly passed to the assembler.

- Files with a `.a` suffix are interpreted as library files and are passed to the linker.

- Files with a `.o` suffix are interpreted as object files and are passed to the linker.

- Files with a `.out` suffix are interpreted as linked object files and are passed to the locating phase of the linker. The linker accepts only one `.out` file in the invocation.

• Files with a `.lsl` suffix are interpreted as linker script files and are passed to the linker.

Options

The control program accepts several command line options. If you specify an unknown option to the control program, the control program looks if it is an option for a specific tool. If so, it passes the option directly to the tool. However, it is recommended to use the control program options **--pass-*** (**-Wcp**, **-Wc**, **-Wa**, **-Wl**) to pass arguments directly to tools.

For a complete list and description of all control program options, see Section 10.6, *Control Program Options*.

Example with verbose output

```
ccarc --verbose test.c
```

The control program calls all tools in the toolset and generates the absolute object file `test.elf`. With option **--verbose** (**-v**) you can see how the control program calls the tools:

```
+ "path\carc" -D__CPU__=tc49x -D__CPU_TC49X__
     --core=ppu_tc49x -o cc3248a.src test.c
+ "path\asarc" -D__CPU__=tc49x -D__CPU_TC49X__
     --core=ppu_tc49x -o cc3248b.o cc3248a.src
+ "path\larc" -o test.elf -dtc49x.lsl --core=ppu
     -D__CPU__=tc49x --map-file cc3248b.o -lc_fpu -lrt
     "-Lpath\lib\tc49x"
```

The control program produces unique filenames for intermediate steps in the compilation process (such as `cc3248a.src` and `cc3248b.o` in the example above) which are removed afterwards, unless you specify command line option **--keep-temporary-files** (**-t**).

Example with argument passing to a tool

```
ccarc --pass-c=-Oc test.c
```

The option **-Oc** is directly passed to the compiler.

8.2. Make Utility amk

amk is a make utility that you can use to maintain, update, and reconstruct groups of programs. **amk** features parallelism which utilizes the multiple cores found on modern host hardware, hardening for path names with embedded white space and it has an (internal) interface to provide progress information for updating a progress bar. It does not use an external command shell (/bin/sh, cmd.exe) but executes commands directly.

The primary purpose of any make utility is to speed up the edit-build-test cycle. To avoid having to build everything from scratch even when only one source file changes, it is necessary to describe dependencies between source files and output files and the commands needed for updating the output files. This is done in a so called "makefile".

8.2.1. Makefile Rules

A makefile dependency rule is a single line of the form:

```
[target ...] : [prerequisite ...]
```

where *target* and *prerequisite* are path names to files. Example:

```
test.o : test.c
```

This states that target test.o depends on prerequisite test.c. So, whenever the latter is modified the first must be updated. Dependencies accumulate: prerequisites and targets can be mentioned in multiple dependency rules (circular dependencies are not allowed however). The command(s) for updating a target when any of its prerequisites have been modified must be specified with leading white space after any of the dependency rule(s) for the target in question. Example:

```
test.o :
  ccarc test.c   # leading white space
```

Command rules may contain dependencies too. Combining the above for example yields:

```
test.o : test.c
  ccarc test.c
```

White space around the colon is not required. When a path name contains special characters such as ':', '**#**' (start of comment), '**=**' (macro assignment) or any white space, then the path name must be enclosed in single or double quotes. Quoted strings can contain anything except the quote character itself and a newline. Two strings without white space in between are interpreted as one, so it is possible to embed single and double quotes themselves by switching the quote character.

When a target does not exist, its modification time is assumed to be very old. So, **amk** will try to make it. When a prerequisite does not exist possibly after having tried to make it, it is assumed to be very new. So, the update commands for the current target will be executed in that case. **amk** will only try to make targets which are specified on the command line. The default target is the first target in the makefile which does not start with a dot.

Static pattern rules

Static pattern rules are rules which specify multiple targets and construct the prerequisite names for each target based on the target name.

```
[target ...] : target-pattern : [prerequisite-patterns ...]
```

The *target* specifies the targets the rules applies to. The *target-pattern* and *prerequisite-patterns* specify how to compute the prerequisites of each target. Each target is matched against the *target-pattern* to extract a part of the target name, called the *stem*. This stem is substituted into each of the *prerequisite-patterns* to make the prerequisite names (one from each *prerequisite-pattern*).

Each pattern normally contains the character '%' just once. When the *target-pattern* matches a target, the '%' can match any part of the target name; this part is called the *stem*. The rest of the pattern must match exactly. For example, the target `foo.o` matches the pattern '`%.o`', with '`foo`' as the stem. The targets `foo.c` and `foo.elf` do not match that pattern.

The prerequisite names for each target are made by substituting the stem for the '%' in each prerequisite pattern.

Example:

```
objects = test.o filter.o

all: $(objects)

$(objects): %.o: %.c
    ccarc -c $< -o $@
    echo the stem is $*
```

Here '`$<`' is the automatic variable that holds the name of the prerequisite, '`$@`' is the automatic variable that holds the name of the target and '`$*`' is the stem that matches the pattern. Internally this translates to the following two rules:

```
test.o: test.c
    ccarc -c test.c -o test.o
    echo the stem is test

filter.o: filter.c
    ccarc -c filter.c -o filter.o
    echo the stem is filter
```

Each target specified must match the target pattern; a warning is issued for each target that does not.

Special targets

There are a number of special targets. Their names begin with a period.

Target	Description
`.DONE`	When the make utility has finished building the specified targets, it continues with the rules following this target.

Target	Description
`.INIT`	The rules following this target are executed before any other targets are built.

`.PHONY`	The prerequisites of this target are considered to be phony targets. A phony target is a target that is not really the name of a file. The rules following a phony target are executed unconditionally, regardless of whether a file with that name exists or what its last-modification time is.
	
	For example:
	
	`.PHONY: clean`
	
	`clean:`
	`        rm *.o`
	
	With `amk clean`, the command is executed regardless of whether there is a file named `clean`.

8.2.2. Makefile Directives

Directives inside makefiles are executed while reading the makefile. When a line starts with the word "`include`" or "`-include`" then the remaining arguments on that line are considered filenames whose contents are to be inserted at the current line. "`-include`" will silently skip files which are not present. You can include several files. Include files may be nested.

Example:

```
include makefile2 makefile3
```

White spaces (tabs or spaces) in front of the directive are allowed.

8.2.3. Macro Definitions

A *macro* is a symbol name that is replaced with its definition before the makefile is executed. Although the macro name can consist of lowercase or uppercase characters, uppercase is an accepted convention. When a line does not start with white space and contains the assignment operator '**=**', '**:=**' or '**+=**' then the line is interpreted as a macro definition. White space around the assignment operator and white space at the end of the line is discarded. Single character macro evaluation happens by prefixing the name with '**$**'. To evaluate macros with names longer than one character put the name between parentheses '**()**' or curly braces '**{}**'. Macro names may contain anything, even white space or other macro evaluations. Example:

```
DINNER = $(FOOD) and $(BEVERAGE)
FOOD = pizza
BEVERAGE = sparkling water
FOOD += with cheese
```

With the **+=** operator you can add a string to an existing macro. An extra space is inserted before the added string automatically.

Macros are evaluated recursively. Whenever $(DINNER) or ${DINNER} is mentioned after the above, it will be replaced by the text "pizza with cheese and sparkling water". The left hand side in a macro definition is evaluated before the definition takes place. Right hand side evaluation depends on the assignment operator:

=	Evaluate the macro at the moment it is used.
:=	Evaluate the replacement text before defining the macro.

Subsequent '**+=**' assignments will inherit the evaluation behavior from the previous assignment. If there is none, then '**+=**' is the same as '**=**'. The default value for any macro is taken from the environment. Macro definitions inside the makefile overrule environment variables. Macro definitions on the **amk** command line will be evaluated first and overrule definitions inside the makefile.

Predefined macros

Macro	Description
$	This macro translates to a dollar sign. Thus you can use "$$" in the makefile to represent a single "$".
@	The name of the current target. When a rule has multiple targets, then it is the name of the target that caused the rule commands to be run.
*	The basename (or stem) of the current target. The stem is either provided via a static pattern rule or is calculated by removing all characters found after and including the last dot in the current target name. If the target name is 'test.c' then the stem is 'test' (if the target was not created via a static pattern rule).
<	The name of the first prerequisite.
MAKE	The **amk** path name (quoted if necessary). Optionally followed by the options **-n** and **-s**.
ORIGIN	The name of the directory where **amk** is installed (quoted if necessary).
SUBDIR	The argument of option **-G**. If you have nested makes with **-G** options, the paths are combined. This macro is defined in the environment (i.e. default macro value).

The @, * and < macros may be suffixed by '**D**' to specify the directory component or by '**F**' to specify the filename component. `$(@D)` evaluates to the directory name holding the file`$(@F)`. `$(@D)/$(@F)` is equivalent to `$@`. Note that on MS-Windows most programs accept forward slashes, even for UNC path names.

The result of the predefined macros @, * and < and 'D' and 'F' variants is not quoted, so it may be necessary to put quotes around it.

Note that stem calculation can cause unexpected values. For example:

```
$@                      $*
/home/.wine/test        /home/
/home/test/.project     /home/test/
/../file                /.
```

Macro string substitution

When the macro name in an evaluation is followed by a colon and equal sign as in

```
$(MACRO:string1=string2)
```

then **amk** will replace *string1* at the end of every word in `$(MACRO)` by *string2* during evaluation. When `$(MACRO)` contains quoted path names, the quote character must be mentioned in both the original string and the replacement string[1]. For example:

```
$(MACRO:.o"=.d")
```

---

[1]Internally, **amk** tokenizes the evaluated text, but performs substitution on the original input text to preserve compatibility here with existing make implementations and POSIX.

8.2.4. Makefile Functions

A function not only expands but also performs a certain operation. The following functions are available:

$(filter *pattern ...,item ...*)

The `filter` function filters a list of items using a pattern. It returns *items* that do match any of the *pattern* words, removing any items that do not match. The patterns are written using '`%`',

```
${filter %.c %.h, test.c test.h test.o readme.txt .project output.c}
```

results in:

```
test.c test.h output.c
```

$(filter-out *pattern ...,item ...*)

The `filter-out` function returns all *items* that do not match any of the *pattern* words, removing the items that do match one or more. This is the exact opposite of the `filter` function.

```
${filter-out %.c %.h, test.c test.h test.o readme.txt .project output.c}
```

results in:

```
test.o readme.txt .project
```

$(foreach *var-name*, *item ...*, *action*)

The `foreach` function runs through a list of items and performs the same *action* for each *item*. The *var-name* is the name of the macro which gets dynamically filled with an item while iterating through the *item* list. In the *action* you can refer to this macro. For example:

```
${foreach T, test filter output, ${T}.c ${T}.h}
```

results in:

```
test.c test.h filter.c filter.h output.c output.h
```

8.2.5. Conditional Processing

Lines containing `ifdef`, `ifndef`, `else` or `endif` are used for conditional processing of the makefile. They are used in the following way:

```
ifdef macro-name
if-lines
else
else-lines
endif
```

The *if-lines* and *else-lines* may contain any number of lines or text of any kind, even other `ifdef`, `ifndef`, `else` and `endif` lines, or no lines at all. The `else` line may be omitted, along with the *else-lines* following it. White spaces (tabs or spaces) in front of preprocessing directives are allowed.

First the *macro-name* after the `ifdef` command is checked for definition. If the macro is defined then the *if-lines* are interpreted and the *else-lines* are discarded (if present). Otherwise the *if-lines* are discarded; and if there is an `else` line, the *else-lines* are interpreted; but if there is no else line, then no lines are interpreted.

When you use the `ifndef` line instead of `ifdef`, the macro is tested for not being defined. These conditional lines can be nested to any level.

You can also add tests based on strings. With `ifeq` the result is true if the two strings match, with `ifneq` the result is true if the two strings do not match. They are used in the following way:

```
ifeq(string1,string2)
if-lines
else
else-lines
endif
```

8.2.6. Makefile Parsing

amk reads and interprets a makefile in the following order:

1. When the last character on a line is a backslash (****) (i.e. without trailing white space) then that line and the next line will be concatenated, removing the backslash and newline.

2. The unquoted '**#**' character indicates start of comment and may be placed anywhere on a line. It will be removed in this phase.

   ```
   # this comment line is continued\
   on the next line
   ```

3. Trailing white space is removed.

4. When a line starts with white space and it is not followed by a directive or preprocessing directive, then it is interpreted as a command for updating a target.

5. Otherwise, when a line contains the unquoted text '**=**', '**+=**' or '**:=**' operator, then it will be interpreted as a macro definition.

6. Otherwise, all macros on the line are evaluated before considering the next steps.

7. When the resulting line contains an unquoted '**:**' the line is interpreted as a dependency rule.

8. When the first token on the line is "`include`" or "`-include`" (which by now must start on the first column of the line), **amk** will execute the directive.

9. Otherwise, the line must be empty.

Macros in commands for updating a target are evaluated right before the actual execution takes place (or would take place when you use the **-n** option).

8.2.7. Makefile Command Processing

A line with leading white space (tabs or spaces) without a (preprocessing) directive is considered as a command for updating a target. When you use the option **-j** or **-J**, **amk** will execute the commands for updating different targets in parallel. In that case standard input will not be available and standard output and error output will be merged and displayed on standard output only after the commands have finished for a target.

You can precede a command by one or more of the following characters:

@   Do not show the command. By default, commands are shown prior to their output.

-          Continue upon error. This means that **amk** ignores a non-zero exit code of the command.

+   Execute the command, even when you use option **-n** (dry run).

| Execute the command on the foreground with standard input, standard output and error output available.

Built-in commands

Command	Description
`true`	This command does nothing. Arguments are ignored.
`false`	This command does nothing, except failing with exit code 1. Arguments are ignored.
`echo` *arg*...	Display a line of text.
`exit` *code*	Exit with defined code. Depending on the program arguments and/or the extra rule options '-' this will cause **amk** to exit with the provided code. Please note that `exit 0` has currently no result.
`argfile` *file arg*...	Create an argument file suitable for the **--option-file** (**-f**) option of all the other tools. The first `argfile` argument is the name of the file to be created. Subsequent arguments specify the contents. An existing argument file is not modified unless necessary. So, the argument file itself can be used to create a dependency to options of the command for updating a target.
`rm` [*option*]... *file*...	Remove the specified file(s). The following options are available: **-r**, **--recursive** Remove directories and their contents recursively. **-f**, **--force** Force deletion. Ignore non-existent files, never prompt. **-i**, **--interactive** Interactive. Prompt before every removal. **-v**, **--verbose** Verbose mode. Explain what is being done. **-m** *file* Read options from *file*.. **-?**, **--help** Show usage.

8.2.8. Calling the amk Make Utility

The invocation syntax of **amk** is:

amk [*option*]... [*target*]... [*macro=def*]...

For example:

```
amk test.elf
```

target	You can specify any target that is defined in the makefile. A target can also be one of the intermediate files specified in the makefile.
macro=def	Macro definition. This definition remains fixed for the **amk** invocation. It overrides any regular definitions for the specified macro within the makefiles and from the environment. It is not inherited by subordinate **amk**'s
option	For a complete list and description of all **amk** make utility options, see Section 10.7, *Parallel Make Utility Options*.

Exit status

The make utility returns an exit status of 1 when it halts as a result of an error. Otherwise it returns an exit status of 0.

8.3. Archiver

The archiver **ararc** is a program to build and maintain your own library files. A library file is a file with extension .a and contains one or more object files (.o) that may be used by the linker.

The archiver has five main functions:

- Deleting an object module from the library

- Moving an object module to another position in the library file

- Replacing an object module in the library or add a new object module

- Showing a table of contents of the library file

- Extracting an object module from the library

The archiver takes the following files for input and output:
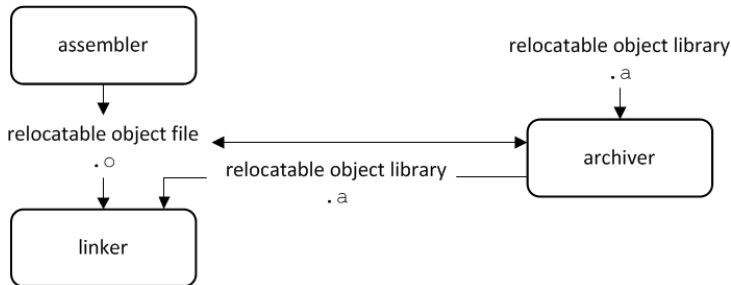
The linker optionally includes object modules from a library if that module resolves an external symbol definition in one of the modules that are read before.

8.3.1. Calling the Archiver

You can create a library in Eclipse, which calls the archiver or you can call the archiver on the command line.

To create a library in Eclipse

Instead of creating an absolute ELF file, you can choose to create a library. You do this when you create a new project with the New C/C++ Project wizard.

1. From the **File** menu, select **New » TASKING ARC C/C++ Project**.

    The New C/C++ Project wizard appears.

2. Enter a project name.

3. In the **Project type** box, select **TASKING ARC Library** and click **Next >**.

4. Follow the rest of the wizard and click **Finish**.

5. Add the files to your project.

6. Build the project as usual. For example, select **Project » Build Project** ().

   Eclipse builds the library. Instead of calling the linker, Eclipse now calls the archiver.

Command line invocation

You can call the archiver from the command line. The invocation syntax is:

ararc *key_option* [*sub_option*...] *library* [*object_file*]

key_option	With a key option you specify the main task which the archiver should perform. You must *always* specify a key option.
sub_option	Sub-options specify into more detail how the archiver should perform the task that is specified with the key option. It is not obligatory to specify sub-options.
library	The name of the library file on which the archiver performs the specified action. You must always specify a library name, except for the options **-?** and **-V**. When the library is not in the current directory, specify the complete path (either absolute or relative) to the library.
object_file	The name of an object file. You must always specify an object file name when you add, extract, replace or remove an object file from the library.

Options of the archiver utility

The following archiver options are available:

Description	Option	Sub-option
Main functions (key options)		
Replace or add an object module	**-r**	**-a -b -c -n -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**	
Sub-options		
Append or move new modules after existing module *name*	**-a** *name*	
Append or move new modules before existing module *name*	**-b** *name*	
Suppress the message that is displayed when a new library is created	**-c**	
Create a new library from scratch	**-n**	
Preserve last-modified date from the library	**-o**	
Print symbols in library modules	**-s{0|1}**	

Description	Option	Sub-option
Replace only newer modules	**-u**	
Verbose	**-v**	
Miscellaneous		
Display options	**-?**	
Display description of one or more diagnostic messages	**--diag**	
Display version header	**-V**	
Read options from *file*	**-f** *file*	

For a complete list and description of all archiver options, see Section 10.8, *Archiver Options*.

8.3.2. Archiver Examples

Create a new library

If you add modules to a library that does not yet exist, the library is created. To create a new library with the name `mylib.a` and add the object modules `cstart.o` and `calc.o` to it:

```
ararc -r mylib.a cstart.o calc.o
```

Add a new module to an existing library

If you add a new module to an existing library, the module is added at the end of the module. (If the module already exists in the library, it is replaced.)

```
ararc -r mylib.a mod3.o
```

Print a list of object modules in the library

To inspect the contents of the library:

```
ararc -t mylib.a
```

The library has the following contents:

```
cstart.o
calc.o
mod3.o
```

Move an object module to another position

To move `mod3.o` to the beginning of the library, position it just before `cstart.o`:

```
ararc -mb cstart.o mylib.a mod3.o
```

Delete an object module from the library

To delete the object module `cstart.o` from the library `mylib.a`:

```
ararc -d mylib.a cstart.o
```

Extract all modules from the library

Extract all modules from the library `mylib.a`:

```
ararc -x mylib.a
```

8.4. HLL Object Dumper

The high level language (HLL) dumper **hldumparc** is a program to dump information about an absolute object file (`.elf`). Key features are a disassembler with HLL source intermixing and HLL symbol display and a HLL symbol listing of static and global symbols.

8.4.1. Invocation

Command line invocation

You can call the HLL dumper from the command line. The invocation syntax is:

hldumparc [*option*]... *file*...

The input file must be an ELF file with or without DWARF debug info (`.elf`).

The HLL dumper can process multiple input files. Files and options can be intermixed on the command line. Options apply to all supplied files. If multiple files are supplied, the disassembly of each file is preceded by a header to indicate which file is dumped. For example:

========== file.elf ==========

For a complete list and description of all options, see Section 10.9, *HLL Object Dumper Options*. With hldumparc --help you will see the options on stdout.

8.4.2. HLL Dump Output Format

The HLL dumper produces output in text format by default, but you can also specify the XML output format with option **--output-type=xml**. The XML output is mainly for use in the Eclipse editor. Alternatively, you can use option **--adx-format** to produce output in the ADX address list format. For more information about this format, see *ADX Specification - Address List Format for A2L Address Calculation - Compiler vendors, Version 1.10, 2015-04-27*.

The output is printed on stdout, unless you specify an output file with **--output=***filename*.

The parts of the output are dumped in the following order:

1. Module list

2. Section list

3. Call graph using the DWARF debug info

4. Section dump (disassembly)

5. HLL symbol table

6. Assembly level symbol table

7. Note sections

8. Debug control flow section

With the option **--dump-format=***flag* you can control which parts are shown. By default, all parts are shown, except for parts 3 and 8.

Example

Suppose we have a simple "Hello World" program in a file called `hello.c`. We call the control program as follows:

```
ccarc -g -t --control-flow-info hello.c
```

Option **-g** tells to include DWARF debug information. Option **-t** tells to keep the intermediate files. Option **--control-flow-info** adds control flow information to the output file. This command results (among other files) in the file `hello.elf` (the absolute object file).

We can dump information about the ELF file with the following command:

```
hldumparc -F3 hello.elf
```

Option **-F3** enables all parts. A possible output could be (just a fraction of the actual output is shown):

```
---------- Module list ----------

Name    Full path
hello.c hello.c

---------- Section list ----------

Address   Size   Align Type     Name
00000494    20      4 text    .text.hello.main
0000056c     6      1 romdata .rodata.hello..1.str
00100004     4      4 bss     .sdata.hello.world
00000572    11      1 romdata .rodata.hello..2.str

---------- Call graph using the DWARF debug info ----------

+-- 0x00000494 main
    |
    +-- 0x000004e4 printf
        |
        +-- 0x000000dc _doprint
            |
            +-- 0x00000238 _io_putc
            |   |
            |   +-- 0x0000046c fputc
            |       |
            |       +-- 0x00000228 _flsbuf
            |           |
            |           +-- 0x00000158 _dofls
            |               |
            |               +-- 0x0000031c _host_write
            |               |   |
```

```
|                      |    +-- 0x00000364 _dbg_trap
|                      |
|                      +-- 0x000003c4 _fflush
|                      |    |
|                      |    +-- 0x0000031c _host_write *
|                      |    |
|                      |    +-- 0x000002e8 _host_lseek
|                      |         |
|                      |         +-- 0x00000364 _dbg_trap
|                      |
|                      +-- 0x0000031c _host_write *
|
+-- 0x00000238 _io_putc *


---------- Section dump ----------

                          .section .text.hello.main, at(0x00000494)
00000494 f1 c0        main:   push_s %blink
00000496 c3 40 00 00 72 05        mov_s %r0,1394

0000049c 00 50                 ld_s %r1,[%gp,0]
0000049e 4a 08 00 00           bl      printf
000004a2 0c 70                 mov_s %r0,0
000004a4 d1 c0                 pop_s %blink
000004a6 e0 7e                 j_s [%blink]
                              .endsec

       .section .data, '.rodata.hello..1.str', at(0x0000056c)
       .db 77,6f,72,6c,64,00                              ; world.
       .endsec

       .section .data, '.rodata.hello..2.str', at(0x00000572)
       .db 48,65,6c,6c,6f,20,25,73,21,0a,00               ; Hello %s!..
       .endsec

       .section .data, '.sdata.hello.world', at(0x00100004)
world:
       .ds 4
       .endsec

---------- HLL symbol table ----------

Address     Size HLL Type            Name
00000294     46 void                _START()
00000494     20 int                 main()
000004e4     66 int                 printf(const char * restrict format, ...)
00100004      4 char            * world [hello.c]
00100008     20 struct              _dbg_request [dbg.c]
0010001c     80 static char         stdin_buf[80] [_iob.c]
0010006c     80 static char         stdout_buf[80] [_iob.c]
```

```
001000bc    200 struct _iobuf          _iob[10] [_iob.c]

---------- Assembly level symbol table ----------

Address  Size     Type Name
00000000
00000000             [.sdata.hello.world]
00000000             hello.c
00000294     46 code _START
00000494     20 code main
000004e4     66 code printf
00100004      4 data world
00100008     20 data _dbg_request
0010001c     80 data stdin_buf
0010006c     80 data stdout_buf
001000bc    200 data _iob

---------- .note sections ----------
00000104 type: TASKING COMPILER NAME
00000110 name: TASKING
00000118 desc: carc

---------- Debug control flow section ----------
start offset : 0
start address: 0x00000494
code size    : 20
#entries     : 0
```

Module list

This part lists all modules (C files) found in the object file(s). It lists the filename and the complete path name at the time the module was built.

Section list

This part lists all sections found in the object file(s).

Address	The start address of the section. Hexadecimal, 8 digits, 32-bit.
Size	The size (length) of the section in bytes. Decimal, filled up with spaces.
Align	The alignment of the section in number of bytes. Decimal, filled up with spaces.
Type	The section type.
Name	The name of the section. Sections within square brackets **[ ]** will be copied during initialization from ROM to the corresponding section name in RAM.

With option **--sections=**name[,name]... you can specify a list of sections that should be dumped.

255

Call graph

The linker can generate a call graph in the linker map file. However, if you only have an ELF file and you need to test it, you can use the option **--dump-format=+callgraph**. You can then step through the call graph to identify the flow for debugging purposes. Some notes about the call graph:

- The call graph starts with the default entry point of the application.

- Recursive calls are marked with 'R'.

- Inline functions are marked with 'I'.

- Indirect function calls are marked with '__INDIRECT__'.

- A function is analyzed only once. When a function is called again, it is not analyzed again and this is marked with '*'.

- By default the DWARF debug information is used to generate the call graph. When no DWARF information is available the ELF information is used. Inline functions can only be detected and dumped when DWARF information is available.

- With option **--call-graph-elf-mode** you can force the call graph to use ELF symbols even when DWARF information is available. This can be useful when you want to dump information from an assembly function.

- With option **--call-graph-root=***function* you can specify the address or function name where to start the call graph (default: `main()`).

Section dump

This part contains the disassembly. It consists of the following columns:

address column	Contains the address of the instruction or directive that is shown in the disassembly. If the section is relocatable the section start address is assumed to be 0. The address is represented in hexadecimal and has a fixed width. The address is padded with zeros. No 0x prefix is displayed. For example, on a 32-bit architecture, the address 0x32 is displayed as 00000032.
encoding column	Shows the hexadecimal encoding of the instruction (code sections) or it shows the hexadecimal representation of data (data sections). The encoding column has a maximum width of eight digits, i.e. it can represent a 32-bit hexadecimal value. The encoding is padded to the size of the data or instruction. For example, a 16-bit instruction only shows four hexadecimal digits. The encoding is aligned left and padded with spaces to fill the eight digits.
label column	Displays the label depending on the option **--symbols=**[**hll**|**asm**|**none**]. The default is **asm**, meaning that the low level (ELF) symbols are used. With **hll**, HLL (DWARF) symbols are used. With **none**, no symbols will be included in the disassembly.

disassembly column   For code sections the instructions are disassembled. Operands are replaced with labels, depending on the option **--symbols=[hll|asm|none]**.

The contents of data sections are represented by directives. A new directive will be generated for each symbol. ELF labels in the section are used to determine the start of a directive. Sections within square brackets **[ ]** will be copied during initialization from ROM to the corresponding section name in RAM. ROM sections are represented with .db, .dh, .dw, .dd kind of directives, depending on the size of the data. RAM sections are represented with .ds directives, with a size operand depending on the data size. This can be either the size specified in the ELF symbol, or the size up to the next label.

With option **--hex**, no directives will be generated for ROM data sections and no disassembly dump will be done for code sections. Instead a hex dump is done with the following format:

```
AAAAAAAA H0 H1 H2 H3 H4 H5 H6 H7 H8 H9 HA HB HC HD HE HF RRRRRRRRRRRRRRRR
```

where,

A = Address (8 digits, 32-bit)

Hx = Hex contents, one byte (16 bytes max)

R = ASCII representation (16 characters max)

For example:

```
                        section 7 (.rodata.hello..2.str):
00000572 48 65 6c 6c 6f 20 25 73 21 0a 00                 Hello %s!..
```

With option **--hex**, RAM sections will be represented with only a start address and a size indicator:

```
AAAAAAAA Space: 48 bytes
```

With option **--disassembly-intermix** you can intermix the disassembly with HLL source code.

HLL symbol table

This part contains a symbol listing based on the HLL (DWARF) symbols found in the object file(s). The symbols are sorted on address.

Address	The start address of the symbol. Hexadecimal, 8 digits, 32-bit.
Size	The size of the symbol from the DWARF info in bytes.
HLL Type	The HLL symbol type.
Name	The name of the HLL symbol.

HLL arrays are indicated by adding the size in square brackets to the symbol name. For example:

```
0010001c     80 static char          stdin_buf[80] [_iob.c]
```

With option **--expand-symbols=+basic-types** HLL struct and union symbols are listed including all fields. Array members are expanded in one array member per line regardless of the HLL type. For example:

```
0010001c    80 static char          stdin_buf[80] [_iob.c]
0010001c     1   char
0010001d     1   char
0010001e     1   char
  ...
0010006b     1   char
```

HLL struct and union symbols are listed by default without fields. For example:

```
00100008    20 struct               _dbg_request [dbg.c]
```

With option **--expand-symbols** all struct, union and array fields are included as well. For the fields the types and names are indented with two spaces. For example:

```
00100008    20 struct               _dbg_request [dbg.c]
00100008     4   int                  _errno
0010000c     1   enum                 nr
00100010    12   union                u
00100010     4     struct               exit
00100010     4       int                  status
00100010     8     struct               open
00100010     4       const char         * pathname
00100014     2       unsigned short int   flags
  ...
```

Functions are displayed with the full function prototype. Size is the size of the function. HLL Type is the return type of the function. For example:

```
000004e4    66 int                   printf(const char * restrict format, ...)
```

The local and static symbols get an identification between square brackets. The filename is printed and if a function scope is known the function name is printed between the square brackets as well. If multiple files with the same name exist, the unique part of the path is added. For example:

```
00004100     4 int                   count [file.c, somefunc()]
00004104     4 int                   count [x\a.c]
00004108     4 int                   count [y\a.c, foo()]
```

Global symbols do not get information in square brackets.

Assembly level symbol table

This part contains a symbol listing based on the assembly level (ELF) symbols found in the object file(s). The symbols are sorted on address.

Address	The start address of the symbol. Hexadecimal, 8 digits, 32-bit.
Size	The size of the symbol from the ELF info in bytes. If this field is empty, the size is zero.

Type	Code or Data, depending on the section the symbol belongs to. If this field is empty, the symbol does not belong to a section.
Name	The name of the ELF symbol. Symbol names within square brackets **[ ]** are the names of sections that will be copied during initialization from ROM to the corresponding section name in RAM.

Debug control flow section

When control flow information is present in the ELF file (control program option **--control-flow-info**), this part shows information about the basic blocks and their relation.

start offset	The start seek offset in bytes from the beginning of the section.
start address	The start address of the basic block.
code size	The code size of the basic block.
#entries	The number of successor basic blocks. This value can be 0 if there are no successors.
dest. offset	The destination offset in bytes to the first, second, ... successor from the beginning of the section.

Chapter 9. Using the Debugger

This chapter describes the debugger and how you can run and debug a C application. This chapter only describes the TASKING specific parts.

9.1. Reading the Eclipse Documentation

Before you start with this chapter, it is recommended to read the Eclipse documentation first. It provides general information about the debugging process. This chapter guides you through a number of examples using the TASKING debugger with simulation as target.

You can find the Eclipse documentation as follows:

1. Start Eclipse.

2. From the **Help** menu, select **Help Contents**.

    The help screen overlays the Eclipse Workbench.

3. In the left pane, select **C/C++ Development User Guide**.

4. Open the **Getting Started** entry and select **Debugging projects**.

    This Eclipse tutorial provides an overview of the debugging process. Be aware that the Eclipse example does not use the TASKING tools and TASKING debugger.

9.2. Debugging an ARC Project

In order to debug an ARC project, follow the steps below.

1. Make sure the Synopsys ARC® nSIM Instruction Set Simulator is present. This simulator is not part of the SmartCode product, you need to order it separately from Synopsys®. Then copy the 64-bit Windows file `libsim.dll` to the following directory:

    `<`*installation-dir*`>\eclipse\plugins\com.tasking.arc.debug.win32.x86_64_1.4.0.0\carc\bin`

2. Create an ARC project (for example, `myproject`), as explained in the *Getting Started with TASKING SmartCode*. Enable at least the **Debug** configuration.

3. Build the ARC project.

    This results in an ELF file (`.elf`).

4. Create a debug configuration for the ARC project with the Synopsys nSIM Simulator as target, if you have not already done so when creating the project. See Section 9.3, *Creating a Customized Debug Configuration*.

5.   Start the debugger. From the **Debug** menu select **Debug** *project*. Alternatively you can click the 🔆
     button in the main toolbar.

Note that when you have a project reference from a TriCore project to the ARC project, the ARC project
is built automatically when you build the TriCore project.

9.3. Creating a Customized Debug Configuration

Before you can debug a project, you need a Debug launch configuration. Such a configuration, identified
by a name, contains all information about the debug project: which debugger is used, which project is
used, which binary debug file is used, which perspective is used, ... and so forth.

If you want to debug on a target board, you have to create a custom debug configuration for your target
board, otherwise you have to create a debug launch configuration for the simulator.

> To debug a project, you need at least one opened and active project in your workbench. In this chapter,
> it is assumed that the `myproject` is opened and active in your workbench.

Create or customize your debug configuration

To create or change a debug configuration follow the steps below.

1.   From the **Debug** menu, select **Debug Configurations...**

     The Debug Configurations dialog appears.

2.   Select **TASKING C/C++ Debugger** and click the **New launch configuration** button (📄
) to add a new configuration.
     Or: In the left pane, select the configuration you want to change, for example, **TASKING C/C++
     Debugger » myproject**.

3.   In the **Name** field enter the name of the configuration. By default, this is the name of the project, but
     you can give your configuration any name you want to distinguish it from the project name. For
     example enter `myproject.simulator` to identify the simulator debug configuration.

4.   On the **Target** tab, select the **Synopsys NSIM Simulator**.

The dialog shows several tabs.

Target tab

On the **Target** tab you can select on which target the application should be debugged. An application
can run on an external evaluation board, or on a simulator using your own PC. On this tab you can also
select the connection settings. The information in this tab is based on the Debug Target Configuration
(DTC) files.
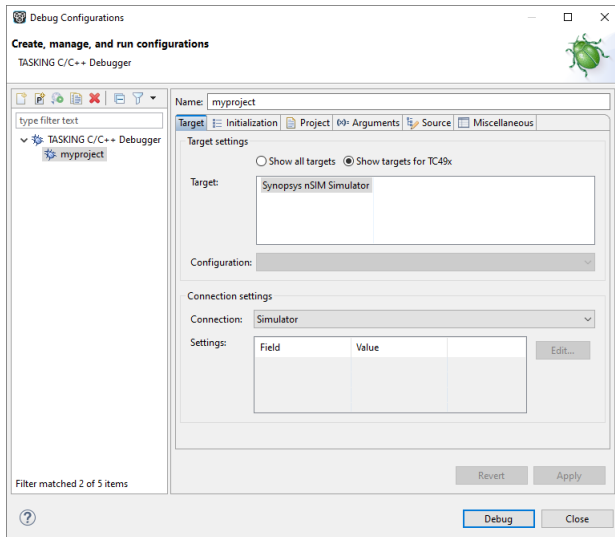
Initialization tab

On the **Initialization** tab enable one or more of the following options:
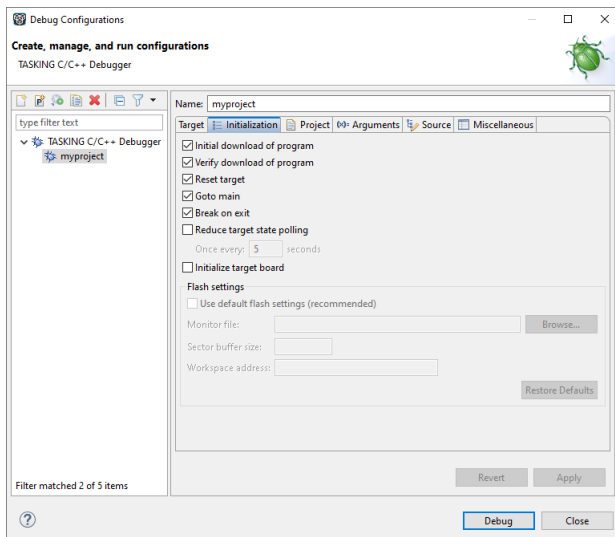
- **Initial download of program**

  If enabled, the target application is downloaded onto the target. If disabled, only the debug information in the file is loaded, which may be useful when the application has already been downloaded (or flashed) earlier. If downloading fails, the debugger will shut down.

- **Verify download of program**

If enabled, the debugger verifies whether the code and data has been downloaded successfully. This takes some extra time but may be useful if the connection to the target is unreliable.

• **Reset target**

If enabled, the target is immediately reset after downloading has completed. Registers that have the `init` resource set in the `.dtc` file, are reset to their default value. Execution stops at the reset vector.

• **Goto main**

If enabled, only the C startup code is processed when the debugger is launched. The application stops executing when it reaches the first C instruction in the function `main()`. Usually you enable this option in combination with the option **Reset Target**.

• **Break on exit**

If enabled, the target halts automatically when the `exit()` function is called.

• **Reduce target state polling**

If you have set a breakpoint, the debugger checks the status of the target every *number* of seconds to find out if the breakpoint is hit. In this field you can change the polling frequency.

• **Initialize Target Board**

Some target boards contain a power supply chip which needs to be initialized every time after power-on, before hardware debugging is started. If enabled, the target board is initialized automatically before the start of a debug session.

This option is not supported for the ARC/PPU.

Initialization tab: Flash settings

• **Use default flash settings (recommended)**

By default, the flash settings are derived from the `.dtc` file for the chosen target processor. So, when you change processors the flash settings change automatically. If you do not want that, you can specify your own flash settings. You can click **Restore Defaults** to restore the default flash settings.

• **Monitor file**

Filename of the monitor, usually an Intel Hex or S-Record file.

• **Sector buffer size**

Specifies the buffer size for buffering a flash sector.

• **Workspace address**

The address of the workspace of the flash programming monitor.

Project tab

On the **Project** tab, you can set the properties for the debug configuration such as a name for the project and the application binary file(s) which are used when you choose this configuration.
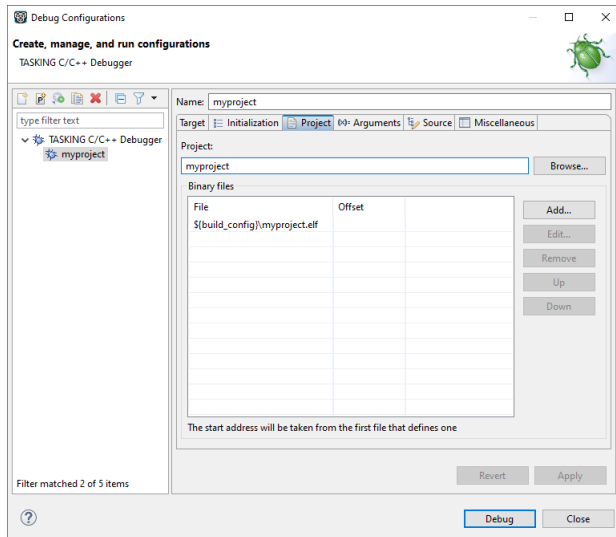
- In the **Project** field, you can choose the project for which you want to make a debug configuration. Because the project myproject is the active project, this project is filled in automatically. Click the Browse... button to select a different project. Only the *opened* projects in your workbench are listed.

- In the **Binary files** group box, you can choose one or more binary files to debug. The file myproject.elf is automatically selected from the active project.

  The order of the binary files matters. Use the **Up** and **Down** buttons to change the order. If there are multiple files, the application start address is taken from the first file that defines one. An ELF file always defines one, whereas Hex files may not. Try to avoid address overlaps.

  Note that conflicts between symbols could arise, for example when you download two ELF files that both contain the function main(). When you download multiple files, we recommend that the first binary file is an ELF file that contains the startup code and main() and that the other files are auxiliary Hex files.

  To add a binary file

  1. Click **Add...** to add a binary file.

     The Add Binary File dialog appears.
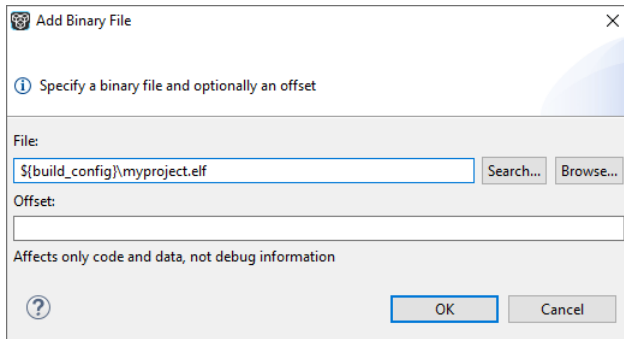
2. Specify the binary file, use the **Search...** button to select one from the active project, or use the **Browse...** button to search the file system.

3. Optionally, specify an address offset. The value will be added to all target addresses in the binary file.

   Note that the address offset will be applied only to code, data and the start address, not to debug information. Specifying a non-zero offset is not recommended for an ELF/DWARF file. If the offset causes an address to underflow or overflow an error occurs.

Arguments tab

If your application's `main()` function takes arguments, you can pass them in this tab. Arguments are conventionally passed in the `argv[]` array. Because this array is allocated in target memory, make sure you have allocated sufficient memory space for it.

- In the C/C++ perspective select **Project » Properties for** to open the Properties dialog. Expand **C/C++ Build » Startup Configuration**. Enable the option **Enable passing argc/argv to main()** and specify a **Buffer size for argv**.
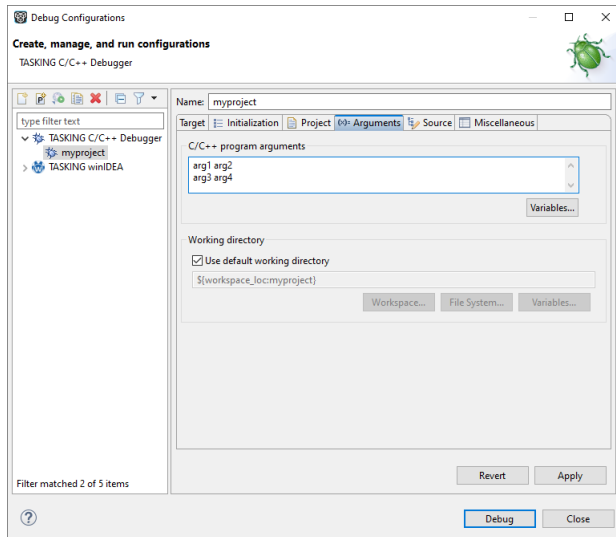
Source tab

On the **Source** tab, you can add additional source code locations in which the debugger should search for debug data.
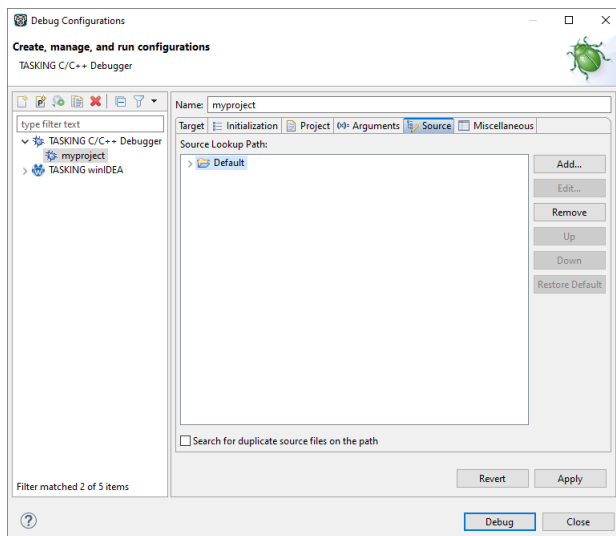
• Usually, the default source code location is correct.

Miscellaneous tab

On the **Miscellaneous** tab you can specify several file locations.
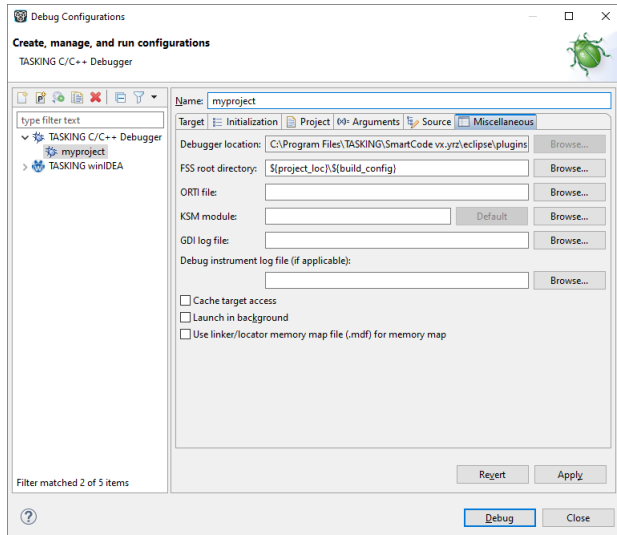
- **Debugger location**

  The location of the debugger itself. This should not be changed.

- **FSS root directory**

  The initial directory used by file system simulation (FSS) calls. See the description of the FSS view.

- **ORTI file** and **KSM module**

  If you wish to use the debugger's special facilities for kernel-aware debugging, specify the name of a Kernel Debug Interface (KDI) compatible KSM module (shared library) in the appropriate edit box. See also the description of the RTOS view.

- **GDI log file** and **Debug instrument log file**

  You can use the options GDI log file and Debug instrument log file (if applicable) to control the generation of internal log files. These are primarily intended for use by or at the request of TASKING support personnel.

- **Cache target access**

  Except when using a simulator, the debugger's performance is generally strongly dependent on the throughput and latency of the connection to the target. Depending on the situation, enabling this option may result in a noticeable improvement, as the debugger will then avoid re-reading registers and memory while the target remains halted. However, be aware that this may cause the debugger to show the wrong data if tasks with a higher priority or external sources can influence the halted target's state.

- **Launch in background**

  When this option is disabled you will see a progress bar when the debugger starts. If you do not want to see the progress bar and want that the debugger launches in the background you can enable this option.

- **Use linker/locator memory map file (.mdf) for memory map**

  You can use this option to find errors in your application that cause access to non-existent memory or cause an attempt to write to read-only memory. When building your project, the linker/locator creates a memory description file (`.mdf`) file which describes the memory regions of the target you selected in your project properties. The debugger uses this file to initialize the debugging target.

  This option is only useful in combination with a simulator as debug target. The debugger may fail to start if you use this option in combination with other debugging targets than a simulator.

9.4. Troubleshooting

If the debugger does not launch properly, this is likely due to mistakes in the settings of the execution environment or to an improper connection between the host computer and the execution environment. Always read the notes for your particular execution environment.

Some common problems you may check for, are:

Problem	Solution
Wrong device name in the launch configuration	Make sure the specified device name is correct.
Invalid baud rate	Specify baud rate that matches the baud rate the execution environment is configured to expect.
No power to the execution environment.	Make sure the execution environment or attached probe is powered.
Cable connected to the wrong port on the execution environment or host.	Some target machines and hosts have several ports. Make sure you connect the cable to the correct port.
Conflict between communication ports.	A device driver or background application may use the same communications port on the host system as the debugger. Disable any service that uses the same port-number or choose a different port-number if possible.
Port already in use by another user.	The port may already be in use by another user on some UNIX hosts, or being allocated by a login process. Some target machines and hosts have several ports. Make sure you connect the cable to the correct port.

9.5. TASKING Debug Perspective

After you have launched the debugger, you are either asked if the TASKING Debug perspective should be opened or it is opened automatically. The Debug perspective consists of several views.

To open views in the Debug perspective:

1. Make sure the Debug perspective is opened

2. From the **Window** menu, select **Show View »**

3. Select a view from the menu or choose **Other...** for more views.

By default, the Debug perspective is opened with the following views:
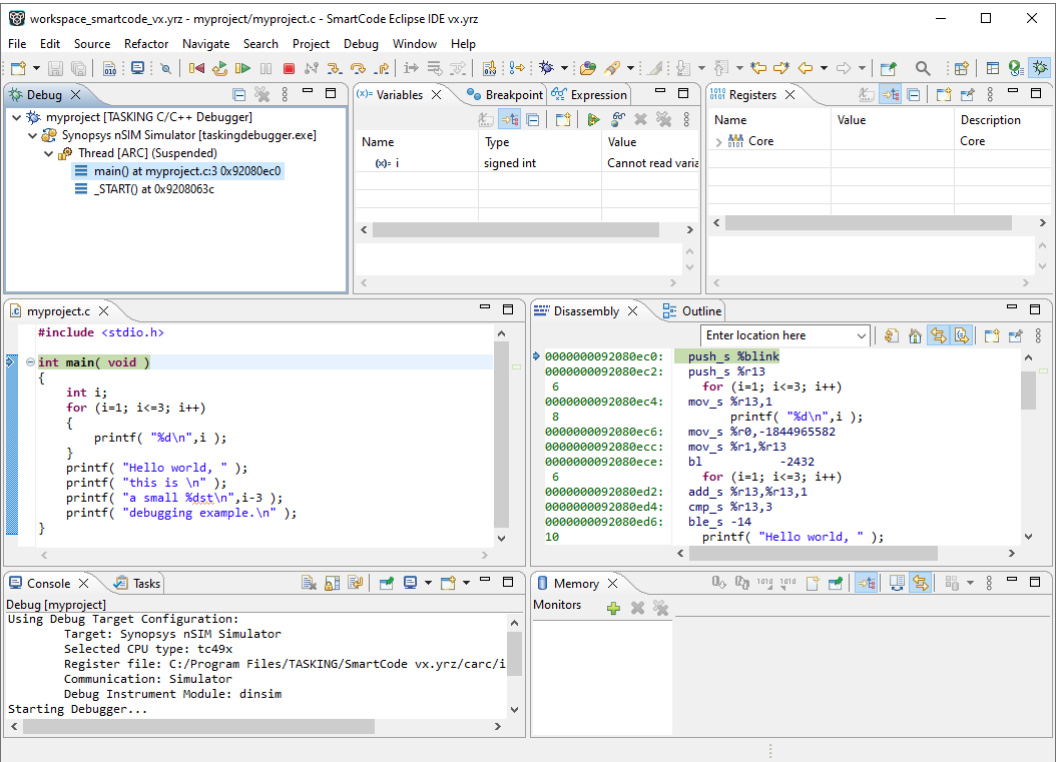
9.5.1. Debug View

The Debug view shows the target information in a tree hierarchy shown below with a sample of the possible icons:

Icon	Session item	Description
	Launch instance	Launch configuration name and launch type
	Debugger instance	Debugger name and state
	Thread instance	Thread number and state

Icon	Session item	Description
≡ ≡	Stack frame instance	Stack frame number, function, file name, and file line number

Stack display

During debugging (running) the actual stack is displayed as it increases or decreases during program execution. By default, all views present information that is related to the current stack item (variables, memory, source code etc.). To obtain the information from other stack items, click on the item you want.

The Debug view displays stack frames as child elements. It displays the reason for the suspension beside the thread, (such as end of stepping range, breakpoint hit, and signal received). When a program exits, the exit code is displayed.

The Debug view contains numerous functions for controlling the individual stepping of your programs and controlling the debug session. You can perform actions such as terminating the session and stopping the program. All functions are available from the right-click menu, though commonly used functions are also available from the toolbar.

Controlling debug sessions

Icon	Action	Description
✳	Remove all	Removes all terminated launches.
▮◀	Reset target system	Resets the target system. Registers that have the `init` resource set in the `.dtc` file, are reset to their default value. Execution stops at the reset vector.
↻	Restart	Resets the target system and restarts the application. The application stops executing when it reaches the first C instruction in the function `main()`.
▯▶	Resume	Resumes the application after it was suspended (manually, breakpoint, signal).
▯▯	Suspend	Suspends the application (pause). Use the Resume button to continue.
◔	Relaunch	Right-click menu. Restarts the selected debug session when it was terminated. If the debug session is still running, a new debug session is launched.
▤	Reload current application	Reloads the current application without restarting the debug session. The application does restart of course.
▪	Terminate	Ends the selected debug session and/or process. Use Relaunch to restart this debug session, or start another debug session.
▦	Terminate all	Right-click menu. As terminate. Ends *all* debug sessions.
▪ₓ	Terminate and remove	Right-click menu. Ends the debug session and removes it from the Debug view.
▪▶	Terminate and Relaunch	Right-click menu. Ends the debug session and relaunches it. This is the same as choosing Terminate and then Relaunch.

Icon	Action	Description
	Disconnect	Detaches the debugger from the selected process (useful for debugging attached processes).

Stepping through the application

Icon	Action	Description
	Step into	Steps to the next source line or instruction.
	Step over	Steps over a called function. The function is executed and the application suspends at the next instruction after the call.
	Step return	Executes the current function. The application suspends at the next instruction after the return of the function.
	Instruction stepping	Toggle. If enabled, the stepping functions are performed on instruction level instead of on C source line level.
	Interrupt aware stepping	Toggle. If an interrupt source continues generating interrupts while the target is stopped (either manually or by hitting a breakpoint), a following single step will always enter the Interrupt Service Routine (ISR). This can lead to some problems during single stepping. With interrupt aware stepping enabled, interrupts are temporarily disabled after the target has stopped. When execution resumes the interrupts are restored.

Miscellaneous

Icon	Action	Description
	Copy Stack	Right-click menu. Copies the stack as text to the windows clipboard. You can paste the copied selection as text in, for example, a text editor.
	Edit *project...*	Right-click menu. Opens the debug configuration dialog to let you edit the current debug configuration.
	Edit Source Lookup...	Right-click menu. Opens the Edit Source Lookup Path window to let you edit the search path for locating source files.

9.5.2. Breakpoints View

You can add, disable and remove breakpoints by clicking in the marker bar (left margin) of the Editor view. This is explained in the Getting Started manual.

Description

The Breakpoints view shows a list of breakpoints that are currently set. The button bar in the Breakpoints view gives access to several common functions. The right-most button opens the Breakpoints menu.

Types of breakpoints

To access the breakpoints dialog, add a breakpoint as follows:

1. Click the **Add TASKING Breakpoint** button (⚙).

   The Breakpoints dialog appears.

Each tab lets you set a breakpoint of a specific type. You can set the following types of breakpoints:

- **File breakpoint**
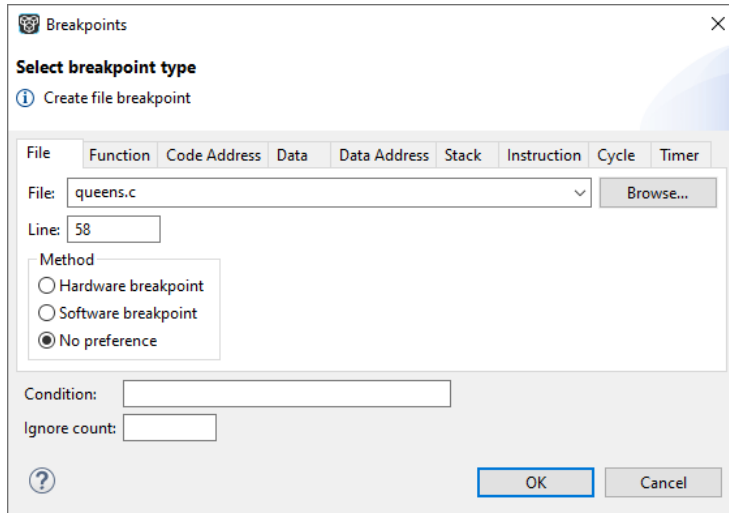
If a debug session is active, the **File** drop-down box is filled with all source files as present in the debug information in the ELF file. This can include files not present in the Eclipse project (for example from libraries). If a file could be matched to a file in the active Eclipse project it will show as an Eclipse project relative filename.

The target halts when it reaches the specified line of the specified source file. Note that it is possible that a source line corresponds to multiple addresses, for example when a header file has been included into two different source files or when inlining has occurred. If so, the breakpoint will be associated with all those addresses. It is also possible that on some files no line breakpoints can be set because the debugger lacks line information.

- **Function**
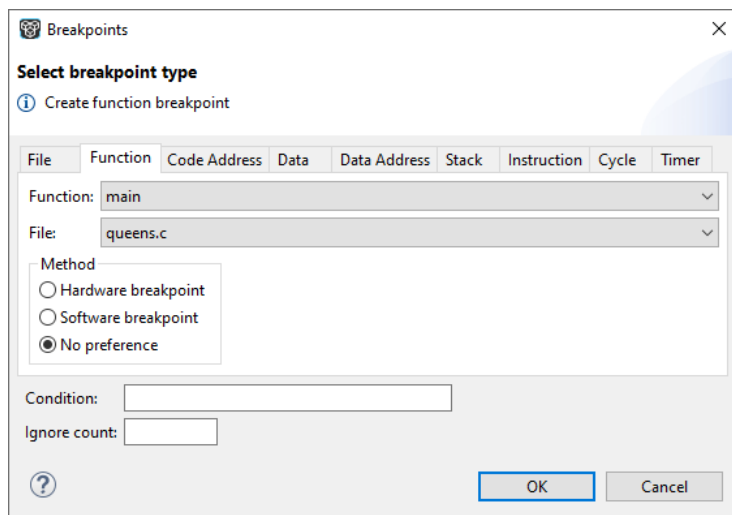
The **Function** drop-down box is filled with all functions from the debug information and the symbol table (if not already in the debug information). You can use the **File** drop-down box to filter the list of functions. If you select **<all>** you will see the filenames (between parentheses) behind each entry in the **Function** drop-down box. Functions marked with *function_name* [*section*] originate from the symbol table. These functions are normally not associated with a filename and will therefore be included if **<unknown>** is selected in the **File** drop-down box. Functions marked '*filename*'::*function_name* are static functions.

The target halts when it reaches the first line of the specified function. Note that function breakpoints generally will not work on inlined instances of a function.

- **Code Address**
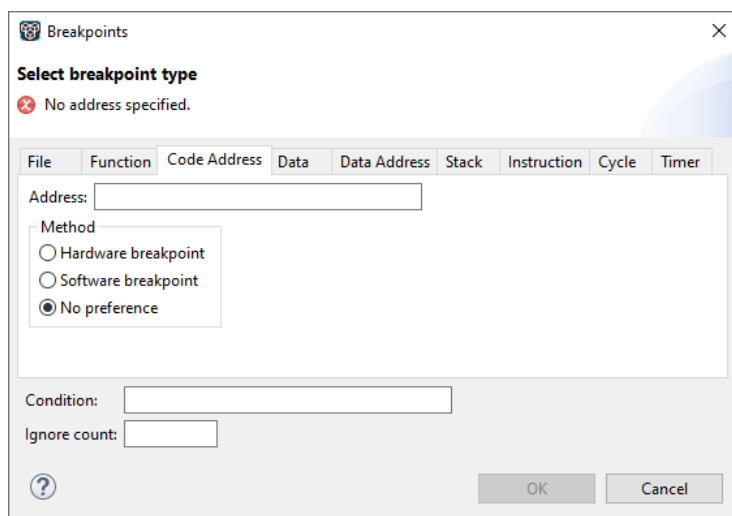
The target halts when it reaches the specified instruction address.

- **Data**
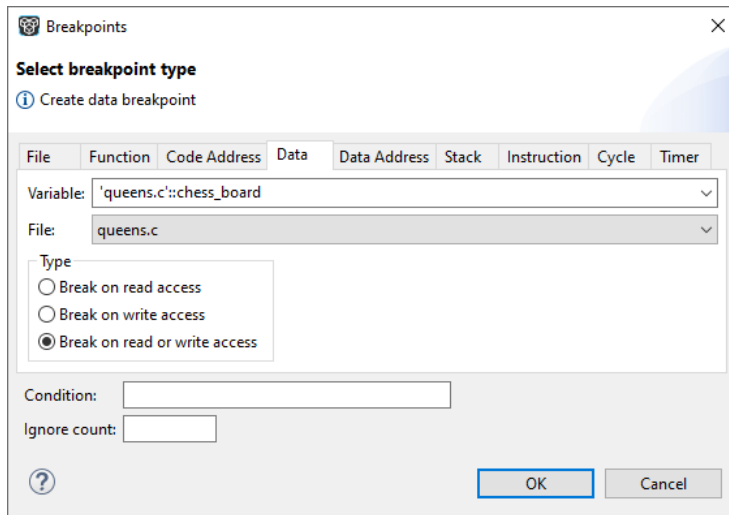
The **Variable** drop-down box is filled with all variables from the debug information and the symbol table (if not already in the debug information), but you can also enter text yourself. If a label is filled in, the size will be 1 MAU. You can use the **File** drop-down box to filter the list of variables. If you select **<all>** you will see the filenames (between parentheses) behind each entry in the **Variables** drop-down box. Variables marked with *variable_name* [*section*] originate from the symbol table. These variables are normally not associated with a filename and will therefore be included if **<unknown>** is selected in the **File** drop-down box. Variables marked '*filename*'::*variable_name* are static.

The target halts when the given variable is read or written to, as specified.

- **Data Address**

The target halts when the given memory range (specified in terms of an absolute **Address** and a **Length** in MAUs) is read or written to, as specified.

This type of breakpoint is not supported for the ARC/PPU.

- **Stack**

The target halts when it reaches the specified stack level.

- **Instruction**
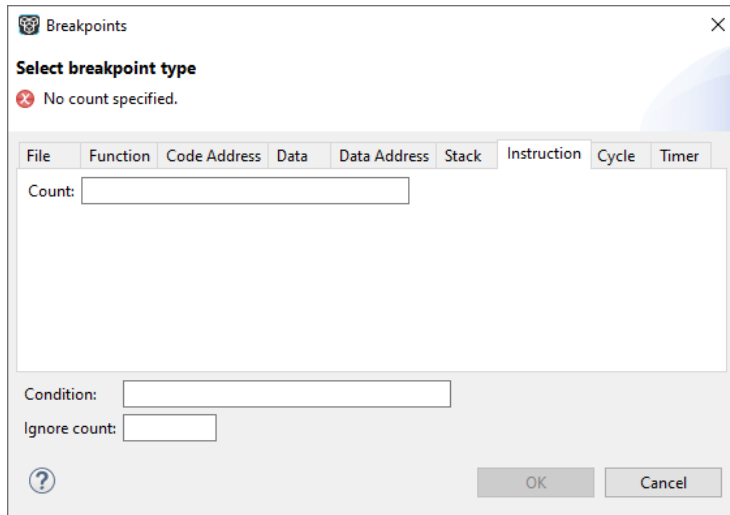
The target halts when the given number of instructions (**Count**) has been executed.

This type of breakpoint is not supported for the ARC/PPU.

• **Cycle**
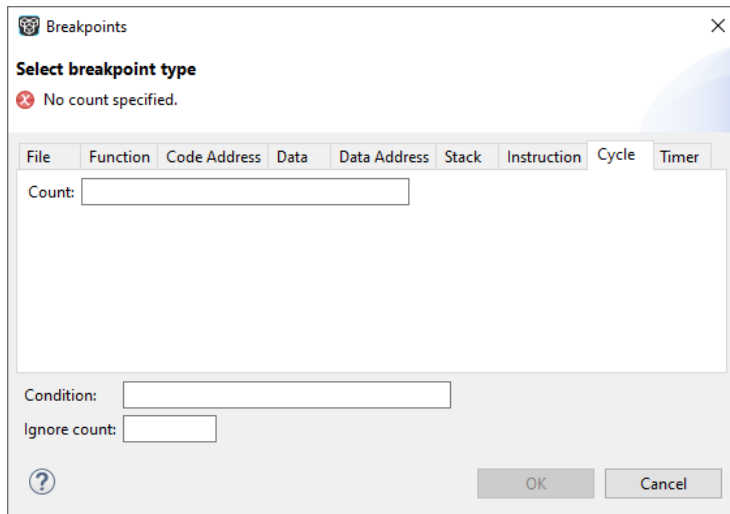
The target halts when the given number of clock cycles (**Count**) has elapsed.

This type of breakpoint is not supported for the ARC/PPU.

• **Timer**
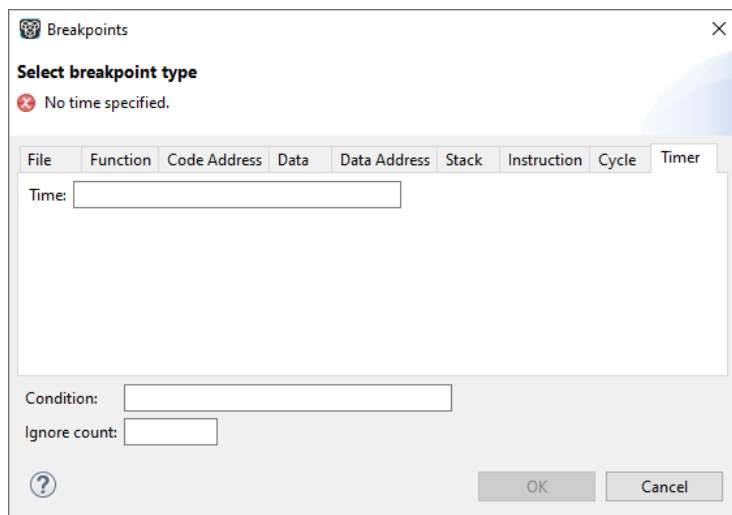
The target halts when the given amount of **Time** elapsed. The value entered is interpreted by the debug instrument.

This type of breakpoint is not supported for the ARC/PPU.

In addition to the type of the breakpoint, you can specify the condition that must be met to halt the program.

In the **Condition** field, type a condition. The condition is an expression which evaluates to 'true' (non-zero) or 'false' (zero). The program only halts on the breakpoint if the condition evaluates to 'true'.

In the **Ignore count** field, you can specify the number of times the breakpoint is ignored before the program halts. For example, if you want the program to halt only in the fifth iteration of a while-loop, type '4': the first four iterations are ignored.

9.5.3. File System Simulation (FSS) View

Description

The File System Simulation (FSS) view is automatically opened when the target requests FSS input or generates FSS output. The virtual terminal that the FSS view represents, follows the VT100 standard. If you right-click in the view area of the FSS view, a menu is presented which gives access to some self-explanatory functions.

Per debugging session, you can have more than one FSS view, each of which is associated with a positive integer. By default, the view "FSS #1" is associated with the standard streams `stdin`, `stdout`, `stderr` and `stdaux`. Other views can be accessed by opening a file named "terminal window <number>", as shown in the example below.

VT100 characteristics

The `queens` example demonstrates some of the VT100 features. (You can find the `queens` example in the `<installation path>\carc\examples` directory from where you can import it into your

workspace.) Per debugging session, you can have more than one FSS view, each of which is associated with a positive integer. By default, the view "FSS #1" is associated with the standard streams stdin, stdout, stderr and stdaux. Other views can be accessed by opening a file named "terminal window <number>", as shown in the example below.

```
FILE * f3 = fopen("terminal window 3", "rw");
fprintf(f3, "Hello, window 3.\n");
fclose(f3);
```

You can set the initial working directory of the target application in the Debug configuration dialog (see also Section 9.3, *Creating a Customized Debug Configuration*):

1. On the **Debugger** tab, select the **Miscellaneous** sub-tab.

2. In the **FSS root directory** field, specify the FSS root directory.

The FSS implementation is designed to work without user intervention. Nevertheless, there are some aspects that you need to be aware of.

First, the interaction between the C library code (in the files dbg*.c and dbg*.h; see Section 12.1.4, *dbg.h*) and the debugger takes place via a breakpoint, which incidentally is not shown in the Breakpoints view. Depending on the situation this may be a hardware breakpoint, which may be in short supply.

Secondly, proper operation requires certain code in the C library to have debug information. This debug information should normally be present but might get lost when this information is stripped later in the development process.

9.5.4. Disassembly View

The Disassembly view shows target memory disassembled into instructions and / or data. If possible, the associated C source code is shown as well if you click the **Show Source** button ( ).

The left part of the Disassembly view shows the addresses, opcodes and/or function offsets depending on what you selected in the right-click menu.

The right part of the Disassembly view shows the disassembly instructions and/or the C source code and symbols depending on what you selected in the right-click menu. Right-click in the right part and select **Preferences** to open the Disassembly Preferences dialog.

To view the contents of a specific memory location, type the address in the **Enter location here** field.

9.5.5. Expressions View

The Expressions view allows you to evaluate and watch regular C expressions.

To add an expression:

Click **OK** to add the expression.

1. Right-click in the Expressions View and select **Add Watch Expression**.

   The Add Watch Expression dialog appears.

2. Enter an expression you want to watch during debugging, for example, the variable name "i"

If you have added one or more expressions to watch, the right-click menu provides options to **Remove** and **Edit** or **Enable** and **Disable** added expressions.

• You can access target registers directly using `#NAME`. For example "`arr[#R0 << 3]`".

• Expressions may contain target function calls like for example "`g1 + invert(&g2)`". Be aware that this will not work if the compiler has optimized the code in such a way that the original function code does not actually exist anymore. This may be the case, for example, as a result of inlining. Also, be aware that the function and its callees use the same stack(s) as your application, which may cause problems if there is too little stack space. Finally, any breakpoints present affect the invoked code in the normal way.

9.5.6. Memory View

Use the Memory view to inspect and change process memory. The Memory view supports the same addressing as the C language. You can address memory using expressions such as:

• `0x0847d3c`

• `(&y)+1024`

• `*ptr`

Monitors

To monitor process memory, you need to add a *monitor*:

1. In the Debug view, select a debug session. Selecting a thread or stack frame automatically selects the associated session.

2. Click the **Add Memory Monitor** button in the Memory Monitors pane.

   The Monitor Memory dialog appears.

3. Type the address or expression that specifies the memory section you want to monitor and click **OK**.

   The monitor appears in the monitor list and the Memory Renderings pane displays the contents of memory locations beginning at the specified address.

To remove a monitor:

1. In the Monitors pane, right-click on a monitor.

2. From the popup menu, select **Remove Memory Monitor**.

Renderings

You can inspect the memory in so-called *renderings*. A rendering specifies how the output is displayed: hexadecimal, ASCII, signed integer, unsigned integer or traditional. You can add or remove renderings per monitor. Though you cannot change a rendering, you can add or remove them:

1. Click the **New Renderings...** tab in the Memory Renderings pane.

    The Add Memory Rendering dialog appears.

2. Select the rendering you want (**Traditional**, **Hex**, **ASCII**, **Signed Integer**, **Unsigned Integer** or **Hex Integer**) and click **Add Rendering(s)**.

To remove a rendering:

1. Right-click on a memory address in the rendering.

2. From the popup menu, select **Remove Rendering**.

Changing memory contents

In a rendering you can change the memory contents. Simply type a new value.

> **Warning**: Changing process memory can cause a program to crash.

The right-click popup menu gives some more options for changing the memory contents or to change the layout of the memory representation.

9.5.7. Compare Application View

You can use the Compare Application view to check if the downloaded application matches the application in memory. Differences may occur, for example, if you changed memory addresses in the Memory view.

*   To check for differences, click the **Compare** button.

9.5.8. Heap View

With the Heap view you can inspect the status of the heap memory. This can be illustrated with the following example:

```
string = (char *) malloc(100);
strcpy ( string, "abcdefgh" );
free (string);
```

If you step through these lines during debugging, the Heap view shows the situation after each line has been executed. Before any of these lines has been executed, there is no memory allocated and the Heap view is empty.

*   After the first line the Heap view shows that memory is occupied, the description tells where the block starts, how large it is (`100 MAUs`) and what its content is (`0x0, 0x0, ...`).

*   After the second line, "`abcdefgh`" has been copied to the allocated block of memory. The description field of the Heap view again shows the actual contents of the memory block (`0x61, 0x62,...`).

*   The third line frees the memory. The Heap view is empty again because after this line no memory is allocated anymore.

9.5.9. Logging View

Use the Logging view to control the generation of internal log files. This view is intended mainly for use by or at the request of TASKING support personnel.

9.5.10. RTOS View

The debugger has special support for debugging real-time operating systems (RTOSs). This support is implemented in an RTOS-specific shared library called a *kernel support module* (KSM) or *RTOS-aware debugging module* (RADM). You have to create your own Run Time Interface (ORTI) and specify this file on the **Miscellaneous** tab while configuring a customized debug configuration (see also Section 9.3, *Creating a Customized Debug Configuration*):

1. From the **Debug** menu, select **Debug Configurations...**

    The Debug Configurations dialog appears.

2. In the left pane, select the configuration you want to change, for example, **TASKING C/C++ Debugger » myproject**.

    Or: click the **New launch configuration** button ( ) to add a new configuration.

3. Open the **Miscellaneous** tab

4. In the **ORTI file** field, specify the name of your own ORTI file.

5. In the **KSM module** field, specify the name of a KSM shared library file suitable for RTOS kernels.

The debugger supports ORTI specifications v2.0 and v2.1.

9.5.11. Registers View

In the Registers view you can examine the value of registers while stepping through your application. The registers are organized in a number of *register groups*, which together contain all known registers. You can select a group to see which registers it contains. This view has a number of features:

- While you step through the application, the registers that are changed turn yellow. If you scroll in the view or switch groups, some registers may appear on a lighter yellow background, indicating that the debugger does not know whether the registers have changed because the debugger did not read the registers before the step began.
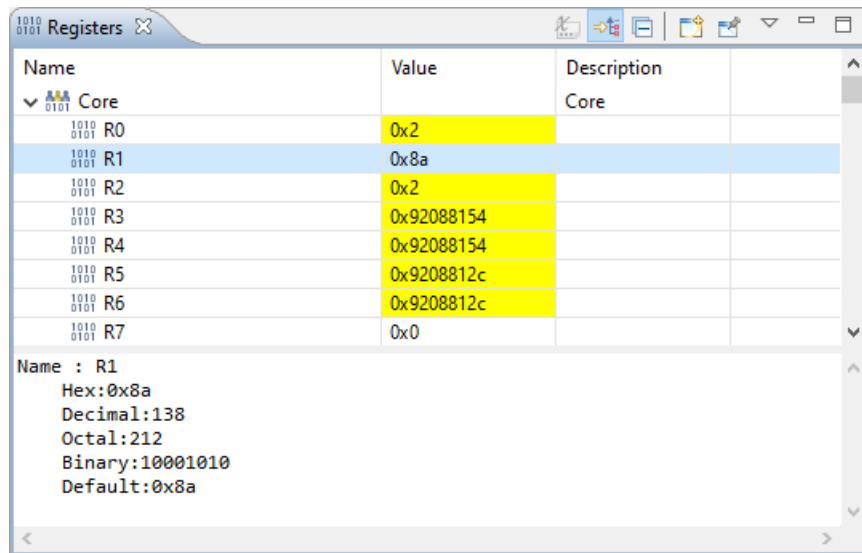
- You can change each register's value.

- You can search for a specific register: right-click on a register and from the popup menu select **Find...**. Enter a group or register name filter, click the register you want to see and click **OK**. The register of your interest will be shown in the view.

Chapter 10. Tool Options

This chapter provides a detailed description of the options for the compiler, assembler, linker, control program, make utility and the archiver.

Tool options in Eclipse (Menu entry)

For each tool option that you can set from within Eclipse, a **Menu entry** description is available. In Eclipse you can customize the tools and tool options in the following dialog:

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

   The Properties dialog appears.

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

   In the right pane the Settings appear.

3. Open the **Tool Settings** tab.

   You can set all tool options here.

Unless stated otherwise, all **Menu entry** descriptions expect that you have this Tool Settings tab open.

The following tables give an overview of all tool options on the Tool Settings tab in Eclipse with hyperlinks to the corresponding command line options (if available).

Global Options

Eclipse option	Description or option
Use global 'product directory' preference	Directory where the TASKING toolset is installed
Treat warnings as errors	Control program option **--warnings-as-errors**
Keep temporary files	Control program option **--keep-temporary-files (-t)**
Verbose mode of control program	Control program option **--verbose (-v)**

C Compiler

Eclipse option	Description or option
Preprocessing	
Automatic inclusion of '.sfr' file	C compiler option **--include-file**
Store preprocessor output in <file>.pre	Control program option **--preprocess (-E)** / **--no-preprocessing-only**

Eclipse option	Description or option
Keep comments in preprocessor output	Control program option **--preprocess=+comments**
Keep #line info in preprocessor output	Control program option **--preprocess=-noline**
Insert a space between adjacent tokens (if needed)	Control program option **--preprocess=+token-separation**
Defined symbols	C compiler option **--define**
Pre-include files	C compiler option **--include-file**
Include Paths	
Include paths	C compiler option **--include-directory**
Precompiled C++ Headers	
Automatically use/create precompiled header file	C++ compiler option **--pch**
Create precompiled header file	C++ compiler option **--create-pch**
Use precompiled header file	C++ compiler option **--use-pch**
Precompiled header file directory	C++ compiler option **--pch-dir**
Language	
Comply to C++ standard	C++ compiler option **--c++**
Comply to C standard	C compiler option **--iso**
Allow GNU C extensions	C compiler option **--language=+gcc**
Allow // comments in ISO C90 mode	C compiler option **--language=+comments**
Check assignment of string literal to non-const string pointer	C compiler option **--language=-strings**
Treat 'char' variables as signed	C compiler option **--schar**
Allow optimization across volatile access	C compiler option **--language=-volatile**
Allow Shift JIS Kanji in strings	C compiler option **--language=+kanji**
Comply to embedded C++ subset	C++ compiler option **--embedded-c++**
Support for C++ RTTI (run-time type information)	C++ compiler option **--rtti**
Allow non-ANSI/ISO C++ features	C++ compiler option **--strict**
Allow GNU C++ extensions	C++ compiler option **--g++**
Floating-Point	
Floating-point model	Control program option **--fp-model**
Allocation	
Threshold for putting data in SDA	C compiler option **--sda-max-data-size**
Optimization	
Optimization level	C compiler option **--optimize**
Trade-off between speed and size	C compiler option **--tradeoff**
Maximum size for code compaction	C compiler option **--compact-max-size**

Eclipse option	Description or option
Maximum call depth for code compaction	C compiler option **--max-call-depth**
Always inline function calls	C compiler option **--inline**
Maximum size increment when inlining (in %)	C compiler option **--inline-max-incr**
Maximum size for functions to always inline	C compiler option **--inline-max-size**
Build for application wide optimizations (MIL linking)	Control program option **--mil-link** / **--mil-split**
Application wide optimization mode	Control program option **--mil-link** / **--mil-split**
Auto-vectorization diagnostics	C compiler option **--vectorize-info**
No auto-vectorization alias checking	C compiler option **--vectorize-noalias**
Assume vector data is in __vccm memory	C compiler option **--vectorize-vccm**
Custom Optimization	C compiler option **--optimize**
Debugging	
Generate symbolic debug information	C compiler option **--debug-info**
Generate control flow information	C compiler option **--control-flow-info**
Generate code for bounds checking	C compiler option **--runtime=+bounds**
Generate code to detect unhandled case in a switch	C compiler option **--runtime=+case**
Generate code for malloc consistency checks	C compiler option **--runtime=+malloc**
MISRA C	
MISRA C checking	C compiler option **--misrac**
MISRA C version	C compiler option **--misrac-version**
Warnings instead of errors for mandatory rules	C compiler option **--misrac-mandatory-warnings**
Warnings instead of errors for required rules	C compiler option **--misrac-required-warnings**
Warnings instead of errors for advisory rules	C compiler option **--misrac-advisory-warnings**
Custom 1998 / Custom 2004 / Custom 2012	C compiler option **--misrac**
CERT C Secure Coding	
CERT C secure code checking	C compiler option **--cert**
Warnings instead of errors	C compiler option **--warnings-as-errors**
Custom CERT C	C compiler option **--cert**
Diagnostics	
Suppress C++ compiler warnings	C++ compiler option **--no-warnings=**_num_
Suppress C compiler warnings	C compiler option **--no-warnings=**_num_
Suppress all warnings	C compiler option **--no-warnings**
Suppress C++ compiler "used before set" warnings	C++ compiler option **--no-use-before-set-warnings**

Eclipse option	Description or option
Issue remarks on C++ code	C++ compiler option **--remarks**
Perform global type checking on C code	C compiler option **--global-type-checking**
Maximum number of emitted errors	C compiler option **--error-limit**
Miscellaneous	
Merge C source code with generated assembly	C compiler option **--source**
Force definition of virtual function tables (C++)	C++ compiler option **--force-vtbl**
Suppress definition of virtual function tables (C++)	C++ compiler option **--suppress-vtbl**
Minimal inlining of function calls (C++)	C++ compiler option **--no-inlining**
Instantiation mode of external template entities	C++ compiler option **--instantiate**
Additional options	C compiler options, C++ compiler options, Control program options

Assembler

Eclipse option	Description or option
Preprocessing	
Use TASKING preprocessor	Assembler option **--preprocessor-type**
Automatic inclusion of '.def' file	Assembler option **--include-file**
Defined symbols	Assembler option **--define**
Pre-include files	Assembler option **--include-file**
Include Paths	
Include paths	Assembler option **--include-directory**
Symbols	
Generate symbolic debug	Assembler option **--debug-info**
Case insensitive identifiers	Assembler option **--case-insensitive**
Emit local EQU symbols	Assembler option **--emit-locals=+equs**
Emit local non-EQU symbols	Assembler option **--emit-locals=+symbols**
Set default symbol scope to global	Assembler option **--symbol-scope**
List File	
Generate list file	Control program option **--list-files**
List ...	Assembler option **--list-format**
List section summary	Assembler option **--section-info=+list**
Diagnostics	
Suppress warnings	Assembler option **--no-warnings=***num*
Suppress all warnings	Assembler option **--no-warnings**
Display section summary	Assembler option **--section-info=+console**

Eclipse option	Description or option
Maximum number of emitted errors	Assembler option **--error-limit**
Miscellaneous	
Allow Shift JIS Kanji in strings	Assembler option **--kanji**
Additional options	Assembler options

Linker

Eclipse option	Description or option
Output Format	
Generate Intel Hex format file	Linker option **--output=**file**:IHEX**
Generate S-records file	Linker option **--output=**file**:SREC**
Generate binary file	Linker option **--chip-output=:BIN**
Generate C array file	Linker option **--chip-output=**basename**:CARR:32**
Create file for each memory chip	Linker option **--chip-output**
Size of addresses (in bytes) for Intel Hex records	Linker option **--output=**file**:IHEX:**size
Size of addresses (in bytes) for Motorola S records	Linker option **--output=**file**:SREC:**size
Emit start address record	Linker option **--hex-format=s**
Emit list of exported symbols	Linker option **--hex-format=y**
Libraries	
Link default libraries	Control program option **--no-default-libraries**
Rescan libraries to solve unresolved externals	Linker option **--no-rescan**
Libraries	The libraries are added as files on the command line.
Library search path	Linker option **--library-directory**
Data Objects	
Data objects	Linker option **--import-object**
Script File	
Defined symbols	Linker option **--define**
Linker script file (.lsl)	Linker option **--lsl-file**
Optimization	
Delete unreferenced sections	Linker option **--optimize=c**
Use a 'first-fit decreasing' algorithm	Linker option **--optimize=l**
Compress copy table	Linker option **--optimize=t**
Delete duplicate code	Linker option **--optimize=x**
Delete duplicate data	Linker option **--optimize=y**

Eclipse option	Description or option
Map File	
Generate map file (.map)	Control program option **--no-map-file**
Generate XML map file format (.mapxml) for map file viewer	Linker option **--map-file=**_file_**.mapxml:XML**
Include ...	Linker option **--map-file-format**
Diagnostics	
Suppress warnings	Linker option **--no-warnings=**_num_
Suppress all warnings	Linker option **--no-warnings**
Maximum number of emitted errors	Linker option **--error-limit**
Miscellaneous	
Strip symbolic debug information	Linker option **--strip-debug**
Link case insensitive	Linker option **--case-insensitive**
Do not use standard copy table for initialization	Linker option **--user-provided-initialization-code**
Show link phases during processing	Linker option **--verbose**
Application is not romable	Linker option **--non-romable**
Additional options	Linker options

10.1. Configuring the Command Line Environment

If you want to use the tools on the command line, you can set *environment variables*.

You can set the following environment variables:

Environment variable	Description
ASARCINC	With this variable you specify one or more additional directories in which the assembler looks for include files. See Section 6.3, *How the Assembler Searches Include Files*.
CARCINC	With this variable you specify one or more additional directories in which the C compiler looks for include files. See Section 4.4, *How the Compiler Searches Include Files*.
CPARCINC	With this variable you specify one or more additional directories in which the C++ compiler looks for include files. See Section 5.2, *How the C++ Compiler Searches Include Files*.
CCARCBIN	When this variable is set, the control program prepends the directory specified by this variable to the names of the tools invoked.

Environment variable	Description
LIBPPU_TC45X LIBPPU_TC49X LIBPPU_TC49XN LIBPPU_TC4DX	With these variables you specify one or more additional directories in which the linker looks for libraries. See Section 7.3.1, *How the Linker Searches Libraries*.
PATH	With this variable you specify the directory in which the executables reside. This allows you to call the executables when you are not in the `bin` directory. Usually your system already uses the PATH variable for other purposes. To keep these settings, you need to add (rather than replace) the path. Use a semicolon (;) to separate path names.
TMPDIR	With this variable you specify the location where programs can create temporary files. Usually your system already uses this variable. In this case you do not need to change it.

With the exception of CCARCBIN, PATH and TMPDIR, the information about the environment variables used by the tools is logged in ELF Note sections of the generated ELF output file. See Section 7.11, *Linker ELF Note Sections*. The compiler related environment variables are also logged in `.COMPILER_ENV` assembler directives.

See the documentation of your operating system on how to set environment variables.

10.2. C Compiler Options

This section lists all C compiler options.

Options in Eclipse versus options on the command line

Most command line options have an equivalent option in Eclipse but some options are only available on the command line. Eclipse invokes the compiler via the control program. Therefore, it uses the syntax of the control program to pass options and files to the C compiler. If there is no equivalent option in Eclipse, you can specify a command line option in Eclipse as follows:

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

   The Properties dialog appears.

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

   In the right pane the Settings appear.

3. On the Tool Settings tab, select **C/C++ Compiler » Miscellaneous**.

4. In the **Additional options** field, enter one or more command line options.

   *Because Eclipse uses the control program, you have to precede the option with **-Wc** to pass the option via the control program directly to the C compiler.*

Be aware that some command line options are not useful in Eclipse or just do not have any effect. For example, the option **-n** sends output to stdout instead of a file and has no effect in Eclipse.

Short and long option names

Options can have both short and long names. Short option names always begin with a single minus (**-**) character, long option names always begin with two minus (**--**) characters. You can abbreviate long option names as long as it forms a unique name. You can mix short and long option names on the command line.

Options can have flags or suboptions. To switch a flag 'on', use a lowercase letter or a **+***longflag*. To switch a flag off, use an uppercase letter or a **-***longflag*. Separate *longflags* with commas. The following two invocations are equivalent:

```
carc -Oac test.c
carc --optimize=+coalesce,+cse test.c
```

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

C compiler option: --cert

Menu entry

1. Select **C/C++ Compiler » CERT C Secure Coding**.

2. Make a selection from the **CERT C secure code checking** list.

3. If you selected **Custom**, expand the **Custom CERT C** entry and enable one or more individual recommendations/rules.

Command line syntax

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

Default format: all

Description

With this option you can enable one or more checks for CERT C Secure Coding Standard recommendations/rules. When you omit the argument, all checks are enabled. *name* is the name of a CERT recommendation/rule, consisting of three letters and two digits. Specify only the three-letter mnemonic to select a whole category. For the list of names you can use, see Chapter 16, *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:

```
carc --cert=str30 test.c
```

Related information

Chapter 16, *CERT C Secure Coding Standard*

C compiler option **--diag** (Explanation of diagnostic messages)

C compiler option: --check

Menu entry

-

Command line syntax

`--check`

Description

With this option you can check the source code for syntax errors, without generating code. This saves time in developing your application because the code will not actually be compiled.

The compiler reports any warnings and/or errors.

This option is available on the command line only.

Related information

Assembler option **--check** (Check syntax)

C compiler option: --compact-max-size

Menu entry

1. Select **C/C++ Compiler » Optimization**.

2. In the **Maximum size for code compaction** field, enter the maximum size of a match.

Command line syntax

--compact-max-size=*value*

Default: 200

Description

This option is related to the compiler optimization **--optimize=+compact** (Code compaction or reverse inlining). Code compaction is the opposite of inlining functions: large sequences of code that occur more than once, are transformed into a function. This reduces code size (possibly at the cost of execution speed).

However, in the process of finding sequences of matching instructions, compile time and compiler memory usage increase quadratically with the number of instructions considered for code compaction. With this option you tell the compiler to limit the number of matching instructions it considers for code compaction.

Example

To limit the maximum number of instructions in functions that the compiler generates during code compaction:

```
carc --optimize=+compact --compact-max-size=100 test.c
```

Related information

C compiler option **--optimize=+compact** (Optimization: code compaction)

C compiler option **--max-call-depth** (Maximum call depth for code compaction)

295

C compiler option: --control-flow-info

Menu entry

1. Select **C/C++ Compiler » Debugging**.

2. Select **Generate control flow information**.

Command line syntax

`--control-flow-info`

Description

With this option the compiler adds control flow information to the output file. The compiler generates a `.debug_control_flow` section which describes the basic blocks and their relations. This information can be used for code coverage analysis on optimized code.

Example

```
carc --control-flow-info test.c
```

Related information

C compiler option **--debug-info** (Debug information)

C compiler option: --core

Menu entry

1. Expand **C/C++ Build** and select **Processor**.

2. From the **Processor selection** list, make a selection.

Command line syntax

--core=_core_

You can specify the following core architectures:

ppu_tc45x	PPU core architecture of TC45x
ppu_tc49x	PPU core architecture of TC49x
ppu_tc49xn	PPU core architecture of TC49xN
ppu_tc4dx	PPU core architecture of TC4Dx

Default: ppu_tc49x

Description

With this option you specify the PPU core architecture for which you create your application. The core architecture determines which instructions are valid and which are not.

Example

To compile the file `test.c` for the PPU core architecture of the TC49x, enter the following on the command line:

```
carc --core=ppu_tc49x test.c
```

Related information

Control program option **--core** (Select core architecture)

Assembler option **--core** (Select core architecture)

C compiler option: --debug-info (-g)

Menu entry

1. Select **C/C++ Compiler » Debugging**.

2. To **generate symbolic debug information**, select **Default**, **Small set** or **Full**.
   To disable the generation of debug information, select **None**.

Command line syntax

--debug-info[**=**_suboption_]

-g[_suboption_]

You can set the following suboptions:

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.

Default (Eclipse): **--debug-info=default**

Default (without suboptions): **--debug-info=default**

Description

With this option you tell the compiler to add directives to the output file for including symbolic information. This facilitates high level debugging but increases the size of the resulting assembler file (and thus the size of the object file). For the final application, compile your C files without debug information.

If you do not specify this option, the compiler does not generate debug information.

The DWARF debug format allows for a flexible approach as to how much symbolic information is included, as long as the structure is valid. Adding all possible DWARF data for a program is not practical. The amount of DWARF information per compilation unit can be huge. And for large projects, with many object modules the link time can grow unacceptably long. That is why the compiler has several debug information levels. In general terms one can say, the higher the level the more DWARF information is produced.

The DWARF data in an object module is not only used for debugging. The toolset can also do "type checking" of the whole application. In that case the linker will use the DWARF information of all object modules to determine if every use of a symbol is done with the same type. In other words, if the application is built with type checking enabled then the compiler will add DWARF information too.

Small set of debug information

With this suboption only DWARF call frame information and type information are generated. This enables you to inspect parameters of nested functions. The type information improves debugging. You can perform a stack trace, but stepping is not possible because debug information on function bodies is not generated. You can use this suboption, for example, to compact libraries.

Default debug information

This provides all debug information you need to debug your application. It meets the debugging requirements in most cases without resulting in oversized assembler/object files.

Full debug information

With this suboption extra debug information is generated about unused typedefs and DWARF "lookup table sections". Under normal circumstances this extra debug information is not needed to debug the program. Information about unused typedefs concerns all typedefs, even the ones that are not used for any variable in the program. (Possibly, these unused typedefs are listed in the standard include files.) With this suboption, the resulting assembler/object file will increase significantly.

In the following table you see in more detail what DWARF information is included for the debug option levels.

Feature	-g1	-g2	-g3	type check	Remarks
basic info	+	+	+	+	info such as symbol name and type
call frame	+	+	+	+	this is information for a debugger to compute a stack trace when a program has stopped at a breakpoint
symbol lifetime		+	+		this is information about where symbols live (e.g. on stack at offset so and so, when the program counter is in this range)
line number info		+	+	+	file name, line number, column number
"lookup tables"			+		DWARF sections ... this is an optimization for the DWARF data, it is not essential
unused typedefs			+		in the C code of the program there can be (many) typedefs that are not used for any variable. Sometimes this can cause enormous expansion of the DWARF data and thus it is only included in **-g3**.

Related information

-

C compiler option: --define (-D)

Menu entry

1. Select **C/C++ Compiler » Preprocessing**.

    The Defined symbols box shows the symbols that are currently defined.

2. To define a new symbol, click on the **Add** button in the **Defined symbols** box.

3. Type the symbol definition (for example, `demo=1`)

> Use the **Edit** and **Delete** button to change a macro definition or to remove a macro from the list.

Command line syntax

--define=`macro_name`[**=**`macro_definition`]

-D`macro_name`[**=**`macro_definition`]

Description

With this option you can define a macro and specify it to the preprocessor. If you only specify a macro name (no macro definition), the macro expands as '1'.

You can specify as many macros as you like. Simply use the **Add** button to add new macro definitions.

On the command line, you can use the option **--define** (**-D**) multiple times. If the command line exceeds the limit of the operating system, you can define the macros in an option file which you then must specify to the compiler with the option **--option-file** (**-f**) *file*.

Defining macros with this option (instead of in the C source) is, for example, useful to compile conditional C source as shown in the example below.

> Make sure you do not use a reserved keyword as a macro name, as this can lead to unexpected results.

Example

Consider the following C program with conditional code to compile a demo program and a real program:

```
void main( void )
{
#if DEMO
    demo_func();   /* compile for the demo program */
#else
    real_func();   /* compile for the real program */
#endif
}
```

You can now use a macro definition to set the DEMO flag:

```
carc --define=DEMO test.c
carc --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.

```
carc --define="MAX(A,B)=((A) > (B) ? (A) : (B))" test.c
```

Related information

C compiler option **--undefine** (Remove preprocessor macro)

C compiler option **--option-file** (Specify an option file)

C compiler option: --dep-file

Menu entry

Eclipse uses this option in the background to create a file with extension `.d` (one for every input file).

Command line syntax

--dep-file[**=***file*]

Description

With this option you tell the compiler to generate dependency lines that can be used in a Makefile. In contrast to the option **--preprocess=+make**, the dependency information will be generated in addition to the normal output file.

By default, the information is written to a file with extension `.d` (one for every input file). When you specify a filename, all dependencies will be combined in the specified file.

Example

```
carc --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 **--dep-format** (Format of dependency info)

C compiler option **--preprocess=+make** (Generate dependencies for make)

C compiler option: --dep-format

Menu entry

-

Command line syntax

`--dep-format=`*format*

You can specify the following *format* arguments:

amk	amk format of dependency info
gnu	GNU format of dependency info

Default: **amk**

Description

With this option you can override the format of the make dependencies file from TASKING amk (the default format) to GNU make. The format determines how names with spaces or other special characters are escaped. In the amk format such names are double quoted. In the GNU format special characters are escaped using a backslash.

Example

```
carc --dep-file=test.dep --dep-format=gnu 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` in the GNU make format.

Related information

C compiler option **--dep-file** (Generate dependencies in a file)

C compiler option **--preprocess=+make** (Generate dependencies for make)

C compiler option: --diag

Menu entry

1. From the **Window** menu, select **Show View » Other » TASKING » Problems**.

    The Problems view is added to the current perspective.

2. In the Problems view right-click on a message.

    A popup menu appears.

3. Select **Detailed Diagnostics Info**.

    A dialog box appears with additional information.

Command line syntax

--diag=[*format*:]{**all** | *msg*[*-msg*],...}

You can set the following output formats:

html	HTML output.
rtf	Rich Text Format.
text	ASCII text.

Default format: text

Description

With this option you can ask for an extended description of error messages in the format you choose. The output is directed to stdout (normally your screen) and in the format you specify. The compiler does not compile any files. You can specify the following formats: **html**, **rtf** or **text** (default). To create a file with the descriptions, you must redirect the output.

With the suboption **all**, the descriptions of all error messages are given (except for the CERT checks). If you want the description of one or more selected error messages, you can specify the error message numbers, separated by commas, or you can specify a range.

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

Example

To display an explanation of message number 282, enter:

```
carc --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:

```
carc --diag=html:all > cerrors.html
```

Related information

Section 4.8, *C Compiler Error Messages*

C compiler option **--cert** (Enable individual CERT checks)

C compiler option **--warning-level** (Control warning level)

C compiler option: --error-file

Menu entry

-

Command line syntax

--error-file[**=***file*]

Description

With this option the compiler redirects diagnostic messages to a file. If you do not specify a filename, the error file will be named after the output file with extension `.err`.

Example

To write diagnostic messages to `errors.err` instead of `stderr`, enter:

```
carc --error-file=errors.err test.c
```

Related information

-

C compiler option: --error-limit

Menu entry

1. Select **C/C++ Compiler » Diagnostics**.

2. Enter a value in the **Maximum number of emitted errors** field.

Command line syntax

--error-limit=*number*

Default: 42

Description

With this option you limit the number of error messages in one compiler run to the specified number. When the limit is exceeded, the compiler aborts with fatal error message F105. Warnings and informational messages are not included in the count. When 0 (zero) or a negative number is specified, the compiler emits all errors. Without this option the maximum number of errors is 42.

Related information

Section 4.8, *C Compiler Error Messages*

C compiler option: --fp-model

Menu entry

1. Select **C/C++ Compiler » Floating-Point**.

2. Make a selection from the **Floating-point model** list.

3. If you selected **Custom**, enable one or more individual options.

Command line syntax

--fp-model=`flags`

You can set the following flags:

+/-contract	**c/C**	allow expression contraction
+/-fastlib	**l/L**	allow less precise library functions
+/-nonan	**n/N**	allow optimizations to ignore NaN/Inf
+/-rewrite	**r/R**	allow expression rewriting
+/-negzero	**z/Z**	ignore sign of -0.0
	0	alias for **--fp-model=CLNRZ** (strict)
	1	alias for **--fp-model=cLNRZ** (precise)
	2	alias for **--fp-model=clnrz** (fast double precision)

Default: **--fp-model=clnrz**

Description

With this option you select the floating-point execution model.

With **--fp-model=+contract** you allow the compiler to contract multiple float operations into a single operation, with different rounding results. A possible example is fused multiply-add.

With **--fp-model=+fastlib** you allow the compiler to select faster but less accurate library functions for certain floating-point operations.

With **--fp-model=+nonan** you allow the compiler to ignore NaN or Inf input values. An example is to replace multiply by zero with zero.

With **--fp-model=+rewrite** you allow the compiler to rewrite expressions by reassociating. This might result in rounding differences and possibly different exceptions. An example is to rewrite (a*c)+(b*c) as (a+b)*c.

With **--fp-model=+negzero** you allow the compiler to ignore the sign of -0.0 values. An example is to replace (a-a) by zero.

Related information

Pragmas `STDC FP_CONTRACT`, `fp_negzero`, `fp_nonan` and `fp_rewrite` in Section 1.8, *Pragmas to Control the Compiler*.

C compiler option: --global-type-checking

Menu entry

1.   Select **C/C++ Compiler » Diagnostics**.

2.   Enable the option **Perform global type checking on C code**.

Command line syntax

`--global-type-checking`

Description

The C compiler already performs type checking within each module. Use this option when you want the linker to perform type checking between modules.

Related information

-

C compiler option: --help (-?)

Menu entry

-

Command line syntax

--help[**=***item*]

-?

You can specify the following arguments:

intrinsics	**i**	Show the list of intrinsic functions
options	**o**	Show extended option descriptions
pragmas	**p**	Show the list of supported pragmas
typedefs	**t**	Show the list of predefined typedefs

Description

Displays an overview of all command line options. With an argument you can specify which extended information is shown.

Example

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

```
carc -?
carc --help
carc
```

The following invocation displays a list of the available pragmas:

```
carc --help=pragmas
```

Related information

-

C compiler option: --include-directory (-I)

Menu entry

1. Select **C/C++ Compiler » Include Paths**.

   The Include paths box shows the directories that are added to the search path for include files.

2. To define a new directory for the search path, click on the **Add** button in the **Include paths** box.

3. Type or select a path.

Use the **Edit** and **Delete** button to change a path or to remove a path from the list.

Command line syntax

`--include-directory=`*path*`,...`

`-I`*path*`,...`

Description

With this option you can specify the path where your include files are located. A relative path will be relative to the current directory,

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

1. The pathname in the C source file and the directory of the C source (only for #include files that are enclosed in "")

2. The path that is specified with this option.

3. The path that is specified in the environment variable `CARCINC` when the product was installed.

4. The default directory `$(PRODDIR)\include` (unless you specified option **--no-stdinc**).

Example

Suppose that the C source file `test.c` contains the following lines:

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

You can call the compiler as follows:

```
carc --include-directory=myinclude test.c
```

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

The compiler now looks for the file `myinc.h` in the directory where `test.c` is located. If the file is not there the compiler searches in the directory `myinclude`. If it was still not found, the compiler searches in the environment variable and then in the default include directory.

Related information

C compiler option **--include-file** (Include file at the start of a compilation)

C compiler option **--no-stdinc** (Skip standard include files directory)

C compiler option: --include-file (-H)

Menu entry

1.   Select **C/C++ Compiler » Preprocessing**.

     The Pre-include files box shows the files that are currently included before the compilation starts.

2.   To define a new file, click on the **Add** button in the **Pre-include files** box.

3.   Type the full path and file name or select a file.

Use the **Edit** and **Delete** button to change a file name or to remove a file from the list.

Command line syntax

--include-file=*file*,...

-H*file*,...

Description

With this option you include one or more extra files at the beginning of each C source file, before other includes. This is the same as specifying #include "file" at the beginning of *each* of your C sources.

Example

carc --include-file=stdio.h test1.c test2.c

The file stdio.h is included at the beginning of both test1.c and test2.c.

Related information

C compiler option **--include-directory** (Add directory to include file search path)

C compiler option: --inline

Menu entry

1. Select **C/C++ Compiler » Optimization**.

2. Enable the option **Always inline function calls**.

Command line syntax

`--inline`

Description

With this option you instruct the compiler to inline calls to functions without the `__noinline` function qualifier whenever possible. This option has the same effect as a `#pragma inline` at the start of the source file.

This option can be useful to increase the possibilities for code compaction (C compiler option **--optimize=+compact**).

Example

To always inline function calls:

```
carc --inline test.c
```

Related information

C compiler option **--optimize=+compact** (Optimization: code compaction)

Section 1.10.2, *Inlining Functions: inline*

C compiler option: --inline-max-incr / --inline-max-size

Menu entry

1. Select **C/C++ Compiler » Optimization**.

2. In the **Maximum size increment when inlining** field, enter a value (default -1).

3. In the **Maximum size for functions to always inline** field, enter a value (default -1).

Command line syntax

```
--inline-max-incr=percentage  (default: -1)
--inline-max-size=threshold   (default: -1)
```

Description

With these options you can control the automatic function inlining optimization process of the compiler. These options only have effect when you have enabled the inlining optimization (option **--optimize=+inline** or **Optimize most**).

> Regardless of the optimization process, the compiler always inlines all functions that have the function qualifier `inline`.

With the option **--inline-max-size** you can specify the maximum size of functions that the compiler inlines as part of the optimization process. The compiler always inlines all functions that are smaller than the specified *threshold*. The threshold is measured in compiler internal units and the compiler uses this measure to decide which functions are small enough to inline. The default threshold is -1, which means that the threshold depends on the option **--tradeoff**.

After the compiler has inlined all functions that have the function qualifier `inline` and all functions that are smaller than the specified threshold, the compiler looks whether it can inline more functions without increasing the code size too much. With the option **--inline-max-incr** you can specify how much the code size is allowed to increase. The default value is -1, which means that the value depends on the option **--tradeoff**.

Example

```
carc --optimize=+inline --inline-max-incr=40 --inline-max-size=15 test.c
```

The compiler first inlines all functions with the function qualifier `inline` and all functions that are smaller than the specified threshold of 15. If the code size has still not increased with 40%, the compiler decides which other functions it can inline.

Related information

C compiler option **--optimize=+inline** (Optimization: automatic function inlining)
Section 1.10.2, *Inlining Functions: inline*
Section 4.6.4, *Optimize for Code Size or Execution Speed*

C compiler option: --iso (-c)

Menu entry

1. Select **C/C++ Compiler » Language**.

2. From the **Comply to C standard** list, select **ISO C99**, **ISO C11**, **ISO C17**, or **ISO C90**.

Command line syntax

`--iso={90|99|11|17}`

`-c{90|99|11|17}`

Default: **`--iso=17`**

Description

With this option you select the ISO C standard. C90 is also referred to as the "ANSI C standard". C99 refers to the ISO/IEC 9899:1999 (E) standard. C11 refers to the ISO/IEC 9899:2011 (E) standard. C17 refers to the ISO/IEC 9899:2018 (E) standard. C17 is the default.

Example

To select the ISO C99 standard on the command line:

```
carc --iso=99 test.c
```

Related information

C compiler option **--language** (Language extensions)

317

C compiler option: --keep-output-files (-k)

Menu entry

Eclipse *always* removes the `.src` file when errors occur during compilation.

Command line syntax

```
--keep-output-files
```

```
-k
```

Description

If an error occurs during compilation, the resulting `.src` file may be incomplete or incorrect. With this option you keep the generated output file (`.src`) when an error occurs.

By default the compiler removes the generated output file (`.src`) when an error occurs. This is useful when you use the make utility. If the erroneous files are not removed, the make utility may process corrupt files on a subsequent invocation.

Use this option when you still want to inspect the generated assembly source. Even if it is incomplete or incorrect.

Example

```
carc --keep-output-files test.c
```

When an error occurs during compilation, the generated output file `test.src` will *not* be removed.

Related information

C compiler option **--warnings-as-errors** (Treat warnings as errors)

C compiler option: --language (-A)

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Enable or disable one or more of the following options:

    - Allow GNU C extensions

    - Allow long long types in ISO C90 mode

    - Allow // comments in ISO C90 mode

    - 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
+/-kanji	**k/K**	support for Shift JIS Kanji in strings
+/-longlong	**l/L**	long long types in ISO C90 mode
+/-comments	**p/P**	// comments in ISO C90 mode
+/-volatile	**v/V**	don't optimize across volatile access
+/-strings	**x/X**	relaxed const check for string literals

Default: **-AGKlpVx**

Default (without flags): **-AGKLPVX**

Description

With this option you control the language extensions the compiler can accept. By default the C compiler allows all language extensions, except for **gcc** extensions.

The option **--language** (**-A**) without flags disables all language extensions.

GNU C extensions

The **--language=+gcc** (**-Ag**) option enables the following gcc language extensions:

- The identifier __FUNCTION__ expands to the current function name.

- Alternative syntax for variadic macros.

- Alternative syntax for designated initializers.

- Allow zero sized arrays.

- Allow empty struct/union.

- Allow unnamed struct/union fields.

- Allow empty initializer list.

- Allow initialization of static objects by compound literals.

- The middle operand of a `?` `:` operator may be omitted.

- Allow a compound statement inside braces as expression.

- Allow arithmetic on void pointers and function pointers.

- Allow a range of values after a single case label.

- Additional preprocessor directive `#warning`.

- Allow comma operator, conditional operator and cast as lvalue.

- An inline function without "`static`" or "`extern`" will be global.

- An "`extern inline`" function will not be compiled on its own.

- Recognize the keywords `asm`, `alignof`, `typeof`, and `__extension__`.

For a more complete description of these extensions, you can refer to the UNIX gcc info pages (**info gcc**).

Shift JIS Kanji support

With **--language=+kanji** (**-Ak**) you tell the compiler to support Shift JIS encoded Kanji multi-byte characters in strings, (wide) character constants and `//` comments. Without this option, encodings with 0x5c as the second byte conflict with the use of the backslash as an escape character. Shift JIS in `/*...*/` comments is supported regardless of this option. Note that Shift JIS also includes Katakana and Hiragana.

long long types in ISO C90 mode

With **--language=+longlong** (**-Al**) you tell the compiler to allow `long long` types in ISO C90 mode (option **--iso=90**). In later ISO C versions these types are always allowed.

Comments in ISO C90 mode

With **--language=+comments** (**-Ap**) you tell the compiler to allow C++ style comments (//) in ISO C90 mode (option **--iso=90**). In later ISO C versions this style of comments is always accepted.

Check assignment of string literal to non-const string pointer

With **--language=+strings** (**-Ax**) you disable warnings about discarded const qualifiers when a string literal is assigned to a non-const pointer.

```
char *p;
int main( void )
{
    p = "hello"; // with -AX the compiler issues warning W525
    return 0;
}
```

Optimization across volatile access

With the **--language=+volatile** (**-Av**) option, the compiler will block optimizations when reading or writing a volatile object, by executing all memory and (SFR) register accesses before the access of the volatile object. The volatile access acts as a memory barrier. With this option you can prevent for example that code below the volatile object is optimized away to somewhere above the volatile object.

Example:

```
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
    st    0,[access]        ; <== Volatile access
    ld    %r1,[variable]
    or    %r0,%r1,%r0
    cmp   %r0,3
    bne   .L2
    mov   %r0,0
.L2:
    or    %r0,%r0,32768
    st    %r0,[variable]  ; <== Moved across volatile access
    st    1,[access]      ; <== Volatile access
    j     [%blink]
```

321

Result with **--language=+volatile**:

```
TestFunc: .type    func
    st    0,[access]         ; <== Volatile access
    ld    %r1,[variable]
    or    %r0,%r1,%r0
    cmp   %r0,3
    bne   .L2
    mov   %r0,0
.L2:
    or    %r0,%r0,32768
    st    1,[access]         ; <== Volatile access
    st    %r0,[variable]     ; <== Not moved
    j     [%blink]
```

Note that the volatile behavior of the compiler with option **--language=-volatile** or **--language=+volatile** is ISO C compliant in both cases.

Related information

C compiler option **--iso** (ISO C standard)

Section 1.4, *Shift JIS Kanji Support*

C compiler option: --make-target

Menu entry

-

Command line syntax

--make-target=*name*

Description

With this option you can overrule the default target name in the make dependencies generated by the options **--preprocess=+make** (**-Em**) and **--dep-file**. The default target name is the basename of the input file, with extension `.o`.

Example

```
carc --preprocess=+make --make-target=mytarget.o test.c
```

The compiler generates dependency lines with the default target name `mytarget.o` instead of `test.o`.

Related information

C compiler option **--preprocess=+make** (Generate dependencies for make)

C compiler option **--dep-file** (Generate dependencies in a file)

C compiler option **--dep-format** (Format of dependency info)

C compiler option: --max-call-depth

Menu entry

1. Select **C/C++ Compiler » Optimization**.

2. In the **Maximum call depth for code compaction** field, enter a value.

Command line syntax

`--max-call-depth=`*value*

Default: -1

Description

This option is related to the compiler optimization **--optimize=+compact** (Code compaction or reverse inlining). Code compaction is the opposite of inlining functions: large sequences of code that occur more than once, are transformed into a function. This reduces code size (possibly at the cost of execution speed).

During code compaction it is possible that the compiler generates nested calls. This may cause the program to run out of its stack. To prevent stack overflow caused by too deeply nested function calls, you can use this option to limit the call depth. This option can have the following values:

<dl>

-1   Poses no limit to the call depth (default)

0   The compiler will not generate any function calls. (Effectively the same as if you turned of code compaction with option **--optimize=-compact**)

\> 0   Code sequences are only reversed if this will not lead to code at a call depth larger than specified with *value*. Function calls will be placed at a call depth no larger than *value*-1. (Note that if you specified a value of 1, the option **--optimize=+compact** may remain without effect when code sequences for reversing contain function calls.)

</dl>

This option does not influence the call depth of user written functions.

If you use this option with various C modules, the call depth is valid for each individual module. The call depth after linking may differ, depending on the nature of the modules.

Related information

C compiler option **--optimize=+compact** (Optimization: code compaction)

C compiler option **--compact-max-size** (Maximum size of a match for code compaction)

C compiler option: --mil / --mil-split

Menu entry

1. Select **C/C++ Compiler » Optimization**.

2. Enable the option **Build for application wide optimizations (MIL linking)**.

3. Select **Optimize less/Build faster** or **Optimize more/Build slower**.

Command line syntax

```
--mil
--mil-split[=file,...]
```

Description

With option **--mil** the C compiler skips the code generator phase and writes the optimized intermediate representation (MIL) to a file with the suffix .mil. The C compiler accepts .mil files as input files on the command line.

Option **--mil-split** does the same as option **--mil**, but in addition, the C compiler splits the MIL representation and writes it to separate files with suffix .ms. One file is written for each input file or MIL library specified on the command line. The .ms files are only updated on a change. The C compiler accepts .ms files as input files on the command line.

With option **--mil-split** you can perform application-wide optimizations during the frontend phase by specifying all modules at once, and still invoke the backend phase one module at a time to reduce the total compilation time. Application wide code compaction is not possible in this case.

Optionally, you can specify another filename for the .ms file the C compiler generates. Without an argument, the basename of the C source file is used to create the .ms filename. Note that if you specify a filename, you have to specify one filename for every input file.

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

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

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

Build for application wide optimizations (MIL linking) and Optimize less/Build faster

This option is standard MIL linking and splitting. Note that you can control the optimizations to be performed with the optimization settings.

Optimize more/Build slower

When you enable this option, the compiler's frontend does not split the MIL stream in separate modules, but feeds it directly to the compiler's backend, allowing the code compaction to be performed application wide.

Related information

Section 4.1, *Compilation Process*

Control program option **--mil-link** / **--mil-split**

C compiler option: --misrac

Menu entry

1. Select **C/C++ Compiler » MISRA C**.

2. Make a selection from the **MISRA C checking** list.

3. If you selected **Custom**, expand the **Custom 1998**, **Custom 2004** or **Custom 2012** entry and enable one or more individual rules.

Command line syntax

```
--misrac={all | nr[-nr]},...
```

Description

With this option you specify to the compiler which MISRA C rules must be checked. With the option **--misrac=all** the compiler checks for all supported MISRA C rules.

Example

```
carc --misrac=9-13 test.c
```

The compiler generates an error for each MISRA C rule 9, 10, 11, 12 or 13 violation in file `test.c`.

Related information

Section 4.7.2, *C Code Checking: MISRA C*

C compiler option **--misrac-mandatory-warnings**

C compiler option **--misrac-advisory-warnings**

C compiler option **--misrac-required-warnings**

Linker option **--misrac-report**

C compiler option: --misrac-advisory-warnings / --misrac-required-warnings / --misrac-mandatory-warnings

Menu entry

1. Select **C/C++ Compiler » MISRA C**.

2. Make a selection from the **MISRA C checking** list.

3. Enable one or more of the options:
    Warnings instead of errors for mandatory rules
    Warnings instead of errors for required rules
    Warnings instead of errors for advisory rules.

Command line syntax

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

Description

Normally, if an advisory rule, required rule or mandatory rule is violated, the compiler generates an error. As a consequence, no output file is generated. With this option, the compiler generates a warning instead of an error.

Related information

Section 4.7.2, *C Code Checking: MISRA C*

C compiler option **--misrac**

Linker option **--misrac-report**

C compiler option: --misrac-version

Menu entry

1. Select **C/C++ Compiler » MISRA C**.

2. Select the **MISRA C version**: **1998**, **2004** or **2012**.

Command line syntax

`--misrac-version={1998|2004|2012}`

Default: 2012

Description

MISRA C rules exist in three versions: MISRA C:1998, MISRA C:2004 and MISRA C:2012. By default, the C source is checked against the MISRA C:2012 rules. With this option you can select which version to use.

Related information

Section 4.7.2, *C Code Checking: MISRA C*

C compiler option **--misrac**

C compiler option: --no-stdinc

Menu entry

1. Select **C/C++ Compiler » Miscellaneous**.

2. Add the option **--no-stdinc** to the **Additional options** field.

Command line syntax

```
--no-stdinc
```

Description

With this option you tell the compiler not to look in the default `include` directory relative to the installation directory, when searching for include files. This way the compiler only searches in the include file search paths you specified.

Related information

C compiler option **--include-directory** (Add directory to include file search path)

Section 4.4, *How the Compiler Searches Include Files*

C compiler option: --no-strict-aliasing

Menu entry

1. Select **C/C++ Compiler » Miscellaneous**.

2. Add the option **--no-strict-aliasing** to the **Additional options** field.

Command line syntax

```
--no-strict-aliasing
```

Description

With this option you can disable optimizations based on one type of undefined behavior. With this option the compiler will not use type information to prove that two memory accesses cannot alias. This prevents certain classes of optimizations that rely on the absence of this kind of undefined behavior.

Related information

-

C compiler option: --no-strict-overflow

Menu entry

1. Select **C/C++ Compiler » Miscellaneous**.

2. Add the option **--no-strict-overflow** to the **Additional options** field.

Command line syntax

`--no-strict-overflow`

Description

By default the C compiler can perform optimizations based on the assumption that signed integer overflow does not occur in the application. In the ISO C standard the behavior on integer overflow is undefined. With this option you tell the compiler to disable such optimizations and the compiler will treat overflow of signed integers as wraparound in two's complement.

Related information

-

C compiler option: --no-warnings (-w)

Menu entry

1. Select **C/C++ Compiler » Diagnostics**.

    The Suppress C compiler warnings box shows the warnings that are currently suppressed.

2. To suppress a warning, click on the **Add** button in the **Suppress warnings** box.

3. Enter the numbers, separated by commas or as a range, of the warnings you want to suppress (for example `537,538`). Or you can use the **Add** button multiple times.

4. To suppress all warnings, enable the option **Suppress all warnings**.

Use the **Edit** and **Delete** button to change a warning number or to remove a number from the list.

Command line syntax

--no-warnings[**=**_number_[_-number_],...]

-w[_number_[_-number_],...]

Description

With this option you can suppresses all warning messages or specific warning messages.

On the command line this option works as follows:

*   If you do not specify this option, all warnings are reported.

*   If you specify this option but without numbers, all warnings are suppressed.

*   If you specify this option with a number or a range, only the specified warnings are suppressed. You can specify the option **--no-warnings=**_number_ multiple times.

Example

To suppress warnings 537 and 538, enter:

```
carc test.c --no-warnings=537,538
```

Related information

C compiler option **--warnings-as-errors** (Treat warnings as errors)

Pragma `warning`

333

C compiler option: --optimize (-O)

Menu entry

1. Select **C/C++ Compiler » Optimization**.

2. Select an optimization level in the **Optimization level** box.

Command line syntax

--optimize[**=***flags*]

-O*flags*

You can set the following flags:

+/-coalesce	**a/A**	Coalescer: remove unnecessary moves
+/-ipro	**b/B**	Interprocedural register optimizations
+/-cse	**c/C**	Common subexpression elimination
+/-expression	**e/E**	Expression simplification
+/-flow	**f/F**	Control flow simplification
+/-glo	**g/G**	Generic assembly code optimizations
+/-inline	**i/I**	Automatic function inlining
+/-schedule	**k/K**	Instruction scheduler
+/-loop	**l/L**	Loop transformations
+/-vectorize	**m/M**	Loop auto-vectorization
+/-forward	**o/O**	Forward store
+/-propagate	**p/P**	Constant propagation
+/-compact	**r/R**	Code compaction (reverse inlining)
+/-subscript	**s/S**	Subscript strength reduction
+/-unroll	**u/U**	Unroll small loops
+/-ifconvert	**v/V**	Convert IF statements using predicates
+/-pipeline	**w/W**	Software pipelining
+/-peephole	**y/Y**	Peephole optimizations

Use the following options for predefined sets of flags:

--optimize=0	**-O0**	No optimization Alias for **-OaBCEFGIKLMOPRSUVWY**

No optimizations are performed except for the coalescer (to allow better debug information). The compiler tries to achieve an optimal resemblance between source code and produced code. Expressions are evaluated in the same order as written in the source code, associative and commutative properties are not used.

| **--optimize=1** | **-O1** | Optimize |
| | | Alias for **-OabCefgIKLMOPRSUVWy** |

Enables optimizations that do not affect the debug ability of the source code. Use this level when you encounter problems during debugging your source code with optimization level 2.

| **--optimize=2** | **-O2** | Optimize more (default) |
| | | Alias for **-OabcefgIKlMoprsUvWy** |

Enables more optimizations to reduce code size and/or execution time. This is the default optimization level.

| **--optimize=3** | **-O3** | Optimize most |
| | | Alias for **-OabcefgiklMoprsuvwy** |

This is the highest optimization level. Use this level to decrease execution time to meet your real-time requirements.

Default: **`--optimize=2`**

Description

With this option you can control the level of optimization. If you do not use this option, the default optimization level is *Optimize more* (option **--optimize=2** or **--optimize**).

When you use this option to specify a set of optimizations, you can overrule these settings in your C source file with `#pragma optimize` *flag* / `#pragma endoptimize`.

In addition to the option **--optimize**, you can specify the option **--tradeoff** (**-t**). With this option you specify whether the used optimizations should optimize for more speed (regardless of code size) or for smaller code size (regardless of speed).

Example

The following invocations are equivalent and result all in the default optimization set:

```
carc test.c

carc --optimize=2 test.c
carc -O2 test.c

carc --optimize test.c
carc -O test.c

carc -OabcefgIKlMoprsUvWy test.c
carc --optimize=+coalesce,+ipro,+cse,+expression,+flow,+glo,
      -inline,-schedule,+loop,-vectorize,+forward,+propagate,
      +compact,+subscript,-unroll,+ifconvert,-pipeline,+peephole test.c
```

Related information

C compiler option **--tradeoff** (Trade off between speed and size)

Pragma `optimize/endoptimize`

Section 4.6, *Compiler Optimizations*

C compiler option: --option-file (-f)

Menu entry

1. Select **C/C++ Compiler » Miscellaneous**.

2. Add the option **--option-file** to the **Additional options** field.

   Be aware that the options in the option file are added to the C compiler options you have set in the other pages. Only in extraordinary cases you may want to use them in combination.

Command line syntax

```
--option-file=file,...
```

```
-f file,...
```

Description

This option is primarily intended for command line use. Instead of typing all options on the command line, you can create an option file which contains all options and flags you want to specify. With this option you specify the option file to the compiler.

Use an option file when the command line would exceed the limits of the operating system, or just to store options and save typing.

You can specify the option **--option-file** multiple times.

Format of an option file

• Multiple arguments on one line in the option file are allowed.

• To include whitespace in an argument, surround the argument with single or double quotes.

• If you want to use single quotes as part of the argument, surround the argument by double quotes and vise versa:

```
"This has a single quote ' embedded"

'This has a double quote " embedded'

'This has a double quote " and a single quote '"' embedded"
```

• When a text line reaches its length limit, use a **** to continue the line. Whitespace between quotes is preserved.

```
"This is a continuation \
line"

        -> "This is a continuation line"
```

• It is possible to nest command line files up to 25 levels.

Example

Suppose the file `myoptions` contains the following lines:

```
--debug-info
--define=DEMO=1
test.c
```

Specify the option file to the compiler:

```
carc --option-file=myoptions
```

This is equivalent to the following command line:

```
carc —-debug-info --define=DEMO=1 test.c
```

Related information

-

C compiler option: --output-file (-o)

Menu entry

Eclipse names the output file always after the C source file.

Command line syntax

`--output-file=`*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:

```
carc --output-file=output.src test.c
```

Related information

-

C compiler option: --preprocess (-E)

Menu entry

1.   Select **C/C++ Compiler » Preprocessing**.

2.   Enable the option **Store preprocessor output in <file>.pre**.

3.   (Optional) Enable the option **Keep comments in preprocessor output**.

4.   (Optional) Enable the option **Keep #line info in preprocessor output**.

5.   (Optional) Enable the option **Insert a space between adjacent tokens (if needed)**.

Command line syntax

--preprocess[**=***flags*]

-E[*flags*]

You can set the following flags:

+/-comments	**c/C**	keep comments
+/-includes	**i/I**	generate a list of included source files
+/-list	**l/L**	generate a list of macro definitions
+/-make	**m/M**	generate dependencies for make
+/-noline	**p/P**	strip #line source position information
+/-token-separation	**t/T**	insert a space between adjacent tokens (if needed)

Default: **-ECILMPT**

Description

With this option you tell the compiler to preprocess the C source.

Under Eclipse the compiler sends the preprocessed output to the file *name*.pre (where *name* is the name of the C source file to compile). Eclipse also compiles the C source.

On the command line, the compiler sends the preprocessed file to stdout. To capture the information in a file, specify an output file with the option **--output-file**.

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 `.o`. With the option **--make-target** you can specify a target name which overrules the default target name.

With **--preprocess=+noline** you tell the preprocessor to strip the #line source position information (lines starting with `#line`). These lines are normally processed by the assembler and not needed in the preprocessed output. When you leave these lines out, the output is easier to read.

With **--preprocess=+token-separation** you tell the preprocessor to insert a space between adjacent tokens, if needed. For example, to prevent concatenation due to a macro expansion.

Example

```
carc --preprocess=+comments,+includes,-list,-make,-noline,+token-separation
     test.c --output-file=test.pre
```

The compiler preprocesses the file `test.c` and sends the output to the file `test.pre`. Comments, a list of all included source files and a space between tokes (if needed) are included but no list of macro definitions and no dependencies are generated and the line source position information is not stripped from the output file.

Related information

C compiler option **--dep-file** (Generate dependencies in a file)

C compiler option **--dep-format** (Format of dependency info)

C compiler option **--make-target** (Specify target name for **-Em** output)

C compiler option: --rename-sections (-R)

Menu entry

1. Select **C/C++ Compiler » Miscellaneous**.

2. Add the option **--rename-sections** to the **Additional options** field.

Command line syntax

--rename-sections[**=**_name_]

-R[_name_]

Description

The compiler defaults to a section naming convention, using a prefix indicating the section type, the module name and a symbol name:

`section_type_prefix.module_name.symbol_name`

For example, `.text.`*module_name.symbol_name* for code sections.

In case a module must be loaded at a fixed address, or a data section needs a special place in memory, you can use this option to generate different section names (*section_type_prefix.name* where *name* replaces the part *module_name.symbol_name*). You can then use this unique section name in the linker script file for locating.

If you use this option without a value or with an empty string, the compiler uses only the section type prefix as the section name.

Example

To generate the section name *section_type_prefix.*`NEW` instead of the default section name *section_type_prefix.module_name.symbol_name*, enter:

`carc -RNEW test.c`

To generate the section name *section_type_prefix* instead of the default section name *section_type_prefix.module_name.symbol_name*, enter:

`carc -R test.c`

or

`carc -R"" test.c`

Related information

Section 1.11, *Compiler Generated Sections*

C compiler option: --runtime (-r)

Menu entry

1. Select **C/C++ Compiler » Debugging**.

2. Enable or disable one or more of the following run-time error checking options:

    - Generate code for bounds checking

    - Generate code to detect unhandled case in a switch

    - Generate code for malloc consistency checks

Command line syntax

--runtime[**=***flag*,...]

-r[*flags*]

You can set the following flags:

+/-bounds	**b/B**	bounds checking
+/-case	**c/C**	report unhandled case in a switch
+/-malloc	**m/M**	malloc consistency checks

Default (without flags): **-rbcm**

Description

This option controls a number of run-time checks to detect errors during program execution. Some of these checks require additional code to be inserted in the generated code, and may therefore slow down the program execution. The following checks are available:

Bounds 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.

Related information

Pragma `runtime`

C compiler option: --save-irq-regs

Menu entry

1.   Select **C/C++ Compiler » Miscellaneous**.

2.   Add the option **--save-irq-regs** to the **Additional options** field.

Command line syntax

--save-irq-regs[**=***reg*[*-reg*],...]

Description

With this option you can specify the registers that are implicitly saved (by hardware) during an interrupt. The argument of this option is a comma-separated list of registers (r0, ..., r29, blink, lp_count) or in the range form (e.g. "r0-r3"). The resulting list of registers from the r0-r29 range must:

•   start at r0

•   be continuous by register number

•   contain an even number of registers

If you do not specify any register, all registers are saved.

Related information

Section 1.10.3, *Interrupt Functions / Exception Handling*

C compiler option: --schar

Menu entry

1.   Select **C/C++ Compiler » Language**.

2.   Enable the option **Treat 'char' variables as signed**.

Command line syntax

`--schar`

Description

By default `char` is the same as specifying `unsigned char` as required by the ABI. With this option `char` is the same as `signed char`.

Note that this option can cause conflicts when you use it in combination with MIL linking. With MIL linking some extra strict type checking is done that can cause building to fail in a way that is unforeseen and difficult to understand. For example, when you use option **--mil** in combination with option **--schar** and you link the MIL library, you might get the following error:

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

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

Related information

Section 1.1, *Data Types*

C compiler option: --sda-max-data-size

Menu entry

1. Select **C/C++ Compiler » Miscellaneous**.

2. In the **Threshold for putting data in SDA** field, enter a value in bytes.

Command line syntax

--sda-max-data-size=_size_

Default: 4 (bytes)

Description

By default, data consisting of 4 bytes or less will be placed in the Small Data Area (SDA). With this compiler option you can change this default limit of 4 bytes.

Instead of this option you can also use pragma sda_max_data_size around an object declaration. For example,

```
#pragma sda_max_data_size 16
int arr[4];
#pragma sda_max_data_size restore
```

You have to compile the entire program with the same **--sda-max-data-size** option value. More precisely, for every object all of its declarations have to be consistent with its definition with respect to the **--sda-max-data-size** option value (specified either as a compiler option, or as a pragma). So, if for example you override the option at a variable definition in some file with a pragma, you have to use the same pragma around all its extern declarations in other files.

The instruction set supports only a limited addressing range for SDA objects, and it's your responsibility to make sure all program objects fit into it. Objects accessed as bytes and half-words have even a more narrow range around the GP pointer: 512 bytes for single bytes and 1 KiB for half-words. If any access does not fit in the range the linker issues an error like:

```
larc E121: relocation error in "task1": relocation value 0x103680,
type R_ARC_SDA16_LD2, offset 0x222, section ".text" at address 0x86d4
is not within a 11-bit signed range from the value of gp as defined
by the symbol _SDA_BASE_
```

In this case you should mark some of the excessive variables with the __no_sda qualifier, reduce the value of the **--sda-max-data-size** option, or disable automatic SDA allocation completely by using **--sda-max-data-size=0**.

Example

To put data objects with a size of 16 bytes or smaller in SDA automatically, enter:

```
carc --sda-max-data-size=16 test.c
```

Related information

Section 1.3.1, *Memory Type Qualifiers*

Section 1.3.3, *Small Data Area (SDA)*

Pragma `sda_max_data_size`

C compiler option: --source (-s)

Menu entry

1. Select **C/C++ Compiler » Miscellaneous**.

2. Enable the option **Merge C source code with generated assembly**.

Command line syntax

```
--source
```

```
-s
```

Description

With this option you tell the compiler to merge C source code with generated assembly code in the output file. The C source lines are included as comments.

Related information

Pragmas `source/nosource`

C compiler option: --static

Menu entry

-

Command line syntax

`--static`

Description

With this option, the compiler treats external definitions at file scope (except for `main`) as if they were declared `static`. As a result, unused functions will be eliminated, and the alias checking algorithm assumes that objects with static storage cannot be referenced from functions outside the current module.

This option only makes sense when you specify all modules of an application on the command line.

To overrule this option for a specific function or variable, you can use the `export` attribute. For example, when a variable is accessed from assembly:

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

With the `export` attribute the compiler will not perform optimizations that affect the unknown code.

Example

```
carc --static module1.c module2.c module3.c ...
```

Related information

-

C compiler option: --stdout (-n)

Menu entry

-

Command line syntax

`--stdout`

`-n`

Description

With this option you tell the compiler to send the output to stdout (usually your screen). No files are created. This option is for example useful to quickly inspect the output or to redirect the output to other tools.

Related information

-

C compiler option: --tradeoff (-t)

Menu entry

1. Select **C/C++ Compiler » Optimization**.

2. Select a trade-off level in the **Trade-off between speed and size** box.

Command line syntax

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

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

Default: **--tradeoff=2**

Description

If the compiler uses certain optimizations (option **--optimize**), you can use this option to specify whether the used optimizations should optimize for more speed (regardless of code size) or for smaller code size (regardless of speed).

By default the compiler balances speed and size while optimizing (**--tradeoff=2**).

> If you have not specified the option **--optimize**, the compiler uses the default *Optimize more* optimization. In this case it is still useful to specify a trade-off level.

Example

To set the trade-off level for the used optimizations:

```
carc --tradeoff=4 test.c
```

The compiler uses the default *Optimize more* optimization level and optimizes for code size.

Related information

C compiler option **--optimize** (Specify optimization level)

Section 4.6.4, *Optimize for Code Size or Execution Speed*

C compiler option: --undefine (-U)

Menu entry

1. Select **C/C++ Compiler » Preprocessing**

   The Defined symbols box shows the symbols that are currently defined.

2. To remove a defined symbol, select the symbol in the **Defined symbols** box and click on the **Delete** button.

Command line syntax

--undefine=*macro_name*

-U*macro_name*

Description

With this option you can undefine an earlier defined macro as with `#undef`. This option is for example useful to undefine predefined macros.

The following predefined ISO C standard macros cannot be undefined:

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

Example

To undefine the predefined macro `__TASKING__`:

```
carc --undefine=__TASKING__ test.c
```

Related information

C compiler option **--define** (Define preprocessor macro)

Section 1.9, *Predefined Preprocessor Macros*

C compiler option: --undefined-macro

Menu entry

1.   Select **C/C++ Compiler » Miscellaneous**.

2.   Add the option **--undefined-macro** to the **Additional options** field.

Command line syntax

```
--undefined-macro
```

Description

With this option you tell the compiler to issue warning W598 when an undefined macro is replaced by zero in an #if condition.

Related information

-

C compiler option: --verbose (-v)

Menu entry

-

Command line syntax

`--verbose`

`-v`

Description

With this option you put the C compiler in verbose mode. The C compiler performs its tasks while it prints the steps it performs to stdout.

Related information

-

C compiler option: --version (-V)

Menu entry

-

Command line syntax

`--version`

`-V`

Description

Display version information. The compiler ignores all other options or input files.

Related information

-

C compiler option: --vccm-no-clear

Menu entry

1. Select **C/C++ Compiler » Miscellaneous**.

2. Add the option **--vccm-no-clear** to the **Additional options** field.

Command line syntax

```
--vccm-no-clear
```

Description

Normally uninitialized vector data is emitted in `.vbss` sections which are cleared (zero initialized) at program startup. With this option you tell the compiler to generate `.vbss` sections with the `noclear` attribute set. This prevents uninitialized vector data from being cleared at program startup.

Related information

Pragma `vccm_noclear`

C compiler option: --vectorize-info

Menu entry

1. Select **C/C++ Compiler » Optimization**.

2. In the **Optimization level** box, select **Custom Optimization**.

3. Select **C/C++ Compiler » Optimization » Custom Optimization**.

4. Enable **Auto-vectorization**.

5. Select **C/C++ Compiler » Optimization**.

6. Enable **Auto-vectorization diagnostics**.

Command line syntax

```
--vectorize-info
```

Description

With this option you enable additional informational diagnostics about the auto-vectorization optimization, such as which loops could be vectorized, and which loops could not be vectorized and why.

Related information

C compiler option **-Om / --optimize=+vectorize** (Loop auto-vectorization)

Loop auto-vectorization optimization

C compiler option: --vectorize-noalias

Menu entry

1.   Select **C/C++ Compiler » Optimization**.

2.   In the **Optimization level** box, select **Custom Optimization**.

3.   Select **C/C++ Compiler » Optimization » Custom Optimization**.

4.   Enable **Auto-vectorization**.

5.   Select **C/C++ Compiler » Optimization**.

6.   Enable **No auto-vectorization alias checking**.

Command line syntax

```
--vectorize-noalias
```

Description

By default, any possible aliases will disable auto-vectorization for a loop. An example of a possible alias is when the same array is accessed by different forms of subscripts, for instance with a different offset or stride, and at least one of them is used to modify the array. Another example is when an array is accessed via a non-restrict qualified pointer variable. This option will disable all aliasing checks for auto-vectorization. With #pragma vectorize_noalias you can selectively disable alias checking for specific loops.

Related information

C compiler option **-Om / --optimize=+vectorize** (Loop auto-vectorization)

Loop auto-vectorization optimization

Pragma vectorize_noalias

C compiler option: --vectorize-vccm

Menu entry

1. Select **C/C++ Compiler » Optimization**.

2. In the **Optimization level** box, select **Custom Optimization**.

3. Select **C/C++ Compiler » Optimization » Custom Optimization**.

4. Enable **Auto-vectorization**.

5. Select **C/C++ Compiler » Optimization**.

6. Enable **Assume vector data is in __vccm memory**.

Command line syntax

```
--vectorize-vccm
```

Description

By default, the auto-vectorization optimization (**-Om**) will only consider arrays in `__vccm` memory to be able to access them with vector load/store instructions. With this option you can specify that the compiler may assume that potentially vectorizable data is located in vector memory, even in the absence of the `__vccm` qualifier. This means that you are responsible for allocating the data in vector memory, for instance through the use of a linker script file.

Related information

C compiler option **-Om / --optimize=+vectorize** (Loop auto-vectorization)

Loop auto-vectorization optimization

C compiler option: --warning-level (-W)

Menu entry

-

Command line syntax

```
--warning-level=level
-Wlevel
```

Default: warning level 1

Description

With this option you can control the amount of warnings that are enabled by default. Possible levels are 1 and 2, a higher level enables more warnings. For warnings with a level higher than 1, the warning message descriptions produced by the **--diag** option show the warning level (level x) after the warning number. Warnings without a (level x) indication are level 1 warnings.

Related information

C compiler option **--diag** (Explanation of diagnostic messages)

C compiler option **--no-warnings** (Suppress some or all warnings)

C compiler option: --warnings-as-errors

Menu entry

1. Select **Global Options**.

2. Enable the option **Treat warnings as errors**.

Command line syntax

--warnings-as-errors[**=**`number`[`-number`]`,...`]

Description

If the compiler encounters an error, it stops compiling. When you use this option without arguments, you tell the compiler to treat all warnings not suppressed by option **--no-warnings** (or `#pragma warning`) as errors. This means that the exit status of the compiler will be non-zero after one or more compiler warnings. As a consequence, the compiler now also stops after encountering a warning.

You can limit this option to specific warnings by specifying a comma-separated list of warning numbers or ranges. In this case, this option takes precedence over option **--no-warnings** (and `#pragma warning`).

Related information

C compiler option **--no-warnings** (Suppress some or all warnings)

Pragma `warning`

10.3. C++ Compiler Options

This section lists all C++ compiler options.

Options in Eclipse versus options on the command line

Most command line options have an equivalent option in Eclipse but some options are only available on the command line. Eclipse invokes the C++ compiler via the control program. Therefore, it uses the syntax of the control program to pass options and files to the C++ compiler. If there is no equivalent option in Eclipse, you can specify a command line option in Eclipse as follows:

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

    The Properties dialog appears.

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

    In the right pane the Settings appear.

3. On the Tool Settings tab, select **C/C++ Compiler » Miscellaneous**.

4. In the **Additional options** field, enter one or more command line options.

    *Because Eclipse uses the control program, you have to precede the option with **-Wcp** to pass the option via the control program directly to the C++ compiler.*

Short and long option names

Options can have both short and long names. Short option names always begin with a single minus (**-**) character, long option names always begin with two minus (**--**) characters. You can abbreviate long option names as long as it forms a unique name. You can mix short and long option names on the command line.

If an option requires an argument, the argument may be separated from the keyword by white space, or the keyword may be immediately followed by **=***option*. When the second form is used there may not be any white space on either side of the equal sign.

Options can have flags or suboptions. To switch a flag 'on', use a lowercase letter or a **+***longflag*. To switch a flag off, use an uppercase letter or a **-***longflag*. Separate *longflags* with commas. The following two invocations are equivalent:

```
cparc -Ecp test.cc
cparc --preprocess=+comments,+noline test.cc
```

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

The priority of the options is left-to-right: when two options conflict, the first (most left) one takes effect. The **-D** and **-U** options are not considered conflicting options, so they are processed left-to-right for each source file. You can overrule the default output file name with the **--output-file** option.

C++ compiler option: --alternative-tokens

Menu entry

-

Command line syntax

`--alternative-tokens`

Description

Enable recognition of alternative tokens. This controls recognition of the digraph tokens in C++, and controls recognition of the operator keywords (e.g., `not`, `and`, `bitand`, etc.).

Example

To enable operator keywords (e.g., "not", "and") and digraphs, enter:

`cparc --alternative-tokens test.cc`

Related information

-

C++ compiler option: --building-runtime

Menu entry

-

Command line syntax

```
--building-runtime
```

Description

Special option for building the C++ run-time library. Used to indicate that the C++ run-time library is being compiled. This causes additional macros to be predefined that are used to pass configuration information from the C++ compiler to the run-time.

Related information

-

C++ compiler option: --c++

Menu entry

1. Select **C/C++ Compiler » Language**.

2. From the **Comply to C++ standard** list, select the ISO C++ standard.

Command line syntax

`--c++=`*year*

You can specify one of the following years: **11** or **14**.

Default: `--c++=14`

Description

With this option you select the ISO C++ standard. C++11 refers to ISO/IEC 14882:2011, C++14 refers to ISO/IEC 14882:2014. Exceptions are implicitly enabled.

Related information

C++ compiler option **--no-exceptions** (Disable exceptions)

C++ compiler option: --check

Menu entry

-

Command line syntax

`--check`

Description

With this option you can check the source code for syntax errors, without generating code. This saves time in developing your application because the code will not actually be compiled.

The C++ compiler reports any warnings and/or errors.

This option is available on the command line only.

Related information

C compiler option **--check** (Check syntax)

Assembler option **--check** (Check syntax)

C++ compiler option: --check-concatenations

Menu entry

-

Command line syntax

`--check-concatenations`

Description

With this option the preprocessor will generate a diagnostic when a macro concatenation (such as `a ## b`) does not result in a valid token.

Related information

-

C++ compiler option: --context-limit

Menu entry

-

Command line syntax

`--context-limit=`*number*

Default: `--context-limit=10`

Description

Set the context limit to *number*. The context limit is the maximum number of template instantiation context entries to be displayed as part of a diagnostic message. If the number of context entries exceeds the limit, the first and last *N* context entries are displayed, where *N* is half of the context limit. A value of zero is used to indicate that there is no limit.

Example

To set the context limit to 5, enter:

```
cparc --context-limit=5 test.cc
```

Related information

-

C++ compiler option: --core

Menu entry

1. Expand **C/C++ Build** and select **Processor**.

2. From the **Processor selection** list, make a selection.

Command line syntax

--core=_core_

You can specify the following core architectures:

ppu_tc45x	PPU core architecture of TC45x
ppu_tc49x	PPU core architecture of TC49x
ppu_tc49xn	PPU core architecture of TC49xN
ppu_tc4dx	PPU core architecture of TC4Dx

Default: ppu_tc49x

Description

With this option you specify the PPU core architecture for which you create your application. The core architecture determines which instructions are valid and which are not. If you use Eclipse or the control program, the ARC/PPU toolset derives the core from the processor you selected.

The C++ compiler defines the macro __CORE_*core*__ depending on the *core*. The *core* is converted to uppercase and '.' is removed. For example, if you specify **--core=ppu_tc4dx**, the C++ compiler defines the macro __CORE_PPU_TC4DX__. When you do not specify **--core**, the C++ compiler defines __CORE_PPU_TC49X__.

Example

To compile the file test.cc for the PPU core architecture of the TC49x, enter the following on the command line:

```
cparc --core=ppu_tc49x test.cc
```

Related information

Control program option **--cpu** (Select processor)

C++ compiler option: --create-pch

Menu entry

1. Select **C/C++ Compiler » Precompiled C++ Headers**.

2. Enter a filename in the **Create precompiled header file** field.

Command line syntax

`--create-pch=``filename`

Description

If other conditions are satisfied, create a precompiled header file with the specified name. If **--pch** (automatic PCH mode) or **--use-pch** appears on the command line following this option, its effect is erased.

Example

To create a precompiled header file with the name `test.pch`, enter:

```
cparc --create-pch=test.pch test.cc
```

Related information

C++ compiler option **--pch** (Automatic PCH mode)

C++ compiler option **--use-pch** (Use precompiled header file)

Section 2.9, *Precompiled Headers*

C++ compiler option: --defer-parse-function-templates

Menu entry

-

Command line syntax

`--defer-parse-function-templates`

`--no-defer-parse-function-templates`

Default: `--defer-parse-function-templates` in GNU C++ mode.

Description

Enable or disable deferral of prototype instantiations until the first actual instantiation of a function. This is used to permit the compilation of programs that contain definitions of unusable function templates.

It is enabled by default in GNU C++ mode.

Related information

-

C++ compiler option: --define (-D)

Menu entry

1. Select **C/C++ Compiler » Preprocessing**.

   The Defined symbols box shows the symbols that are currently defined.

2. To define a new symbol, click on the **Add** button in the **Defined symbols** box.

3. Type the symbol definition (for example, `demo=1`)

   Use the **Edit** and **Delete** button to change a macro definition or to remove a macro from the list.

Command line syntax

--define=*macro_name*[**(***parm-list***)**][**=***macro_definition*]

-D*macro_name***(***parm-list***)**][**=***macro_definition*]

Description

With this option you can define a macro and specify it to the preprocessor. If you only specify a macro name (no macro definition), the macro expands as '1'.

Function-style macros can be defined by appending a macro parameter list to *macro_name*.

You can specify as many macros as you like. Simply use the **Add** button to add new macro definitions.

On the command line, you can use the option **--define** (**-D**) multiple times. If the command line exceeds the limit of the operating system, you can define the macros in an option file which you then must specify to the C++ compiler with the option **--option-file** (**-f**) *file*.

The definition can be tested by the preprocessor with `#if`, `#ifdef` and `#ifndef`, for conditional compilations.

   Make sure you do not use a reserved keyword as a macro name, as this can lead to unexpected results.

Example

Consider the following program with conditional code to compile a demo program and a real program:

```
void main( void )
{
#if DEMO
    demo_func();   /* compile for the demo program */
#else
    real_func();   /* compile for the real program */
```

```
#endif
}
```

You can now use a macro definition to set the DEMO flag:

```
cparc --define=DEMO test.cc
cparc --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.

```
cparc --define="MAX(A,B)=((A) > (B) ? (A) : (B))" test.cc
```

Related information

C++ compiler option **--undefine** (Remove preprocessor macro)

C++ compiler option **--option-file** (Specify an option file)

C++ compiler option: --dep-file

Menu entry

-

Command line syntax

--dep-file[**=***file*]

Description

With this option you tell the C++ compiler to generate dependency lines that can be used in a Makefile. In contrast to the option **--preprocess=+make**, the dependency information will be generated in addition to the normal output file.

By default, the information is written to a file with extension .d (one for every input file). When you specify a filename, all dependencies will be combined in the specified file.

Example

```
cparc --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 **--dep-format** (Format of dependency info)

C++ compiler option **--preprocess=+make** (Generate dependencies for make)

C++ compiler option: --dep-format

Menu entry

-

Command line syntax

`--dep-format=`*format*

You can specify the following *format* arguments:

amk	amk format of dependency info
gnu	GNU format of dependency info

Default: **amk**

Description

With this option you can override the format of the make dependencies file from TASKING amk (the default format) to GNU make. The format determines how names with spaces or other special characters are escaped. In the amk format such names are double quoted. In the GNU format special characters are escaped using a backslash.

Example

```
cparc --dep-file=test.dep --dep-format=gnu 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` in the GNU make format.

Related information

C++ compiler option **--dep-file** (Generate dependencies in a file)

C++ compiler option **--preprocess=+make** (Generate dependencies for make)

C++ compiler option: --diag

Menu entry

-

Command line syntax

`--diag`

Description

With this option the C++ compiler displays a list of all diagnostic messages on `stdout` (usually your screen). The C++ compiler does not compile any files.

Related information

Section 5.3, *C++ Compiler Error Messages*

C++ compiler option: --digit-separators

Menu entry

-

Command line syntax

`--digit-separators`

Description

Enable support for digit separators (a C++14 feature). When this feature is enabled, an apostrophe can appear between the digits of a numeric literal and is ignored, e.g., `123'456'789`.

This feature is implicitly enabled in C++14 mode and later modes.

Related information

C++ compiler option **--c++=14** (C++ standard)

C++ compiler option: --embedded-c++

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Enable the option **Comply to embedded C++ subset**.

Command line syntax

`--embedded-c++`

Description

The "Embedded C++" subset does not support templates, exceptions, namespaces, new-style casts, RTTI, multiple inheritance, virtual base classes, and the `mutable` keyword. Select this option when you want the C++ compiler to give an error when you use any of them in your C++ source.

Related information

-

C++ compiler option: --error-file

Menu entry

-

Command line syntax

--error-file[**=***file*]

Description

With this option the C++ compiler redirects diagnostic messages to a file. If you do not specify a filename, the error file will be named after the output file with extension .ecp.

Example

To write diagnostic messages to errors.ecp instead of stderr, enter:

cparc --error-file=errors.ecp test.cc

Related information

-

C++ compiler option: --error-limit (-e)

Menu entry

-

Command line syntax

--error-limit=*number*

-e*number*

Default: **--error-limit=100**

Description

Set the error limit to *number*. The C++ compiler will abandon compilation after this number of errors (remarks and warnings are not counted). By default, the limit is 100.

Example

When you want compilation to stop when 10 errors occurred, enter:

```
cparc --error-limit=10 test.cc
```

Related information

-

381

C++ compiler option: --exc-spec-in-func-type

Menu entry

-

Command line syntax

`--exc-spec-in-func-type`

`--no-exc-spec-in-func-type`

Default: `--no-exc-spec-in-func-type`

Description

Enable or disable exceptions being part of a function type (this is a C++17 feature).

Related information

-

C++ compiler option: --extended-variadic-macros

Menu entry

-

Command line syntax

`--extended-variadic-macros`

`--no-extended-variadic-macros`

Description

Enable or disable support for macros with a variable number of arguments and allow the naming of the variable argument list.

Related information

-

C++ compiler option: --force-vtbl

Menu entry

1. Select **C/C++ Compiler » Miscellaneous**.

2. Enable the option **Force definition of virtual function tables (C++)**.

Command line syntax

```
--force-vtbl
```

Description

Force definition of virtual function tables in cases where the heuristic used by the C++ compiler to decide on definition of virtual function tables provides no guidance.

Related information

C++ compiler option **--suppress-vtbl** (Suppress definition of virtual function tables)

C++ compiler option: --g++

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Enable the option **Allow GNU C++ extensions**.

Command line syntax

`--g++`

Description

Enable GNU C++ compiler language extensions.

This option sets the predefined macros `__GNUC__`, `__GNUC_MINOR__`, `__GNUC_PATCHLEVEL__`, `__GNUC_VERSION__`, `__GNUC_STDC_INLINE__`, `__GNUG__`, `__CHAR16_TYPE__` and `__CHAR32_TYPE__`.

Related information

Section 2.2.2, *GNU C++ Mode*

Section 2.8, *Predefined Macros*

C++ compiler option: --generate-move-operations

Menu entry

-

Command line syntax

`--generate-move-operations`

Description

With this option the C++ compiler generates move constructors and move assignment operators according to standard C++11 rules.

Related information

-

C++ compiler option: --gnu-version

Menu entry

-

Command line syntax

`--gnu-version=`*version*

Default: 80100 (version 8.1.0)

Description

It depends on the GNU C++ compiler version if a particular GNU extension is supported or not. With this option you set the GNU C++ compiler version that should be emulated in GNU C++ mode. Version *x.y.z* of the GNU C++ compiler is represented by the value *x**10000+*y**100+*z*.

This option implicitly enables GNU C++ mode and sets the predefined macros `__GNUC__`, `__GNUC_MINOR__`, `__GNUC_PATCHLEVEL__`, `__GNUC_VERSION__`, `__GNUC_STDC_INLINE__`, `__GNUG__`, `__CHAR16_TYPE__` and `__CHAR32_TYPE__`.

Example

To specify version 4.7.1 of the GNU C++ compiler, enter:

```
cparc --g++ --gnu-version=40701 test.cc
```

Related information

Section 2.2.2, *GNU C++ Mode*

Section 2.8, *Predefined Macros*

387

C++ compiler option: --help (-?)

Menu entry

-

Command line syntax

--help[**=***item*]

-?

You can specify the following arguments:

   options   Show extended option descriptions

Description

Displays an overview of all command line options. When you specify an argument you can list extended information such as a list of option descriptions.

Example

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

```
cparc -?
cparc --help
cparc
```

The following invocation displays an extended list of the available options:

```
cparc --help=options
```

Related information

-

C++ compiler option: --ignore-std

Menu entry

-

Command line syntax

`--ignore-std`

Description

Enable a GNU C++ compatibility feature that makes the std namespace a synonym for the global namespace.

Related information

-

C++ compiler option: --implicit-extern-c-type-conversion

Menu entry

-

Command line syntax

`--implicit-extern-c-type-conversion`

Description

Enable the implicit type conversion between pointers to `extern "C"` and `extern "C++"` function types.

Related information

-

C++ compiler option: --implicit-noexcept

Menu entry

-

Command line syntax

`--implicit-noexcept`

Description

With this option destructors and deallocation functions (i.e., operator `delete` and operator `delete[]`) have implicit `noexcept` specifications if no explicit exception specification is provided.

Related information

-

C++ compiler option: --include-directory (-I)

Menu entry

1. Select **C/C++ Compiler » Include Paths**.

    The Include paths box shows the directories that are added to the search path for include files.

2. To define a new directory for the search path, click on the **Add** button in the **Include paths** box.

3. Type or select a path.

Use the **Edit** and **Delete** button to change a path or to remove a path from the list.

Command line syntax

`--include-directory=`*path*`,...`

`-I`*path*`,...`

Description

Add *path* to the list of directories searched for `#include` files whose names do not have an absolute pathname. You can specify multiple directories separated by commas.

Example

To add the directory `/proj/include` to the include file search path, enter:

`cparc --include-directory=/proj/include test.cc`

Related information

C++ compiler option **--include-file** (Include file at the start of a compilation)

C++ compiler option **--sys-include** (Add directory to system include file search path)

Section 5.2, *How the C++ Compiler Searches Include Files*

C++ compiler option: --include-file (-H)

Menu entry

1. Select **C/C++ Compiler » Preprocessing**.

   The Pre-include files box shows the files that are currently included before the compilation starts.

2. To define a new file, click on the **Add** button in the **Pre-include files** box.

3. Type the full path and file name or select a file.

Use the **Edit** and **Delete** button to change a file name or to remove a file from the list.

Command line syntax

`--include-file=`*file*

`-H`*file*

Description

Include the source code of the indicated file at the beginning of the compilation. This is the same as specifying `#include "file"` at the beginning of *each* of your C++ sources.

All files included with **--include-file** are processed after any of the files included with **--include-macros-file**.

The filename is searched for in the directories on the include search list.

Example

```
cparc --include-file=extra.h test1.cc test2.cc
```

The file `extra.h` is included at the beginning of both `test1.cc` and `test2.cc`.

Related information

C++ compiler option **--include-directory** (Add directory to include file search path)

Section 5.2, *How the C++ Compiler Searches Include Files*

C++ compiler option: --include-macros-file

Menu entry

-

Command line syntax

`--include-macros-file=`*file*

Description

Include the macros of the indicated file at the beginning of the compilation. Only the preprocessing directives from the file are evaluated. All of the actual code is discarded. The effect of this option is that any macro definitions from the specified file will be in effect when the primary source file is compiled. All of the macro-only files are processed before any of the normal includes (**--include-file**). Within each group, the files are processed in the order in which they were specified.

Related information

C++ compiler option **--include-file** (Include file at the start of a compilation)

Section 5.2, *How the C++ Compiler Searches Include Files*

C++ compiler option: --init-priority

Menu entry

-

Command line syntax

`--init-priority=`*number*

Default: 0

Description

Normally, the C++ compiler assigns no priority to the global initialization functions and the exact order is determined by the linker. This option sets the default priority for global initialization functions. Default value is "0". You can also set the default priority with the `#pragma init_priority`.

Values from 1 to 100 are for internal use only and should not be used. Values 101 to 65535 are available for user code. A lower number means a higher priority.

Example

```
cparc --init-priority=101 test.cc
```

Related information

-

C++ compiler option: --inline-statement-limit

Menu entry

-

Command line syntax

`--inline-statement-limit=`*number*

Default: 100

Description

With this option you can set the maximum number of statements that a routine can have and still be inlinable. The setting is somewhat arbitrary, but a limit is needed to prevent iterative inlining from exhausting memory. A value of zero allows an inlined routine to be any number of statements.

Related information

-

C++ compiler option: --instantiate (-t)

Menu entry

1. Select **C/C++ Compiler » Miscellaneous**.

2. Select an instantiation mode in the **Instantiation mode of external template entities** box.

Command line syntax

--instantiate=*mode*

-t*mode*

You can specify the following modes:

> **used**
> **all**
> **local**

Default: **--instantiate=used**

Description

Control instantiation of external template entities. External template entities are external (that is, non-inline and non-static) template functions and template static data members. The instantiation mode determines the template entities for which code should be generated based on the template definition. Normally, when a file is compiled, template entities are instantiated wherever they are used (the linker will discard duplicate definitions). The overall instantiation mode can, however, be changed with this option. You can specify the following modes:

used	Instantiate those template entities that were used in the compilation. This will include all static data members for which there are template definitions. This is the default.
all	Instantiate all template entities declared or referenced in the compilation unit. For each fully instantiated template class, all of its member functions and static data members will be instantiated whether or not they were used. Non-member template functions will be instantiated even if the only reference was a declaration.
local	Similar to **--instantiate=used** except that the functions are given internal linkage. This is intended to provide a very simple mechanism for those getting started with templates. The compiler will instantiate the functions that are used in each compilation unit as local functions, and the program will link and run correctly (barring problems due to multiple copies of local static variables). However, one may end up with many copies of the instantiated functions, so this is not suitable for production use.

You cannot use **--instantiate=local** in conjunction with automatic template instantiation.

Related information

Section 2.4, *Template Instantiation*

C++ compiler option: --integer-enumeration

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Enable the option **Always use 32-bit integers for enumeration**.

Command line syntax

```
--integer-enumeration
```

Description

Normally the C++ compiler treats enumerated types as the smallest data type possible (`char`, `short`, `int` or `long long`). This reduces code size. With this option the C++ compiler always treats enum-types as `int`.

Related information

Section 1.1, *Data Types*

C++ compiler option: --late-tiebreaker

Menu entry

-

Command line syntax

`--late-tiebreaker`

Default: early tiebreaker processing.

Description

Select the way that tie-breakers (e.g., cv-qualifier differences) apply in overload resolution. In "early" tie-breaker processing, the tie-breakers are considered at the same time as other measures of the goodness of the match of an argument value and the corresponding parameter type (this is the standard approach).

In "late" tie-breaker processing, tie-breakers are ignored during the initial comparison, and considered only if two functions are otherwise equally good on all arguments; the tie-breakers can then be used to choose one function over another.

Related information

-

C++ compiler option: --list-file (-L)

Menu entry

-

Command line syntax

--list-file=*file*

-L*file*

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, or **2** for exit from an include file, and omitted otherwise.

        The first line in the raw listing file is always an L line identifying the primary input file. L lines are also output for #line directives (key is omitted). L lines indicate the source position of the following source line in the raw listing file.

R, **W**,   An indication of a diagnostic (**R** for remark, **W** for warning, **E** for error, and **C** for catastrophic
E, or **C**   error). The line has the form:

```
S  "file-name" line_number column-number message-text
```

where *S* is R, W, E, or C, as explained above. Errors at the end of file indicate the last line of the primary source file and a column number of zero. Command line errors are catastrophes with an empty file name ("") and a line and column number of zero. Internal errors are catastrophes with position information as usual, and message-text beginning with (internal error). When a diagnostic displays a list (e.g., all the contending routines when there is ambiguity on an overloaded call), the initial diagnostic line is followed by one or more lines with the same overall format (code letter, file name, line number, column number, and message text), but in which the code letter is the lowercase version of the code letter in the initial line. The source position in such lines is the same as that in the corresponding initial line.

Example

To write raw listing information to the file `test.lst`, enter:

`cparc --list-file=test.lst test.cc`

Related information

-

C++ compiler option: --long-long

Menu entry

-

Command line syntax

`--long-long`

Description

Permit the use of `long long` in strict mode in dialects in which it is non-standard.

Related information

-

C++ compiler option: --lossy-conversion-warning

Menu entry

-

Command line syntax

`--lossy-conversion-warning`

Description

With this option a warning is issued whenever a conversion occurs from one arithmetic type to a smaller one or from a floating type to an integral type.

Related information

-

C++ compiler option: --make-target

Menu entry

-

Command line syntax

--make-target=*name*

Description

With this option you can overrule the default target name in the make dependencies generated by the options **--preprocess=+make** (**-Em**) and **--dep-file**. The default target name is the basename of the input file, with extension `.o`.

Example

```
cparc --preprocess=+make --make-target=mytarget.o test.cc
```

The compiler generates dependency lines with the default target name `mytarget.o` instead of `test.o`.

Related information

C++ compiler option **--preprocess=+make** (Generate dependencies for make)

C++ compiler option **--dep-file** (Generate dependencies in a file)

C++ compiler option **--dep-format** (Format of dependency info)

C++ compiler option: --max-cost-constexpr-call

Menu entry

-

Command line syntax

`--max-cost-constexpr-call=`*value*

Default: 50000

Description

With this option you can set the maximum "cost" for folding a top-level `constexpr` function or constructor call. In C++11 mode, the "cost" is the number of calls expanded from a single top-level call. In C++14 mode or later modes, the "cost" is a combination of the number of calls interpreted and the number of loop iterations performed during the interpretation of a top-level call.

Related information

-

C++ compiler option: --max-depth-constexpr-call

Menu entry

-

Command line syntax

`--max-depth-constexpr-call=`*value*

Default: 1000

Description

With this option you can set the maximum nesting depth of `constexpr` function and constructor calls.

Related information

-

C++ compiler option: --no-arg-dep-lookup

Menu entry

-

Command line syntax

`--no-arg-dep-lookup`

Default: argument dependent lookup of unqualified function names is performed.

Description

With this option you disable argument dependent lookup of unqualified function names.

Related information

-

C++ compiler option: --no-array-new-and-delete

Menu entry

-

Command line syntax

`--no-array-new-and-delete`

Default: array new and delete are supported.

Description

Disable support for array new and delete.

The macro `__ARRAY_OPERATORS` is defined when array new and delete is enabled.

Related information

-

C++ compiler option: --no-enum-overloading

Menu entry

-

Command line syntax

`--no-enum-overloading`

Description

Disable support for using operator functions to overload built-in operations on enum-typed operands.

Related information

-

C++ compiler option: --no-exceptions

Menu entry

-

Command line syntax

`--no-exceptions`

Default: Exceptions are enabled.

Description

With this option you disable support for exception handling in the C++ compiler.

The macro `__EXCEPTIONS` is defined when exception handling support is enabled.

Exceptions are implicitly enabled in C++11 mode and later modes.

Related information

C++ compiler option **--c++=11** (C++ standard)

C++ compiler option: --no-implicit-typename

Menu entry

-

Command line syntax

`--no-implicit-typename`

Default: implicit typename determination is enabled.

Description

Disable implicit determination, from context, whether a template parameter dependent name is a type or nontype.

Related information

-

C++ compiler option: --no-inlining

Menu entry

1. Select **C/C++ Compiler » Miscellaneous**.

2. Disable the option **Minimal inlining of function calls (C++)**.

Command line syntax

```
--no-inlining
```

Description

Disable minimal inlining of function calls.

Related information

-

C++ compiler option: --no-lambdas

Menu entry

-

Command line syntax

`--no-lambdas`

Default: lambdas are supported.

Description

Disable support for C++11 lambdas.

Related information

C++ compiler option **--c++=11** (C++ standard)

C++ compiler option: --nonstd-qualifier-deduction

Menu entry

-

Command line syntax

`--nonstd-qualifier-deduction`

Description

Controls whether non-standard template argument deduction should be performed in the qualifier portion of a qualified name. With this feature enabled, a template argument for the template parameter `T` can be deduced in contexts like `A<T>::B` or `T::B`. The standard deduction mechanism treats these as non-deduced contexts that use the values of template parameters that were either explicitly specified or deduced elsewhere.

Related information

-

C++ compiler option: --no-parse-templates

Menu entry

-

Command line syntax

`--no-parse-templates`

Default: parsing of nonclass templates is enabled.

Description

Disable the parsing of nonclass templates in their generic form (i.e., even if they are not really instantiated). It is done by default if dependent name processing is enabled.

Related information

-

C++ compiler option: --no-pch-messages

Menu entry

-

Command line syntax

`--no-pch-messages`

Default: a message is displayed indicating that a precompiled header file was created or used in the current compilation. For example,

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

Description

Disable the display of a message indicating that a precompiled header file was created or used in the current compilation.

Related information

C++ compiler option **--pch** (Automatic PCH mode)

C++ compiler option **--use-pch** (Use precompiled header file)

C++ compiler option **--create-pch** (Create precompiled header file)

Section 2.9, *Precompiled Headers*

417

C++ compiler option: --no-preprocessing-only

Menu entry

Eclipse always does a full compilation.

Command line syntax

`--no-preprocessing-only`

Description

You can use this option in conjunction with the options that normally cause the C++ compiler to do preprocessing only (e.g., **--preprocess**, etc.) to specify that a full compilation should be done (not just preprocessing).

Example

cparc --preprocess --no-preprocessing-only test.cc

Related information

C++ compiler option **--preprocess** (Preprocessing only)

C++ compiler option: --no-stdarg-builtin

Menu entry

-

Command line syntax

`--no-stdarg-builtin`

Description

Disable special treatment of the `stdarg.h` header. When enabled, the `stdarg.h` header is treated as a built-in, and references to its macros ("`va_start`" et al) are passed through as such in generated C code.

Related information

-

C++ compiler option: --no-stdinc / --no-stdcppinc

Menu entry

1. Select **C/C++ Compiler » Miscellaneous**.

2. Add the option **--no-stdinc** or **--no-stdcppinc** to the **Additional options** field.

Command line syntax

`--no-stdinc`

`--no-stdcppinc`

Description

With option **--no-stdinc** you tell the C++ compiler not to look in the default `include,include.cpp` and `include.cxx` directories relative to the installation directory, when searching for standard include files.

With option **--no-stdcppinc** you tell the C++ compiler not to look in the default `include.cxx` directories relative to the installation directory, when searching for standard C++11/C++14 include files.

This way the C++ compiler only searches in the include file search paths you specified.

Related information

Section 5.2, *How the C++ Compiler Searches Include Files*

C++ compiler option **--c++**

C++ compiler option: --no-use-before-set-warnings (-j)

Menu entry

1. Select **C/C++ Compiler » Diagnostics**.

2. Enable the option **Suppress C++ compiler "used before set" warnings**.

Command line syntax

`--no-use-before-set-warnings`

`-j`

Description

Suppress warnings on local automatic variables that are used before their values are set.

Related information

C++ compiler option **--no-warnings** (Suppress all warnings)

C++ compiler option: --no-warnings (-w)

Menu entry

1. Select **C/C++ Compiler » Diagnostics**.

    The Suppress C++ compiler warnings box shows the warnings that are currently suppressed.

2. To suppress a warning, click on the **Add** button in the **Suppress C++ compiler warnings** box.

3. Enter the numbers, separated by commas or as a range, of the warnings you want to suppress (for example `611,2709`). Or you can use the **Add** button multiple times.

4. To suppress all warnings, enable the option **Suppress all warnings**.

> Use the **Edit** and **Delete** button to change a warning number or to remove a number from the list.

Command line syntax

--no-warnings[**=**_number_,...]

-w[_number_,...]

Description

With this option you can suppresses all warning messages or specific warning messages. Error messages are still issued.

On the command line this option works as follows:

*   If you do not specify this option, all warnings are reported.

*   If you specify this option but without numbers, all warnings are suppressed.

*   If you specify this option with one or more numbers, only the specified warnings are suppressed. You can specify the option **--no-warnings=**_number_ multiple times.

*   Leave out any leading zeros.

Example

To suppress warning W0611, enter:

```
cparc --no-warnings=611 test.cc
```

Related information

C++ compiler option **--warnings-as-errors** (Treat warnings as errors)

C++ compiler option: --option-file (-f)

Menu entry

-

Command line syntax

--option-file=*file*

-f *file*

Description

This option is primarily intended for command line use. Instead of typing all options on the command line, you can create an option file which contains all options and flags you want to specify. With this option you specify the option file to the C++ compiler.

Use an option file when the command line would exceed the limits of the operating system, or just to store options and save typing.

You can specify the option **--option-file** multiple times.

Format of an option file

• Multiple arguments on one line in the option file are allowed.

• To include whitespace in an argument, surround the argument with single or double quotes.

• If you want to use single quotes as part of the argument, surround the argument by double quotes and vise versa:

```
"This has a single quote ' embedded"

'This has a double quote " embedded'

'This has a double quote " and a single quote '"' embedded"
```

• When a text line reaches its length limit, use a **** to continue the line. Whitespace between quotes is preserved.

```
"This is a continuation \
line"

        -> "This is a continuation line"
```

• It is possible to nest command line files up to 25 levels.

Example

Suppose the file `myoptions` contains the following lines:

```
--embedded-c++
--define=DEMO=1
test.cc
```

Specify the option file to the C++ compiler:

```
cparc --option-file=myoptions
```

This is equivalent to the following command line:

```
cparc --embedded-c++ --define=DEMO=1 test.cc
```

Related information

-

C++ compiler option: --output-file (-o)

Menu entry

Eclipse names the output file always after the C++ source file.

Command line syntax

--output-file=*file*

-o *file*

Default: module name with `.ic` suffix.

Description

With this option you can specify another filename for the output file of the C++ compiler. Without this option the basename of the C++ source file is used with extension `.ic`.

You can also use this option in combination with the option **--preprocess (-E)** to redirect the preprocessing output to a file.

Example

To create the file `output.ic` instead of `test.ic`, enter:

```
cparc --output-file=output.ic test.cc
```

To use the file `my.pre` as the preprocessing output file, enter:

```
cparc --preprocess --output-file=my.pre test.cc
```

Related information

C++ compiler option **--preprocess** (Preprocessing)

C++ compiler option: --pch

Menu entry

1. Select **C/C++ Compiler » Precompiled C++ Headers**.

2. Enable the option **Automatically use/create precompiled header file**.

Command line syntax

`--pch`

Description

Automatically use and/or create a precompiled header file. If **--use-pch** or **--create-pch** (manual PCH mode) appears on the command line following this option, its effect is erased.

Related information

C++ compiler option **--use-pch** (Use precompiled header file)

C++ compiler option **--create-pch** (Create precompiled header file)

Section 2.9, *Precompiled Headers*

C++ compiler option: --pch-dir

Menu entry

1. Select **C/C++ Compiler » Precompiled C++ Headers**.

2. Enter a path in the **Precompiled header file directory**.

Command line syntax

--pch-dir=*directory-name*

Description

Specify the directory in which to search for and/or create a precompiled header file. This option may be used with automatic PCH mode (**--pch**) or manual PCH mode (**--create-pch** or **--use-pch**).

Example

To use the directory /usr/include/pch to automatically create precompiled header files, enter:

```
cparc --pch-dir=/usr/include/pch --pch test.cc
```

Related information

C++ compiler option **--pch** (Automatic PCH mode)

C++ compiler option **--use-pch** (Use precompiled header file)

C++ compiler option **--create-pch** (Create precompiled header file)

Section 2.9, *Precompiled Headers*

427

C++ compiler option: --pch-verbose

Menu entry

-

Command line syntax

`--pch-verbose`

Description

In automatic PCH mode, for each precompiled header file that cannot be used for the current compilation, a message is displayed giving the reason that the file cannot be used.

Example

cparc --pch --pch-verbose test.cc

Related information

C++ compiler option **--pch** (Automatic PCH mode)

Section 2.9, *Precompiled Headers*

C++ compiler option: --pending-instantiations

Menu entry

-

Command line syntax

--pending-instantiations=*n*

where *n* is the maximum number of instantiations of a single template.

Default: 64

Description

Specifies the maximum number of instantiations of a given template that may be in process of being instantiated at a given time. This is used to detect runaway recursive instantiations. If *n* is zero, there is no limit.

Example

To specify a maximum of 32 pending instantiations, enter:

```
cparc --pending-instantiations=32 test.cc
```

Related information

Section 2.4, *Template Instantiation*

C++ compiler option: --preprocess (-E)

Menu entry

1. Select **C/C++ Compiler » Preprocessing**.

2. Enable the option **Store preprocessor output in <file>.pre**.

3. (Optional) Enable the option **Keep comments in preprocessor output**.

4. (Optional) Enable the option **Keep #line info in preprocessor output**.

Command line syntax

--preprocess[**=***flags*]

-E[*flags*]

You can set the following flags:

+/-comments	**c/C**	keep comments
+/-includes	**i/I**	generate a list of included source files
+/-make	**m/M**	generate dependencies for make
+/-noline	**p/P**	strip #line source position information

Default: **-ECIMP**

Description

With this option you tell the C++ compiler to preprocess the C++ source. Under Eclipse the C++ compiler sends the preprocessed output to the file *name*.pre (where *name* is the name of the C++ source file to compile). Eclipse also compiles the C++ source.

On the command line, the C++ compiler sends the preprocessed file to stdout. To capture the information in a file, specify an output file with the option **--output-file**.

With **--preprocess=+comments** you tell the preprocessor to keep the comments from the C++ source file in the preprocessed output.

With **--preprocess=+includes** the C++ compiler will generate a list of all included source files. The preprocessor output is discarded.

With **--preprocess=+make** the C++ compiler will generate dependency lines that can be used in a Makefile. The preprocessor output is discarded. The default target name is the basename of the input file, with the extension .o. 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

```
cparc --preprocess=+comments,-make,-noline test.cc --output-file=test.pre
```

The C++ compiler preprocesses the file test.cc and sends the output to the file test.pre. Comments are included but no dependencies are generated and the line source position information is not stripped from the output file.

Related information

C++ compiler option **--no-preprocessing-only** (Force full compilation)

C++ compiler option **--dep-file** (Generate dependencies in a file)

C++ compiler option **--dep-format** (Format of dependency info)

C++ compiler option **--make-target** (Specify target name for **-Em** output)

C++ compiler option: --remarks (-r)

Menu entry

1.   Select **C/C++ Compiler » Diagnostics**.

2.   Enable the option **Issue remarks on C++ code**.

Command line syntax

`--remarks`

`-r`

Description

Issue remarks, which are diagnostic messages even milder than warnings.

Related information

Section 5.3, *C++ Compiler Error Messages*

C++ compiler option: --remove-unneeded-entities

Menu entry

-

Command line syntax

`--remove-unneeded-entities`

Description

Enable an optimization to remove types, variables, routines, and related constructs that are not really needed. Something may be referenced but unneeded if it is referenced only by something that is itself unneeded; certain entities, such as global variables and routines defined in the translation unit, are always considered to be needed.

Related information

-

C++ compiler option: --rtti

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Enable the option **Support for C++ RTTI (run-time type information)**.

Command line syntax

```
--rtti
```

Default: RTTI (run-time type information) features are disabled.

Description

Enable support for RTTI (run-time type information) features: `dynamic_cast`, `typeid`.

The macro `__RTTI` is defined when RTTI support is enabled.

Related information

-

C++ compiler option: --schar (-s)

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Enable the option **Treat 'char' variables as signed**.

Command line syntax

`--schar`

`-s`

Description

By default `char` is the same as specifying `unsigned char`. With this option `char` is the same as `signed char`.

When plain `char` is signed, the macro `__SIGNED_CHARS__` is defined.

Related information

C++ compiler option **--uchar** (Plain char is unsigned)

Section 1.1, *Data Types*

C++ compiler option: --signed-bitfields

Menu entry

-

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 **--unsigned-bitfields**

Section 1.1, *Data Types*

C++ compiler option: --special-subscript-cost

Menu entry

-

Command line syntax

`--special-subscript-cost`

Description

Enable a special nonstandard weighting of the conversion to the integral operand of the `[]` operator in overload resolution.

This is a compatibility feature that may be useful with some existing code. With this feature enabled, the following code compiles without error:

```
struct A {
    A();
    operator int *();
    int operator[](unsigned);
};
void main() {
    A a;
    a[0];   // Ambiguous, but allowed with this option
            // operator[] is chosen
}
```

Related information

-

C++ compiler option: --strict (-A)

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Disable the option **Allow non-ANSI/ISO C++ features**.

Command line syntax

```
--strict
```

```
-A
```

Default: non-ANSI/ISO C++ features are enabled.

Description

Enable strict ANSI/ISO mode, which provides diagnostic messages when non-standard features are used, and disables features that conflict with ANSI/ISO C++. All ANSI/ISO violations are issued as errors.

Example

To enable strict ANSI/ISO mode, with error diagnostic messages, enter:

```
cparc --strict test.cc
```

Related information

C++ compiler option **--strict-warnings** (Strict ANSI/ISO mode with warnings)

C++ compiler option: --strict-warnings (-a)

Menu entry

-

Command line syntax

`--strict-warnings`

`-a`

Default: non-ANSI/ISO C++ features are enabled.

Description

This option is similar to the option **--strict**, but all violations are issued as warnings instead of errors.

Example

To enable strict ANSI mode, with warning diagnostic messages, enter:

```
cparc --strict-warnings test.cc
```

Related information

C++ compiler option **--strict** (Strict ANSI/ISO mode with errors)

C++ compiler option: --suppress-vtbl

Menu entry

1. Select **C/C++ Compiler » Miscellaneous**.

2. Enable the option **Suppress definition of virtual function tables (C++)**.

Command line syntax

```
--suppress-vtbl
```

Description

Suppress definition of virtual function tables in cases where the heuristic used by the C++ compiler to decide on definition of virtual function tables provides no guidance. The virtual function table for a class is defined in a compilation if the compilation contains a definition of the first non-inline non-pure virtual function of the class. For classes that contain no such function, the default behavior is to define the virtual function table (but to define it as a local static entity). The **--suppress-vtbl** option suppresses the definition of the virtual function tables for such classes, and the **--force-vtbl** option forces the definition of the virtual function table for such classes. **--force-vtbl** differs from the default behavior in that it does not force the definition to be local.

Related information

C++ compiler option **--force-vtbl** (Force definition of virtual function tables)

C++ compiler option: --sys-include

Menu entry

-

Command line syntax

--sys-include=*directory*,...

Description

Change the algorithm for searching system include files whose names do not have an absolute pathname to look in *directory*.

Example

To add the directory /proj/include to the system include file search path, enter:

```
cparc --sys-include=/proj/include test.cc
```

Related information

C++ compiler option **--include-directory** (Add directory to include file search path)

Section 5.2, *How the C++ Compiler Searches Include Files*

C++ compiler option: --template-typedefs-in-diagnostics

Menu entry

-

Command line syntax

`--template-typedefs-in-diagnostics`

Default: typedefs declared in template classes are not replaced with their underlying type.

Description

Enable the replacement of typedefs declared in template classes with their underlying type. Diagnostic messages are often more useful when such typedefs are replaced.

Related information

-

C++ compiler option: --timing

Menu entry

-

Command line syntax

`--timing`

Default: no timing information is generated.

Description

Generate compilation timing information. This option causes the C++ compiler to display the amount of CPU time and elapsed time used by each phase of the compilation and a total for the entire compilation.

Example

```
cparc --timing test.cc
```

```
processed 180 lines at 8102 lines/min
```

Related information

-

C++ compiler option: --trace-includes

Menu entry

-

Command line syntax

`--trace-includes`

Description

Output a list of the names of files #included to the error output file. The source file is compiled normally (i.e. it is not just preprocessed) unless another option that causes preprocessing only is specified.

Example

```
cparc --trace-includes test.cc
```

```
string.h
```

Related information

C++ compiler option **--preprocess** (Preprocessing only)

C++ compiler option: --type-traits-helpers

Menu entry

-

Command line syntax

`--type-traits-helpers`

`--no-type-traits-helpers`

Default: in C++ mode type traits helpers are enabled by default. In GNU C++ mode, type traits helpers are never enabled by default.

Description

Enable or disable type traits helpers (like `__is_union` and `__has_virtual_destructor`). Type traits helpers are meant to ease the implementation of ISO/IEC TR 19768.

The macro `__TYPE_TRAITS_ENABLED` is defined when type traits pseudo-functions are enabled.

Related information

-

C++ compiler option: --uchar (-u)

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Disable the option **Treat 'char' variables as signed**.

Command line syntax

`--uchar`

`-u`

Description

With this option char is the same as unsigned char. This is the default.

Related information

C++ compiler option **--schar** (Plain char is signed)

Section 1.1, *Data Types*

C++ compiler option: --undefine (-U)

Menu entry

1. Select **C/C++ Compiler » Preprocessing**

    The Defined symbols box shows the symbols that are currently defined.

2. To remove a defined symbol, select the symbol in the **Defined symbols** box and click on the **Delete** button.

Command line syntax

--undefine=*macro_name*

-U*macro_name*

Description

Remove any initial definition of *macro_name* as in `#undef`. **--undefine** options are processed after all **--define** options have been processed.

You cannot undefine a predefined macro as specified in Section 2.8, *Predefined Macros*, except for:

    __STDC__
    __cplusplus
    __SIGNED_CHARS__

Example

To undefine the predefined macro `__cplusplus`:

```
cparc --undefine=__cplusplus test.cc
```

Related information

C++ compiler option **--define** (Define preprocessor macro)

Section 2.8, *Predefined Macros*

447

C++ compiler option: --unsigned-bitfields

Menu entry

-

Command line syntax

```
--unsigned-bitfields
```

Description

For bit-fields it depends on the implementation whether a plain `int` is treated as `signed int` or `unsigned int`. By default (this option) an `int` bit-field is treated as `unsigned int`. This offers the best performance.

Related information

C++ compiler option **--signed-bitfields**

Section 1.1, *Data Types*

C++ compiler option: --use-pch

Menu entry

1.   Select **C/C++ Compiler » Precompiled C++ Headers**.

2.   Enter a filename in the **Use precompiled header file** field.

Command line syntax

--use-pch=*filename*

Description

Use a precompiled header file of the specified name as part of the current compilation. If **--pch** (automatic PCH mode) or **--create-pch** appears on the command line following this option, its effect is erased.

Example

To use the precompiled header file with the name test.pch, enter:

```
cparc --use-pch=test.pch test.cc
```

Related information

C++ compiler option **--pch** (Automatic PCH mode)

C++ compiler option **--create-pch** (Create precompiled header file)

Section 2.9, *Precompiled Headers*

C++ compiler option: --using-std

Menu entry

-

Command line syntax

`--using-std`

Default: implicit use of the `std` namespace is disabled.

Description

Enable implicit use of the `std` namespace when standard header files are included. Note that this does not do the equivalent of putting a "`using namespace std;`" in the program to allow old programs to be compiled with new header files; it has a special and localized meaning related to the TASKING versions of certain header files, and is unlikely to be of much use to end-users of the TASKING C++ compiler.

Related information

Section 2.3, *Namespace Support*

C++ compiler option: --utf8-char-literals

Menu entry

-

Command line syntax

`--utf8-char-literals`

Description

Enable C++17-style UTF-8 character literals (e.g., `u8'a'`).

Related information

-

C++ compiler option: --version (-V)

Menu entry

-

Command line syntax

`--version`

`-V`

Description

Display version information. The C++ compiler ignores all other options or input files.

Related information

-

C++ compiler option: --vla

Menu entry

-

Command line syntax

`--vla`

Description

Enable support for variable length arrays. This allows the declaration and use of arrays of automatic storage duration with dimensions that are fixed at run-time.

C++ compiler option: --warnings-as-errors

Menu entry

1. Select **Global Options**.

2. Enable the option **Treat warnings as errors**.

Command line syntax

`--warnings-as-errors`[`=`*number*`,...`]

Description

If the C++ compiler encounters an error, it stops compiling. When you use this option without arguments, you tell the C++ compiler to treat all warnings as errors. This means that the exit status of the C++ compiler will be non-zero after one or more compiler warnings. As a consequence, the C++ compiler now also stops after encountering a warning.

You can limit this option to specific warnings by specifying a comma-separated list of warning numbers.

Related information

C++ compiler option **--no-warnings** (Suppress all warnings)

C++ compiler option: --xref-file (-X)

Menu entry

-

Command line syntax

--xref-file=*file*

-X*file*

Description

Generate cross-reference information in a *file*. For each reference to an identifier in the source program, a line of the form

```
symbol_id name X file-name line-number column-number
```

is written, where *X* is

D	for definition;
d	for declaration (that is, a declaration that is not a definition);
M	for modification;
A	for address taken;
U	for used;
C	for changed (but actually meaning used and modified in a single operation, such as an increment);
R	for any other kind of reference, or
E	for an error in which the kind of reference is indeterminate.

symbol-id is a unique decimal number for the symbol. The fields of the above line are separated by tab characters.

Related information

-

10.4. Assembler Options

This section lists all assembler options.

Options in Eclipse versus options on the command line

Most command line options have an equivalent option in Eclipse but some options are only available on the command line. Eclipse invokes the assembler via the control program. Therefore, it uses the syntax of the control program to pass options and files to the assembler. If there is no equivalent option in Eclipse, you can specify a command line option in Eclipse as follows:

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

    The Properties dialog appears.

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

    In the right pane the Settings appear.

3. On the Tool Settings tab, select **Assembler » Miscellaneous**.

4. In the **Additional options** field, enter one or more command line options.

    *Because Eclipse uses the control program, Eclipse automatically precedes the option with **-Wa** to pass the option via the control program directly to the assembler.*

Note that the options you enter in the Assembler page are not only used for hand-coded assembly files, but also for the assembly files generated by the compiler.

Be aware that some command line options are not useful in Eclipse or just do not have any effect. For example, the option **-V** displays version header information and has no effect in Eclipse.

Short and long option names

Options can have both short and long names. Short option names always begin with a single minus (**-**) character, long option names always begin with two minus (**--**) characters. You can abbreviate long option names as long as it forms a unique name. You can mix short and long option names on the command line.

Options can have flags or suboptions. To switch a flag 'on', use a lowercase letter or a **+***longflag*. To switch a flag off, use an uppercase letter or a **-***longflag*. Separate *longflags* with commas. The following two invocations are equivalent:

```
asarc -gal test.src
asarc --debug-info=+asm,+local test.src
```

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

Assembler option: --assume-short

Menu entry

-

Command line syntax

`--assume-short`

Default: 16-bit instruction mnemonics in assembly code must have the _S suffix.

Description

With this option you tell the assembler to select the 16-bit version of an instruction if there is a valid encoding, even if the _S suffix is not specified in the instruction's mnemonic. If there is no valid 16-bit encoding, the assembler selects the 32-bit version of the instruction.

Example

Assume short instructions for mnemonics without _S suffix:

```
asarc --assume-short test.src
```

Related information

-

Assembler option: --case-insensitive (-c)

Menu entry

1. Select **Assembler » Symbols**.

2. Enable the option **Case insensitive identifiers**.

Command line syntax

```
--case-insensitive
```

```
-c
```

Default: case sensitive

Description

With this option you tell the assembler not to distinguish between uppercase and lowercase characters. By default the assembler considers uppercase and lowercase characters as different characters.

Assembly source files that are generated by the compiler must always be assembled case sensitive. When you are writing your own assembly code, you may want to specify the case insensitive mode.

Example

When assembling case insensitive, the label `LabelName` is the same label as `labelname`.

```
asarc --case-insensitive test.src
```

Related information

Assembler control **$CASE**

Assembler option: --check

Menu entry

-

Command line syntax

`--check`

Description

With this option you can check the source code for syntax errors, without generating code. This saves time in developing your application.

The assembler reports any warnings and/or errors.

This option is available on the command line only.

Related information

C compiler option **--check** (Check syntax)

Assembler option: --core

Menu entry

1. Expand **C/C++ Build** and select **Processor**.

2. From the **Processor selection** list, make a selection.

Command line syntax

--core=*core*

You can specify the following core architectures:

ppu_tc45x	PPU core architecture of TC45x
ppu_tc49x	PPU core architecture of TC49x
ppu_tc49xn	PPU core architecture of TC49xN
ppu_tc4dx	PPU core architecture of TC4Dx

Default: ppu_tc49x

Description

With this option you specify the PPU core architecture for which you create your application. The core architecture determines which instructions are valid and which are not.

To avoid conflicts, make sure you specify the same core architecture as you did for the compiler (Eclipse and the control program do this automatically).

Related information

Control program option **--core** (Select core architecture)

C compiler option **--core** (Select core architecture)

Assembler option: --debug-info (-g)

Menu entry

1. Select **Assembler » Symbols**.

2. Select an option from the **Generate symbolic debug** list.

Command line syntax

--debug-info[**=***flags*]

-g[*flags*]

You can set the following flags:

+/-asm	**a/A**	Assembly source line information
+/-hll	**h/H**	Pass high level language debug information (HLL)
+/-local	**l/L**	Assembler local symbols debug information
+/-smart	**s/S**	Smart debug information

Default: **--debug-info=+hll**

Default (without flags): **--debug-info=+smart**

Description

With this option you tell the assembler which kind of debug information to emit in the object file.

You cannot specify **--debug-info=+asm,+hll**. Either the assembler generates assembly source line information, or it passes HLL debug information.

When you specify **--debug-info=+smart**, the assembler selects which flags to use. If high level language information is available in the source file, the assembler passes this information (same as **--debug-info=-asm,+hll,-local**). If not, the assembler generates assembly source line information (same as **--debug-info=+asm,-hll,+local**).

With **--debug-info=AHLS** the assembler does not generate any debug information.

Related information

Assembler control **$DEBUG**

Assembler option: --define (-D)

Menu entry

1. Select **Assembler » Preprocessing**.

    The Defined symbols box right-below shows the symbols that are currently defined.

2. To define a new symbol, click on the **Add** button in the **Defined symbols** box.

3. Type the symbol definition (for example, `demo=1`)

Use the **Edit** and **Delete** button to change a macro definition or to remove a macro from the list.

Command line syntax

--define=*macro_name*[**=***macro_definition*]

-D*macro_name*[**=***macro_definition*]

Description

With this option you can define a macro and specify it to the assembler preprocessor. If you only specify a macro name (no macro definition), the macro expands as '1'.

You can specify as many macros as you like. Simply use the **Add** button to add new macro definitions.

On the command line, use the option **--define** (**-D**) multiple times. If the command line exceeds the limit of the operating system, you can define the macros in an option file which you then must specify to the assembler with the option **--option-file** (**-f**) *file*.

Defining macros with this option (instead of in the assembly source) is, for example, useful in combination with conditional assembly as shown in the example below.

This option has the same effect as defining symbols via the `.DEFINE`, `.SET`, and `.EQU` directives (similar to `#define` in the C language). With the `.MACRO` directive you can define more complex macros.

Make sure you do not use a reserved keyword as a macro name, as this can lead to unexpected results.

Example

Consider the following assembly program with conditional code to assemble a demo program and a real program:

```
.IF DEMO == 1
...          ; instructions for demo application
.ELSE
...          ; instructions for the real application
.ENDIF
```

You can now use a macro definition to set the DEMO flag:

```
asarc --define=DEMO test.src
asarc --define=DEMO=1 test.src
```

Note that both invocations have the same effect.

Related information

Assembler option **--option-file** (Specify an option file)

Assembler option: --dep-file

Menu entry

-

Command line syntax

`--dep-file`[`=`*file*]

Description

With this option you tell the assembler to generate dependency lines that can be used in a Makefile. The dependency information will be generated in addition to the normal output file.

By default, the information is written to a file with extension `.d`. When you specify a filename, all dependencies will be combined in the specified file.

Example

```
asarc --dep-file=test.dep test.src
```

The assembler assembles the file `test.src`, which results in the output file `test.o`, and generates dependency lines in the file `test.dep`.

Related information

Assembler option **--dep-format** (Format of dependency info)

Assembler option **--make-target** (Specify target name for **--dep-file** output)

Assembler option: --dep-format

Menu entry

-

Command line syntax

`--dep-format=`*format*

You can specify the following *format* arguments:

amk	amk format of dependency info
gnu	GNU format of dependency info

Default: **amk**

Description

With this option you can override the format of the make dependencies file from TASKING amk (the default format) to GNU make. The format determines how names with spaces or other special characters are escaped. In the amk format such names are double quoted. In the GNU format special characters are escaped using a backslash.

Example

```
asarc --dep-file=test.dep --dep-format=gnu test.src
```

The assembler assembles the file `test.src`, which results in the output file `test.o`, and generates dependency lines in the file `test.dep` in the GNU make format.

Related information

Assembler option **--dep-file** (Generate dependencies in a file)

465

Assembler option: --diag

Menu entry

1. From the **Window** menu, select **Show View » Other » TASKING » Problems**.

    The Problems view is added to the current perspective.

2. In the Problems view right-click on a message.

    A popup menu appears.

3. Select **Detailed Diagnostics Info**.

    A dialog box appears with additional information.

Command line syntax

--diag=[*format*:]{**all** | *nr*,...}

You can set the following output formats:

html	HTML output.
rtf	Rich Text Format.
text	ASCII text.

Default format: text

Description

With this option you can ask for an extended description of error messages in the format you choose. The output is directed to stdout (normally your screen) and in the format you specify. You can specify the following formats: **html**, **rtf** or **text** (default). To create a file with the descriptions, you must redirect the output.

With the suboption **all**, the descriptions of all error messages are given. If you want the description of one or more selected error messages, you can specify the error message numbers, separated by commas.

Example

To display an explanation of message number 244, enter:

```
asarc --diag=244
```

This results in the following message and explanation:

```
W244: additional input files will be ignored
```

```
The assembler supports only a single input file. All other input files are ignored.
```

To write an explanation of all errors and warnings in HTML format to file `aserrors.html`, use redirection and enter:

```
asarc --diag=html:all > aserrors.html
```

Related information

Section 6.5, *Assembler Error Messages*

Assembler option: --emit-locals

Menu entry

1. Select **Assembler » Symbols**.

2. Enable or disable one or both of the following options:

    *   Emit local EQU symbols

    *   Emit local non-EQU symbols

Command line syntax

--emit-locals[**=***flag*,...]

You can set the following flags:

+/-equs	**e/E**	emit local EQU symbols
+/-symbols	**s/S**	emit local non-EQU symbols

Default: **--emit-locals=ES**

Default (without flags): **--emit-locals=+symbols**

Description

With the option **--emit-locals=+equs** the assembler also emits local EQU symbols to the object file. Normally, only global symbols and non-EQU local symbols are emitted. Having local symbols in the object file can be useful for debugging.

Related information

Assembler directive **.EQU**

Assembler option: --error-file

Menu entry

-

Command line syntax

--error-file[**=***file*]

Description

With this option the assembler redirects diagnostic messages to a file. If you do not specify a filename, the error file will be named after the output file with extension `.ers`.

Example

To write diagnostic messages to `errors.ers` instead of `stderr`, enter:

```
asarc --error-file=errors.ers test.src
```

Related information

Section 6.5, *Assembler Error Messages*

Assembler option: --error-limit

Menu entry

1. Select **Assembler » Diagnostics**.

2. Enter a value in the **Maximum number of emitted errors** field.

Command line syntax

`--error-limit=`*number*

Default: 42

Description

With this option you tell the assembler to only emit the specified maximum number of errors. When 0 (null) is specified, the assembler emits all errors. Without this option the maximum number of errors is 42.

Related information

Section 6.5, *Assembler Error Messages*

Assembler option: --help (-?)

Menu entry

-

Command line syntax

`--help`[`=`*item*]

`-?`

You can specify the following arguments:

     options   Show extended option descriptions

Description

Displays an overview of all command line options. When you specify the argument **options** you can list detailed option descriptions.

Example

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

```
asarc -?
asarc --help
asarc
```

To see a detailed description of the available options, enter:

```
asarc --help=options
```

Related information

-

Assembler option: --include-directory (-I)

Menu entry

1. Select **Assembler » Include Paths**.

    The Include paths box shows the directories that are added to the search path for include files.

2. To define a new directory for the search path, click on the **Add** button in the **Include paths** box.

3. Type or select a path.

> Use the **Edit** and **Delete** button to change a path or to remove a path from the list.

Command line syntax

`--include-directory=`*path*`,...`

`-I`*path*`,...`

Description

With this option you can specify the path where your include files are located. A relative path will be relative to the current directory,

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

1. The pathname in the assembly file and the directory of the assembly source.

2. The path that is specified with this option.

3. The path that is specified in the environment variable `ASARCINC` when the product was installed.

4. The default directory `$(PRODDIR)\include`.

Example

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

`.INCLUDE 'myinc.inc'`

You can call the assembler as follows:

`asarc --include-directory=c:\proj\include test.src`

First the assembler looks for the file `myinc.inc` in the directory where `test.src` is located. If it does not find the file, it looks in the directory `c:\proj\include` (this option). If the file is still not found, the assembler searches in the environment variable and then in the default include directory.

Related information

Assembler option **--include-file** (Include file at the start of the input file)

Assembler option: --include-file (-H)

Menu entry

1. Select **Assembler » Preprocessing**.

    The Pre-include files box shows the files that are currently included before the assembling starts.

2. To define a new file, click on the **Add** button in the **Pre-include files** box.

3. Type the full path and file name or select a file.

Use the **Edit** and **Delete** button to change a file name or to remove a file from the list.

Command line syntax

`--include-file=`*file*`,...`

`-H`*file*`,...`

Description

With this option (set at project level) you include one extra file at the beginning of the assembly source file. The specified include file is included before all other includes. This is the same as specifying `.INCLUDE 'file'` at the beginning of your assembly source.

Example

```
asarc --include-file=sfr/regppu.def test.src
```

The file `regppu.def` in the `sfr` subdirectory of the `include` directory is included at the beginning of `test.src` before it is assembled.

Related information

Assembler option **--include-directory** (Add directory to include file search path)

Assembler option: --kanji

Menu entry

1. Select **Assembler » Miscellaneous**.

2. Enable the option **Allow Shift JIS Kanji in strings**.

Command line syntax

--kanji

Description

With this option you tell the assembler to support Shift JIS encoded Kanji multi-byte characters in strings. Without this option, encodings with 0x5c as the second byte conflict with the use of the backslash as an escape character. Shift JIS in comments is supported regardless of this option.

Note that Shift JIS also includes Katakana and Hiragana.

Related information

C compiler option **--language=+kanji** (Allow Shift JIS Kanji in strings)

Assembler option: --keep-output-files (-k)

Menu entry

Eclipse *always* removes the object file when errors occur during assembling.

Command line syntax

```
--keep-output-files
```

```
-k
```

Description

If an error occurs during assembling, the resulting object file (`.o`) may be incomplete or incorrect. With this option you keep the generated object file when an error occurs.

By default the assembler removes the generated object file when an error occurs. This is useful when you use the make utility. If the erroneous files are not removed, the make utility may process corrupt files on a subsequent invocation.

Use this option when you still want to use the generated object. For example when you know that a particular error does not result in a corrupt object file.

Related information

Assembler option **--warnings-as-errors** (Treat warnings as errors)

Assembler option: --list-file (-l)

Menu entry

1. Select **Assembler » List File**.

2. Enable the option **Generate list file**.

3. Enable or disable the types of information to be included.

Command line syntax

--list-file[**=***file*]

-l[*file*]

Default: no list file is generated

Description

With this option you tell the assembler to generate a list file. A list file shows the generated object code and the relative addresses. Note that the assembler generates a relocatable object file with relative addresses.

With the optional *file* you can specify an alternative name for the list file. By default, the name of the list file is the basename of the output file with the extension .lst.

Related information

Assembler option **--list-format** (Format list file)

477

Assembler option: --list-format (-L)

Menu entry

1. Select **Assembler » List File**.

2. Enable the option **Generate list file**.

3. Enable or disable the types of information to be included.

Command line syntax

--list-format=`flag`**,...**

-L`flags`

You can set the following flags:

+/-section	**d/D**	List section directives (`.SECTION`)
+/-symbol	**e/E**	List symbol definition directives
+/-generic-expansion	**g/G**	List expansion of generic instructions
+/-generic	**i/I**	List generic instructions
+/-line	**l/L**	List C preprocessor #line directives
+/-macro	**m/M**	List macro definitions
+/-empty-line	**n/N**	List empty source lines and comment lines (newline)
+/-conditional	**p/P**	List conditional assembly
+/-equate	**q/Q**	List equate and set directives (`.EQU`, `.SET`)
+/-relocations	**r/R**	List relocations characters ('r')
+/-hll	**s/S**	List HLL symbolic debug informations
+/-equate-values	**v/V**	List equate and set values
+/-wrap-lines	**w/W**	Wrap source lines
+/-macro-expansion	**x/X**	List macro expansions
+/-cycle-count	**y/Y**	List cycle counts
+/-define-expansion	**z/Z**	List define expansions

Use the following options for predefined sets of flags:

--list-format=0	**-L0**	All options disabled Alias for **--list-format=DEGILMNPQRSVWXYZ**
--list-format=1	**-L1**	All options enabled Alias for **--list-format=degilmnpqrsvwxyz**

Default: **--list-format=dEGilMnPqrsVwXyZ**

Description

With this option you specify which information you want to include in the list file.

On the command line you must use this option in combination with the option **--list-file** (**-l**).

Related information

Assembler option **--list-file** (Generate list file)

Assembler option **--section-info=+list** (Display section information in list file)

Assembler option: --make-target

Menu entry

-

Command line syntax

`--make-target=`*name*

Description

With this option you can overrule the default target name in the make dependencies generated by the option **--dep-file**. The default target name is the basename of the input file, with extension `.o`.

Example

```
asarc --dep-file --make-target=../mytarget.o test.src
```

The assembler generates dependency lines with the default target name `../mytarget.o` instead of `test.o`.

Related information

Assembler option **--dep-file** (Generate dependencies in a file)

Assembler option **--dep-format** (Format of dependency info)

Assembler option: --no-notes

Menu entry

1. Select **Assembler » Miscellaneous**.

2. Add the option **--no-notes** to the **Additional options** field.

Command line syntax

```
--no-notes
```

Description

By default, the assembler generates a note section in the object file. The note section contains compiler version and invocation information, if supplied in the input file, version and invocation information of the assembler, and information of environment variables, if used by the compiler or assembler. With this option you can suppress the generation of a note section in the output object file.

Related information

-

Assembler option: --no-reg-prefix

Menu entry

1. Select **Assembler » Miscellaneous**.

2. Add the option **--no-reg-prefix** to the **Additional options** field.

Command line syntax

```
--no-reg-prefix
```

Description

By default, the register names in an assembly instruction must have a % prefix. With this option, the assembler allows you to use registers with or without the % prefix.

When this option is enabled, make sure that there is no ambiguity between non-prefixed register names and user-defined symbol names used in the assembly code.

Example

With this option enabled, instead of

```
add %r1,%r2,%r4
```

you can also write

```
add r1,r2,r4
```

Related information

-

Assembler option: --no-require-end

Menu entry

1. Select **Assembler » Miscellaneous**.

2. Add the option **--no-require-end** to the **Additional options** field.

Command line syntax

`--no-require-end`

Description

With this option the assembly source does not have to be terminated with a `.END` directive.

Related information

`.END` (End of assembly)

Assembler option: --no-warnings (-w)

Menu entry

1. Select **Assembler » Diagnostics**.

    The Suppress warnings box shows the warnings that are currently suppressed.

2. To suppress a warning, click on the **Add** button in the **Suppress warnings** box.

3. Enter the numbers, separated by commas, of the warnings you want to suppress (for example `201,202`). Or you can use the **Add** button multiple times.

4. To suppress all warnings, enable the option **Suppress all warnings**.

Use the **Edit** and **Delete** button to change a warning number or to remove a number from the list.

Command line syntax

--no-warnings[**=***number*,...]

-w[*number*,...]

Description

With this option you can suppresses all warning messages or specific warning messages.

On the command line this option works as follows:

*   If you do not specify this option, all warnings are reported.

*   If you specify this option but without numbers, all warnings are suppressed.

*   If you specify this option with a number, only the specified warning is suppressed. You can specify the option **--no-warnings=***number* multiple times.

Example

To suppress warnings 201 and 202, enter:

```
asarc test.src --no-warnings=201,202
```

Related information

Assembler option **--warnings-as-errors** (Treat warnings as errors)

Assembler option: --option-file (-f)

Menu entry

1. Select **Assembler » Miscellaneous**.

2. Add the option **--option-file** to the **Additional options** field.

    Be aware that the options in the option file are added to the assembler options you have set in the other pages. Only in extraordinary cases you may want to use them in combination.

Command line syntax

--option-file=*file*,...

-f *file*,...

Description

This option is primarily intended for command line use. Instead of typing all options on the command line, you can create an option file which contains all options and flags you want to specify. With this option you specify the option file to the assembler.

Use an option file when the command line would exceed the limits of the operating system, or just to store options and save typing.

Option files can also be generated on the fly, for example by the make utility. You can specify the option **--option-file** multiple times.

Format of an option file

*   Multiple arguments on one line in the option file are allowed.

*   To include whitespace in an argument, surround the argument with single or double quotes.

*   If you want to use single quotes as part of the argument, surround the argument by double quotes and vise versa:

    ```
    "This has a single quote ' embedded"

    'This has a double quote " embedded'

    'This has a double quote " and a single quote '"' embedded"
    ```

*   When a text line reaches its length limit, use a **** to continue the line. Whitespace between quotes is preserved.

    ```
    "This is a continuation \
    line"

            -> "This is a continuation line"
    ```

- It is possible to nest command line files up to 25 levels.

Example

Suppose the file `myoptions` contains the following lines:

```
--debug-info=+asm,-local
test.src
```

Specify the option file to the assembler:

```
asarc --option-file=myoptions
```

This is equivalent to the following command line:

```
asarc --debug-info=+asm,-local test.src
```

Related information

-

Assembler option: --output (-o)

Menu entry

Eclipse names the output file always after the input file.

Command line syntax

--output=*file*

-o *file*

Description

With this option you can specify another filename for the output file of the assembler. Without this option, the basename of the assembly source file is used with extension `.o`.

Example

To create the file `relobj.o` instead of `asm.o`, enter:

```
asarc --output=relobj.o asm.src
```

Related information

-

Assembler option: --page-length

Menu entry

1. Select **Assembler » Miscellaneous**.

2. Add the option **--page-length** to the **Additional options** field.

Command line syntax

`--page-length=`*number*

Default: 72

Description

If you generate a list file with the assembler option **--list-file**, this option sets the number of lines in a page in the list file. The default is 72, the minimum is 10. As a special case, a page length of 0 turns off page breaks.

Related information

Assembler option **--list-file** (Generate list file)

Assembler control **$PAGE**

Assembler option: --page-width

Menu entry

1. Select **Assembler » Miscellaneous**.

2. Add the option **--page-width** to the **Additional options** field.

Command line syntax

`--page-width=`*number*

Default: 132

Description

If you generate a list file with the assembler option **--list-file**, this option sets the number of columns per line on a page in the list file. The default is 132, the minimum is 40.

Related information

Assembler option **--list-file** (Generate list file)

Assembler control **$PAGE**

Assembler option: --preprocess (-E)

Menu entry

-

Command line syntax

`--preprocess`

`-E`

Description

With this option the assembler will only preprocess the assembly source file. The assembler sends the preprocessed file to stdout.

Related information

-

Assembler option: --preprocessor-type (-m)

Menu entry

1. Select **Assembler » Preprocessing**.

2. Enable or disable the option **Use TASKING preprocessor**.

Command line syntax

`--preprocessor-type=`*type*

`-m`*type*

You can set the following preprocessor types:

none	**n**	No preprocessor
tasking	**t**	TASKING preprocessor

Default: `--preprocessor-type=tasking`

Description

With this option you select the preprocessor that the assembler will use. By default, the assembler uses the TASKING preprocessor.

When the assembly source file does not contain any preprocessor symbols, you can specify to the assembler not to use a preprocessor.

Related information

-

Assembler option: --section-info (-t)

Menu entry

1. Select **Assembler » List File**.

2. Enable the option **Generate list file**.

3. Enable the option **List section summary**.

and/or

1. Select **Assembler » Diagnostics**.

2. Enable the option **Display section summary**.

Command line syntax

`--section-info`[`=`*flag*`,...`]

`-t`[*flags*]

You can set the following flags:

+/-console	**c/C**	Display section summary on console
+/-list	**l/L**	List section summary in list file

Default: `--section-info=CL`

Default (without flags): `--section-info=cl`

Description

With this option you tell the assembler to display section information. For each section its memory space, size, total cycle counts and name is listed on stdout and/or in the list file.

The cycle count consists of two parts: the total accumulated count for the section and the total accumulated count for all repeated instructions. In the case of nested loops it is possible that the total supersedes the section total.

Example

To writes the section information to the list file and also display the section information on stdout, enter:

```
asarc --list-file --section-info asm.src
```

Related information

Assembler option **--list-file** (Generate list file)

Assembler option: --symbol-scope (-i)

Menu entry

1. Select **Assembler » Symbols**.

2. Enable or disable the option **Set default symbol scope to global**.

Command line syntax

--symbol-scope=*scope*

-i*scope*

You can set the following scope:

global	**g**	Default symbol scope is global
local	**l**	Default symbol scope is local

Default: **--symbol-scope=local**

Description

With this option you tell the assembler how to treat symbols that you have not specified explicitly as global or local. By default the assembler treats all symbols as local symbols unless you have defined them explicitly as global.

Related information

Assembler directive **.GLOBAL**

Assembler directive **.LOCAL**

Assembler control **$IDENT**

Assembler option: --version (-V)

Menu entry

-

Command line syntax

`--version`

`-V`

Description

Display version information. The assembler ignores all other options or input files.

Related information

-

Assembler option: --warnings-as-errors

Menu entry

1. Select **Global Options**.

2. Enable the option **Treat warnings as errors**.

Command line syntax

`--warnings-as-errors`[`=`*number,...*]

Description

If the assembler encounters an error, it stops assembling. When you use this option without arguments, you tell the assembler to treat all warnings as errors. This means that the exit status of the assembler will be non-zero after one or more assembler warnings. As a consequence, the assembler now also stops after encountering a warning.

You can limit this option to specific warnings by specifying a comma-separated list of warning numbers.

Related information

Assembler option **--no-warnings** (Suppress some or all warnings)

10.5. Linker Options

This section lists all linker options.

Options in Eclipse versus options on the command line

Most command line options have an equivalent option in Eclipse but some options are only available on the command line. Eclipse invokes the linker via the control program. Therefore, it uses the syntax of the control program to pass options and files to the linker. If there is no equivalent option in Eclipse, you can specify a command line option in Eclipse as follows:

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

    The Properties dialog appears.

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

    In the right pane the Settings appear.

3. On the Tool Settings tab, select **Linker » Miscellaneous**.

4. In the **Additional options** field, enter one or more command line options.

    *Because Eclipse uses the control program, Eclipse automatically precedes the option with **-Wl** to pass the option via the control program directly to the linker.*

Be aware that some command line options are not useful in Eclipse or just do not have any effect. For example, the option **--keep-output-files** keeps files after an error occurred. When you specify this option in Eclipse, it will have no effect because Eclipse always removes the output file after an error had occurred.

Short and long option names

Options can have both short and long names. Short option names always begin with a single minus (**-**) character, long option names always begin with two minus (**--**) characters. You can abbreviate long option names as long as it forms a unique name. You can mix short and long option names on the command line.

Options can have flags or suboptions. To switch a flag 'on', use a lowercase letter or a **+***longflag*. To switch a flag off, use an uppercase letter or a **-***longflag*. Separate *longflags* with commas. The following two invocations are equivalent:

```
larc -mfkl test.o
larc --map-file-format=+files,+link,+locate test.o
```

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

Linker option: --binfill

Menu entry

-

Command line syntax

--binfill=*pattern*

Default: 0x00

Description

With this option you can specify an unsigned 32-bit fill pattern for the binary output file. To use this option, you first need to set the command to inform the linker to produce a binary file. You can do this by setting the output file type as BIN with linker option **--chip-output (-c)**. If this is not done then option **--binfill** is ignored.

Example

To convert two Intel Hex input files to a binary output file and fill the memory gaps with 0x2D, enter the following on the command line:

```
larc myproj_1.hex myproj_2.hex -dtc49x.lsl --core=ppu --chip-output=myproj:bin --binfill=0x2D
```

Related information

Linker option **--chip-output**

Section 7.6, *Converting Intel Hex to Binary Format*

Linker option: --case-insensitive

Menu entry

1. Select **Linker » Miscellaneous**.

2. Enable the option **Link case insensitive**.

Command line syntax

`--case-insensitive`

Default: case sensitive

Description

With this option you tell the linker not to distinguish between uppercase and lowercase characters in symbols. By default the linker considers uppercase and lowercase characters as different characters.

Assembly source files that are generated by the compiler must *always* be assembled and thus linked case sensitive. When you have written your own assembly code and specified to assemble it case insensitive, you must also link the `.o` file case insensitive.

Related information

Assembler option **--case-insensitive**

Linker option: --c-array-element-type

Menu entry

-

Command line syntax

`--c-array-element-type=`*string*

Default: `unsigned long`

Description

With this option you can overrule the C data type to be used for all C array elements in a C array output file. The type must be an integral type. Without this option the default data type is `unsigned long`.

Related information

Section 14.4, *C Array Format*

Linker option **--chip-output** (Generate an output file for each chip)

Linker option: --chip-output (-c)

Menu entry

1. Select **Linker » Output Format**.

2. Enable the option **Generate Intel Hex format file** and/or **Generate S-records file** and/or **Generate C array file** or enable **Generate binary file**.

3. Enable the option **Create file for each memory chip**.

4. Optionally, specify the **Size of addresses**.

   Eclipse always uses the project name as the basename for the output file.

Command line syntax

--chip-output=[*basename*]**:***format*[**:***addr_size*]**,...**

-c[*basename*]**:***format*[**:***addr_size*]**,...**

You can specify the following formats:

IHEX	Intel Hex
SREC	Motorola S-records
BIN	Binary
CARR	C array

For Intel Hex and 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-records you can specify: **2** (S1 records), **3** (S2 records) or **4** bytes (S3 records, default). For binary files an address size is not allowed. For C array files, the address size specifies the number of bits stored in each array element.

Description

With this option the linker generates an output file for each memory chip in the specified format: Intel Hex, Motorola S-records, binary or C array. You can use the Intel Hex or Motorola S-record output for loading into a PROM-programmer, or you can use the binary or C array output for importing the application into a host application. The C array format contains the generated machine code in the form of C code. For more information see Section 14.4, *C Array Format*.

The linker generates a file for each ROM or RAM memory defined in the LSL file, where one or more initialized sections are located. For example:

```
memory memname
{  type=rom;  }
```

The name of the file is the name of the Eclipse project or, on the command line, the name of the memory device that was emitted with extension `.hex` (Intel Hex), `.sre` (Motorola S-records) or `.bin` (binary without metadata). For the C array format the output is a `.c` file for the array definition and a `.h` file for

the accompanying header file. Optionally, you can specify a *basename* which prepends the generated file name.

The linker also always generates a task-related absolute object file in ELF/DWARF format and a memory definition file, unless you specify linker option **--no-default-output**.

Example

To generate Intel Hex output files for each defined memory, enter the following on the command line:

```
larc --chip-output=myprog:IHEX test1.o
```

In this case, this generates the file myprog_*memname*.hex.

To generate C array output files for each defined memory, enter the following on the command line:

```
larc --chip-output=myprog:CARR:32 test1.o
```

In this case, this generates the files myprog_*memname*.c and myprog_*memname*.h.

Related information

Chapter 14, *Object File Formats*

Linker option **--output** (Output file)

Linker option **--hex-format** (Specify Hex file or C array format settings)

Linker option **--no-default-output** (No default task-related output files)

Linker option **--binfill** (Binary fill pattern)

Linker option: --core (-C)

Menu entry

-

Command line syntax

--core=core

-Ccore

You can specify the following cores:

    ppu   PPU core

Default: **ppu**

Description

With this option you specify the core of the target processor for the current link task. Only one task can be assigned to a certain core.

Example

To link objects for the ARC core `ppu`, enter:

```
larc -o test.elf -dtc49x.lsl --core=ppu -D__CPU__=tc49x -D__PROC_TC49X__
     --map-file test.o -lc_fpu -lrt -L~/carc/lib/tc49x
```

Related information

-

Linker option: --define (-D)

Menu entry

1. Select **Linker » Script File**.

    The Defined symbols box shows the symbols that are currently defined.

2. To define a new symbol, click on the **Add** button in the **Defined symbols** box.

3. Type the symbol definition (for example, `demo=1`)

Use the **Edit** and **Delete** button to change a macro definition or to remove a macro from the list.

Command line syntax

--define=`macro_name`[**=**`macro_definition`]

-D`macro_name`[**=**`macro_definition`]

Description

With this option you can define a macro and specify it to the linker LSL file preprocessor. If you only specify a macro name (no macro definition), the macro expands as '1'.

You can specify as many macros as you like; just use the option **--define** (**-D**) multiple times. If the command line exceeds the limit of the operating system, you can define the macros in an option file which you then must specify to the linker with the option **--option-file** (**-f**) *file*.

The definition can be tested by the preprocessor with `#if`, `#ifdef` and `#ifndef`, for conditional locating.

Make sure you do not use a reserved keyword as a macro name, as this can lead to unexpected results.

Example

To define the RESET vector, which is used in the linker script file `tc49x.lsl`, enter:

```
larc test.o -otest.elf --lsl-file=tc49x.lsl --define=RESET=0x80000000
```

Related information

Linker option **--option-file** (Specify an option file)

503

Linker option: --dep-file

Menu entry

Eclipse uses this option in the background to create a file with extension `.d` (one for every input file).

Command line syntax

`--dep-file`[**`=`**`file`]

Description

With this option you tell the linker to generate dependency lines that can be used in a Makefile. The dependency information will be generated in addition to the normal output file.

By default, the information is written to the file `larc.d`. When you specify a filename, all dependencies will be combined in the specified file.

Example

```
larc --dep-file=test.dep test.o
```

The linker links the file `test.o`, which results in the output file `test.elf`, and generates dependency lines in the file `test.dep`.

Related information

Linker option **--dep-format** (Format of dependency info)

Linker option **--make-target** (Target to use in dependencies file)

Linker option: --dep-format

Menu entry

-

Command line syntax

`--dep-format=`*format*

You can specify the following *format* arguments:

amk	amk format of dependency info
gnu	GNU format of dependency info

Default: **amk**

Description

With this option you can override the format of the make dependencies file from TASKING amk (the default format) to GNU make. The format determines how names with spaces or other special characters are escaped. In the amk format such names are double quoted. In the GNU format special characters are escaped using a backslash.

Example

```
larc --dep-file=test.dep --dep-format=gnu test.o
```

The linker links the file `test.o`, which results in the output file `test.elf`, and generates dependency lines in the file `test.dep` in the GNU make format.

Related information

Linker option **--dep-file** (Generate dependencies in a file)

Linker option: --diag

Menu entry

1. From the **Window** menu, select **Show View » Other » TASKING » Problems**.

    The Problems view is added to the current perspective.

2. In the Problems view right-click on a message.

    A popup menu appears.

3. Select **Detailed Diagnostics Info**.

    A dialog box appears with additional information.

Command line syntax

--diag=[*format*:]{**all** | *nr*,...}

You can set the following output formats:

html	HTML output.
rtf	Rich Text Format.
text	ASCII text.

Default format: text

Description

With this option you can ask for an extended description of error messages in the format you choose. The output is directed to stdout (normally your screen) and in the format you specify. You can specify the following formats: **html**, **rtf** or **text** (default). To create a file with the descriptions, you must redirect the output.

With the suboption **all**, the descriptions of all error messages are given. If you want the description of one or more selected error messages, you can specify the error message numbers, separated by commas.

With this option the linker does not link/locate any files.

Example

To display an explanation of message number 106, enter:

```
larc --diag=106
```

This results in the following message and explanation:

```
E106: unresolved external: <message>

The linker could not resolve all external symbols.
```

This is an error when the incremental linking option is disabled.
The <message> indicates the symbol that is unresolved.

To write an explanation of all errors and warnings in HTML format to file `lkerrors.html`, use redirection and enter:

```
larc --diag=html:all > lkerrors.html
```

Related information

Section 7.12, *Linker Error Messages*

Linker option: --duplicate-section-limit

Menu entry

-

Command line syntax

`--duplicate-section-limit=`*number*

Default: 10000

Description

With this option, duplicate code and/or data sections are not removed if more than *number* sections would have to be compared.

The duplicate section removal optimization performs detailed comparisons between sections with equal contents to check if they are indeed equal. Since these comparisons can be computationally intensive, the optimization is only performed when the number of sections to be compared is under a limit. With this option you can specify a custom maximum amount.

Related information

Section 7.7, *Linker Optimizations*

Linker option **--optimize=+delete-duplicate-code** (**-Ox**) (Delete duplicate code sections)

Linker option **--optimize=+delete-duplicate-data** (**-Oy**) (Delete duplicate data sections)

Linker option: --error-file

Menu entry

-

Command line syntax

--error-file[**=***file*]

Description

With this option the linker redirects diagnostic messages to a file. If you do not specify a filename, the error file is larc.elk.

Example

To write diagnostic messages to errors.elk instead of stderr, enter:

larc --error-file=errors.elk test.o

Related information

Section 7.12, *Linker Error Messages*

Linker option: --error-limit

Menu entry

1. Select **Linker » Diagnostics**.

2. Enter a value in the **Maximum number of emitted errors** field.

Command line syntax

`--error-limit=`*number*

Default: 42

Description

With this option you tell the linker to only emit the specified maximum number of errors. When 0 (null) is specified, the linker emits all errors. Without this option the maximum number of errors is 42.

Related information

Section 7.12, *Linker Error Messages*

Linker option: --extern (-e)

Menu entry

-

Command line syntax

--extern=*symbol*,...

-e*symbol*,...

Description

With this option you force the linker to consider the given symbol as an undefined reference. The linker tries to resolve this symbol, either the symbol is defined in an object file or the linker extracts the corresponding symbol definition from a library.

This option is, for example, useful if the startup code is part of a library. Because your own application does not refer to the startup code, you can force the startup code to be extracted by specifying the symbol `_START` as an unresolved external.

Example

Consider the following invocation:

```
larc mylib.a
```

Nothing is linked and no output file will be produced, because there are no unresolved symbols when the linker searches through `mylib.a`.

```
larc --extern=_START mylib.a
```

In this case the linker searches for the symbol `_START` in the library and (if found) extracts the object that contains `_START`, the startup code. If this module contains new unresolved symbols, the linker looks again in `mylib.a`. This process repeats until no new unresolved symbols are found.

Related information

Section 7.3, *Linking with Libraries*

Linker option: --first-library-first

Menu entry

-

Command line syntax

--first-library-first

Description

When the linker processes a library it searches for symbols that are referenced by the objects and libraries processed so far. If the library contains a definition for an unresolved reference the linker extracts the object that contains the definition from the library.

By default the linker processes object files and libraries in the order in which they appear on the command line. If you specify the option **--first-library-first** the linker always tries to take the symbol definition from the library that appears first on the command line before scanning subsequent libraries.

This is for example useful when you are working with a newer version of a library that partially overlaps the older version. Because they do not contain exactly the same functions, you have to link them both. However, when a function is present in both libraries, you may want the linker to extract the most recent function.

Example

Consider the following example:

```
larc --first-library-first a.a test.o b.a
```

If the file `test.o` calls a function which is both present in `a.a` and `b.a`, normally the function in `b.a` would be extracted. With this option the linker first tries to extract the symbol from the first library `a.a`.

Note that routines in `b.a` that call other routines that are present in both `a.a` and `b.a` are now also resolved from `a.a`.

Related information

Linker option **--no-rescan** (Rescan libraries to solve unresolved externals)

Linker option: --global-map-file

Menu entry

-

Command line syntax

--global-map-file=*file*[**:XML**]**,...**

Default: no global map file is generated

Description

With this option you tell the linker to generate a global linker map file that includes information about each of the tasks.

A global linker map file is a text or XML file that shows how the linker has mapped the sections and symbols from the various object files (`.o`) to the linked object file. A locate part shows the absolute position of each section. External symbols are listed per space with their absolute address, both sorted on symbol and sorted on address.

Related information

Linker option **--global-map-file-format** (Format global map file)

Linker option **--map-file** (Generate map file for a single task)

Section 13.2, *Linker Map File Format*

Linker option: --global-map-file-format

Menu entry

-

Command line syntax

`--global-map-file-format=`*flag*`,...`

You can set the following flags:

+/-callgraph	**c/C**	Include call graph information
+/-removed	**d/D**	Include information on removed sections
+/-files	**f/F**	Include processed files information
+/-invocation	**i/I**	Include information on invocation and tools
+/-link	**k/K**	Include link result information
+/-locate	**l/L**	Include locate result information
+/-memory	**m/M**	Include memory usage information
+/-nonalloc	**n/N**	Include information of non-alloc sections
+/-overlay	**o/O**	Include overlay information
+/-statics	**q/Q**	Include module local symbols information
+/-crossref	**r/R**	Include cross references information
+/-lsl	**s/S**	Include processor and memory information
+/-rules	**u/U**	Include locate rules

Use the following options for predefined sets of flags:

--global-map-file-format=0	Link information Alias for **--global-map-file-format=cDfikLMNoQrSU**
--global-map-file-format=1	Locate information Alias for **--global-map-file-format=CDfiKlMNoQRSU**
--global-map-file-format=2	Most information Alias for **--global-map-file-format=cdfiklmNoQrSu**

Default: `--global-map-file-format=2`

Description

With this option you specify which information you want to include in the global map file.

On the command line you must use this option in combination with the option **--global-map-file**.

Related information

Linker option **--global-map-file** (Generate global map file)

Section 13.2, *Linker Map File Format*

Linker option: --global-type-checking

Menu entry

-

Command line syntax

`--global-type-checking`

Description

Use this option when you want the linker to check the types of variable and function references against their definitions, using DWARF 3 debug information.

Related information

-

Linker option: --help (-?)

Menu entry

-

Command line syntax

--help[**=***item*]

-?

You can specify the following arguments:

options   Show extended option descriptions

Description

Displays an overview of all command line options. When you specify the argument **options** you can list detailed option descriptions.

Example

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

```
larc -?
larc --help
larc
```

To see a detailed description of the available options, enter:

```
larc --help=options
```

Related information

-

Linker option: --hex-format

Menu entry

1. Select **Linker » Output Format**.

2. Enable the option **Generate Intel Hex format file** and enable or disable the option **Emit start address record**.

3. Enable the option **Generate C array file** and enable or disable the option **Emit list of exported symbols**.

Command line syntax

```
--hex-format=flag,...
```

You can set the following flag:

+/-start-address	**s/S**	Emit start address record
+/-c-array-symbols	**y/Y**	Emit list of exported symbols

Default: **--hex-format=s**

Description

With this option you can specify to emit or omit the start address record from the hex file or you can emit a list of exported symbols for C array files.

Related information

Chapter 14, *Object File Formats*

Linker option **--output** (Output file)

Linker option **--chip-output** (Generate an output file for each chip)

Linker option: --hex-record-size

Menu entry

1. Select **Linker » Miscellaneous**.

2. Add the option **--hex-record-size** to the **Additional options** field.

Command line syntax

`--hex-record-size=`*size*

Default: 32

Description

With this option you can set the size (width) of the Intel Hex data records.

Related information

Linker option **--output** (Output file)

Linker option: --import-object

Menu entry

1. Select **Linker » Data Objects**.

   The Data objects box shows the list of object files that are imported.

2. To add a data object, click on the **Add** button in the **Data objects** box.

3. Type or select a binary file (including its path).

Use the **Edit** and **Delete** button to change a filename or to remove a data object from the list.

Command line syntax

--import-object=*file,...*

Description

With this option the linker imports a binary *file* containing raw data and places it in a section. The section name is derived from the filename, in which dots are replaced by an underscore. So, when importing a file called my.jpg, a section with the name my_jpg is created. In your application you can refer to the created section by using linker labels.

Related information

Section 7.5, *Importing Binary Files*

Linker option: --include-directory (-I)

Menu entry

-

Command line syntax

--include-directory=*path*,...

-I*path*,...

Description

With this option you can specify the path where your LSL include files are located. A relative path will be relative to the current directory.

The order in which the linker searches for LSL include files is:

1. The pathname in the LSL file and the directory where the LSL file is located (only for #include files that are enclosed in "")

2. The path that is specified with this option.

3. The default directory `$(PRODDIR)\include.lsl`.

Example

Suppose that your linker script file `mylsl.lsl` contains the following line:

```
#include "myinc.inc"
```

You can call the linker as follows:

```
larc --include-directory=c:\proj\include --lsl-file=mylsl.lsl test.o
```

First the linker looks for the file `myinc.inc` in the directory where `mylsl.lsl` is located. If it does not find the file, it looks in the directory `c:\proj\include` (this option). Finally it looks in the directory `$(PRODDIR)\include.lsl`.

Related information

Linker option **--lsl-file** (Specify linker script file)

Linker option: --incremental (-r)

Menu entry

-

Command line syntax

`--incremental`

`-r`

Description

Normally the linker links and locates the specified object files. With this option you tell the linker only to link the specified files. The linker creates a linker output file `.out`. You then can link this file again with other object files until you have reached the final linker output file that is ready for locating.

In the last pass, you call the linker without this option with the final linker output file `.out`. The linker will now locate the file.

Example

In this example, the files `test1.o`, `test2.o` and `test3.o` are incrementally linked:

1. `larc --incremental test1.o test2.o --output=test.out`

   test1.o and test2.o are linked

2. `larc --incremental test3.o test.out`

   test3.o and test.out are linked, task1.out is created

3. `larc task1.out`

   task1.out is located

Related information

Section 7.4, *Incremental Linking*

Linker option: --keep-output-files (-k)

Menu entry

Eclipse *always* removes the output files when errors occurred.

Command line syntax

```
--keep-output-files
```

```
-k
```

Description

If an error occurs during linking, the resulting output file may be incomplete or incorrect. With this option you keep the generated output files when an error occurs.

By default the linker removes the generated output file when an error occurs. This is useful when you use the make utility. If the erroneous files are not removed, the make utility may process corrupt files on a subsequent invocation.

Use this option when you still want to use the generated file. For example when you know that a particular error does not result in a corrupt object file, or when you want to inspect the output file, or send it to TASKING support.

Related information

Linker option **--warnings-as-errors** (Treat warnings as errors)

Linker option: --library (-l)

Menu entry

1. Select **Linker » Libraries**.

   The Libraries box shows the list of libraries that are linked with the project.

2. To add a library, click on the **Add** button in the **Libraries** box.

3. Type or select a library (including its path).

4. Optionally, disable the option **Link default libraries**.

Use the **Edit** and **Delete** button to change a library name or to remove a library from the list.

Command line syntax

--library=*name*

-l*name*

Description

With this option you tell the linker to use system library lib*name*.a, where *name* is a string. The linker first searches for system libraries in any directories specified with **--library-directory**, then in the directories specified with the environment variables LIBPPU_TC45X / LIBPPU_TC49X / LIBPPU_TC49XN / LIBPPU_TC4DX, unless you used the option **--ignore-default-library-path**.

Example

To search in the system library libc.a (C library):

larc test.o mylib.a --library=c

The linker links the file test.o and first looks in library mylib.a (in the current directory only), then in the system library libc.a to resolve unresolved symbols.

Related information

Linker option **--library-directory** (Additional search path for system libraries)

Section 7.3, *Linking with Libraries*

Linker option: --library-directory (-L) / --ignore-default-library-path

Menu entry

1. Select **Linker » Libraries**.

    The Library search path box shows the directories that are added to the search path for library files.

2. To define a new directory for the search path, click on the **Add** button in the **Library search path** box.

3. Type or select a path.

Use the **Edit** and **Delete** button to change a path or to remove a path from the list.

Command line syntax

```
--library-directory=path,...
-Lpath,...

--ignore-default-library-path
-L
```

Description

With this option you can specify the path(s) where your system libraries, specified with the option **--library** (**-l**), are located. If you want to specify multiple paths, use the option **--library-directory** for each separate path.

The default path is `$(PRODDIR)\lib`.

If you specify only **-L** (without a pathname) or the long option **--ignore-default-library-path**, the linker will not search the default path and also not in the paths specified in the environment variables `LIBPPU_TC45X / LIBPPU_TC49X / LIBPPU_TC49XN / LIBPPU_TC4DX`. So, the linker ignores steps 2 and 3 as listed below.

The priority order in which the linker searches for system libraries specified with the option **--library** (**-l**) is:

1. The path that is specified with the option **--library-directory**.

2. The path that is specified in the environment variables `LIBPPU_TC45X / LIBPPU_TC49X / LIBPPU_TC49XN / LIBPPU_TC4DX`.

3. The default directory `$(PRODDIR)\lib`.

Example

Suppose you call the linker as follows:

```
larc test.o --library-directory=c:\mylibs --library=c
```

First the linker looks in the directory `c:\mylibs` for library `libc.a` (this option). If it does not find the requested libraries, it looks in the directory that is set with the environment variables `LIBPPU_TC45X` / `LIBPPU_TC49X` / `LIBPPU_TC49XN` / `LIBPPU_TC4DX`. Then the linker looks in the default directory `$(PRODDIR)\lib` for libraries.

Related information

Linker option **--library** (Link system library)

Section 7.3.1, *How the Linker Searches Libraries*

Linker option: --link-only

Menu entry

-

Command line syntax

`--link-only`

Description

With this option you suppress the locating phase. The linker stops after linking and informs you about unresolved references.

Related information

Control program option **--create=relocatable** (**-cl**) (Stop after linking)

Linker option: --lsl-check

Menu entry

-

Command line syntax

`--lsl-check`

Description

With this option the linker just checks the syntax of the LSL file(s) and exits. No linking or locating is performed. Use the option **--lsl-file** to specify the name of the Linker Script File you want to test.

Related information

Linker option **--lsl-file** (Linker script file)

Linker option **--lsl-dump** (Dump LSL info)

Section 7.8, *Controlling the Linker with a Script*

Linker option: --lsl-dump

Menu entry

-

Command line syntax

`--lsl-dump``[=`*`file`*`]`

Description

With this option you tell the linker to dump the LSL part of the map file in a separate file, independent of the option **--map-file** (generate map file for a single task). If you do not specify a filename, the file `larc.ldf` is used.

Related information

Linker option **--map-file-format** (Map file formatting)

Linker option: --lsl-file (-d)

Menu entry

An LSL file can be generated when you create your TriCore project in Eclipse:

1. From the **File** menu, select **File » New » TASKING TriCore C/C++ Project**.

   The New C/C++ Project wizard appears.

2. Fill in the project settings in each dialog and click **Next >** until the **TriCore Project Settings** appear.

3. Enable the option **Add linker script file to the project** and click **Finish**.

   Eclipse creates your project and the file project.lsl in the project directory.

The LSL file can be specified in the Properties dialog:

1. Select **Linker » Script File**.

2. Specify a LSL file in the **Linker script file (.lsl)** field.

Command line syntax

--lsl-file=`file`

-d`file`

Description

A linker script file contains vital information about the core for the locating phase of the linker. A linker script file is coded in LSL and contains the following types of information:

- the architecture definition describes the core's hardware architecture.

- the memory definition describes the physical memory available in the system.

- the section layout definition describes how to locate sections in memory.

With this option you specify a linker script file to the linker. If you do not specify this option, the linker uses a default script file. You can specify the existing file *target*.lsl or the name of a manually written linker script file. You can use this option multiple times. The linker processes the LSL files in the order in which they appear on the command line.

Related information

Linker option **--lsl-check** (Check LSL file(s) and exit)

Section 7.8, *Controlling the Linker with a Script*

Linker option: --make-target

Menu entry

-

Command line syntax

--make-target=_name_

Description

With this option you can overrule the default target name in the make dependencies generated by the option **--dep-file**. The default target name is the basename of the input file, with extension `.elf`.

Example

```
larc --make-target=mytarget.elf test.o
```

The linker generates dependency lines with the default target name `mytarget.elf` instead of `test.elf`.

Related information

Linker option **--dep-file** (Generate dependencies in a file)

Linker option: --map-file (-M)

Menu entry

1. Select **Linker » Map File**.

2. Enable the option **Generate XML map file format (.mapxml) for map file viewer**.

3. (Optional) Enable the option **Generate map file**.

4. Enable or disable the types of information to be included.

Command line syntax

--map-file[**=***file*][**:XML**]

-M[*file*][**:XML**]

Default (Eclipse): XML map file is generated

Default (linker): no map file is generated

Description

With this option you tell the linker to generate a linker map file. If you do not specify a filename and you specified the option **--output**, the linker uses the same basename as the output file with the extension .map. If you did not specify the option **--output**, the linker uses the file task1.map. Eclipse names the .map file after the project.

A linker map file is a text file that shows how the linker has mapped the sections and symbols from the various object files (.o) to the linked object file. A locate part shows the absolute position of each section. External symbols are listed per space with their absolute address, both sorted on symbol and sorted on address.

Related information

Linker option **--map-file-format** (Format map file)

Section 13.2, *Linker Map File Format*

Linker option: --map-file-format (-m)

Menu entry

1. Select **Linker » Map File**.

2. Enable the option **Generate XML map file format (.mapxml) for map file viewer**.

3. (Optional) Enable the option **Generate map file**.

4. Enable or disable the types of information to be included.

Command line syntax

--map-file-format=$flag$**,...**

-m$flags$

You can set the following flags:

+/-callgraph	**c/C**	Include call graph information
+/-removed	**d/D**	Include information on removed sections
+/-files	**f/F**	Include processed files information
+/-invocation	**i/I**	Include information on invocation and tools
+/-link	**k/K**	Include link result information
+/-locate	**l/L**	Include locate result information
+/-memory	**m/M**	Include memory usage information
+/-nonalloc	**n/N**	Include information of non-alloc sections
+/-overlay	**o/O**	Include overlay information
+/-statics	**q/Q**	Include module local symbols information
+/-crossref	**r/R**	Include cross references information
+/-lsl	**s/S**	Include processor and memory information
+/-rules	**u/U**	Include locate rules

Use the following options for predefined sets of flags:

--map-file-format=0	**-m0**	Link information Alias for **-mcDfikLMNoQrSU**
--map-file-format=1	**-m1**	Locate information Alias for **-mCDfiKlMNoQRSU**
--map-file-format=2	**-m2**	Most information Alias for **-mcdfiklmNoQrSu**

Default: **--map-file-format=2**

Description

With this option you specify which information you want to include in the map file.

On the command line you must use this option in combination with the option **--map-file** (**-M**).

Related information

Linker option **--map-file** (Generate map file for a single task)

Section 13.2, *Linker Map File Format*

Linker option: --misra-c-report

Menu entry

-

Command line syntax

--misra-c-report[**=***file*]

Description

With this option you tell the linker to create a MISRA C Quality Assurance report. This report lists the various modules in the project with the respective MISRA C settings at the time of compilation. If you do not specify a filename, the file *basename*.mcr is used.

Related information

C compiler option **--misrac** (MISRA C checking)

Linker option: --munch

Menu entry

-

Command line syntax

`--munch`

Description

With this option you tell the linker to activate the muncher in the pre-locate phase.

The muncher phase is a special part of the linker that creates sections containing a list of pointers to the initialization and termination routines. The list of pointers is consulted at run-time by startup code invoked from `main`, and the routines on the list are invoked at the appropriate times.

Related information

-

Linker option: --non-romable

Menu entry

-

Command line syntax

`--non-romable`

Description

With this option you tell the linker that the application must not be located in ROM. The linker will locate all ROM sections, including a copy table if present, in RAM. When the application is started, the data sections are re-initialized and the BSS sections are cleared as usual.

This option is, for example, useful when you want to test the application in RAM before you put the final application in ROM. This saves you the time of flashing the application in ROM over and over again.

If you want to locate your application in RAM only, without using ROM/flash resources of the chip, for example when you run the debugger in RAM only, also specify the options **--no-rom-copy** and **--user-provided-initialization-code**.

Related information

Linker option **--no-rom-copy** (Do not generate ROM copy)

Linker option **--user-provided-initialization-code** (Own initialization code, no standard copy table)

Linker option: --no-default-output

Menu entry

-

Command line syntax

`--no-default-output`

Description

By default the linker generates an absolute object file and a memory definition file for each task. With this option you specify to the linker not to generate these files, unless explicitly specified.

Example

Invocation to create an Intel Hex output for each chip only:

```
larc -cmyprog:IHEX --no-default-output test.o
```

This generates the file myprog_*memname*.hex. Without **--no-default-output** also the files task1.elf and task1.mdf are generated.

Related information

Linker option **--chip-output** (Generate an output file for each chip)

Control program option **--no-map-file** (Do not generate map file)

Linker option: --no-rescan

Menu entry

1. Select **Linker » Libraries**.

2. Disable the option **Rescan libraries to solve unresolved externals**.

Command line syntax

```
--no-rescan
```

Description

When the linker processes a library it searches for symbol definitions that are referenced by the objects and libraries processed so far. If the library contains a definition for an unresolved reference the linker extracts the object that contains the definition from the library. The linker processes object files and libraries in the order in which they appear on the command line.

When all objects and libraries are processed the linker checks if there are unresolved symbols left. If so, the default behavior of the linker is to rescan all libraries in the order given at the command line. The linker stops rescanning the libraries when all symbols are resolved, or when the linker could not resolve any symbol(s) during the rescan of all libraries. Notice that resolving one symbol may introduce new unresolved symbols.

With this option, you tell the linker to scan the object files and libraries only once. When the linker has not resolved all symbols after the first scan, it reports which symbols are still unresolved. This option is useful if you are building your own libraries. The libraries are most efficiently organized if the linker needs only one pass to resolve all symbols.

Related information

Linker option **--first-library-first** (Scan libraries in given order)

Linker option: --no-rom-copy (-N)

Menu entry

-

Command line syntax

`--no-rom-copy`

`-N`

Description

With this option the linker will not generate a ROM copy for data sections. A copy table is generated and contains entries to clear BSS sections. However, no entries to copy data sections from ROM to RAM are placed in the copy table.

The data sections are initialized when the application is downloaded. The data sections are not re-initialized when the application is restarted.

Related information

Linker option **--non-romable** (Application is not romable)

Linker option **--user-provided-initialization-code** (Own initialization code, no standard copy table)

Linker option: --no-warnings (-w)

Menu entry

1. Select **Linker » Diagnostics**.

    The Suppress warnings box shows the warnings that are currently suppressed.

2. To suppress a warning, click on the **Add** button in the **Suppress warnings** box.

3. Enter the numbers, separated by commas, of the warnings you want to suppress (for example `135,136`). Or you can use the **Add** button multiple times.

4. To suppress all warnings, enable the option **Suppress all warnings**.

Use the **Edit** and **Delete** button to change a warning number or to remove a number from the list.

Command line syntax

`--no-warnings`[`=`*number*`,...`]

`-w`[*number*`,...`]

Description

With this option you can suppresses all warning messages or specific warning messages.

On the command line this option works as follows:

*   If you do not specify this option, all warnings are reported.

*   If you specify this option but without numbers, all warnings are suppressed.

*   If you specify this option with a number, only the specified warning is suppressed. You can specify the option **--no-warnings=***number* multiple times.

Example

To suppress warnings 135 and 136, enter:

```
larc --no-warnings=135,136 test.o
```

Related information

Linker option **--warnings-as-errors** (Treat warnings as errors)

Linker option: --optimize (-O)

Menu entry

1. Select **Linker » Optimization**.

2. Select one or more of the following options:

   - Delete unreferenced sections

   - Use a 'first-fit decreasing' algorithm

   - Compress copy table

   - Delete duplicate code

   - Delete duplicate data

Command line syntax

--optimize=*flag*,...

-O*flags*

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

Use the following options for predefined sets of flags:

--optimize=0	**-O0**	No optimization Alias for **-OCLTXY**
--optimize=1	**-O1**	Default optimization Alias for **-OcLtxy**
--optimize=2	**-O2**	All optimizations Alias for **-Ocltxy**

Default: **--optimize=1**

Description

With this option you can control the level of optimization.

Note that when you use the flag **+copytable-compression**, sections affected by the copy table are located as if they were in a clustered LSL group, if they do not have a locate restriction yet.

Related information

For details about each optimization see Section 7.7, *Linker Optimizations*.

Define the mutual order of sections in an LSL group in Section 15.8.2, *Creating and Locating Groups of Sections*.

Linker option: --option-file (-f)

Menu entry

1. Select **Linker » Miscellaneous**.

2. Add the option **--option-file** to the **Additional options** field.

    Be aware that the options in the option file are added to the linker options you have set in the other pages. Only in extraordinary cases you may want to use them in combination.

Command line syntax

`--option-file=`*file*`,...`

`-f` *file*`,...`

Description

This option is primarily intended for command line use. Instead of typing all options on the command line, you can create an option file which contains all options and flags you want to specify. With this option you specify the option file to the linker.

Use an option file when the command line would exceed the limits of the operating system, or just to store options and save typing.

Option files can also be generated on the fly, for example by the make utility. You can specify the option **--option-file** multiple times.

Format of an option file

*   Multiple arguments on one line in the option file are allowed.

*   To include whitespace in an argument, surround the argument with single or double quotes.

*   If you want to use single quotes as part of the argument, surround the argument by double quotes and vise versa:

    ```
    "This has a single quote ' embedded"

    'This has a double quote " embedded'

    'This has a double quote " and a single quote '"' embedded"
    ```

*   When a text line reaches its length limit, use a **** to continue the line. Whitespace between quotes is preserved.

    ```
    "This is a continuation \
    line"

            -> "This is a continuation line"
    ```

• It is possible to nest command line files up to 25 levels.

Example

Suppose the file `myoptions` contains the following lines:

```
--map-file=my.map             (generate a map file)
test.o                        (input file)
--library-directory=c:\mylibs  (additional search path for system libraries)
```

Specify the option file to the linker:

```
larc --option-file=myoptions
```

This is equivalent to the following command line:

```
larc --map-file=my.map test.o --library-directory=c:\mylibs
```

Related information

-

Linker option: --output (-o)

Menu entry

1. Select **Linker » Output Format**.

2. Enable one or more output formats.

   For some output formats you can specify a number of suboptions.

   Eclipse always uses the project name as the basename for the output file.

Command line syntax

--output=[*filename*][**:**derivation*format*[**:**addr_size]][**,**space_name]]...

-o[*filename*][**:**format[**:**addr_size]][**,**space_name]]...

You can specify the following formats:

ELF	ELF/DWARF
IHEX	Intel Hex
SREC	Motorola S-records

Description

By default, the linker generates an output file in ELF/DWARF format, with the name `task1.elf`.

With this option you can specify an alternative *filename*, and an alternative output *format*. The default output format is the format of the first input file.

You can use the **--output** option multiple times. This is useful to generate multiple output formats. With the first occurrence of the **--output** option you specify the basename (the filename without extension), which is used for subsequent **--output** options with no filename specified. If you do not specify a filename, or you do not specify the **--output** option at all, the linker uses the default basename `task`*n*.

IHEX and SREC formats

If you specify the Intel Hex format or the Motorola S-records format, you can use the argument *addr_size* to specify the size of addresses in bytes (record length). For Intel Hex you can use the values: 1, 2, and 4 (default). For Motorola S-records you can specify: 2 (S1 records), 3 (S2 records, default) or 4 bytes (S3 records). Note that if you make the *addr_size* too small, the linker might give a fatal object writer error indicating an address overflow.

With the argument *space_name* you can specify the name of the address space. The name of the output file will be filename with the extension `.hex` or `.sre` and contains the code and data allocated in the specified space. If they exist, any other address spaces are also emitted whereas their output files are named *filename_spacename* with the extension `.hex` or `.sre`.

If you do not specify *space_name*, or you specify a non-existing space, the default address space is filled in.

Use option **--chip-output** (**-c**) to create Intel Hex or Motorola S-record output files for each chip defined in the LSL file (suitable for loading into a PROM-programmer).

Example

To create the output file `myprog.hex` of the address space named `linear`, enter:

```
larc test.o --output=myprog.hex:IHEX:2,linear
```

If they exist, any other address spaces are emitted as well and are named `myprog_`*spacename*`.hex`.

Related information

Linker option **--chip-output** (Generate an output file for each chip)

Linker option **--hex-format** (Specify Hex file or C array format settings)

Linker option: --print-mangled-symbols (-P)

Menu entry

-

Command line syntax

```
--print-mangled-symbols
```

```
-P
```

Description

C++ compilers generate unreadable symbol names. These symbols cannot easily be related to your C++ source file anymore. Therefore the linker will by default decode these symbols conform the IA64 ABI when printed to stdout. With this option you can override this default setting and print the mangled names instead.

Related information

-

Linker option: --strip-debug (-S)

Menu entry

1. Select **Linker » Miscellaneous**.

2. Enable the option **Strip symbolic debug information**.

Command line syntax

`--strip-debug`

`-s`

Description

With this option you specify not to include symbolic debug information in the resulting output file.

Related information

-

Linker option: --user-provided-initialization-code (-i)

Menu entry

1. Select **Linker » Miscellaneous**.

2. Enable the option **Do not use standard copy table for initialization**.

Command line syntax

```
--user-provided-initialization-code
```

```
-i
```

Description

It is possible to use your own initialization code, for example, to save ROM space. With this option you tell the linker *not* to generate a copy table for initialize/clear sections. Use linker labels in your source code to access the positions of the sections when located.

If the linker detects references to the TASKING initialization code, an error is emitted: it is either the TASKING initialization routine or your own, not both.

Note that the options **--no-rom-copy** and **--non-romable**, may vary independently. The 'copytable-compression' optimization (**--optimize=t**) is automatically disabled when you enable this option.

Related information

Linker option **--no-rom-copy** (Do not generate ROM copy)

Linker option **--non-romable** (Application is not romable)

Linker option **--optimize** (Specify optimization)

Linker option: --verbose (-v)

Menu entry

-

Command line syntax

`--verbose`

`-v`

Description

With this option you put the linker in *verbose* mode. The linker prints the link phases while it processes the files. The linker prints one entry for each action it executes for a task. When you use this option twice (`-vv`) you put the linker in *extra verbose* mode. In this mode the linker also prints the filenames and it shows which objects are extracted from libraries and it shows verbose information that would normally be hidden when you use the normal verbose mode or when you run without verbose. With this option you can monitor the current status of the linker.

Related information

-

Linker option: --version (-V)

Menu entry

-

Command line syntax

`--version`

`-V`

Description

Display version information. The linker ignores all other options or input files.

Related information

-

Linker option: --warnings-as-errors

Menu entry

1. Select **Global Options**.

2. Enable the option **Treat warnings as errors**.

Command line syntax

`--warnings-as-errors`[`=`*number,...*]

Description

When the linker detects an error or warning, it tries to continue the link process and reports other errors and warnings. When you use this option without arguments, you tell the linker to treat all warnings as errors. This means that the exit status of the linker will be non-zero after the detection of one or more linker warnings. As a consequence, the linker will not produce any output files.

You can also limit this option to specific warnings by specifying a comma-separated list of warning numbers.

Related information

Linker option **--no-warnings** (Suppress some or all warnings)

Linker option: --whole-archive

Menu entry

1. Select **Linker » Miscellaneous**.

2. Add the option **--whole-archive** to the **Additional options** field.

Command line syntax

```
--whole-archive=file
```

Description

This option tells the linker to directly load all object modules in a library, as if they were placed on the command line. This is different from libraries specified as input files or with the **-l** option, which are only used to resolve references in object files that were loaded earlier.

Example

Suppose the library `myarchive.a` contains the objects `my1.o`, `my2.o` and `my3.o`. Specifying

```
larc --whole-archive=myarchive.a
```

is the same as specifying

```
larc my1.o my2.o my3.o
```

Related information

Linker option **--library** (Link system library)

10.6. Control Program Options

The control program **ccarc** facilitates the invocation of the various components of the TASKING toolset for ARC/PPU from a single command line.

Options in Eclipse versus options on the command line

Eclipse invokes the compiler, assembler and linker via the control program. Therefore, it uses the syntax of the control program to pass options and files to the tools. The control program processes command line options either by itself, or, when the option is unknown to the control program, it looks whether it can pass the option to one of the other tools. However, for directly passing an option to the C++ compiler, C compiler, assembler or linker, it is recommended to use the control program options **--pass-c++**, **--pass-c**, **--pass-assembler**, **--pass-linker**.

See the previous sections for details on the options of the tools.

Short and long option names

Options can have both short and long names. Short option names always begin with a single minus (**-**) character, long option names always begin with two minus (**--**) characters. You can abbreviate long option names as long as it forms a unique name. You can mix short and long option names on the command line.

Options can have flags or suboptions. To switch a flag 'on', use a lowercase letter or a **+**_longflag_. To switch a flag off, use an uppercase letter or a **-**_longflag_. Separate _longflags_ with commas. The following two invocations are equivalent:

```
ccarc -Wc-Oac test.c
ccarc --pass-c=--optimize=+coalesce,+cse test.c
```

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

Control program option: --address-size

Menu entry

1. Select **Linker » Output Format**.

2. Enable the option **Generate Intel Hex format file** and/or **Generate S-records file**.

3. Specify the **Size of addresses**.

   Eclipse always uses the project name as the basename for the output file.

Command line syntax

--address-size=*addr_size*

Description

If you specify IHEX or SREC with the control option **--format**, you can additionally specify the record length to be emitted in the output files.

With this option you can specify the size of the addresses in bytes (record length). For Intel Hex you can use the values **1**, **2** or **4** bytes (default). For Motorola-S you can specify: **2** (S1 records), **3** (S2 records) or **4** bytes (S3 records, default).

If you do not specify *addr_size*, the default address size is generated.

Example

To create the SREC file `test.sre` with S1 records, type:

```
ccarc --format=SREC --address-size=2 test.c
```

Related information

Control program option **--format** (Set linker output format)

Control program option **--output** (Output file)

Control program option: --c++

Menu entry

1. Select **C/C++ Compiler » Language**.

2. From the **Comply to C++ standard** list, select the ISO C++ standard.

Command line syntax

`--c++=`*year*

You can specify one of the following years: **11** or **14**.

Default: `--c++=14`

Description

With this option you select the ISO C++ standard. C++11 refers to ISO/IEC 14882:2011, C++14 refers to ISO/IEC 14882:2014. Exceptions are implicitly enabled.

Related information

Control program option **--no-exceptions** (Disable exceptions)

Control program option: --case-insensitive

Menu entry

1.   Select **Assembler » Symbols**.

2.   Enable the option **Case insensitive identifiers**.

Command line syntax

`--case-insensitive`

Default: case sensitive

Description

With this option you tell the assembler not to distinguish between uppercase and lowercase characters. By default the assembler considers uppercase and lowercase characters as different characters.

Assembly source files that are generated by the compiler must always be assembled case sensitive. When you are writing your own assembly code, you may want to specify the case insensitive mode.

The control program passes this option to both the assembler and the linker.

Example

When assembling case insensitive, the label `LabelName` is the same label as `labelname`.

`ccarc --case-insensitive test.src`

Related information

Assembler option **--case-insensitive**

Assembler control **$CASE**

Control program option: --check

Menu entry

-

Command line syntax

`--check`

Description

With this option you can check the source code for syntax errors, without generating code. This saves time in developing your application because the code will not actually be compiled.

The compiler/assembler reports any warnings and/or errors.

This option is available on the command line only.

Related information

C compiler option **--check** (Check syntax)

Assembler option **--check** (Check syntax)

Control program option: --control-flow-info

Menu entry

1.   Select **C/C++ Compiler » Debugging**.

2.   Enable the option **Generate control flow information**.

Command line syntax

`--control-flow-info`

Description

With this option the compiler adds control flow information to the output file. The compiler generates a
`.debug_control_flow` section which describes the basic blocks and their relations. This information
can be used for code coverage analysis on optimized code.

Example

ccarc --control-flow-info test.c

Related information

Section 8.4.2, *HLL Dump Output Format*

Control program option **--debug-info** (Debug information)

Control program option: --core

Menu entry

1. Expand **C/C++ Build** and select **Processor**.

2. From the **Processor selection** list, make a selection.

Command line syntax

--core=_core_

You can specify the following core architectures:

ppu_tc45x	PPU core architecture of TC45x
ppu_tc49x	PPU core architecture of TC49x
ppu_tc49xn	PPU core architecture of TC49xN
ppu_tc4dx	PPU core architecture of TC4Dx

Default: ppu_tc49x

Description

With this option you specify the PPU core architecture for which you create your application. The core architecture determines which instructions are valid and which are not. If you use Eclipse or the control program, the ARC/PPU toolset derives the core from the processor you selected. The control program passes this option to both the C compiler and the assembler.

Example

After

```
ccarc --core=ppu_tc49x -v -t test.c
```

the control program will call the tools as follows:

```
carc  -D__CPU__=tc49x -D__CPU_TC49X__ --core=ppu_tc49x -o test.src test.c
asarc -D__CPU__=tc49x -D__CPU_TC49X__ --core=ppu_tc49x -o test.o test.src
larc  -o test.elf -dtc49x.lsl --core=ppu -D__CPU__=tc49x
      --map-file test.o -lc_fpu -lrt -Linstall-dir\carc\lib\tc49x
```

Related information

Control program option **--cpu** (Select processor)

Control program option **--cpu-list** (Show list of processors)

C compiler option **--core** (Select core architecture)

Assembler option **--core** (Select core architecture)

561

Control program option: --cpu (-C)

Menu entry

1. Expand **C/C++ Build** and select **Processor**.

2. From the **Processor selection** list, make a selection.

Command line syntax

--cpu=*id* | *name* | *cpu*

-C*id* | *name* | *cpu*

Description

With this option you define the target processor for which you create your application. You can specify a full processor *name*, like TC49x, or a base *CPU* name, like tc49x or its unique *id*, like tc49x.

The standard list of supported processors is defined in the file `processors.xml`. This file defines for each processor its full name (for example, TC49x), its ID, the base CPU name (for example, tc49x) and the core architecture settings (for example, ppu_tc49x). To show a list of all supported processors you can use option **--cpu-list**.

The control program reads the file `processors.xml`. The lookup sequence for names specified to this option is as follows:

1. match with the `'id'` attribute in `processors.xml` (case insensitive, for example `tc49x`)

2. if none matched, match with the `'name'` attribute in `processors.xml` (case insensitive, for example `TC49x`)

3. if still none matched, match any of the base CPU names (the `'cpu'` attribute in `processors.xml`, for example `tc49x`). If multiple processors exist with the same base CPU, a warning will be issued and the first one is selected.

4. if still none matched, the control program issues a fatal error.

The preferred use of the option **--cpu**, is to specify an ID because that is always a unique name. For example, **--cpu=tc49x**. The control program will lookup this processor name in the file `processors.xml`. The control program passes the options to the underlaying tools. For example, **-D__CPU__=tc49x -D__CPU_TC49X__** to the C compiler and assembler, or **-dtc49x.lsl --core=ppu -D__CPU__=tc49x -D__PROC_TC49X__** to the linker.

Example

To generate the file `test.elf` for the TC49x processor, enter:

```
ccarc --cpu=tc49x test.c
```

Related information

Control program option **--cpu-list** (Show list of processors)

Control program option: --cpu-list

Menu entry

-

Command line syntax

--cpu-list[=*pattern*]

Description

With this option the control program shows a list of supported processors as defined in the file processors.xml. This can be useful when you want to select a processor name or id for the **--cpu** option.

The *pattern* works similar to the UNIX **grep** utility. You can use it to limit the output list.

Example

To show a list of all processors, enter:

```
ccarc --cpu-list
```

To show all processors of the ppu_tc49x core, enter:

```
ccarc --cpu-list=ppu_tc49x

--- ~/carc/etc/processors.xml ---
    id          name        CPU         core
    tc49x       TC49x       tc49x       ppu_tc49x
```

Related information

Control program option **--cpu** (Select processor)

Control program option: --create (-c)

Menu entry

-

Command line syntax

--create[**=***stage*]

-c[*stage*]

You can specify the following stages:

intermediate-c	**c**	Stop after C++ files are compiled to intermediate C files (`.ic`)
relocatable	**l**	Stop after the files are linked to a linker object file (`.out`)
mil	**m**	Stop after C++ files or C files are compiled to MIL (`.mil`)
object	**o**	Stop after the files are assembled to objects (`.o`)
assembly	**s**	Stop after C++ files or C files are compiled to assembly (`.src`)

Default (without flags): **--create=object**

Description

Normally the control program generates an absolute object file of the specified output format from the file you supplied as input. With this option you tell the control program to stop after a certain number of phases.

Example

To generate the object file `test.o`:

```
ccarc --create test.c
```

The control program stops after the file is assembled. It does not link nor locate the generated output.

Related information

Linker option **--link-only** (Link only, no locating)

Control program option: --debug-info (-g)

Menu entry

1. Select **C/C++ Compiler » Debugging**.

2. To **generate symbolic debug information**, select **Default**, **Small set** or **Full**.
   To disable the generation of debug information, select **None**.

Command line syntax

--debug-info[**=**_suboption_]

-g[_suboption_]

You can set the following suboptions:

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.

Default (Eclipse): **--debug-info=default**

Default (without suboptions): **--debug-info=default**

Description

With this option you specify to include debug information in the generated object file.

If you do not specify this option, no debug information is generated.

The control program passes the option **-g**_suboption_ to the C compiler and calls the assembler with **-gsl**.

Related information

C compiler option **--debug-info** (Generate symbolic debug information)

Assembler option **--debug-info** (Generate symbolic debug information)

Control program option: --define (-D)

Menu entry

1. Select **C/C++ Compiler » Preprocessing** and/or **Assembler » Preprocessing**.

    The Defined symbols box right-below shows the symbols that are currently defined.

2. To define a new symbol, click on the **Add** button in the **Defined symbols** box.

3. Type the symbol definition (for example, demo=1)

Use the **Edit** and **Delete** button to change a macro definition or to remove a macro from the list.

Command line syntax

--define=`macro_name`[**=**`macro_definition`]

-D`macro_name`[**=**`macro_definition`]

Description

With this option you can define a macro and specify it to the preprocessor. If you only specify a macro name (no macro definition), the macro expands as '1'.

You can specify as many macros as you like. Simply use the **Add** button to add new macro definitions.

On the command line, use the option **--define** (**-D**) multiple times. If the command line exceeds the limit of the operating system, you can define the macros in an option file which you then must specify to the compiler with the option **--option-file** (**-f**) *file*.

Defining macros with this option (instead of in the C source) is, for example, useful to compile conditional C source as shown in the example below.

The control program passes the option **--define** (**-D**) to the compiler and the assembler.

Make sure you do not use a reserved keyword as a macro name, as this can lead to unexpected results.

Example

Consider the following C program with conditional code to compile a demo program and a real program:

```
void main( void )
{
#if DEMO
    demo_func();   /* compile for the demo program */
#else
    real_func();   /* compile for the real program */
```

567

```
#endif
}
```

You can now use a macro definition to set the DEMO flag:

```
ccarc --define=DEMO test.c
ccarc --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.

```
ccarc --define="MAX(A,B)=((A) > (B) ? (A) : (B))" test.c
```

Related information

Control program option **--undefine** (Remove preprocessor macro)

Control program option **--option-file** (Specify an option file)

Control program option: --dep-file

Menu entry

-

Command line syntax

--dep-file[**=***file*]

Description

With this option you tell the compiler to generate dependency lines that can be used in a Makefile. In contrast to the option **--preprocess=+make**, the dependency information will be generated in addition to the normal output file.

By default, the information is written to a file with extension .d (one for every input file). When you specify a filename, all dependencies will be combined in the specified file.

Example

```
ccarc --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 **--dep-format** (Format of dependency info)

Control program option **--preprocess=+make** (Generate dependencies for make)

569

Control program option: --dep-format

Menu entry

-

Command line syntax

`--dep-format=`*format*

You can specify the following *format* arguments:

amk	amk format of dependency info
gnu	GNU format of dependency info

Default: **amk**

Description

With this option you can override the format of the make dependencies file from TASKING amk (the default format) to GNU make. The format determines how names with spaces or other special characters are escaped. In the amk format such names are double quoted. In the GNU format special characters are escaped using a backslash.

Example

```
ccarc --dep-file=test.dep --dep-format=gnu -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` in the GNU make format.

Related information

Control program option **--dep-file** (Generate dependencies in a file)

Control program option **--preprocess=+make** (Generate dependencies for make)

Control program option: --diag

Menu entry

1. From the **Window** menu, select **Show View » Other » TASKING » Problems**.

    The Problems view is added to the current perspective.

2. In the Problems view right-click on a message.

    A popup menu appears.

3. Select **Detailed Diagnostics Info**.

    A dialog box appears with additional information.

Command line syntax

--diag=[*format*:]{**all** | *nr*,...}

You can set the following output formats:

html	HTML output.
rtf	Rich Text Format.
text	ASCII text.

Default format: text

Description

With this option you can ask for an extended description of error messages in the format you choose. The output is directed to stdout (normally your screen) and in the format you specify. You can specify the following formats: **html**, **rtf** or **text** (default). To create a file with the descriptions, you must redirect the output.

With the suboption **all**, the descriptions of all error messages are given. If you want the description of one or more selected error messages, you can specify the error message numbers, separated by commas.

Example

To display an explanation of message number 103, enter:

```
ccarc --diag=103
```

This results in message 103 with explanation.

To write an explanation of all errors and warnings in HTML format to file ccerrors.html, use redirection and enter:

```
ccarc --diag=html:all > ccerrors.html
```

571

Related information

Section 4.8, *C Compiler Error Messages*

Control program option: --dry-run (-n)

Menu entry

-

Command line syntax

`--dry-run`

`-n`

Description

With this option you put the control program in verbose mode. The control program prints the invocations of the tools it would use to process the files without actually performing the steps.

Related information

Control program option **--verbose** (Verbose output)

Control program option: --error-file

Menu entry

-

Command line syntax

`--error-file`

Description

With this option the control program tells the compiler, assembler and linker to redirect diagnostic messages to a file.

Example

To write diagnostic messages to error files instead of stderr, enter:

```
ccarc --error-file test.c
```

Related information

Control Program option **--warnings-as-errors** (Treat warnings as errors)

Control program option: --error-limit

Menu entry

1.   Select **C/C++ Compiler » Diagnostics**.

2.   Enter a value in the **Maximum number of emitted errors** field.

Command line syntax

--error-limit=*number*

Default: 42

Description

With this option you limit the number of error messages in one invocation to the specified number. When the limit is exceeded, the control program aborts with fatal error message F105. Warnings and informational messages are not included in the count. When 0 (zero) or a negative number is specified, the control program emits all errors. Without this option the maximum number of errors is 42. The control program also passes this option to the C++ compiler, C compiler, assembler and linker.

Related information

Section 4.8, *C Compiler Error Messages*

Control program option: --force-c

Menu entry

-

Command line syntax

`--force-c`

Description

With this option you tell the control program to treat all `.cc` files as C files instead of C++ files. This means that the control program does not call the C++ compiler and forces the linker to link C libraries.

Related information

Control program option **--force-c++** (Force C++ compilation and linking)

Control program option: --force-c++

Menu entry

Eclipse always uses this option for a C++ project.

Command line syntax

`--force-c++`

Description

With this option you tell the control program to treat all `.c` files as C++ files instead of C files. This means that the control program calls the C++ compiler prior to the C compiler and forces the linker to link C++ libraries.

Related information

Control program option **--force-c** (Treat C++ files as C files)

Control program option: --force-munch

Menu entry

Eclipse always uses this option for a C++ project.

Command line syntax

`--force-munch`

Description

With this option you force the control program to activate the muncher in the pre-locate phase.

Related information

-

Control program option: --format

Menu entry

1. Select **Linker » Output Format**.

2. Enable the option **Generate Intel Hex format file** and/or **Generate S-records file**.

3. Optionally, specify the **Size of addresses**.

   Eclipse always uses the project name as the basename for the output file.

Command line syntax

--format=`format`

You can specify the following formats:

ELF	ELF/DWARF
IHEX	Intel Hex
SREC	Motorola S-records

Description

With this option you specify the output format for the resulting (absolute) object file. The default output format is ELF/DWARF, which can directly be used by the debugger.

If you choose IHEX or SREC, you can additionally specify the address size of the chosen format (option **--address-size**).

Example

To generate a Motorola S-record output file:

```
ccarc --format=SREC test1.c test2.c --output=test.sre
```

Related information

Control program option **--address-size** (Set address size for linker IHEX/SREC files)

Control program option **--output** (Output file)

Linker option **--chip-output** (Generate an output file for each chip)

Control program option: --fp-model

Menu entry

1. Select **C/C++ Compiler » Floating-Point**.

2. Make a selection from the **Floating-point model** list.

3. If you selected **Custom**, enable one or more individual options.

Command line syntax

--fp-model=*flags*

You can set the following flags:

+/-contract	**c/C**	allow expression contraction
+/-fastlib	**l/L**	allow less precise library functions
+/-nonan	**n/N**	allow optimizations to ignore NaN/Inf
+/-rewrite	**r/R**	allow expression rewriting
+/-negzero	**z/Z**	ignore sign of -0.0
	0	alias for **--fp-model=CLNRZ** (strict)
	1	alias for **--fp-model=cLNRZ** (precise)
	2	alias for **--fp-model=clnrz** (fast double precision)

Default: **--fp-model=clnrz**

Description

With this option you select the floating-point execution model.

With **--fp-model=+contract** you allow the compiler to contract multiple float operations into a single operation, with different rounding results. A possible example is fused multiply-add.

With **--fp-model=+fastlib** you allow the compiler to select faster but less accurate library functions for certain floating-point operations.

With **--fp-model=+nonan** you allow the compiler to ignore NaN or Inf input values. An example is to replace multiply by zero with zero.

With **--fp-model=+rewrite** you allow the compiler to rewrite expressions by reassociating. This might result in rounding differences and possibly different exceptions. An example is to rewrite (a*c)+(b*c) as (a+b)*c.

With **--fp-model=+negzero** you allow the compiler to ignore the sign of -0.0 values. An example is to replace (a-a) by zero.

Related information

Pragmas `STDC FP_CONTRACT`, `fp_negzero`, `fp_nonan` and `fp_rewrite` in Section 1.8, *Pragmas to Control the Compiler*.

Control program option: --help (-?)

Menu entry

-

Command line syntax

--help[**=***item*]

-?

You can specify the following argument:

options　　　**o**　　Show extended option descriptions

Description

Displays an overview of all command line options. When you specify the argument **options** you can list detailed option descriptions.

Example

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

```
ccarc -?
ccarc --help
ccarc
```

To see a detailed description of the available options, enter:

```
ccarc --help=options
```

Related information

-

Control program option: --include-directory (-I)

Menu entry

1. Select **C/C++ Compiler » Include Paths**.

    The Include paths box shows the directories that are added to the search path for include files.

2. To define a new directory for the search path, click on the **Add** button in the **Include paths** box.

3. Type or select a path.

Use the **Edit** and **Delete** button to change a path or to remove a path from the list.

Command line syntax

`--include-directory=`*path*`,...`

`-I`*path*`,...`

Description

With this option you can specify the path where your include files are located. A relative path will be relative to the current directory.

The control program passes this option to the compiler and the assembler.

Example

Suppose that the C source file `test.c` contains the following lines:

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

You can call the control program as follows:

```
ccarc --include-directory=myinclude test.c
```

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

The compiler now looks for the file `myinc.h` in the directory where `test.c` is located. If the file is not there the compiler searches in the directory `myinclude`. If it was still not found, the compiler searches in the environment variable and then in the default include directory.

Related information

C compiler option **--include-directory** (Add directory to include file search path)

C compiler option **--include-file** (Include file at the start of a compilation)

Control program option: --include-file (-H)

Menu entry

1.   Select **C/C++ Compiler » Preprocessing**.

     The Pre-include files box shows the files that are currently included before the compilation starts.

2.   To define a new file, click on the **Add** button in the **Pre-include files** box.

3.   Type the full path and file name or select a file.

Use the **Edit** and **Delete** button to change a file name or to remove a file from the list.

Command line syntax

```
--include-file=file,...
```

```
-Hfile,...
```

Description

With this option you include one or more extra files at the beginning of each C source file, before other includes. This is the same as specifying `#include "file"` at the beginning of *each* of your C sources.

Example

```
ccarc --include-file=stdio.h test1.c test2.c
```

The file `stdio.h` is included at the beginning of both `test1.c` and `test2.c`.

Related information

C compiler option **--include-directory** (Add directory to include file search path)

C compiler option **--include-file** (Include file at the start of a compilation)

Control program option: --instantiate

Menu entry

1.   Select **C/C++ Compiler » Miscellaneous**.

2.   Select an instantiation mode in the **Instantiation mode of external template entities** box.

Command line syntax

--instantiate=*mode*

You can specify the following modes:

>   **used**
>
>   **all**
>
>   **local**

Default: **--instantiate=used**

Description

Control instantiation of external template entities. External template entities are external (that is, non-inline and non-static) template functions and template static data members. The instantiation mode determines the template entities for which code should be generated based on the template definition. Normally, when a file is compiled, template entities are instantiated wherever they are used (the linker will discard duplicate definitions). The overall instantiation mode can, however, be changed with this option. You can specify the following modes:

used	Instantiate those template entities that were used in the compilation. This will include all static data members for which there are template definitions. This is the default.
all	Instantiate all template entities declared or referenced in the compilation unit. For each fully instantiated template class, all of its member functions and static data members will be instantiated whether or not they were used. Non-member template functions will be instantiated even if the only reference was a declaration.
local	Similar to **--instantiate=used** except that the functions are given internal linkage. This is intended to provide a very simple mechanism for those getting started with templates. The compiler will instantiate the functions that are used in each compilation unit as local functions, and the program will link and run correctly (barring problems due to multiple copies of local static variables). However, one may end up with many copies of the instantiated functions, so this is not suitable for production use.

You cannot use **--instantiate=local** in conjunction with automatic template instantiation.

Related information

Section 2.4, *Template Instantiation*

Control program option: --iso

Menu entry

1. Select **C/C++ Compiler » Language**.

2. From the **Comply to C standard** list, select **ISO C99**, **ISO C11**, **ISO C17**, or **ISO C90**.

Command line syntax

`--iso={90|99|11|17}`

Default: **`--iso=17`**

Description

With this option you select the ISO C standard. C90 is also referred to as the "ANSI C standard". C99 refers to the ISO/IEC 9899:1999 (E) standard. C11 refers to the ISO/IEC 9899:2011 (E) standard. C17 refers to the ISO/IEC 9899:2018 (E) standard. C17 is the default.

Independent of the chosen ISO C standard, the control program always links libraries with C11/C17 support.

Example

To select the ISO C99 standard on the command line:

```
ccarc --iso=99 test.c
```

Related information

C compiler option **--iso** (ISO C standard)

Control program option: --keep-output-files (-k)

Menu entry

Eclipse *always* removes generated output files when an error occurs.

Command line syntax

`--keep-output-files`

`-k`

Description

If an error occurs during the compilation, assembling or linking process, the resulting output file may be incomplete or incorrect. With this option you keep the generated output files when an error occurs.

By default the control program removes generated output files when an error occurs. This is useful when you use the make utility. If the erroneous files are not removed, the make utility may process corrupt files on a subsequent invocation.

Use this option when you still want to use the generated files. For example when you know that a particular error does not result in a corrupt file, or when you want to inspect the output file, or send it to TASKING support.

The control program passes this option to the compiler, assembler and linker.

Example

```
ccarc --keep-output-files test.c
```

When an error occurs during compiling, assembling or linking, the erroneous generated output files will not be removed.

Related information

C compiler option **--keep-output-files**

Assembler option **--keep-output-files**

Linker option **--keep-output-files**

Control program option: --keep-temporary-files (-t)

Menu entry

1. Select **Global Options**.

2. Enable the option **Keep temporary files**.

Command line syntax

`--keep-temporary-files`

`-t`

Description

By default, the control program removes intermediate files like the `.src` file (result of the compiler phase) and the `.o` file (result of the assembler phase).

With this option you tell the control program to keep temporary files it generates during the creation of the absolute object file.

Example

`ccarc --keep-temporary-files test.c`

The control program keeps all intermediate files it generates while creating the absolute object file `test.elf`.

Related information

-

Control program option: --library (-l)

Menu entry

1. Select **Linker » Libraries**.

    The Libraries box shows the list of libraries that are linked with the project.

2. To add a library, click on the **Add** button in the **Libraries** box.

3. Type or select a library (including its path).

4. Optionally, disable the option **Link default libraries**.

Use the **Edit** and **Delete** button to change a library name or to remove a library from the list.

Command line syntax

--library=*name*

-l*name*

Description

With this option you tell the linker via the control program to use system library lib*name*.a, where *name* is a string. The linker first searches for system libraries in any directories specified with **--library-directory**, then in the directories specified with the environment variables LIBPPU_TC45X / LIBPPU_TC49X / LIBPPU_TC49XN / LIBPPU_TC4DX, unless you used the option **--ignore-default-library-path**.

Example

To search in the system library libc.a (C library):

```
ccarc test.o mylib.a --library=c
```

The linker links the file test.o and first looks in library mylib.a (in the current directory only), then in the system library libc.a to resolve unresolved symbols.

Related information

Control program option **--no-default-libraries** (Do not link default libraries)

Control program option **--library-directory** (Additional search path for system libraries)

Section 7.3, *Linking with Libraries*

Chapter 12, *Libraries*

Control program option: --library-directory (-L) / --ignore-default-library-path

Menu entry

1. Select **Linker » Libraries**.

   The Library search path box shows the directories that are added to the search path for library files.

2. To define a new directory for the search path, click on the **Add** button in the **Library search path** box.

3. Type or select a path.

Use the **Edit** and **Delete** button to change a path or to remove a path from the list.

Command line syntax

```
--library-directory=path,...
-Lpath,...

--ignore-default-library-path
-L
```

Description

With this option you can specify the path(s) where your system libraries, specified with the option **--library** (**-l**), are located. If you want to specify multiple paths, use the option **--library-directory** for each separate path.

The default path is `$(PRODDIR)\lib`.

If you specify only **-L** (without a pathname) or the long option **--ignore-default-library-path**, the linker will not search the default path and also not in the paths specified in the environment variables `LIBPPU_TC45X` / `LIBPPU_TC49X` / `LIBPPU_TC49XN` / `LIBPPU_TC4DX`. So, the linker ignores steps 2 and 3 as listed below.

The priority order in which the linker searches for system libraries specified with the option **--library** (**-l**) is:

1. The path that is specified with the option **--library-directory**.

2. The path that is specified in the environment variables `LIBPPU_TC45X` / `LIBPPU_TC49X` / `LIBPPU_TC49XN` / `LIBPPU_TC4DX`.

3. The default directory `$(PRODDIR)\lib`.

Example

Suppose you call the control program as follows:

```
ccarc test.c --library-directory=c:\mylibs --library=c
```

First the linker looks in the directory `c:\mylibs` for library `libc.a` (this option). If it does not find the requested libraries, it looks in the directory that is set with the environment variables `LIBPPU_TC45X` / `LIBPPU_TC49X` / `LIBPPU_TC49XN` / `LIBPPU_TC4DX`. Then the linker looks in the default directory `$(PRODDIR)\lib` for libraries.

Related information

Control program option **--library** (Link system library)

Section 7.3.1, *How the Linker Searches Libraries*

Control program option: --link-only

Menu entry

-

Command line syntax

`--link-only`

Description

With this option you suppress the locating phase. The linker stops after linking and informs you about unresolved references.

Related information

Control program option **--create=relocatable** (**-cl**) (Stop after linking)

Linker option **--link-only** (Link only, no locating)

Control program option: --list-files

Menu entry

-

Command line syntax

--list-files[**=***file*]

Default: no list files are generated

Description

With this option you tell the assembler via the control program to generate a list file for each specified input file. A list file shows the generated object code and the relative addresses. Note that the assembler generates a relocatable object file with relative addresses.

With the optional *file* you can specify a name for the list file. This is only possible if you specify only one input file to the control program. If you do not specify a file name, or you specify more than one input file, the control program names the generated list file(s) after the specified input file(s) with extension .lst.

Note that object files and library files are not counted as input files.

Related information

Assembler option **--list-file** (Generate list file)

Assembler option **--list-format** (Format list file)

Control program option: --lsl-file (-d)

Menu entry

An LSL file can be generated when you create your TriCore project in Eclipse:

1. From the **File** menu, select **File » New » TASKING TriCore C/C++ Project**.

    The New C/C++ Project wizard appears.

2. Fill in the project settings in each dialog and click **Next >** until the **TriCore Project Settings** appear.

3. Enable the option **Add linker script file to the project** and click **Finish**.

    Eclipse creates your project and the file project.lsl in the project directory.

The LSL file can be specified in the Properties dialog:

1. Select **Linker » Script File**.

2. Specify an LSL file in the **Linker script file (.lsl)** field (default `../${ProjName}.lsl`).

Command line syntax

--lsl-file=*file*,...

-d*file*,...

Description

A linker script file contains vital information about the core for the locating phase of the linker. A linker script file is coded in LSL and contains the following types of information:

*   the architecture definition describes the core's hardware architecture.

*   the memory definition describes the physical memory available in the system.

*   the section layout definition describes how to locate sections in memory.

With this option you specify a linker script file via the control program to the linker. If you do not specify this option, the linker uses a default script file. You can specify the existing file *target*.lsl or the name of a manually written linker script file. You can use this option multiple times. The linker processes the LSL files in the order in which they appear on the command line.

Related information

Section 7.8, *Controlling the Linker with a Script*

Control program option: --make-target

Menu entry

-

Command line syntax

--make-target=*name*

Description

With this option you can overrule the default target name in the make dependencies generated by the options **--preprocess=+make** (**-Em**) and **--dep-file**. The default target name is the basename of the input file, with extension `.o`.

Example

```
ccarc --preprocess=+make --make-target=../mytarget.o test.c
```

The compiler generates dependency lines with the default target name `../mytarget.o` instead of `test.o`.

Related information

Control program option **--preprocess=+make** (Generate dependencies for make)

Control program option **--dep-file** (Generate dependencies in a file)

Control program option **--dep-format** (Format of dependency info)

Control program option: --mil-link / --mil-split

Menu entry

1. Select **C/C++ Compiler » Optimization**.

2. Enable the option **Build for application wide optimizations (MIL linking)**.

3. Select **Optimize less/Build faster** or **Optimize more/Build slower**.

Command line syntax

```
--mil-link
--mil-split[=file,...]
```

Description

With option **--mil-link** the C compiler links the optimized intermediate representation (MIL) of all input files and MIL libraries specified on the command line in the compiler. The result is one single module that is optimized another time.

Option **--mil-split** does the same as option **--mil-link**, but in addition, the resulting MIL representation is written to a file with the suffix `.mil` and the C compiler also splits the MIL representation and writes it to separate files with suffix `.ms`. One file is written for each input file or MIL library specified on the command line. The `.ms` files are only updated on a change.

With option **--mil-split** you can perform application-wide optimizations during the frontend phase by specifying all modules at once, and still invoke the backend phase one module at a time to reduce the total compilation time. Application wide code compaction is not possible in this case.

Optionally, you can specify another filename for the `.ms` file the C compiler generates. Without an argument, the basename of the C source file is used to create the `.ms` filename. Note that if you specify a filename, you have to specify one filename for every input file.

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

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

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

Build for application wide optimizations (MIL linking) and Optimize less/Build faster

This option is standard MIL linking and splitting. Note that you can control the optimizations to be performed with the optimization settings.

Optimize more/Build slower

When you enable this option, the compiler's frontend does not split the MIL stream in separate modules, but feeds it directly to the compiler's backend, allowing the code compaction to be performed application wide.

Related information

Section 4.1, *Compilation Process*

C compiler option **--mil** / **--mil-split**

Control program option: --no-default-libraries

Menu entry

1. Select **Linker » Libraries**.

2. Disable the option **Link default libraries**.

Command line syntax

```
--no-default-libraries
```

Description

By default the control program specifies the standard C libraries (C11/C17) and run-time library to the linker. With this option you tell the control program not to specify the standard C libraries and run-time library to the linker.

In this case you must specify the libraries you want to link to the linker with the option **--library=***library_name* or pass the libraries as files on the command line. The control program recognizes the option **--library** (**-l**) as an option for the linker and passes it as such.

Example

```
ccarc --no-default-libraries test.c
```

The control program does not specify any libraries to the linker. In normal cases this would result in unresolved externals.

To specify your own libraries (`libc.a`) and avoid unresolved externals:

```
ccarc --no-default-libraries --library=c test.c
```

Related information

Control program option **--library** (Link system library)

Section 7.3.1, *How the Linker Searches Libraries*

Control program option: --no-exceptions

Menu entry

-

Command line syntax

`--no-exceptions`

Default: Exceptions are enabled.

Description

With this option you disable support for exception handling in the C++ compiler.

The macro `__EXCEPTIONS` is defined when exception handling support is enabled.

Exceptions are implicitly enabled in C++11 mode and later modes.

Related information

Control program option **--c++=11** (C++ standard)

Control program option: --no-map-file

Menu entry

1. Select **Linker » Map File**.

2. Disable the option **Generate map file**.

Command line syntax

```
--no-map-file
```

Description

By default the control program tells the linker to generate a linker map file.

A linker map file is a text file that shows how the linker has mapped the sections and symbols from the various object files (`.o`) to the linked object file. A locate part shows the absolute position of each section. External symbols are listed per space with their absolute address, both sorted on symbol and sorted on address.

With this option you prevent the generation of a map file.

Related information

-

Control program option: --no-stdinc

Menu entry

1.   Select **C/C++ Compiler » Miscellaneous**.

2.   Add the option **--no-stdinc** to the **Additional options** field.

Command line syntax

```
--no-stdinc
```

Description

With this option the control program tells the C compiler or C++ compiler to not use the standard include paths to search for include files, depending on the compilation mode for the source file.

Related information

Section 4.4, *How the Compiler Searches Include Files*

Section 5.2, *How the C++ Compiler Searches Include Files*

C compiler option **--no-stdinc** (Skip standard include files directory)

C++ compiler option **--no-stdinc** (Skip standard include files directory)

Control program option: --no-warnings (-w)

Menu entry

1. Select **C/C++ Compiler » Diagnostics**.

   The Suppress C compiler warnings box shows the warnings that are currently suppressed.

2. To suppress a warning, click on the **Add** button in the **Suppress C compiler warnings** box.

3. Enter the numbers, separated by commas or as a range, of the warnings you want to suppress (for example `537,538`). Or you can use the **Add** button multiple times.

4. To suppress all warnings, enable the option **Suppress all warnings**.

Use the **Edit** and **Delete** button to change a warning number or to remove a number from the list.

Command line syntax

--no-warnings[**=***number*[-*number*],...]

-w[*number*[-*number*],...]

Description

With this option you can suppresses all warning messages for the various tools or specific control program warning messages.

On the command line this option works as follows:

• If you do not specify this option, all warnings are reported.

• If you specify this option but without numbers, all warnings of all tools are suppressed.

• If you specify this option with a number or a range, only the specified control program warnings are suppressed. You can specify the option **--no-warnings=**number multiple times.

Example

To suppress all warnings for all tools, enter:

```
ccarc test.c --no-warnings
```

Related information

Control program option **--warnings-as-errors** (Treat warnings as errors)

Control program option: --option-file (-f)

Menu entry

-

Command line syntax

--option-file=*file*,...

-f *file*,...

Description

This option is primarily intended for command line use. Instead of typing all options on the command line, you can create an option file which contains all options and flags you want to specify. With this option you specify the option file to the control program.

Use an option file when the command line would exceed the limits of the operating system, or just to store options and save typing.

You can specify the option **--option-file** multiple times.

Format of an option file

• Multiple arguments on one line in the option file are allowed.

• To include whitespace in an argument, surround the argument with single or double quotes.

• If you want to use single quotes as part of the argument, surround the argument by double quotes and vise versa:

```
"This has a single quote ' embedded"

'This has a double quote " embedded'

'This has a double quote " and a single quote '"' embedded"
```

• When a text line reaches its length limit, use a **** to continue the line. Whitespace between quotes is preserved.

```
"This is a continuation \
line"

        -> "This is a continuation line"
```

• It is possible to nest command line files up to 25 levels.

Example

Suppose the file myoptions contains the following lines:

```
--debug-info
--define=DEMO=1
test.c
```

Specify the option file to the control program:

```
ccarc --option-file=myoptions
```

This is equivalent to the following command line:

```
ccarc --debug-info --define=DEMO=1 test.c
```

Related information

-

Control program option: --output (-o)

Menu entry

Eclipse always uses the project name as the basename for the output file.

Command line syntax

--output=*file*

-o *file*

Description

By default, the control program generates a file with the same basename as the first specified input file. With this option you specify another name for the resulting absolute object file.

The default output format is ELF/DWARF, but you can specify another output format with option **--format**.

Example

```
ccarc test.c prog.c
```

The control program generates an ELF/DWARF object file (default) with the name test.elf.

To generate the file result.elf:

```
ccarc --output=result.elf test.c prog.c
```

Related information

Control program option **--format** (Set linker output format)

Linker option **--output** (Output file)

Linker option **--chip-output** (Generate an output file for each chip)

Control program option: --pass (-W)

Menu entry

1. Select **C/C++ Compiler » Miscellaneous** or **Assembler » Miscellaneous** or **Linker » Miscellaneous**.

2. Add an option to the **Additional options** field.

   *Be aware that the options in the option file are added to the options you have set in the other pages. Only in extraordinary cases you may want to use them in combination. The assembler options are preceded by **-Wa** and the linker options are preceded by **-Wl**. For the C/C++ options you have to do this manually.*

Command line syntax

--pass-assembler=*option*	**-Wa***option*	Pass option directly to the assembler
--pass-c=*option*	**-Wc***option*	Pass option directly to the C compiler
--pass-c++=*option*	**-Wcp***option*	Pass option directly to the C++ compiler
--pass-linker=*option*	**-Wl***option*	Pass option directly to the linker

Description

With this option you tell the control program to call a tool with the specified option. The control program does not use or interpret the option itself, but specifies it directly to the tool which it calls.

Example

To pass the option **--verbose** directly to the linker, enter:

```
ccarc --pass-linker=--verbose test.c
```

Related information

-

Control program option: --preprocess (-E) / --no-preprocessing-only

Menu entry

1. Select **C/C++ Compiler » Preprocessing**.

2. Enable the option **Store preprocessor output in <file>.pre**.

3. (Optional) Enable the option **Keep comments in preprocessor output**.

4. (Optional) Enable the option **Keep #line info in preprocessor output**.

5. (Optional) Enable the option **Insert a space between adjacent tokens (if needed)**.

Command line syntax

--preprocess[**=**_flags_]

-E[_flags_]

--no-preprocessing-only

You can set the following flags:

+/-comments	**c/C**	keep comments
+/-includes	**i/I**	generate a list of included source files
+/-list	**l/L**	generate a list of macro definitions
+/-make	**m/M**	generate dependencies for make
+/-noline	**p/P**	strip #line source position information
+/-token-separation	**t/T**	insert a space between adjacent tokens (if needed)

Default: **-ECILMPT**

Description

With this option you tell the compiler to preprocess the C source. The C compiler sends the preprocessed output to the file _name_.pre (where _name_ is the name of the C source file to compile). Eclipse also compiles the C source.

On the command line, the control program stops after preprocessing. If you also want to compile the C source you can specify the option **--no-preprocessing-only**. In this case the control program calls the compiler twice, once with option **--preprocess** and once for a regular compilation.

With **--preprocess=+comments** you tell the preprocessor to keep the comments from the C source file in the preprocessed output.

With **--preprocess=+includes** the compiler will generate a list of all included source files. The preprocessor output is discarded.

With **--preprocess=+list** the compiler will generate a list of all macro definitions. The preprocessor output is discarded.

With **--preprocess=+make** the compiler will generate dependency lines that can be used in a Makefile. The information is written to a file with extension `.d`. The preprocessor output is discarded. The default target name is the basename of the input file, with the extension `.o`. With the option **--make-target** you can specify a target name which overrules the default target name.

With **--preprocess=+noline** you tell the preprocessor to strip the #line source position information (lines starting with `#line`). These lines are normally processed by the assembler and not needed in the preprocessed output. When you leave these lines out, the output is easier to read.

With **--preprocess=+token-separation** you tell the preprocessor to insert a space between adjacent tokens, if needed. For example, to prevent concatenation due to a macro expansion.

Example

```
ccarc --preprocess=+comments,-make,-noline --no-preprocessing-only test.c
```

The compiler preprocesses the file `test.c` and sends the output to the file `test.pre`. Comments are included but no dependencies are generated and the line source position information is not stripped from the output file. Next, the control program calls the compiler, assembler and linker to create the final object file `test.elf`

Related information

Control program option **--dep-file** (Generate dependencies in a file)

Control program option **--dep-format** (Format of dependency info)

Control program option **--make-target** (Specify target name for **-Em** output)

Control program option: --schar

Menu entry

1. Select **C/C++ Compiler » Language**.

2. Enable the option **Treat 'char' variables as signed**.

Command line syntax

--schar

Description

By default char is the same as specifying unsigned char as required by the ABI. With this option char is the same as signed char.

Note that this option can cause conflicts when you use it in combination with MIL linking. With MIL linking some extra strict type checking is done that can cause building to fail in a way that is unforeseen and difficult to understand. For example, when you use option **--mil-link** in combination with option **--schar** you might get the following error:

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

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

Related information

Section 1.1, *Data Types*

Control program option: --show-c++-warnings

Menu entry

-

Command line syntax

`--show-c++-warnings`

Description

The C++ compiler may generate a compiled C++ file (`.ic`) that causes warnings during compilation or assembling. With this option you tell the control program to show these warnings. By default C++ warnings are suppressed.

Related information

-

Control program option: --static

Menu entry

-

Command line syntax

`--static`

Description

This option is directly passed to the compiler.

With this option, the compiler treats external definitions at file scope (except for `main`) as if they were declared `static`. As a result, unused functions will be eliminated, and the alias checking algorithm assumes that objects with static storage cannot be referenced from functions outside the current module.

This option only makes sense when you specify all modules of an application on the command line.

Example

```
ccarc --static module1.c module2.c module3.c ...
```

Related information

-

Control program option: --tasking-sfr

Menu entry

1. Select **C/C++ Compiler » Preprocessing**.

2. Enable the option **Automatic inclusion of '.sfr' file**.

3. Select **Assembler » Preprocessing**.

4. Enable the option **Automatic inclusion of '.def' file**.

Command line syntax

`--tasking-sfr`

Description

By default, the C compiler and assembler do not include a special function register (SFR) file before compiling/assembling.

With this option the compiler includes the register file `regppu.sfr` and the assembler includes the file `regppu.def`. The control program passes the appropriate **-H** option to the tools.

Example

To generate the file `test.elf` for the and automatically include SFR files, enter:

```
ccarc --tasking-sfr -v -t test.c
```

The control program will call the tools as follows:

```
carc  -D__CPU__=tc49x -D__CPU_TC49X__  -Hsfr/regppu.sfr
      -o test.src test.c
asarc -D__CPU__=tc49x -D__CPU_TC49X__  -Hsfr/regppu.def
      -o test.o test.src
larc  -o test.elf -dtc49x.lsl --core=ppu_tc49x -D__CPU__=tc49x
      -D__PROC_TC49X__ --map-file test.o -lc_fpu -lrt
      "-Linstall-dir\carc\lib\tc49x"
```

Related information

Section 1.3.6, *Accessing Hardware from C*

Control program option: --undefine (-U)

Menu entry

1. Select **C/C++ Compiler » Preprocessing**

    The Defined symbols box shows the symbols that are currently defined.

2. To remove a defined symbol, select the symbol in the **Defined symbols** box and click on the **Delete** button.

Command line syntax

--undefine=`macro_name`

-U`macro_name`

Description

With this option you can undefine an earlier defined macro as with `#undef`. This option is for example useful to undefine predefined macros.

The following predefined ISO C standard macros cannot be undefined:

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

The control program passes the option **--undefine** (**-U**) to the compiler.

Example

To undefine the predefined macro `__TASKING__`:

```
ccarc --undefine=__TASKING__ test.c
```

Related information

Control program option **--define** (Define preprocessor macro)

Section 1.9, *Predefined Preprocessor Macros*

Control program option: --verbose (-v)

Menu entry

1. Select **Global Options**.

2. Enable the option **Verbose mode of control program**.

Command line syntax

`--verbose`

`-v`

Description

With this option you put the control program in verbose mode. The control program performs its tasks while it prints the steps it performs to stdout.

Related information

Control program option **--dry-run** (Verbose output and suppress execution)

Control program option: --version (-V)

Menu entry

-

Command line syntax

```
--version
```

```
-V
```

Description

Display version information. The control program ignores all other options or input files.

Related information

-

Control program option: --warnings-as-errors

Menu entry

1. Select **Global Options**.

2. Enable the option **Treat warnings as errors**.

Command line syntax

```
--warnings-as-errors[=number[-number],...]
```

Description

If one of the tools encounters an error, it stops processing the file(s). With this option you tell the tools to treat warnings as errors or treat specific control program warning messages as errors:

• If you specify this option but without numbers, all warnings are treated as errors.

• If you specify this option with a number or a range, only the specified control program warnings are treated as an error. You can specify the option **--warnings-as-errors=**_number_ multiple times.

Use one of the **--pass-**_tool_ options to pass this option directly to a tool when a specific warning for that tool must be treated as an error. For example, use **--pass-c=--warnings-as-errors=**_number_ to treat a specific C compiler warning as an error.

Related information

Control program option **--no-warnings** (Suppress some or all warnings)

Control program option **--pass** (Pass option to tool)

10.7. Parallel Make Utility Options

When you build a project in Eclipse, Eclipse generates a makefile and uses the make utility **amk** to build all your files. However, you can also use the make utility directly from the command line to build your project.

The invocation syntax is:

amk [*option*...] [*target*...] [*macro*=*def*]

This section describes all options for the parallel make utility.

For detailed information about the parallel make utility and using makefiles see Section 8.2, *Make Utility amk*.

Parallel make utility option: --always-rebuild (-a)

Command line syntax

`--always-rebuild`

`-a`

Description

Normally the make utility rebuilds only those files that are out of date. With this option you tell the make utility to rebuild *all* files, without checking whether they are out of date.

Example

```
amk -a
```

Rebuilds all your files, regardless of whether they are out of date or not.

Related information

-

Parallel make utility option: --change-dir (-G)

Command line syntax

--change-dir=*path*

-G *path*

Description

Normally you must call the make utility 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.

The macro SUBDIR is defined with the value of *path*.

Example

Suppose your makefile and other files are stored in the directory ..\myfiles. You can call the make utility, for example, as follows:

```
amk -G ..\myfiles
```

Related information

-

Parallel make utility option: --diag

Command line syntax

--diag=[*format*:]{**all** | *msg*[*-msg*],...}

You can set the following output formats:

html	HTML output.
rtf	Rich Text Format.
text	ASCII text.

Default format: text

Description

With this option you can ask for an extended description of error messages in the format you choose. The output is directed to stdout (normally your screen) and in the format you specify. You can specify the following formats: **html**, **rtf** or **text** (default). To create a file with the descriptions, you must redirect the output.

With the suboption **all**, the descriptions of all error messages are given. If you want the description of one or more selected error messages, you can specify the error message numbers, separated by commas, or you can specify a range.

Example

To display an explanation of message number 169, enter:

```
amk --diag=451
```

This results in the following message and explanation:

```
E451: make stopped

An error has occured while executing one of the commands
of the target, and -k option is not specified.
```

To write an explanation of all errors and warnings in HTML format to file `amkerrors.html`, use redirection and enter:

```
amk --diag=html:all > amkerrors.html
```

Related information

-

Parallel make utility option: --dry-run (-n)

Command line syntax

`--dry-run`

`-n`

Description

With this option you tell the make utility to perform a dry run. The make utility shows what it would do but does not actually perform these tasks.

This option is for example useful to quickly inspect what would happen if you call the make utility.

Example

```
amk -n
```

The make utility does not perform any tasks but displays what it would do if called without the option **-n**.

Related information

Parallel make utility option **-s** (Do not print commands before execution)

Parallel make utility option: --help (-? / -h)

Command line syntax

--help[**=**_item_]

-h[_item_]

-?

You can specify the following arguments:

    options   **o**   Show extended option descriptions

Description

Displays an overview of all command line options. When you specify the argument **options** you can list detailed option descriptions.

Example

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

```
amk -?
amk -h
amk --help
```

To see a detailed description of the available options, enter:

```
amk --help=options
```

Related information

-

Parallel make utility option: --jobs (-j) / --jobs-limit (-J)

Menu

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

    The Properties dialog appears.

2. In the left pane, select **C/C++ Build**.

    In the right pane the C/C++ Build page appears.

3. On the Behavior tab, select **Enable parallel build**.

4. You can specify the number of parallel jobs, or you can use an optimal number of jobs. In the last case, **amk** will fork as many jobs in parallel as cores are available.

Command line syntax

```
--jobs[=number]
-j[number]

--jobs-limit[=number]
-J[number]
```

Description

When these options you can limit the number of parallel jobs. The default is 1. Zero means no limit. When you omit the *number*, **amk** uses the number of cores detected.

Option **-J** is the same as **-j**, except that the number of parallel jobs is limited by the number of cores detected.

Example

```
amk -j3
```

Limit the number of parallel jobs to 3.

Related information

-

Parallel make utility option: --keep-going (-k)

Command line syntax

`--keep-going`

`-k`

Description

When during the make process the make utility encounters an error, it stops rebuilding your files.

With the option **-k**, the make utility only stops building the target that produced the error. All other targets defined in the makefile are built.

Example

```
amk -k
```

If the make utility encounters an error, it stops building the current target but proceeds with the other targets that are defined in the makefile.

Related information

-

Parallel make utility option: --list-targets (-l)

Command line syntax

`--list-targets`

`-l`

Description

With this option, the make utility lists all "primary" targets that are out of date.

Example

```
amk -l
list of targets
```

Related information

-

Parallel make utility option: --makefile (-f)

Command line syntax

--makefile=<i>my_makefile</i>

-f <i>my_makefile</i>

Description

By default the make utility uses the file `makefile` to build your files.

With this option you tell the make utility to use the specified file instead of the file `makefile`. Multiple **-f** options act as if all the makefiles were concatenated in a left-to-right order.

If you use '-' instead of a makefile name it means that the information is read from `stdin`.

Example

```
amk -f mymake
```

The make utility uses the file `mymake` to build your files.

Related information

-

Parallel make utility option: --no-warnings (-w)

Command line syntax

`--no-warnings``[=number,...]`

`-w``[number,...]`

Description

With this option you can suppresses all warning messages or specific warning messages.

On the command line this option works as follows:

- If you do not specify this option, all warnings are reported.

- If you specify this option but without numbers, all warnings are suppressed.

- If you specify this option with a number, only the specified warning is suppressed. You can specify the option **--no-warnings=**_number_ multiple times.

Example

To suppress warnings 751 and 756, enter:

```
amk --no-warnings=751,756
```

Related information

Parallel make utility option **--warnings-as-errors** (Treat warnings as errors)

627

Parallel make utility option: --silent (-s)

Command line syntax

```
--silent
```

```
-s
```

Description

With this option you tell the make utility to perform its tasks without printing the commands it executes. Error messages are normally printed.

Example

```
amk -s
```

The make utility rebuilds your files but does not print the commands it executes during the make process.

Related information

Parallel make utility option **-n** (Perform a dry run)

Parallel make utility option: --version (-V)

Command line syntax

`--version`

`-v`

Description

Display version information. The make utility ignores all other options or input files.

Related information

-

Parallel make utility option: --warnings-as-errors

Command line syntax

`--warnings-as-errors`[`=`*number,...*]

Description

If the make utility encounters an error, it stops. When you use this option without arguments, you tell the make utility to treat all warnings as errors. This means that the exit status of the make utility will be non-zero after one or more warnings. As a consequence, the make utility now also stops after encountering a warning.

You can also limit this option to specific warnings by specifying a comma-separated list of warning numbers.

Related information

Parallel make utility option **--no-warnings** (Suppress some or all warnings)

10.8. Archiver Options

The archiver and library maintainer **ararc** is a tool to build library files and it offers the possibility to replace, extract and remove modules from an existing library.

The invocation syntax is:

ararc *key_option* [*sub_option...*] *library* [*object_file*]

This section describes all options for the archiver. Some suboptions can only be used in combination with certain key options. They are described together. Suboptions that can always be used are described separately.

For detailed information about the archiver, see Section 8.3, *Archiver*.

Short and long option names

Options can have both short and long names. Short option names always begin with a single minus (**-**) character, long option names always begin with two minus (**--**) characters. You can abbreviate long option names as long as it forms a unique name. You can mix short and long option names on the command line.

Overview of the options of the archiver utility

The following archiver options are available:

Description	Option	Sub-option
Main functions (key options)		
Replace or add an object module	**-r**	**-a -b -c -n -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**	
Sub-options		
Append or move new modules after existing module *name*	**-a** *name*	
Append or move new modules before existing module *name*	**-b** *name*	
Suppress the message that is displayed when a new library is created	**-c**	
Create a new library from scratch	**-n**	
Preserve last-modified date from the library	**-o**	
Print symbols in library modules	**-s{0|1}**	
Replace only newer modules	**-u**	
Verbose	**-v**	

Description	Option	Sub-option
Miscellaneous		
Display options	**-?**	
Display description of one or more diagnostic messages	**--diag**	
Display version header	**-V**	
Read options from *file*	**-f** *file*	

Archiver option: --delete (-d)

Command line syntax

--delete [**--verbose**]

-d [**-v**]

Description

Delete the specified object modules from a library. With the suboption **--verbose** (**-v**) the archiver shows which files are removed.

--verbose   **-v**   Verbose: the archiver shows which files are removed.

Example

ararc --delete mylib.a obj1.o obj2.o

The archiver deletes obj1.o and obj2.o from the library mylib.a.

ararc -d -v mylib.a obj1.o obj2.o

The archiver deletes obj1.o and obj2.o from the library mylib.a and displays which files are removed.

Related information

-

Archiver option: --diag

Command line syntax

--diag=[*format*:]{**all** | *msg*[-*msg*],...}

You can set the following output formats:

html	HTML output.
rtf	Rich Text Format.
text	ASCII text.

Default format: text

Description

With this option you can ask for an extended description of error messages in the format you choose. The output is directed to stdout (normally your screen) and in the format you specify. The archiver does not perform any actions. You can specify the following formats: **html**, **rtf** or **text** (default). To create a file with the descriptions, you must redirect the output.

With the suboption **all**, the descriptions of all error messages are given. If you want the description of one or more selected error messages, you can specify the error message numbers, separated by commas, or you can specify a range.

Example

To display an explanation of message number 102, enter:

```
ararc --diag=102
```

This results in the following message and explanation:

```
F102: cannot create "<file>"
```

```
The output file or a temporary file could not be created. Check if you have
sufficient disk space and if you have write permissions for the specified
file.
```

To write an explanation of all errors and warnings in HTML format to file `arerrors.html`, use redirection and enter:

```
ararc --diag=html:all > arerrors.html
```

Related information

-

Archiver option: --dump (-p)

Command line syntax

`--dump`

`-p`

Description

Print the specified object module(s) in the library to standard output.

This option is only useful when you redirect or pipe the output to other files or tools that serve your own purposes. Normally you do not need this option.

Example

```
ararc --dump mylib.a obj1.o > file.o
```

The archiver prints the file `obj1.o` to standard output where it is redirected to the file `file.o`. The effect of this example is very similar to extracting a file from the library but in this case the 'extracted' file gets another name.

Related information

-

Archiver option: --extract (-x)

Command line syntax

`--extract` [`--modtime`] [`--verbose`]

`-x` [`-o`] [`-v`]

Description

Extract an existing module from the library.

--modtime	**-o**	Give the extracted object module the same date as the last-modified date that was recorded in the library. Without this suboption it receives the last-modified date of the moment it is extracted.
--verbose	**-v**	Verbose: the archiver shows which files are extracted.

Example

To extract the file `obj1.o` from the library `mylib.a`:

```
ararc --extract mylib.a obj1.o
```

If you do not specify an object module, all object modules are extracted:

```
ararc -x mylib.a
```

Related information

-

Archiver option: --help (-?)

Command line syntax

--help[**=***item*]

-?

You can specify the following argument:

>   **options**   Show extended option descriptions

Description

Displays an overview of all command line options. When you specify the argument **options** you can list detailed option descriptions.

Example

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

```
ararc -?
ararc --help
ararc
```

To see a detailed description of the available options, enter:

```
ararc --help=options
```

Related information

-

Archiver option: --move (-m)

Command line syntax

--move [**-a** *posname*] [**-b** *posname*]

-m [**-a** *posname*] [**-b** *posname*]

Description

Move the specified object modules to another position in the library.

The ordering of members in a library can make a difference in how programs are linked if a symbol is defined in more than one member.

By default, the specified members are moved to the end of the archive. Use the suboptions **-a** or **-b** to move them to a specified place instead.

--after=*posname*	**-a** *posname*	Move the specified object module(s) after the existing module *posname*.
--before=*posname*	**-b** *posname*	Move the specified object module(s) before the existing module *posname*.

Example

Suppose the library `mylib.a` contains the following objects (see option **--print**):

```
obj1.o
obj2.o
obj3.o
```

To move `obj1.o` to the end of `mylib.a`:

```
ararc --move mylib.a obj1.o
```

To move `obj3.o` just before `obj2.o`:

```
ararc -m -b obj2.o mylib.a obj3.o
```

The library `mylib.a` after these two invocations now looks like:

```
obj3.o
obj2.o
obj1.o
```

Related information

Archiver option **--print** (**-t**) (Print library contents)

Archiver option: --option-file (-f)

Command line syntax

--option-file=*file*

-f *file*

Description

Instead of typing all options on the command line, you can create an option file which contains all options and flags you want to specify. With this option you specify the option file to the archiver.

Use an option file when the command line would exceed the limits of the operating system, or just to store options and save typing.

You can specify the option **--option-file** (**-f**) multiple times.

If you use '-' instead of a filename it means that the options are read from stdin.

Format of an option file

• Multiple arguments on one line in the option file are allowed.

• To include whitespace in an argument, surround the argument with single or double quotes.

• If you want to use single quotes as part of the argument, surround the argument by double quotes and vise versa:

```
"This has a single quote ' embedded"

'This has a double quote " embedded'

'This has a double quote " and a single quote '"' embedded"
```

• When a text line reaches its length limit, use a **** to continue the line. Whitespace between quotes is preserved.

```
"This is a continuation \
line"

        -> "This is a continuation line"
```

• It is possible to nest command line files up to 25 levels.

Example

Suppose the file myoptions contains the following lines:

```
-x mylib.a obj1.o
-v
```

Specify the option file to the archiver:

```
ararc --option-file=myoptions
```

This is equivalent to the following command line:

```
ararc -x mylib.a obj1.o -v
```

Related information

-

Archiver option: --print (-t)

Command line syntax

--print [**--symbols=0**|**1**]

-t [**-s0**|**-s1**]

Description

Print a table of contents of the library to standard output. With the suboption **-s0** the archiver displays all symbols per object file.

--symbols=0	**-s0**	Displays per object the name of the object itself and all symbols in the object.
--symbols=1	**-s1**	Displays the symbols of all object files in the library in the form *library_name*:*object_name*:*symbol_name*

Example

```
ararc --print mylib.a
```

The archiver prints a list of all object modules in the library `mylib.a`:

```
ararc -t -s0 mylib.a
```

The archiver prints per object all symbols in the library. For example:

```
cstart.o
   symbols:
        _START
        _start
        _Exit
```

Related information

-

Archiver option: --replace (-r)

Command line syntax

```
--replace [--after=posname] [--before=posname]
          [--create] [--new] [--newer-only] [--verbose]
```

```
-r [-a posname] [-b posname][-c] [-n] [-u] [-v]
```

Description

You can use the option **--replace** (**-r**) for several purposes:

- Adding new objects to the library

- Replacing objects in the library with the same object of a newer date

- Creating a new library

The option **--replace** (**-r**) normally *adds* a new module to the library. However, if the library already contains a module with the specified name, the existing module is *replaced*. If you specify a library that does not exist, the archiver creates a new library with the specified name.

If you add a module to the library without specifying the suboption **-a** or **-b**, the specified module is added at the end of the archive. Use the suboptions **-a** or **-b** to insert them after/before a specified place instead.

--after=posname	**-a** posname	Insert the specified object module(s) after the existing module *posname*.
--before=posname	**-b** posname	Insert the specified object module(s) before the existing module *posname*.
--create	**-c**	Suppress the message that is displayed when a new library is created.
--new	**-n**	Create a new library from scratch. If the library already exists, it is overwritten.
--newer-only	**-u**	Insert the specified object module only if it is newer than the module in the library.
--verbose	**-v**	Verbose: the archiver shows which files are replaced.

The suboptions **-a** or **-b** have no effect when an object is added to the library.

Example

Suppose the library `mylib.a` contains the following object (see option **--print**):

```
obj1.o
```

To add `obj2.o` to the end of `mylib.a`:

```
ararc --replace mylib.a obj2.o
```

To insert `obj3.o` just before `obj2.o`:

```
ararc -r -b obj2.o mylib.a obj3.o
```

The library `mylib.a` after these two invocations now looks like:

```
obj1.o
obj3.o
obj2.o
```

Creating a new library

To *create a new library file*, add an object file and specify a library that does not yet exist:

```
ararc --replace newlib.a obj1.o
```

The archiver creates the library `newlib.a` and adds the object `obj1.o` to it.

To *create a new library file and overwrite an existing library*, add an object file and specify an existing library with the suboption **--new** (**-n**):

```
ararc -r -n mylib.a obj1.o
```

The archiver overwrites the library `mylib.a` and adds the object `obj1.o` to it. The new library `mylib.a` only contains `obj1.o`.

Related information

Archiver option **--print** (**-t**) (Print library contents)

643

Archiver option: --version (-V)

Command line syntax

`--version`

`-V`

Description

Display version information. The archiver ignores all other options or input files.

Related information

-

10.9. HLL Object Dumper Options

The high level language (HLL) dumper **hldumparc** is a program to dump information about an absolute object file (`.elf`).

Short and long option names

Options can have both short and long names. Short option names always begin with a single minus (**-**) character, long option names always begin with two minus (**--**) characters. You can abbreviate long option names as long as it forms a unique name. You can mix short and long option names on the command line.

Options can have flags or suboptions. To switch a flag 'on', use a lowercase letter or a **+***longflag*. To switch a flag off, use an uppercase letter or a **-***longflag*. Separate *longflags* with commas. The following two invocations are equivalent:

```
hldumparc -FdhMsy test.elf
hldumparc --dump-format=+dump,+hllsymbols,-modules,+sections,+symbols test.elf
```

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

HLL object dumper option: --adx-format (-A)

Command line syntax

`--adx-format[=flag],...`

`-A[flag]...`

You can specify one of the following flags:

+/-force-elf-mode	**e/E**	Force the use of ELF symbols instead of the DWARF debug info
+/-reduced	**r/R**	Do not output tags CATEGORY, COMP-UNIT-NAME, COMP-UNIT-DIR and CALLED-SYMBOLS.

Default (no flags): `--adx-format=ER`

Description

With this option you dump the application data in the ADX address list format. The address list format is based on XML.

With **--adx-format=+force-elf-mode**, ELF symbols are used instead of the DWARF debug info, resulting in reduced info.

With **--adx-format=+reduced**, the tags CATEGORY, COMP-UNIT-NAME, COMP-UNIT-DIR and CALLED-SYMBOLS are not printed in the XML output.

Note that when you use this option all other output formatting options are ignored.

Example

```
hldumparc --adx-format hello.elf

<?xml version="1.0"?>
<!-- Using DWARF debug info -->
<ADDRESS-CALCULATOR version="1.0.4" spec="1.10">
  <GENERAL-INFO>
    <MACHINE-TYPE>ARCv2</MACHINE-TYPE>
    <ELF-TYPE>ET_EXEC</ELF-TYPE>
  </GENERAL-INFO>
  <MEMORY-ELEMENT>
    <LABEL-NAME>_dbg_request</LABEL-NAME>
    <CATEGORY>STRUCTURE</CATEGORY>
    <ABSOLUTE-ADDRESS>0x00100008</ABSOLUTE-ADDRESS>
    <SIZE>20</SIZE>
    <DEMANGLED-NAME>_dbg_request</DEMANGLED-NAME>
    <SH-INDEX>27</SH-INDEX>
    <COMP-UNIT-NAME>dbg.c</COMP-UNIT-NAME>
    <COMP-UNIT-DIR>~\carc\lib\src\libc\lib\</COMP-UNIT-DIR>
  </MEMORY-ELEMENT>
```

```
      <SECTION-ELEMENT>
        <SECTION-NAME>.text.hello.main</SECTION-NAME>
        <SECTION-START-ADDRESS>0x00000494</SECTION-START-ADDRESS>
        <SECTION-SIZE>0x14</SECTION-SIZE>
        <SECTION-INDEX>4</SECTION-INDEX>
        <SECTION-TYPE>PROGBITS</SECTION-TYPE>
      </SECTION-ELEMENT>

hldumparc --adx-format=+reduced hello.elf

<?xml version="1.0"?>
<!-- Using DWARF debug info -->
<ADDRESS-CALCULATOR version="1.0.4" spec="1.10">
   <GENERAL-INFO>
      <MACHINE-TYPE>ARCv2</MACHINE-TYPE>
      <ELF-TYPE>ET_EXEC</ELF-TYPE>
   </GENERAL-INFO>
   <MEMORY-ELEMENT>
      <LABEL-NAME>_dbg_request</LABEL-NAME>
      <ABSOLUTE-ADDRESS>0x00100008</ABSOLUTE-ADDRESS>
      <SIZE>20</SIZE>
      <DEMANGLED-NAME>_dbg_request</DEMANGLED-NAME>
      <SH-INDEX>27</SH-INDEX>
   </MEMORY-ELEMENT>
   <SECTION-ELEMENT>
      <SECTION-NAME>.text.hello.main</SECTION-NAME>
      <SECTION-START-ADDRESS>0x00000494</SECTION-START-ADDRESS>
      <SECTION-SIZE>0x14</SECTION-SIZE>
      <SECTION-INDEX>4</SECTION-INDEX>
      <SECTION-TYPE>PROGBITS</SECTION-TYPE>
   </SECTION-ELEMENT>

hldumparc --adx-format=+force-elf-mode hello.elf

<?xml version="1.0"?>
<!-- Using ELF symbols -->
<ADDRESS-CALCULATOR version="1.0.4" spec="1.10">
   <GENERAL-INFO>
      <MACHINE-TYPE>ARCv2</MACHINE-TYPE>
      <ELF-TYPE>ET_EXEC</ELF-TYPE>
   </GENERAL-INFO>
   <MEMORY-ELEMENT>
      <LABEL-NAME>_dbg_request</LABEL-NAME>
      <CATEGORY>DATA</CATEGORY>
      <ABSOLUTE-ADDRESS>0x00100008</ABSOLUTE-ADDRESS>
      <OFFSET>0x00100008</OFFSET>
      <SIZE>20</SIZE>
   </MEMORY-ELEMENT>
   <SECTION-ELEMENT>
      <SECTION-NAME>.text.hello.main</SECTION-NAME>
      <SECTION-START-ADDRESS>0x00000494</SECTION-START-ADDRESS>
```

```
   <SECTION-SIZE>0x14</SECTION-SIZE>
   <SECTION-INDEX>4</SECTION-INDEX>
   <SECTION-TYPE>PROGBITS</SECTION-TYPE>
  </SECTION-ELEMENT>
</ADDRESS-CALCULATOR>
```

Related information

ADX Specification - Address List Format for A2L Address Calculation - Compiler vendors, Version 1.10, 2015-04-27

HLL object dumper option: --blank-out (-b)

Command line syntax

--blank-out[**=***flag*]

-b[*flag*]

You can specify the following format flags:

+/-labels	**l/L**	Black out hexadecimal address and labels.

Default: **--blank-out=L**

Description

With this option you can blank out addresses and optionally labels in all dump phases. Instead of the addresses and labels crosses (X's) are shown.

The **+labels** sub-option blanks out hexadecimal addresses and labels. With the **-labels** sub-option only hexadecimal addresses are blanked out. This is the default.

This option is useful when you want to compare the output, but want to ignore the addresses and labels.

Example

```
hldumparc -F2 hello.elf

----------- Section dump ----------

                                        .section .text.hello.main, at(0x00000494)
00000494 f1 c0        main:            push_s %blink
00000496 c3 40 00 00 72 05                 mov_s %r0,1394

0000049c 00 50                         ld_s %r1,[%gp,0]
0000049e 4a 08 00 00                   bl     printf
000004a2 0c 70                         mov_s %r0,0
000004a4 d1 c0                         pop_s %blink
000004a6 e0 7e                         j_s [%blink]
                                        .endsec

hldumparc -F2 --blank-out hello.elf

----------- Section dump ----------

                                        .section .text.hello.main, at(0x00000494)
XXXXXXXX f1 c0        main:            push_s %blink
XXXXXXXX c3 40 00 00 72 05                 mov_s %r0,1394

XXXXXXXX 00 50                         ld_s %r1,[%gp,0]
XXXXXXXX 4a 08 00 00                   bl     printf
```

649

```
XXXXXXXX 0c 70                          mov_s %r0,0
XXXXXXXX d1 c0                          pop_s %blink
XXXXXXXX e0 7e                          j_s [%blink]
                                        .endsec

hldumparc -F2 --blank-out=+labels hello.elf

----------- Section dump ----------

                                        .section .text.hello.main, at(0x00000494)
XXXXXXXX f1 c0        XXXXXXXXXXX:       push_s %blink
XXXXXXXX c3 40 00 00 72 05                  mov_s %r0,1394

XXXXXXXX 00 50                          ld_s %r1,[%gp,0]
XXXXXXXX 4a 08 00 00                    bl     printf
XXXXXXXX 0c 70                          mov_s %r0,0
XXXXXXXX d1 c0                          pop_s %blink
XXXXXXXX e0 7e                          j_s [%blink]
                                        .endsec
```

Related information

-

HLL object dumper option: --call-graph-elf-mode

Command line syntax

`--call-graph-elf-mode`

Description

With this option you can force the call graph to use the ELF symbols instead of the DWARF debug info, for example when dumping from an assembly function.

Related information

Section 8.4.2, *HLL Dump Output Format*

HLL object dumper option: --call-graph-root

Command line syntax

--call-graph-root=*function*

Description

With this option you can specify the address or function name where to start the call graph. By default, the call graph starts with main().

Example

To start the call graph from printf() instead of main(), enter:

hldumparc --call-graph-root=printf -F3 hello.elf

The call graph looks something like this:

```
+-- 0x000004e4 printf
    |
    +-- 0x000000dc _doprint
        |
        +-- 0x00000238 _io_putc
        |   |
        |   +-- 0x0000046c fputc
        |       |
        |       +-- 0x00000228 _flsbuf
        |           |
        |           +-- 0x00000158 _dofls
        |               |
        |               +-- 0x0000031c _host_write
        |               |   |
        |               |   +-- 0x00000364 _dbg_trap
        |               |
        |               +-- 0x000003c4 _fflush
        |               |   |
        |               |   +-- 0x0000031c _host_write *
        |               |   |
        |               |   +-- 0x000002e8 _host_lseek
        |               |       |
        |               |       +-- 0x00000364 _dbg_trap
        |               |
        |               +-- 0x0000031c _host_write *
        |
        +-- 0x00000238 _io_putc *
```

Related information

HLL object dumper option: --class (-c)

Command line syntax

--class[**=***class*]

-c[*class*]

You can specify one of the following classes:

all	**a**	Dump contents of all sections.
code	**c**	Dump contents of code sections.
data	**d**	Dump contents of data sections.

Default: **--class=all**

Description

With this option you can restrict the output to code or data only. This option affects all parts of the output, except the module list. The effect is listed in the following table.

Output part	Effect of --class
Module list	Not restricted
Section list	Only lists sections of the specified class
Section dump	Only dumps the contents of the sections of the specified class
HLL symbol table	Only lists symbols of the specified class
Assembly level symbol table	Only lists symbols defined in sections of the specified class
Note sections	Not restricted

By default all sections are included.

Related information

Section 8.4.2, *HLL Dump Output Format*

HLL object dumper option: --copy-table

Command line syntax

```
--copy-table
```

Description

With this option the HLL object dumper attempts to translate the specified code address to the destination address of a copy table copy command during disassembly.

Related information

-

HLL object dumper option: --diag

Command line syntax

--diag=[*format*:]{**all** | *msg*[-*msg*],...}

You can set the following output formats:

html	HTML output.
rtf	Rich Text Format.
text	ASCII text.

Default format: text

Description

With this option you can ask for an extended description of error messages in the format you choose. The output is directed to stdout (normally your screen) and in the format you specify. The HLL object dumper does not process any files. You can specify the following formats: **html**, **rtf** or **text** (default). To create a file with the descriptions, you must redirect the output.

With the suboption **all**, the descriptions of all error messages are given. If you want the description of one or more selected error messages, you can specify the error message numbers, separated by commas, or you can specify a range.

Example

To display an explanation of message number 101, enter:

```
hldumparc --diag=101
```

This results in the following message and explanation:

```
F101: cannot create "<file>"

The output file or a temporary file could not be created.
Check if you have sufficient disk space and if you have
write permissions for the specified file.
```

To write an explanation of all errors and warnings in HTML format to file hldumperrors.html, use redirection and enter:

```
hldumparc --diag=html:all > hldumperrors.html
```

Related information

-

HLL object dumper option: --disassembly-intermix (-i)

Command line syntax

`--disassembly-intermix`[**`=`***flag*]

`-i`[*flag*]

You can specify the following format flags:

> **+/-single-line**   **s/S**   Force the insert to be limited to the first preceding source line.

Default: **`--disassembly-intermix=S`**

Description

With this option the disassembly is intermixed with HLL source code. The source is searched for as described with option **--source-lookup-path**

The **+single-line** sub-option forces the insert to be limited to the first preceding source line. With the **-single-line** sub-option all source lines that belong to the address are prefixed. For example comments are thus also visible. This is the default.

Example

```
hldumparc --disassembly-intermix --source-lookup-path=c:\mylib\src hello.elf
```

Related information

HLL object dumper option **--source-lookup-path**

HLL object dumper option: --disassembly-without-encoding (-r)

Command line syntax

```
--disassembly-without-encoding
```

```
-r
```

Description

With this option the address and encoding are not part of the disassembly of a code section. This is useful when you only want the disassembly part.

Example

```
hldumparc -F2 hello.elf

----------- Section dump ----------

                                         .section .text.hello.main, at(0x00000494)
00000494 f1 c0        main:              push_s %blink
00000496 c3 40 00 00 72 05                   mov_s %r0,1394

0000049c 00 50                           ld_s %r1,[%gp,0]
0000049e 4a 08 00 00                     bl      printf
000004a2 0c 70                           mov_s %r0,0
000004a4 d1 c0                           pop_s %blink
000004a6 e0 7e                           j_s [%blink]
                                         .endsec

hldumparc -F2 --disassembly-without-encoding hello.elf

----------- Section dump ----------

              .section .text.hello.main, at(0x00000494)
main:          push_s %blink
               mov_s %r0,1394

               ld_s %r1,[%gp,0]
               bl    printf
               mov_s %r0,0
               pop_s %blink
               j_s [%blink]
              .endsec
```

Related information

-

HLL object dumper option: --dump-format (-F)

Command line syntax

--dump-format[**=***flag*,...]

-F[*flag*]...

You can specify the following format flags:

+/-callgraph	**c/C**	Dump the call graph of the application.
+/-dump	**d/D**	Dump the contents of the sections in the object file. Code sections can be disassembled, data sections are dumped.
+/-debug-control-flow	**f/F**	Dump the debug control flow section.
+/-hllsymbols	**h/H**	List the high level language symbols, with address, size and type.
+/-modules	**m/M**	Print a list of modules found in object file.
+/-note	**n/N**	Dump all ELF `.note` sections.
+/-sections	**s/S**	Print a list of sections with start address, length and type.
+/-symbols	**y/Y**	List the low level symbols, with address and length (if known).
	0	Alias for **CDFHMNSY** (nothing)
	1	Alias for **CDFhMNSY** (only HLL symbols)
	2	Alias for **CdFHMNSY** (only section contents)
	3	Alias for **cdfhmnsy** (everything)

Default: **--dump-format=CdFhmnsy**

Description

With this option you can control which parts of the dump output you want to see.

1. Module list

2. Section list

3. Call graph using the DWARF debug info

4. Section dump (disassembly)

5. HLL symbol table

6. Assembly level symbol table

7. Note sections

8. Debug control flow section

By default, all parts are dumped, except for parts 3 and 8.

You can limit the number of sections that will be dumped with the options **--sections** and **--section-types**.

Related information

Section 8.4.2, *HLL Dump Output Format*

HLL object dumper option: --expand-symbols (-e)

Command line syntax

--expand-symbols[**=***flag*],...

-e[*flag*]...

You can specify one of the following flags:

+/-basic-types	**b/B**	Expand arrays with basic C types.
+/-fullpath	**f/F**	Include the full path to the field level.
+/-gap-info	**g/G**	Insert gap markers where data is not consecutive.
+/-nesting-indicator	**n/N**	Print nesting bars.

Default (no flags): **--expand-symbols=BFGN**

Description

With this option you specify that all struct, union and array symbols are expanded with their fields in the HLL symbol dump.

With **--expand-symbols=+basic-types**, HLL struct and union symbols are listed including all fields. Array members are expanded in one array member per line regardless of the HLL type. For the fields the types and names are indented with 2 spaces.

With **--expand-symbols=+fullpath**, all fields of structs and unions and all members of non-basic type arrays are expanded and prefixed with their parent's names.

With **--expand-symbols=+gap-info**, unused memory in complex data types (structures and unions) between data objects and between code objects is shown as $\{gap\}$ parts. This option is useful to optimize data memory usage. This option only works if debug information is available in the ELF file.

With **--expand-symbols=+nesting-indicator**, vertical bars (|) are shown to make it easier to see the expanded structs, unions and arrays.

Example

```
hldumparc -F1 hello.elf

---------- HLL symbol table ----------

Address    Size HLL Type            Name
00100008     20 struct             _dbg_request [dbg.c]
0010001c     80 static char        stdin_buf[80] [_iob.c]

hldumparc -e -F1 hello.elf

---------- HLL symbol table ----------
```

```
Address     Size HLL Type             Name
00100008     20 struct               _dbg_request [dbg.c]
00100008      4  int                 _errno
0010000c      1  enum                nr
00100010     12  union               u
00100010      4    struct              exit
00100010      4     int                  status
00100010      8    struct              open
00100010      4     const char        * pathname
00100014      2     unsigned short int  flags
   ...
0010001c     80 static char          stdin_buf[80] [_iob.c]

hldumparc -eb -F1 hello.elf

---------- HLL symbol table ----------

Address     Size HLL Type             Name
00100008     20 struct               _dbg_request [dbg.c]
00100008      4  int                 _errno
0010000c      1  enum                nr
00100010     12  union               u
00100010      4    struct              exit
00100010      4     int                  status
00100010      8    struct              open
00100010      4     const char        * pathname
00100014      2     unsigned short int  flags
   ...
0010001c     80 static char          stdin_buf[80] [_iob.c]
0010001c      1  char
0010001d      1  char
0010001e      1  char
  ...
0010006b      1  char

hldumparc -ef -F1 hello.elf

---------- HLL symbol table ----------

Address     Size HLL Type             Name
00100008     20 struct               _dbg_request [dbg.c]
00100008      4  int                 _dbg_request._errno
0010000c      1  enum                _dbg_request.nr
00100010     12  union               _dbg_request.u
00100010      4    struct              _dbg_request.u.exit
00100010      4     int                  _dbg_request.u.exit.status
00100010      8    struct              _dbg_request.u.open
00100010      4     const char        * _dbg_request.u.open.pathname
00100014      2     unsigned short int  _dbg_request.u.open.flags
   ...
0010001c     80 static char          stdin_buf[80] [_iob.c]
```

661

```
hldumparc -eg -F1 hello.elf

---------- HLL symbol table ----------

Address     Size HLL Type             Name
00100008     20 struct               _dbg_request [dbg.c]
00100008      4   int                  _errno
0010000c      1   enum                 nr
0010000d      3                        {gap}
00100010     12   union                u
00100010      4     struct               exit
00100010      4       int                  status
00100014      8                            {gap}
00100010      8     struct               open
00100010      4       const char         * pathname
00100014      2       unsigned short int   flags
00100016      2                            {gap}
00100018      4                          {gap}
   ...
0010001c     80 static char          stdin_buf[80] [_iob.c]

hldumparc -en -F1 hello.elf

---------- HLL symbol table ----------

Address     Size HLL Type             Name
00100008     20 struct               _dbg_request [dbg.c]
00100008      4 | int                  _errno
0010000c      1 | enum                 nr
00100010     12 | union                u
00100010      4 | | struct               exit
00100010      4 | | | int                  status
00100010      8 | | struct               open
00100010      4 | | | const char         * pathname
00100014      2 | | | unsigned short int   flags
   ...
0010001c     80 static char          stdin_buf[80] [_iob.c]
```

Related information

Section 8.4.2, *HLL Dump Output Format*

662

HLL object dumper option: --help (-?)

Command line syntax

`--help`

`-?`

Description

Displays an overview of all command line options.

Example

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

```
hldumparc -?
hldumparc --help
hldumparc
```

Related information

-

HLL object dumper option: --hex (-x)

Command line syntax

--hex

-x

Description

With this option you can control the way data sections and code sections are dumped. By default, the contents of data sections are represented by directives. A new directive will be generated for each symbol. ELF labels in the section are used to determine the start of a directive. ROM sections are represented with .db, .dh, .dw, .dd kind of directives, depending on the size of the data. RAM sections are represented with .ds directives, with a size operand depending on the data size. This can be either the size specified in the ELF symbol, or the size up to the next label. Code sections are dumped as disassembly.

With option **--hex**, no directives will be generated for ROM data sections and no disassembly dump will be done for code sections. Instead ROM data sections and code sections are dumped as hexadecimal code with ASCII translation. RAM sections will be represented with only a start address and a size indicator.

Example

```
hldumparc -F2 --section=.rodata.hello..2.str hello.elf

---------- Section dump ----------

        .section .data, '.rodata.hello..2.str', at(0x00000572)
        .db 48,65,6c,6c,6f,20,25,73,21,0a,00              ; Hello %s!..
        .endsec

hldumparc -F2 --section=.rodata.hello..2.str --hex hello.elf

---------- Section dump ----------

                          section 7 (.rodata.hello..2.str):
00000572 48 65 6c 6c 6f 20 25 73 21 0a 00               Hello %s!..
```

Related information

HLL object dumper option: --option-file (-f)

Command line syntax

--option-file=`file`**,...**

-f `file`**,...**

Description

This option is primarily intended for command line use. Instead of typing all options on the command line, you can create an option file which contains all options and flags you want to specify. With this option you specify the option file to the HLL object dumper.

Use an option file when the command line would exceed the limits of the operating system, or just to store options and save typing.

You can specify the option **--option-file** multiple times.

Format of an option file

• Multiple arguments on one line in the option file are allowed.

• To include whitespace in an argument, surround the argument with single or double quotes.

• If you want to use single quotes as part of the argument, surround the argument by double quotes and vise versa:

```
"This has a single quote ' embedded"

'This has a double quote " embedded'

'This has a double quote " and a single quote '"' embedded"
```

• When a text line reaches its length limit, use a **** to continue the line. Whitespace between quotes is preserved.

```
"This is a continuation \
line"

        -> "This is a continuation line"
```

• It is possible to nest command line files up to 25 levels.

Example

Suppose the file myoptions contains the following lines:

```
--symbols=hll
--class=code
hello.elf
```

Specify the option file to the HLL object dumper:

```
hldumparc --option-file=myoptions
```

This is equivalent to the following command line:

```
hldumparc --symbols=hll --class=code hello.elf
```

Related information

-

HLL object dumper option: --output (-o)

Command line syntax

--output=*file*

-o *file*

Description

By default, the HLL object dumper dumps the output on stdout. With this option you specify to dump the information in the specified file.

The default output format is text, but you can specify another output format with option **--output-type**.

Example

hldumparc --output=dump.txt hello.elf

The HLL object dumper dumps the output in file dump.txt.

Related information

HLL object dumper option **--output-type**

667

HLL object dumper option: --output-type (-T)

Command line syntax

`--output-type`[`=`*type*]

`-T`[*type*]

You can specify one of the following types:

text	**t**	Output human readable text.
xml	**x**	Output XML.

Default: `--output-type=text`

Description

With this option you can specify whether the output is formatted as plain text or as XML.

Related information

HLL object dumper option **--output**

HLL object dumper option: --sections (-s)

Command line syntax

`--sections=`*name*`,...`

`-s`*name*`,...`

Description

With this option you can restrict the output to the specified sections only. This option affects the following parts of the output:

Output part	Effect of --sections
Module list	Not restricted
Section list	Only lists the specified sections
Section dump	Only dumps the contents of the specified sections
HLL symbol table	Not restricted
Assembly level symbol table	Only lists symbols defined in the specified sections
Note sections	Not restricted

By default all sections are included.

Related information

Section 8.4.2, *HLL Dump Output Format*

HLL object dumper option: --source-lookup-path (-L)

Command line syntax

--source-lookup-path=*path*

-L*path*

Description

With this option you can specify an additional path where your source files are located. If you want to specify multiple paths, use the option **--source-lookup-path** for each separate path.

The order in which the HLL object dumper will search for source files when intermixed disassembly is used, is:

1. The path obtained from the HLL debug information.

2. The path that is specified with the option **--source-lookup-path**. If multiple paths are specified, the paths will be searched for in the order in which they are given on the command line.

Example

Suppose you call the HLL object dumper as follows:

```
hldumparc --disassembly-intermix --source-lookup-path=c:\mylib\src hello.elf
```

First the HLL object dumper looks in the directory found in the HLL debug information of file `hello.elf` for the location of the source file(s). If it does not find the file(s), it looks in the directory `c:\mylib\src`.

Related information

HLL object dumper option **--disassembly-intermix**

HLL object dumper option: --symbols (-S)

Command line syntax

--symbols[**=***type*]

-S[*type*]

You can specify one of the following types:

asm	**a**	Display assembly symbols in code dump.
hll	**h**	Display HLL symbols in code dump.
none	**n**	Display plain addresses in code dump.

Default: **--symbols=asm**

Description

With this option you can control symbolic information in the disassembly and data dump. For data sections this only applies to symbols used as labels at the data addresses. Data within the data sections will never be replaced with symbols.

Only symbols that are available in the ELF or DWARF information are used. If you build an application without HLL debug information the **--symbols=hll** option will result in the same output as with **--symbols=none**. The same applies to the **--symbols=asm** option when all symbols are stripped from the ELF file.

Example

```
hldumparc -F2 hello.elf

----------- Section dump ----------

                                        .section .text._Exit._Exit, at(0x000000d4)
000000d4 69 20 40 00  _Exit:            flag    1
000000d8 e0 7e                          j_s [%blink]
                                        .endsec

hldumparc --symbols=none -F2 hello.elf

----------- Section dump ----------

                                        .section .text._Exit._Exit, at(0x000000d4)
000000d4 69 20 40 00                    flag    1
000000d8 e0 7e                          j_s [%blink]
                                        .endsec
```

Related information

Section 8.4.2, *HLL Dump Output Format*

HLL object dumper option: --version (-V)

Command line syntax

```
--version
```

```
-V
```

Description

Display version information. The HLL object dumper ignores all other options or input files.

Related information

-

HLL object dumper option: --xml-base-filename (-X)

Command line syntax

`--xml-base-filename`

`-X`

Description

With this option the `<File name>` field in the XML output only contains the filename of the object file. By default, any path name, if present, is printed as well.

Example

`hldumparc --output-type=xml --output=hello.xml ../hello.elf`

The field `<File name="../hello.elf">` is used in `hello.xml`.

`hldumparc --output-type=xml --output=hello.xml -X ../hello.elf`

The field `<File name="hello.elf">` is used in `hello.xml`. The path is stripped from the filename.

Related information

HLL object dumper option **--output-type**

Chapter 11. Influencing the Build Time

In general many settings have influence on the build time of a project. Any change in the tool settings of your project source will have more or less impact on the build time. The following sections describe several issues that can have significant influence on the build time.

11.1. SFR File

SFR files can define such a large number of SFRs that compiling the SFR file alone already takes up a significant part of the build time. To reduce the build time:

* By default, the tools do not automatically include the SFR file. You should include the SFR file only in the source modules where the SFRs are used, with a `#include` directive. In Eclipse make sure that the automatic inclusion option is disabled. You can find this option on the "**C/C++ Compiler » Preprocessing**" and the "**Assembler » Preprocessing**" pages.

  When you include the SFR file in the source, be aware that the SFR files are in the `sfr` subdirectory of the include files, so you must use: `#include <sfr/regppu.sfr>`

11.2. MIL Linking

With MIL linking (see Section 4.6.1, *Generic Optimizations (frontend)*) it is possible to let the compiler apply optimizations application wide. This can yield significant optimization improvements, but the build times can also be significantly longer. MIL linking itself can require significant time, but also the changed build process implies longer build times. The MIL linking settings in Eclipse are:

* **Build for application wide optimizations (MIL linking)**

  This enables MIL linking. The build process changes: the C files are translated to intermediate code (MIL files) and the generated MIL files of the whole project are linked together by the C compiler. The next step depends on the setting of the option below.

* **Application wide optimization mode: Optimize more/Build slower**

  When this option is enabled, the compiler runs the code generator immediately on the completely linked MIL stream, which represents the entire application. This way the code generator can perform several optimizations, such as "code compaction", at application scope. But this also requires significantly more memory and requires more time to generate code. Besides that, it is no longer possible to do incremental builds. With each build the full MIL linking phase and code generation has to be done, even with the smallest change that would in a normal build (not MIL linking) require only a single module to be translated.

* **Application wide optimization mode: Optimize less/Build faster**

  When this option is disabled, the compiler splits the MIL stream after MIL linking in separate modules. This allows the code generation to be performed for the modified modules only, and will therefore be faster than with the other option enabled. Although the MIL stream is split in separate modules after MIL linking, it still may happen that modifying a single C source file results in multiple MIL files to be

675

compiled. This is a natural result of global optimizations, where the code generated for multiple modules was affected by the change.

In general, if you do not need code compaction, for example because you are optimizing fully for speed, it is recommended to choose **Optimize less/Build faster**.

11.3. Optimization Options

In general any optimization may require more work to be done by the compiler. But this does not mean that disabling all optimizations (level 0) gives the fastest compilation time. Disabling optimizations may result in more code being generated, resulting in more work for other parts of the compiler, like for example the register allocator.

11.4. Automatic Inlining

Automatic inlining is an optimization which can result in significant longer build time. The overall functions will get bigger, often making it possible to do more optimizations. But also often resulting in more registers to be in use in a function, giving the register allocation a tougher job.

11.5. Code Compaction

When you disable the code compaction optimization, the build times may be shorter. Certainly when MIL linking is used where the full application is passed as a single MIL stream to the code generation. Code compaction is however an optimization which can make a huge difference when optimizing for code size. When size matters it makes no sense to disable this option. When you choose to optimize for speed (**--tradeoff=0**) the code compaction is automatically disabled.

11.6. Header Files

Many applications include all header files in each module, often by including them all within a single include file. Processing header files takes time. It is a good programming practice to only include the header files that are really required in a module, because:

• it is clear what interfaces are used by a module

• an incremental build after modifying a header file results in less modules required to be rebuild

• it reduces compile time

11.7. Parallel Build

The make utility **amk**, which is used by Eclipse, has a feature to build jobs in parallel. This means that multiple modules can be compiled in parallel. With today's multi-core processors this means that each core can be fully utilized. In practice even on single core machines the compile time decreases when

using parallel jobs. On multi-core machines the build time even improves further when specifying more parallel jobs than the number of cores.

In Eclipse you can control the parallel build behavior:

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

    The Properties dialog appears.

2. In the left pane, select **C/C++ Build**.

    In the right pane the C/C++ Build page appears.

3. On the Behavior tab, select **Enable parallel build**.

4. You can specify the number of parallel jobs, or you can use an optimal number of jobs. In the last case, **amk** will fork as many jobs in parallel as cores are available.

Chapter 12. Libraries

This chapter contains an overview of all library functions that you can call in your C source. This includes all functions of the standard C library (ISO C11/C17) and some functions of the floating-point library.

Section 12.1, *Library Functions*, gives an overview of all library functions you can use, grouped per header file. A number of functions declared in `wchar.h` are parallel to functions in other header files. These are discussed together.

Section 12.2, *C Library Reentrancy*, gives an overview of which functions are reentrant and which are not.

C library / floating-point library / run-time library

The following libraries are included in the TASKING toolset for ARC/PPU. The control program **ccarc** automatically select the appropriate libraries depending on the specified options.

Libraries	Description
libc_fpu.a	C library with double-precision FPU instructions for ppu_tc45x, ppu_tc49x, ppu_tc49xn and ppu_tc4dx core architectures.
librt.a	Run-time library (contains other run-time functions needed by the C compiler)

For the C library also a MIL library variant is present (file with extension `.ma`).

Sources for the libraries are present in the directories `lib\src`, `lib\src.*` in the form of an executable. If you run the executable it will extract the sources in the corresponding directory. Note that under Windows you need to run the self extractor executables as Administrator.

C++ library

The TASKING C++ standard libraries are based on the LLVM libc++ libraries. libc++ is a multi-platform ISO C++ Standard Library implementation. It is a free, open-source product, which is delivered with the TASKING C++ compiler. The libc++ library supports standard templates, STL features and I/O streams as defined by the C++11 standard and later. This library has a few deviations from the C++11/C++14 standard as mentioned in Section 19.3, *Deviations from C++11/C++14*.

The libc++ libraries have been built with C++ compiler option **--c++=14**

The include files for the libc++ libraries are present in directory `include.cxx` relative to the product installation directory.

You can find more information on the libc++ library on the following site: http://libcxx.llvm.org/

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

Libraries	Description
libcp[x]_fpu.a	C++ run-time library with FPU instructions Optional letter: x = exception handling (default, without control program option **--no-exceptions**)
libcxx[x]_fpu.a	libc++ library with FPU instructions Optional letter: x = exception handling (default, without control program option **--no-exceptions**)

12.1. Library Functions

The tables in the sections below list all library functions, grouped per header file in which they are declared. Some functions are not completely implemented because their implementation depends on the context where your application will run. These functions are for example all I/O related functions. Where possible, these functions are implemented using file system simulation (FSS). This system can be used by the debugger to simulate an I/O environment which enables you to debug your application.

12.1.1. assert.h

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

For C11/C17 only, the following macro is defined:

```
#define static_assert _Static_assert
```

12.1.2. complex.h

The complex number *z* is also written as *x*+*y*i 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*, *function*`f`, *function*`l`. All long type functions, though declared in `complex.h`, are implemented as the `double` type variant which nearly always meets the requirement in embedded applications.

This implementation uses the obvious implementation for complex multiplication; and a more sophisticated implementation for division and absolute value calculations which handles underflow, overflow and infinities with more care. The ISO C99 `#pragma CX_LIMITED_RANGE` therefore has no effect.

Trigonometric functions

`csin`   `csinf`   `csinl`   Returns the complex sine of *z*.

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

Exponential and logarithmic functions

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

Power and absolute-value functions

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

Manipulation functions

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

12.1.3. 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
isalnum	iswalnum	Returns a non-zero value when c is an alphabetic character or a number ([A-Z][a-z][0-9]).
isalpha	iswalpha	Returns a non-zero value when c is an alphabetic character ([A-Z][a-z]).
isblank	iswblank	Returns a non-zero value when c is a blank character (tab, space...)
iscntrl	iswcntrl	Returns a non-zero value when c is a control character.
isdigit	iswditit	Returns a non-zero value when c is a numeric character ([0-9]).
isgraph	iswgraph	Returns a non-zero value when c is printable, but not a space.
islower	iswlower	Returns a non-zero value when c is a lowercase character ([a-z]).
isprint	iswprint	Returns a non-zero value when c is printable, including spaces.
ispunct	iswpunct	Returns a non-zero value when c is a punctuation character (such as '.', ',', '!').
isspace	iswspace	Returns a non-zero value when c is a space type character (space, tab, vertical tab, formfeed, linefeed, carriage return).
isupper	iswupper	Returns a non-zero value when c is an uppercase character ([A-Z]).
isxdigit	iswxdigit	Returns a non-zero value when c is a hexadecimal digit ([0-9][A-F][a-f]).
tolower	towlower	Returns c converted to a lowercase character if it is an uppercase character, otherwise c is returned.
toupper	towupper	Returns c converted to an uppercase character if it is a lowercase character, otherwise c is returned.
_tolower	–	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.
_toupper	–	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.
isascii		Returns a non-zero value when c is in the range of 0 and 127. This function is not defined in ISO C99.
toascii		Converts c to an ASCII value (strip highest bit). This function is not defined in ISO C99.

12.1.4. 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.

_dbg_trap	Low level function to trap debug events
_argcv(const char *buf,size_t size)	Low level function for command line argument passing

12.1.5. errno.h

`int errno`   External variable that holds implementation defined error codes.

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

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

Floating-point errors

`EDOM`	23	Argument too large
`ERANGE`	24	Result too large

Errors returned by printf/scanf

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

Encoding errors set by functions like fgetwc, getwc, mbrtowc, etc ...

`EILSEQ`	29	Invalid or incomplete multibyte or wide character

Errors returned by RTOS

`ECANCELED`	30	Operation canceled
`ENODEV`	31	No such device

12.1.6. fcntl.h

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

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

12.1.7. fenv.h

Contains mechanisms to control the floating-point environment.

fegetenv	Stores the current floating-point environment.
feholdexcept	Saves the current floating-point environment and installs an environment that ignores all floating-point exceptions.
fesetenv	Restores a previously saved (`fegetenv` or `feholdexcept`) floating-point environment.
feupdateenv	Saves the currently raised floating-point exceptions, restores a previously saved floating-point environment and finally raises the saved exceptions.
feclearexcept	Clears the current exception status flags corresponding to the flags specified in the argument.
fegetexceptflag	Stores the current setting of the floating-point status flags.
feraiseexcept	Raises the exceptions represented in the argument. As a result, other exceptions may be raised as well.
fesetexceptflag	Sets the current floating-point status flags.
fetestexcept	Returns the bitwise-OR of the exception macros corresponding to the exception flags which are currently set *and* are specified in the argument.

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

```
FE_DIVBYZERO    FE_INEXACT      FE_INVALID
FE_OVERFLOW     FE_UNDERFLOW    FE_ALL_EXCEPT
```

fegetround	Returns the current rounding direction, represented as one of the values of the rounding direction macros.
fesetround	Sets the current rounding directions.

For each supported rounding mode, a macro is defined. The following rounding mode macro is defined:

```
FE_TONEAREST
```

12.1.8. 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 12.1.15, *math.h and tgmath.h*.

The following functions are only available for ISO C90:

copysignf(float *f*,float *s*)	Copies the sign of the second argument *s* to the value of the first argument *f* and returns the result.
copysign(double *d*,double *s*)	Copies the sign of the second argument *s* to the value of the first argument *d* and returns the result.
isinff(float *f*)	Test the variable *f* on being an infinite (IEEE-754) value.
isinf(double *d*);	Test the variable *d* on being an infinite (IEEE-754) value.
isfinitef(float *f*)	Test the variable *f* on being a finite (IEEE-754) value.
isfinite(double *d*)	Test the variable *d* on being a finite (IEEE-754) value.
isnanf(float *f*)	Test the variable *f* on being NaN (Not a Number, IEEE-754) .
isnan(double *d*)	Test the variable *d* on being NaN (Not a Number, IEEE-754) .
scalbf(float *f*,int *p*)	Returns *f* * 2^*p* for integral values without computing 2^N.
scalb(double *d*,int *p*)	Returns *d* * 2^*p* for integral values without computing 2^N. (See also scalbn in Section 12.1.15, *math.h and tgmath.h*)

12.1.9. 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.

imaxabs(intmax_t j)	Returns the absolute value of j
imaxdiv(intmax_t numer, intmax_t denom)	Computes numer/denom and numer % denom. The result is stored in the quot and rem components of the imaxdiv_t structure type.
strtoimax(const char * restrict nptr, char ** restrict endptr, int base)	Convert string to maximum sized integer. (Compare strtoll)
strtoumax(const char * restrict nptr, char ** restrict endptr, int base)	Convert string to maximum sized unsigned integer. (Compare strtoull)
wcstoimax(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base)	Convert wide string to maximum sized integer. (Compare wcstoll)

```
wcstoumax(const wchar_t *      Convert wide string to maximum sized unsigned integer. (Compare
restrict nptr, wchar_t **      wcstoull)
restrict endptr, int base)
```

12.1.10. io.h

The header file `io.h` contains prototypes for low level I/O functions. This header file is not defined in ISO
C99.

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

12.1.11. 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  ^=
```

12.1.12. limits.h

Contains the sizes of integral types, defined as macros.

12.1.13. locale.h

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

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

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

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

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

```
struct lconv *leaveconv( void )
```

Returns a pointer to type `struct lconv` with values appropriate for the formatting of numeric quantities according to the rules of the current locale. The `struct lconv` in this header file is conforming the ISO standard.

12.1.14. 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 12.1.24, *stdlib.h and wchar.h*.

`malloc(size)`	Allocates space for an object with size *size*. The allocated space is not initialized. Returns a pointer to the allocated space.
`aligned_alloc(alignment, size)`	(C11/C17 only) Allocates space for an object whose alignment is specified by *alignment* and with size *size*. The allocated space is not initialized. Returns a pointer to the allocated space.
`calloc(nobj,size)`	Allocates space for n objects with size *size*. The allocated space is initialized with zeros. Returns a pointer to the allocated space.
`free(*ptr)`	Deallocates the memory space pointed to by *ptr* which should be a pointer earlier returned by the `malloc` or `calloc` function.
`realloc(*ptr,size)`	Deallocates the old object pointed to by *ptr* and returns a pointer to a new object with size *size*, while preserving its contents. If the new size is smaller than the old size, some contents at the end of the old region will be discarded. If the new size is larger than the old size, all of the old contents are preserved and any bytes in the new object beyond the size of the old object will have indeterminate values.

12.1.15. 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*, *function*f, *function*l. 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
`sin`	`sinf`	`sinl`	`sin`	Returns the sine of `x`.
`cos`	`cosf`	`cosl`	`cos`	Returns the cosine of `x`.
`tan`	`tanf`	`tanl`	`tan`	Returns the tangent of `x`.
`asin`	`asinf`	`asinl`	`asin`	Returns the arc sine $\sin^{-1}(x)$ of `x`.
`acos`	`acosf`	`acosl`	`acos`	Returns the arc cosine $\cos^{-1}(x)$ of `x`.
`atan`	`atanf`	`atanl`	`atan`	Returns the arc tangent $\tan^{-1}(x)$ of `x`.
`atan2`	`atan2f`	`atan2l`	`atan2`	Returns the result of: $\tan^{-1}(y/x)$.
`sinh`	`sinhf`	`sinhl`	`sinh`	Returns the hyperbolic sine of `x`.
`cosh`	`coshf`	`coshl`	`cosh`	Returns the hyperbolic cosine of `x`.
`tanh`	`tanhf`	`tanhl`	`tanh`	Returns the hyperbolic tangent of `x`.
`asinh`	`asinhf`	`asinhl`	`asinh`	Returns the arc hyperbolic sine of `x`.
`acosh`	`acoshf`	`acoshl`	`acosh`	Returns the non-negative arc hyperbolic cosine of `x`.
`atanh`	`atanhf`	`atanhl`	`atanh`	Returns the arc hyperbolic tangent of `x`.

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.
`expm1`	`expm1f`	`expm1l`	`expm1`	Returns the result of the exponential function e^x-1.
`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`.
`log2`	`log2f`	`log2l`	`log2`	Returns the base-2 logarithm of `x`. `x>0`.
`ilogb`	`ilogbf`	`ilogbl`	`ilogb`	Returns the signed exponent of x as an integer. `x>0`.
`logb`	`logbf`	`logbl`	`logb`	Returns the exponent of `x` as a signed integer in value in floating-point notation. `x > 0`.

frexp, ldexp, modf, scalbn, scalbln

math.h			tgmath.h	Description
frexp	frexpf	frexpl	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 $f*2^n = x$. Returns f, stores n.
ldexp	ldexpf	ldexpl	ldexp	Inverse of `frexp`. Returns the result of $x*2^n$. (x and n are both arguments).
modf	modff	modfl	–	Splits a float x into fraction f and integer n, so that: $|f| < 1.0$ and $f+n=x$. Returns f, stores n.
scalbn	scalbnf	scalbnl	scalbn	Computes the result of $x*\texttt{FLT\_RADIX}^n$. efficiently, not normally by computing $\texttt{FLT\_RADIX}^n$ explicitly.
scalbln	scalblnf	scalblnl	scalbln	Same as `scalbn` 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	rintf	rintl	rint	Returns the rounded integer value as an `int` according to the current rounding direction. See `fenv.h`.
lrint	lrintf	lrintl	lrint	Returns the rounded integer value as a `long int` according to the current rounding direction. See `fenv.h`.
llrint	llrintf	llrintl	llrint	Returns the rounded integer value as a `long long int` according to the current rounding direction. See `fenv.h`.
nearbyint	nearbyintf	nearbyintl	nearbyint	Returns the rounded integer value as a floating-point according to the current rounding direction. See `fenv.h`.
round	roundf	roundl	round	Returns the nearest integer value of x as int.
lround	lroundf	lroundl	lround	Returns the nearest integer value of x as long int.
llround	lroundf	llroundl	llround	Returns the nearest integer value of x as long long int.
trunc	truncf	truncl	trunc	Returns the truncated integer value x.

Remainder after division

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

Power and absolute-value functions

math.h			tgmath.h	Description
cbrt	cbrtf	cbrtl	cbrt	Returns the real cube root of x ($=x^{1/3}$).
fabs	fabsf	fabsl	fabs	Returns the absolute value of x ($\lvert x \rvert$). (abs, labs, llabs, div, ldiv, lldiv are defined in stdlib.h)
fma	fmaf	fmal	fma	Floating-point multiply add. Returns $x*y+z$.
hypot	hypotf	hypotl	hypot	Returns the square root of x^2+y^2.
pow	powf	powl	power	Returns x raised to the power y (x^y).
sqrt	sqrtf	sqrtl	sqrt	Returns the non-negative square root of x. x 0.

Manipulation functions: copysign, nan, nextafter, nexttoward

math.h			tgmath.h	Description
copysign	copysignf	copysignll	copysign	Returns the value of x with the sign of y.
nan	nanf	nanl	–	Returns a quiet NaN, if available, with content indicated through *tagp*.
nextafter	nextafterf	nextafterl	nextafter	Returns the next representable value in the specified format after x in the direction of y. Returns y is $x=y$.
nexttoward	nexttowardf	nexttowardl	nexttoward	Same as nextafter, except that the second argument in all three variants is of type long double. Returns y if $x=y$.

Positive difference, maximum, minimum

math.h			tgmath.h	Description
fdim	fdimf	fdiml	fdim	Returns the positive difference between: $\lvert x-y \rvert$.
fmax	fmaxf	fmaxl	fmax	Returns the maximum value of their arguments.
fmin	fminf	fminl	fmin	Returns the minimum value of their arguments.

Error and gamma

math.h			tgmath.h	Description
erf	erff	erfl	erf	Computes the error function of x.
erfc	erfcf	erfcl	erc	Computes the complementary error function of x.
lgamma	lgammaf	lgammal	lgamma	Computes the $*\log_e \lvert \Gamma(x) \rvert$
tgamma	tgammaf	tgammal	tgamma	Computes $\Gamma(x)$

Comparison macros

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

math.h	tgmath.h	Description
isgreater	–	Returns the value of `(x) > (y)`
isgreaterequal	–	Returns the value of `(x) >= (y)`
isless	–	Returns the value of `(x) < (y)`
islessequal	–	Returns the value of `(x) <= (y)`
islessgreater	–	Returns the value of `(x) < (y) || (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 therefore 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 infinite 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.

12.1.16. setjmp.h

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

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

12.1.17. 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 behavior is used
SIG_IGN	The signal is ignored

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

12.1.18. stdalign.h

This C11/C17 header file contains the following macro definitions about alignment:

```
#define alignas _Alignas
#define __alignas_is_defined    1

#define alignof _Alignof
#define __alignof_is_defined    1
```

12.1.19. stdarg.h

The facilities in this header file gives you a portable way to access variable arguments lists, such as needed for as `fprintf` and `vfprintf`. `va_copy` is new in ISO C99. This header file contains the following macros:

va_arg(va_list ap,type)	Returns the value of the next argument in the variable argument list. Its return type has the type of the given argument `type`. A next call to this macro will return the value of the next argument.

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

12.1.20. stdbool.h

This header file contains the following macro definitions. These names for boolean type and values are consistent with C++. You are allowed to `#undefine` or redefine the macros below.

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

12.1.21. stddef.h

This header file defines the types for common use:

`ptrdiff_t`	Signed integer type of the result of subtracting two pointers.
`size_t`	Unsigned integral type of the result of the `sizeof` operator.
`wchar_t`	Integer type to represent character codes in large character sets.

Besides these types, the following macros are defined:

`NULL`	Expands to the null pointer constant `(void *) 0`.
`offsetof(_type, _member)`	Expands to an integer constant expression with type `size_t` that is the offset in bytes of `_member` within structure type `_type`.

12.1.22. stdint.h

See Section 12.1.9, *inttypes.h and stdint.h*

12.1.23. stdio.h and wchar.h

Types

The header file `stdio.h` contains functions for performing input and output. A number of functions also have a parallel wide character function or macro, defined in `wchar.h`. The header file `wchar.h` also includes `stdio.h`.

In the C language, many I/O facilities are based on the concept of streams. The `stdio.h` header file defines the data type **FILE** which holds the information about a stream. A `FILE` object is created with the function `fopen`. The pointer to this object is used as an argument in many of the in this header file. The `FILE` object can contain the following information:

- the current position within the stream

- pointers to any associated buffers

- indications of for read/write errors

- end of file indication

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

Macros

stdio.h	Description
NULL	Expands to the null pointer constant `(void *) 0`.
BUFSIZ	Size of the buffer used by the `setbuf`/`setvbuf` function: 512
EOF	End of file indicator. Expands to -1.
WEOF	End of file indicator. Expands to UINT_MAX (defined in `limits.h`) NOTE: WEOF need not to be a negative number as long as its value does not correspond to a member of the wide character set. (Defined in `wchar.h`).
FOPEN_MAX	Number of files that can be opened simultaneously: 10
FILENAME_MAX	Maximum length of a filename: 100
_IOFBF _IOLBF _IONBF	Expand to an integer expression, suitable for use as argument to the `setvbuf` function.
L_tmpnam	Size of the string used to hold temporary file names: 8 (tmp*xxxxx*)
TMP_MAX	Maximum number of unique temporary filenames that can be generated: 0x8000
SEEK_CUR SEEK_END SEEK_SET	Expand to an integer expression, suitable for use as the third argument to the `fseek` function.
stderr stdin stdout	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
fopen(*name*,*mode*)	Opens a file for a given mode. Available modes are:

stdio.h	Description
	`"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
	(FSS implementation)
`fclose(name)`	Flushes the data stream and closes the specified file that was previously opened with fopen. (*FSS implementation*)
`fflush(name)`	If stream is an output stream, any buffered but unwritten date is written. Else, the effect is undefined. (*FSS implementation*)
`freopen(name,mode, stream)`	Similar to fopen, but rather than generating a new value of type FILE *, the existing value is associated with a new stream. (*FSS implementation*)
`setbuf(stream,buffer)`	If buffer is NULL, buffering is turned off for the stream. Otherwise, setbuf is equivalent to: `(void) setvbuf(stream,buffer,_IOFBF,BUFSIZ)`.
`setvbuf(stream,buffer,mode, size)`	Controls buffering for the *stream*; this function must be called before reading or writing. *Mode* can have the following values: `_IOFBF` causes full buffering `_IOLBF` causes line buffering of text files `_IONBF` causes no buffering. If *buffer* is not NULL, it will be used as a buffer; otherwise a buffer will be allocated. *size* determines the buffer size.

Formatted input/output

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

• Flags (in any order):

    –   specifies left adjustment of the converted argument.

    +   a number is always preceded with a sign character.
            + has higher precedence than `space`.

    `space` a negative number is preceded with a sign, positive numbers with a space.

    `0`   specifies padding to the field width with zeros (only for numbers).

    #   specifies an alternate output form. For o, the first digit will be zero. For x or X, "0x" and "0X" will be prefixed to the number. For e, E, f, g, G, the output always contains a decimal point, trailing zeros are not removed.

- A number specifying a minimum field width. The converted argument is printed in a field with at least the length specified here. If the converted argument has fewer characters than specified, it will be padded at the left side (or at the right when the flag '-' was specified) with spaces. Padding to numeric fields will be done with zeros when the flag '0' is also specified (only when padding left). Instead of a numeric value, also '*' may be specified, the value is then taken from the next argument, which is assumed to be of type `int`.

- A period. This separates the minimum field width from the precision.

- A number specifying the maximum length of a string to be printed. Or the number of digits printed after the decimal point (only for floating-point conversions). Or the minimum number of digits to be printed for an integer conversion. Instead of a numeric value, also '*' may be specified, the value is then taken from the next argument, which is assumed to be of type `int`.

- A length modifier 'h', 'hh', '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, 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
%	No argument is converted, a '%' is printed.

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.
f, 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 value which must be entered without 0x- prefix.

Character	Scanned as
[...]	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.

scanf conversion characters

stdio.h	wchar.h	Description
`fscanf(`*stream*`,` *format*`, ...)`	`fwscanf(`*stream*`,` *format*`, ...)`	Performs a formatted read from the given *stream*. Returns the number of items converted successfully. (*FSS implementation*)
`scanf(`*format*`,...)`	`wscanf(`*format*`, ...)`	Performs a formatted read from `stdin`. Returns the number of items converted successfully. (*FSS implementation*)
`sscanf(*s,` *format*`, ...)`	`swscanf(*s,` *format*`, ...)`	Performs a formatted read from the string *s*. Returns the number of items converted successfully.
`vfscanf(`*stream*`,` *format*`,` *arg*`)`	`vfwscanf(`*stream*`,` *format*`,` *arg*`)`	Same as `fscanf`/`fwscanf`, but extra arguments are given as variable argument list *arg*. (See Section 12.1.19, *stdarg.h*)
`vscanf(`*format*`,` *arg*`)`	`vwscanf(`*format*`,` *arg*`)`	Same as `sscanf`/`swscanf`, but extra arguments are given as variable argument list *arg*. (See Section 12.1.19, *stdarg.h*)
`vsscanf(*s,` *format*`,` *arg*`)`	`vswscanf(*s,` *format*`,` *arg*`)`	Same as `scanf`/`wscanf`, but extra arguments are given as variable argument list *arg*. (See Section 12.1.19, *stdarg.h*)
`fprintf(`*stream*`,` *format*`, ...)`	`fwprintf(`*stream*`,` *format*`, ...)`	Performs a formatted write to the given *stream*. Returns EOF/WEOF on error. (*FSS implementation*)
`printf(`*format*`, ...)`	`wprintf(`*format*`, ...)`	Performs a formatted write to the stream `stdout`. Returns EOF/WEOF on error. (*FSS implementation*)
`sprintf(*s,` *format*`, ...)`	–	Performs a formatted write to string *s*. Returns EOF/WEOF on error.
`snprintf(*s,` *n*`,` *format*`, ...)`	`swprintf(*s,` *n*`,` *format*`, ...)`	Same as `sprintf`, but n specifies the maximum number of characters (including the terminating null character) to be written.
`vfprintf(`*stream*`,` *format*`,` *arg*`)`	`vfwprintf(`*stream*`,` *format*`,` *arg*`)`	Same as `fprintf`/`fwprintf`, but extra arguments are given as variable argument list *arg*. (See Section 12.1.19, *stdarg.h*) (*FSS implementation*)

stdio.h	wchar.h	Description
vprintf(*format*, *arg*)	vwprintf(*format*, *arg*)	Same as printf/wprintf, but extra arguments are given as variable argument list *arg*. (See Section 12.1.19, *stdarg.h*) (*FSS implementation)*
vsprintf(**s*, *format*, *arg*)	vswprintf(**s*, *format*, *arg*)	Same as sprintf/swprintf, but extra arguments are given as variable argument list *arg*. (See Section 12.1.19, *stdarg.h*)

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 indirectly, 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 1.8, *Pragmas to Control the Compiler*.

Character input/output

stdio.h	wchar.h	Description
fgetc(*stream*)	fgetwc(*stream*)	Reads one character from *stream*. Returns the read character, or EOF/WEOF on error. (*FSS implementation)*
getc(*stream*)	getwc(*stream*)	Same as fgetc/fgetwc except that is implemented as a macro. (*FSS implementation)* NOTE: Currently #defined as getchar()/getwchar() because FILE I/O is not supported. Returns the read character, or EOF/WEOF on error.

stdio.h	wchar.h	Description
`getchar(stdin)`	`getwchar(stdin)`	Reads one character from the `stdin` stream. Returns the character read or EOF/WEOF on error. Implemented as macro. (*FSS implementation*)
`fgets(*s, n, stream)`	`fgetws(*s, n, stream)`	Reads at most the next *n*-1 characters from the *stream* into array *s* until a newline is found. Returns s or NULL or EOF/WEOF on error. (*FSS implementation*)
`gets(*s)`	–	(C90/C99 only) Reads characters from the `stdin` stream into array *s* until end-of-file is encountered or a newline is found. The newline is replaced by a NULL character. Returns *s* or NULL on EOF. (*FSS implementation*)
`ungetc(c, stream)`	`ungetwc(c, stream)`	Pushes character *c* back onto the input *stream*. Returns EOF/WEOF on error.
`fputc(c, stream)`	`fputwc(c, stream)`	Put character *c* onto the given *stream*. Returns EOF/WEOF on error. (*FSS implementation*)
`putc(c, stream)`	`putwc(c, stream)`	Same as `fpuc`/`fputwc` except that is implemented as a macro. (*FSS implementation*)
`putchar(c, stdout)`	`putwchar(c, stdout)`	Put character *c* onto the `stdout` stream. Returns EOF/WEOF on error. Implemented as macro. (*FSS implementation*)
`fputs(*s, stream)`	`fputws(*s, stream)`	Writes string *s* to the given *stream*. Returns EOF/WEOF on error. (*FSS implementation*)
`puts(*s)`	–	Writes string *s* to the `stdout` stream. Returns EOF/WEOF on error. (*FSS implementation*)

Direct input/output

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

Random access

stdio.h	Description
`fseek(stream, offset, origin)`	Sets the position indicator for *stream*. (*FSS implementation*)

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

SEEK_SET 0 *offset* characters from the beginning of the file
SEEK_CUR 1 *offset* characters from the current position in the file
SEEK_END 2 *offset* characters from the end of the file

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

Operations on files

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

Error handling

stdio.h	Description
clearerr(*stream*)	Clears the end of file and error indicators for stream.
ferror(*stream*)	Returns a non-zero value if the error indicator for stream is set.
feof(*stream*)	Returns a non-zero value if the end of file indicator for stream is set.
perror(**s*)	Prints *s* and the error message belonging to the integer errno. (See Section 12.1.5, *errno.h*)

12.1.24. stdlib.h and wchar.h

The header file `stdlib.h` contains general utility functions which fall into the following categories (Some have parallel wide character, declared in `wchar.h`)

- Numeric conversions

- Random number generation

- Memory management

- Environment communication

- Searching and sorting

- Integer arithmetic

- Multibyte/wide character and string conversions.

Macros

`EXIT_SUCCES`
`0`
`EXIT_FAILURE`
`1`
	Predefined exit codes that can be used in the `exit` function.

`RAND_MAX` `32767`	Highest number that can be returned by the `rand`/`srand` function.
`MB_CUR_MAX 1`	Maximum number of bytes in a multibyte character for the extended character set specified by the current locale (category LC_CTYPE, see Section 12.1.13, *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		wchar.h	
`float`	`strtof(*s,**endp)`	`float`	`wcstof(*s,**endp)`
`double`	`strtod(*s,**endp)`	`double`	`wcstod(*s,**endp)`
`long double`	`strtold(*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	wchar.h
`long strtol (*s,**endp,base)`	`long wcstol (*s,**endp,base)`
`long long strtoll` `        (*s,**endp,base)`	`long long wcstoll` `            (*s,**endp,base)`
`unsigned long strtoul` `        (*s,**endp,base)`	`unsigned long wcstoul` `            (*s,**endp,base)`
`unsigned long long strtoull` `        (*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.
`aligned_alloc(`*alignment,* *size*`)`	(C11/C17 only) Allocates space for an object whose alignment is specified by *alignment* and with size *size*. The allocated space is not initialized. Returns a pointer to the allocated space.
`calloc(`*nobj*`,`*size*`)`	Allocates space for n objects with size *size*. The allocated space is initialized with zeros. Returns a pointer to the allocated space.
`free(*`*ptr*`)`	Deallocates the memory space pointed to by *ptr* which should be a pointer earlier returned by the `malloc` or `calloc` function.
`realloc(*`*ptr*`,`*size*`)`	Deallocates the old object pointed to by *ptr* and returns a pointer to a new object with size *size*, while preserving its contents. If the new size is smaller than the old size, some contents at the end of the old region will be discarded. If the new size is larger than the old size, all of the old contents are preserved and any bytes in the new object beyond the size of the old object will have indeterminate values.

Environment communication

`abort()`	Causes abnormal program termination. If the signal SIGABRT is caught, the signal handler may take over control. (See Section 12.1.17, *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_SUCCES 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.
`at_quick_exit(*`*`func`*`)`	(C11/C17 only) Registers the function pointed to by *func* to be called (without arguments) when `quick_exit` is called. Returns zero if the registration succeeds, nonzero if it fails.
`quick_exit(`*`status`*`)`	(C11/C17 only) Causes normal program termination. Calls all functions registered by the `at_quick_exit` function, in the reverse order of their registration, and then calls `_Exit`.
`getenv(*`*`s`*`)`	Searches an environment list for a string *s*. Returns a pointer to the contents of *s*. NOTE: this function is not implemented because there is no OS.
`system(*`*`s`*`)`	Passes the string *s* to the environment for execution. NOTE: this function is not implemented because there is no OS.

Searching and sorting

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

`qsort(*`*`base`*`, `*`n`*`,` `*`size`*`, *`*`cmp`*`)`	This function sorts an array of *n* members using the quick sort algorithm. The initial base of the array is given by *base*. The size of each member is specified by *size*. The array is sorted in ascending order, according to the results of the function pointed to by *cmp*.

Integer arithmetic

`int abs(`*`j`*`)`
`long labs(`*`j`*`)`
`long long llabs(`*`j`*`)`
	Compute the absolute value of an `int`, `long int`, and `long long int` *j* respectively.

`div_t div(`*`x`*`,`*`y`*`)`
`ldiv_t ldiv(`*`x`*`,`*`y`*`)`
`lldiv_t lldiv(`*`x`*`,`*`y`*`)`
	Compute *x/y* and *x%y* in a single operation. *X* and *y* have respectively type `int`, `long int` and `long long int`. The result is stored in the members `quot` and `rem` of `struct div_t`, `ldiv_t` and `lldiv_t` which have the same types.

Multibyte/wide character and string conversions

`mblen(*`*`s`*`,`*`n`*`)`	Determines the number of bytes in the multibyte character pointed to by *s*. At most *n* characters will be examined. (See also `mbrlen` in Section 12.1.30, *wchar.h*).
`mbtowc(*`*`pwc`*`,*`*`s`*`,`*`n`*`)`	Converts the multibyte character in *s* to a wide character code and stores it in pwc. At most *n* characters will be examined.

`wctomb(*s,wc)`		Converts the wide character *wc* into a multibyte representation and stores it in the string pointed to by *s*. At most MB_CUR_MAX characters are stored.
`mbstowcs(*pwcs,*s,n)`		Converts a sequence of multibyte characters in the string pointed to by *s* into a sequence of wide characters and stores at most *n* wide characters into the array pointed to by *pwcs*. (See also `mbsrtowcs` in Section 12.1.30, *wchar.h*).
`wcstombs(*s,*pwcs,n)`		Converts a sequence of wide characters in the array pointed to by *pwcs* into multibyte characters and stores at most *n* multibyte characters into the string pointed to by *s*. (See also wc`srtowmb` in Section 12.1.30, *wchar.h*).

12.1.25. stdnoreturn.h

This C11/C17 header file contains the following macro definition:

```
#define noreturn _Noreturn
```

12.1.26. 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

string.h	wchar.h	Description
`memcpy(*s1,*s2,n)`	`wmemcpy(*s1,*s2,n)`	Copies *n* characters from *s2 into *s1 and returns *s1. If *s1 and *s2 overlap the result is undefined.
`memmove(*s1,*s2,n)`	`wmemmove(*s1,*s2,n)`	Same as `memcpy`, but overlapping strings are handled correctly. Returns *s1.
`strcpy(*s1,*s2)`	`wcscpy(*s1,*s2)`	Copies *s2 into *s1 and returns *s1. If *s1 and *s2 overlap the result is undefined.
`strncpy(*s1,*s2,n)`	`wcsncpy(*s1,*s2,n)`	Copies not more than *n* characters from *s2 into *s1 and returns *s1. If *s1 and *s2 overlap the result is undefined.
`strcat(*s1,*s2)`	`wcscat(*s1,*s2)`	Appends a copy of *s2 to *s1 and returns *s1. If *s1 and *s2 overlap the result is undefined.
`strncat(*s1,*s2,n)`	`wcsncat(*s1,*s2,n)`	Appends not more than *n* characters from *s2 to *s1 and returns *s1. If *s1 and *s2 overlap the result is undefined.

Comparison functions

string.h	wchar.h	Description
`memcmp(*s1,*s2,n)`	`wmemcmp(*s1,*s2,n)`	Compares the first *n* characters of *s1 to the first *n* characters of *s2. Returns < 0 if *s1 < *s2, 0 if *s1 = = *s2, or > 0 if *s1 > *s2.

string.h	wchar.h	Description
strcmp(*s1,*s2)	wcscmp(*s1,*s2)	Compares string *s1 to *s2. Returns < 0 if *s1 < *s2, 0 if *s1 = = *s2, or > 0 if *s1 > *s2.
strncmp(*s1,*s2,n)	wcsncmp(*s1,*s2,n)	Compares the first n characters of *s1 to the first n characters of *s2. Returns < 0 if *s1 < *s2, 0 if *s1 = = *s2, or > 0 if *s1 > *s2.
strcoll(*s1,*s2)	wcscoll(*s1,*s2)	Performs a local-specific comparison between string *s1 and string *s2 according to the LC_COLLATE category of the current locale. Returns < 0 if *s1 < *s2, 0 if *s1 = = *s2, or > 0 if *s1 > *s2. (See Section 12.1.13, *locale.h*)
strxfrm(*s1,*s2,n)	wcsxfrm(*s1,*s2,n)	Transforms (a local) string *s2 so that a comparison between transformed strings with strcmp gives the same result as a comparison between non-transformed strings with strcoll. Returns the transformed string *s1.

Search functions

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

Miscellaneous functions

string.h	wchar.h	Description
memset(*s,c,n)	wmemset(*s,c,n)	Fills the first n bytes of *s with character c and returns *s.

string.h	wchar.h	Description
strerror(*errno*)	–	Typically, the values for errno come from int errno. This function returns a pointer to the associated error message. (See also Section 12.1.5, *errno.h*)
strlen(**s*)	wcslen(**s*)	Returns the length of string **s*.

12.1.27. time.h and wchar.h

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

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

The type `struct tm` below is defined according to ISO C99 with one exception: this implementation does not support leap seconds. The `struct tm` type is defines as follows:

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

Time manipulation

clock	Returns the application's best approximation to the processor time used by the program since it was started. This low-level routine reads the 64-bit real-time counter (RTC). To determine the time in seconds, the result of clock should be divided by the value defined by CLOCKS_PER_SEC. This value is hard-coded to 500000000 (500MHz).
difftime(*t1*,*t0*)	Returns the difference *t1-t0* in seconds.
mktime(tm **tp*)	Converts the broken-down time in the structure pointed to by *tp*, to a value of type time_t. The return value has the same encoding as the return value of the time function.
time(**timer*)	Returns the current calendar time. This value is also assigned to *timer.

Time conversion

`asctime(tm *`*tp*`)`	Converts the broken-down time in the structure pointed to by *tp* into a string in the form `Mon Feb 04 16:15:14 2013\n\0`. Returns a pointer to this string.
`ctime(*`*timer*`)`	Converts the calender time pointed to by *timer* to local time in the form of a string. This is equivalent to: `asctime(localtime(timer))`
`gmtime(*`*timer*`)`	Converts the calender time pointed to by *timer* to the broken-down time, expressed as UTC. Returns a pointer to the broken-down time.
`localtime(*`*timer*`)`	Converts the calendar time pointed to by *timer* to the broken-down time, expressed as local time. Returns a pointer to the broken-down time.

Formatted time

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

time.h	wchar.h
`strftime(*s,`*smax*`,*`*fmt*`,tm *`*tp*`)`	`wcsftime(*s,`*smax*`,*`*fmt*`,tm *`*tp*`)`

Formats date and time information from `struct tm *`*tp* into *`*s` according to the specified format *`*fmt*. No more than *smax* characters are placed into *`*s*. The formatting of `strftime` is locale-specific using the `LC_TIME` category (see Section 12.1.13, *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   locale-specific date and time representation (same as `%a %b %e %T %Y`)

%C   last two digits of the year

%d   day of the month (01-31)

%D   same as `%m/%d/%y`

%e   day of the month (1-31), with single digits preceded by a space

%F   ISO 8601 date format: `%Y-%m-%d`

%g   last two digits of the week based year (00-99)

%G   week based year (0000–9999)

%h   same as %b

%H   hour, 24-hour clock (00-23)

%I   hour, 12-hour clock (01-12)

%j   day of the year (001-366)

%m   month (01-12)

%M   minute (00-59)

%n   replaced by newline character

%p   locale's equivalent of AM or PM

%r   locale's 12-hour clock time; same as `%I:%M:%S %p`

%R   same as `%H:%M`

%S   second (00-59)

%t   replaced by horizontal tab character

%T   ISO 8601 time format: `%H:%M:%S`

%u   ISO 8601 weekday number (1-7), Monday as first day of the week

%U   week number of the year (00-53), week 1 has the first Sunday

%V   ISO 8601 week number (01-53) in the week-based year

%w   weekday (0-6, Sunday is 0)

%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

%%   %

12.1.28. uchar.h

The C11/C17 header file `uchar.h` declares types and functions for manipulating Unicode characters.

This header file declares the types:

`char16_t`   Unsigned integer type used for 16-bit characters.

`char32_t`   Unsigned integer type used for 32-bit characters.

`size_t`   Unsigned integer type of the result of the `sizeof` operator.

`wchar_t`   Integer type to represent character codes in large character sets.

The functions perform conversions between multibyte characters and Unicode characters. In these functions, *ps* points to struct `mbstate_t` which holds the conversion state information necessary to convert between sequences of multibyte characters and Unicode characters:

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

```
                                            character <=> multibyte
                                            conversion */
} mbstate_t;
```

mbrtoc16(*pc16,*s,n,*ps)	Converts a multibyte character *s to a 16-bit character *pc16 according to conversion state ps.
c16rtomb(*s,c16,*ps)	Converts a 16-bit character c16 to a multibyte character according to conversion state ps and stores the multibyte character in *s.
mbrtoc32(*pc32,*s,n,*ps)	Converts a multibyte character *s to a 32-bit character *pc32 according to conversion state ps.
c32rtomb(*s,c32,*ps)	Converts a 32-bit character c32 to a multibyte character according to conversion state ps and stores the multibyte character in *s.

12.1.29. unistd.h

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

access(*name,mode)	Use file system simulation to check the permissions of a file on the host. mode specifies the type of access and is a bit pattern constructed by a logical OR of the following values:

R_OK	Checks read permission.
W_OK	Checks write permission.
X_OK	Checks execute (search) permission.
F_OK	Checks to see if the file exists.

(*FSS implementation*)

chdir(*path)	Use file system simulation to change the current directory on the host to the directory indicated by path. (*FSS implementation*)
close(fd)	File close function. The given file descriptor should be properly closed. This function calls _close(). (*FSS implementation*)
getcwd(*buf,size)	Use file system simulation to retrieve the current directory on the host. Returns the directory name. (*FSS implementation*)
lseek(fd,offset,whence)	Moves read-write file offset. Calls _lseek(). (*FSS implementation*)
read(fd,*buff,cnt)	Reads a sequence of characters from a file. This function calls _read(). (*FSS implementation*)
stat(*name,*buff)	Use file system simulation to stat() a file on the host platform. (*FSS implementation*)
lstat(*name,*buff)	This function is identical to stat(), except in the case of a symbolic link, where the link itself is 'stat'-ed, not the file that it refers to. (*Not implemented*)
fstat(fd,*buff)	This function is identical to stat(), except that it uses a file descriptor instead of a name. (*Not implemented*)

`unlink(*name)`	Removes the named file, so that a subsequent attempt to open it fails. (*FSS implementation)*
`write(fd,*buff,cnt)`	Write a sequence of characters to a file. Calls `_write()`. (*FSS implementation)*

12.1.30. wchar.h

Many functions in `wchar.h` represent the wide character variant of other functions so these are discussed together. (See Section 12.1.23, *stdio.h and wchar.h*, Section 12.1.24, *stdlib.h and wchar.h*, Section 12.1.26, *string.h and wchar.h* and Section 12.1.27, *time.h and wchar.h*).

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

```
typedef struct
{
    wchar_t         wc_value;  /* wide character value solved
                                  so far */
    unsigned short  n_bytes;   /* number of bytes of solved
                                  multibyte */
    unsigned short  encoding;  /* encoding rule for wide
                                  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, multibyte characters are 1 byte long (MB_CUR_MAX and MB_LEN_MAX are defined as 1) and this will never occur.

`mbsinit(*ps)`	Determines whether the object pointed to by *ps*, is an initial conversion state. Returns a non-zero value if so.
`mbsrtowcs(*pwcs,**src,n,*ps)`	Restartable version of `mbstowcs`. See Section 12.1.24, *stdlib.h and wchar.h*. The initial conversion state is specified by *ps*. The input sequence of multibyte characters is specified indirectly by *src*.
`wcsrtombs(*s,**src,n,*ps)`	Restartable version of `wcstombs`. See Section 12.1.24, *stdlib.h and wchar.h*. The initial conversion state is specified by *ps*. The input wide string is specified indirectly by *src*.
`mbrtowc(*pwc,*s,n,*ps)`	Converts a multibyte character *s to a wide character *pwc according to conversion state *ps*. See also `mbtowc` in Section 12.1.24, *stdlib.h and wchar.h*.
`wcrtomb(*s,wc,*ps)`	Converts a wide character *wc* to a multibyte character according to conversion state *ps* and stores the multibyte character in *s*.
`btowc(c)`	Returns the wide character corresponding to character *c*. Returns WEOF on error.

`wctob(c)`	Returns the multibyte character corresponding to the wide character *c*. The returned multibyte character is represented as one byte. Returns EOF on error.
`mbrlen(*s,n,*ps)`	Inspects up to *n* bytes from the string **s* to see if those characters represent valid multibyte characters, relative to the conversion state held in **ps*.

12.1.31. wctype.h

Most functions in `wctype.h` represent the wide character variant of functions declared in `ctype.h` and are discussed in Section 12.1.3, *ctype.h and wctype.h*. In addition, this header file provides extensible, locale specific functions and wide character classification.

`wctype(*property)`	Constructs a value of type `wctype_t` that describes a class of wide characters identified by the string **property*. If property identifies a valid class of wide characters according to the LC_TYPE category (see Section 12.1.13, *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
`iswalnum(wc)`	`iswctype(wc,wctype("alnum"))`
`iswalpha(wc)`	`iswctype(wc,wctype("alpha"))`
`iswcntrl(wc)`	`iswctype(wc,wctype("cntrl"))`
`iswdigit(wc)`	`iswctype(wc,wctype("digit"))`
`iswgraph(wc)`	`iswctype(wc,wctype("graph"))`
`iswlower(wc)`	`iswctype(wc,wctype("lower"))`
`iswprint(wc)`	`iswctype(wc,wctype("print"))`
`iswpunct(wc)`	`iswctype(wc,wctype("punct"))`
`iswspace(wc)`	`iswctype(wc,wctype("space"))`
`iswupper(wc)`	`iswctype(wc,wctype("upper"))`
`iswxditig(wc)`	`iswctype(wc,wctype("xdigit"))`

`wctrans(*property)`	Constructs a value of type `wctype_t` that describes a mapping between wide characters identified by the string **property*. If property identifies a valid mapping of wide characters according to the LC_TYPE category (see Section 12.1.13, *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
`towlower(wc)`	`towctrans(wc,wctrans("tolower")`

Function	Equivalent to locale specific transformation
`towupper(`*`wc`*`)`	`towctrans(`*`wc`*`,wctrans("toupper")`

12.2. C Library Reentrancy

Some of the functions in the C library are reentrant, others are not. The table below shows the functions in the C library, and whether they are reentrant or not. A dash '-' means that the function is reentrant. Note that some of the functions are not reentrant because they set the global variable 'errno' (or call other functions that eventually set 'errno'). If your program does not check this variable and errno is the only reason for the function not being reentrant, these functions can be assumed reentrant as well.

The explanation of the cause why a function is not reentrant sometimes refers to a footnote because the explanation is too lengthy for the table.

Function	Not reentrant because
`_close`	Uses global File System Simulation buffer, _dbg_request
`_doflt`	Uses I/O functions which modify iob[ ]. See (1).
`_doprint`	Uses indirect access to static iob[ ] array. See (1).
`_doscan`	Uses indirect access to iob[ ] and calls ungetc (access to local static ungetc[ ] buffer). See (1).
`_Exit`	See exit.
`_filbuf`	Uses iob[ ], which is not reentrant. See (1).
`_flsbuf`	Uses iob[ ]. See (1).
`_getflt`	Uses iob[ ]. See (1).
`_iob`	Defines static iob[ ]. See (1).
`_lseek`	Uses global File System Simulation buffer, _dbg_request
`_open`	Uses global File System Simulation buffer, _dbg_request
`_read`	Uses global File System Simulation buffer, _dbg_request
`_unlink`	Uses global File System Simulation buffer, _dbg_request
`_write`	Uses global File System Simulation buffer, _dbg_request
`abort`	Calls exit
`abs labs llabs`	-
`access`	Uses global File System Simulation buffer, _dbg_request
`acos acosf acosl`	Sets errno.
`acosh acoshf acoshl`	Sets errno via calls to other functions.
`aligned_alloc`	See malloc (5).
`asctime`	asctime defines static array for broken-down time string.
`asin asinf asinl`	Sets errno.
`asinh asinhf asinhl`	Sets errno via calls to other functions.

Function	Not reentrant because
`at_quick_exit`	at_quick_exit defines static array with function pointers to execute when quick_exit is called.
`atan atanf atanl`	-
`atan2 atan2f atan2l`	-
`atanh atanhf atanhl`	Sets errno via calls to other functions.
`atexit`	atexit defines static array with function pointers to execute at exit of program.
`atof`	-
`atoi`	-
`atol`	-
`bsearch`	-
`btowc`	-
`c16rtomb`	Sets errno. Uses static internal_state variable.
`c32rtomb`	Sets errno. Uses static internal_state variable.
`cabs cabsf cabsl`	Sets errno via calls to other functions.
`cacos cacosf cacosl`	Sets errno via calls to other functions.
`cacosh cacoshf cacoshl`	Sets errno via calls to other functions.
`calloc`	calloc uses static buffer management structures. See malloc (5).
`carg cargf cargl`	-
`casin casinf casinl`	Sets errno via calls to other functions.
`casinh casinhf casinhl`	Sets errno via calls to other functions.
`catan catanf catanl`	Sets errno via calls to other functions.
`catanh catanhf catanhl`	Sets errno via calls to other functions.
`cbrt cbrtf cbrtl`	-
`ccos ccosf ccosl`	Sets errno via calls to other functions.
`ccosh ccoshf ccoshl`	Sets errno via calls to other functions.
`ceil ceilf ceill`	-
`cexp cexpf cexpl`	Sets errno via calls to other functions.
`chdir`	Uses global File System Simulation buffer, _dbg_request
`cimag cimagf cimagl`	-
`cleanup`	Calls fclose. See (1)
`clearerr`	Modifies iob[ ]. See (1)
`clock`	-
`clog clogf clogl`	Sets errno via calls to other functions.
`close`	Calls _close
`conj conjf conjl`	-

Function	Not reentrant because
`copysign copysignf copysignl`	-
`cos cosf cosl`	-
`cosh coshf coshl`	cosh calls exp(), which sets errno. If errno is discarded, cosh is reentrant.
`cpow cpowf cpowl`	Sets errno via calls to other functions.
`cproj cprojf cprojl`	-
`creal crealf creall`	-
`csin csinf csinl`	Sets errno via calls to other functions.
`csinh csinhf csinhl`	Sets errno via calls to other functions.
`csqrt csqrtf csqrtl`	Sets errno via calls to other functions.
`ctan ctanf ctanl`	Sets errno via calls to other functions.
`ctanh ctanhf ctanhl`	Sets errno via calls to other functions.
`ctime`	Calls asctime
`difftime`	-
`div ldiv lldiv`	-
`erf erfl erff`	-
`erfc erfcf erfcl`	-
`exit`	Calls fclose indirectly which uses iob[ ] calls functions in _atexit array. See (1). To make exit reentrant kernel support is required.
`exp expf expl`	Sets errno.
`exp2 exp2f exp2l`	Sets errno.
`expm1 expm1f expm1l`	Sets errno via calls to other functions.
`fabs fabsf fabsl`	-
`fclose`	Uses values in iob[ ]. See (1).
`fdim fdimf fdiml`	-
`feclearexcept`	Writes FPU_STATUS_WORD bits.
`fegetenv`	- (reads FPU_STATUS_WORD bits)
`fegetexceptflag`	- (reads FPU_STATUS_WORD bits via calls to other functions)
`fegetround`	- (reads FPU_CTRL bits)
`feholdexcept`	Reads/writes FPU_STATUS_WORD bits via calls to other functions.
`feof`	Uses values in iob[ ]. See (1).
`feraiseexcept`	Writes FPU_STATUS_WORD bits.
`ferror`	Uses values in iob[ ]. See (1).
`fesetenv`	Writes FPU_STATUS_WORD bits.
`fesetexceptflag`	Writes FPU_STATUS_WORD bits via calls to other functions.

Function	Not reentrant because
fesetround	Writes FPU_CTRL bits via calls to other functions.
fetestexcept	- (reads FPU_STATUS_WORD bits)
feupdateenv	Writes FPU_STATUS_WORD bits via calls to other functions.
fflush	Modifies iob[ ]. See (1).
fgetc fgetwc	Uses pointer to iob[ ]. See (1).
fgetpos	Sets the variable errno and uses pointer to iob[ ]. See (1) / (2).
fgets fgetws	Uses iob[ ]. See (1).
floor floorf floorl	-
fma fmaf fmal	-
fmax fmaxf fmaxl	-
fmin fminf fminl	-
fmod fmodf fmodl	-
fopen	Uses iob[ ] and calls malloc when file open for buffered IO. See (1)
fpclassify	-
fprintf fwprintf	Uses iob[ ]. See (1).
fputc fputwc	Uses iob[ ]. See (1).
fputs fputws	Uses iob[ ]. See (1).
fread	Calls fgetc. See (1).
free	free uses static buffer management structures. See malloc (5).
freopen	Modifies iob[ ]. See (1).
frexp frexpf frexpl	-
fscanf fwscanf	Uses iob[ ]. See (1)
fseek	Uses iob[ ] and calls _lseek. Accesses ungetc[ ] array. See (1).
fsetpos	Uses iob[ ] and sets errno. See (1) / (2).
fstat	(*Not implemented*)
ftell	Uses iob[ ] and sets errno. Calls _lseek. See (1) / (2).
fwrite	Uses iob[ ]. See (1).
getc getwc	Uses iob[ ]. See (1).
getchar getwchar	Uses iob[ ]. See (1).
getcwd	Uses global File System Simulation buffer, _dbg_request
getenv	Skeleton only.
gets getws	Uses iob[ ]. See (1).
gmtime	gmtime defines static structure
hypot hypotf hypotl	Sets errno via calls to other functions.
ilogb ilogbf ilogbl	Sets errno.

Function	Not reentrant because
`imaxabs`	-
`imaxdiv`	-
`isalnum iswalnum`	-
`isalpha iswalpha`	-
`isascii iswascii`	-
`isblank iswblank`	-
`iscntrl iswcntrl`	-
`isdigit iswdigit`	-
`isfinite`	-
`isgraph iswgraph`	-
`isgreater`	-
`isgreaterequal`	-
`isinf`	-
`isless`	-
`islessequal`	-
`islessgreater`	-
`islower iswlower`	-
`isnan`	-
`isnormal`	-
`isprint iswprint`	-
`ispunct iswpunct`	-
`isspace iswspace`	-
`isunordered`	-
`isupper iswupper`	-
`iswalnum`	-
`iswalpha`	-
`iswcntrl`	-
`iswctype`	-
`iswdigit`	-
`iswgraph`	-
`iswlower`	-
`iswprint`	-
`iswpunct`	-
`iswspace`	-
`iswupper`	-

Function	Not reentrant because
`iswxditig`	-
`isxdigit iswxdigit`	-
`ldexp ldexpf ldexpl`	Sets errno. See (2).
`lgamma lgammaf lgammal`	Sets errno.
`llrint llrintf llrintl`	-
`llround llroundf llroundl`	Sets errno.
`localeconv`	N.A.; skeleton function
`localtime`	-
`log logf logl`	Sets errno. See (2).
`log10 log10f log10l`	Sets errno via calls to other functions.
`log1p log1pf log1pl`	Sets errno.
`log2 log2f log2l`	Sets errno.
`logb logbf logbl`	Sets errno.
`longjmp`	-
`lrint lrintf lrintl`	-
`lround lroundf lroundl`	Sets errno.
`lseek`	Calls _lseek
`lstat`	(*Not implemented*)
`malloc`	Needs kernel support. See (5).
`mblen`	N.A., skeleton function
`mbrlen`	Sets errno.
`mbrtoc16`	Sets errno. Uses static internal_state variable.
`mbrtoc32`	Sets errno. Uses static internal_state variable.
`mbrtowc`	Sets errno. Uses static internal_state variable.
`mbsinit`	-
`mbsrtowcs`	Sets errno.
`mbstowcs`	N.A., skeleton function
`mbtowc`	N.A., skeleton function
`memchr wmemchr`	-
`memcmp wmemcmp`	-
`memcpy wmemcpy`	-
`memmove wmemmove`	-
`memset wmemset`	-
`mktime`	-
`modf modff modfl`	-

Function	Not reentrant because
`nan nanf nanl`	-
`nearbyint nearbyintf nearbyintl`	-
`nextafter nextafterf nextafterl`	-
`nexttoward nexttowardf nexttowardl`	-
`offsetof`	-
`open`	Calls _open
`perror`	Uses errno. See (2)
`pow powf powl`	Sets errno. See (2)
`printf wprintf`	Uses iob[ ]. See (1)
`putc putwc`	Uses iob[ ]. See (1)
`putchar putwchar`	Uses iob[ ]. See (1)
`puts`	Uses iob[ ]. See (1)
`qsort`	-
`quick_exit`	Calls _Exit.
`raise`	Updates the signal handler table
`rand`	Uses static variable to remember latest random number. Must diverge from ISO C standard to define reentrant rand. See (4).
`read`	Calls _read
`realloc`	See malloc (5).
`remainder remainderf remainderl`	-
`remove`	Uses global File System Simulation buffer, _dbg_request
`remquo remquof remquol`	-
`rename`	Uses global File System Simulation buffer, _dbg_request
`rewind`	Eventually calls _lseek
`rint rintf rintl`	-
`round roundf roundl`	-
`scalbln scalblnf scalblnl`	-
`scalbn scalbnf scalbnl`	-
`scanf wscanf`	Uses iob[ ], calls _doscan. See (1).
`setbuf`	Sets iob[ ]. See (1).
`setjmp`	-
`setlocale`	N.A.; skeleton function

Function	Not reentrant because
setvbuf	Sets iob and calls malloc. See (1) / (5).
signal	Updates the signal handler table
signbit	-
sin sinf sinl	-
sinh sinhf sinhl	Sets errno via calls to other functions.
snprintf swprintf	Sets errno. See (2).
sprintf	Sets errno. See (2).
sqrt sqrtf sqrtl	Sets errno. See (2).
srand	See rand
sscanf swscanf	Sets errno via calls to other functions.
stat	Uses global File System Simulation buffer, _dbg_request
strcat wcscat	-
strchr wcschr	-
strcmp wcscmp	-
strcoll wcscoll	-
strcpy wcscpy	-
strcspn wcscspn	-
strerror	-
strftime wcsftime	-
strlen wcslen	-
strncat wcsncat	-
strncmp wcsncmp	-
strncpy wcsncpy	-
strpbrk wcspbrk	-
strrchr wcsrchr	-
strspn wcsspn	-
strstr wcsstr	-
strtod wcstod	-
strtof wcstof	-
strtoimax	Sets errno via calls to other functions.
strtok wcstok	strtok saves last position in string in local static variable. This function is not reentrant by design. See (4).
strtol wcstol	Sets errno. See (2).
strtold wcstold	-
strtoul wcstoul	Sets errno. See (2).

Function	Not reentrant because
strtoull wcstoull	Sets errno. See (2).
strtoumax	Sets errno via calls to other functions.
strxfrm wcsxfrm	-
system	N.A; skeleton function
tan tanf tanl	Sets errno. See (2).
tanh tanhf tanhl	Sets errno via call to other functions.
tgamma tgammaf tgammal	Sets errno.
time	Uses static variable which defines initial start time
tmpfile	Uses iob[ ]. See (1).
tmpnam	Uses local buffer to build filename. Function can be adapted to use user buffer. This makes the function non ISO C. See (4).
toascii	-
tolower	-
toupper	-
towctrans	-
towlower	-
towupper	-
trunc truncf truncl	-
ungetc ungetwc	Uses static buffer to hold unget characters for each file. Can be moved into iob structure. See (1).
unlink	Uses global File System Simulation buffer, _dbg_request
vfprintf vfwprintf	Uses iob[ ]. See (1).
vfscanf vfwscanf	Calls _doscan
vprintf vwprintf	Uses iob[ ]. See (1).
vscanf vwscanf	Calls _doscan
vsprintf vswprintf	Sets errno.
vsscanf vswscanf	Sets errno.
wcrtomb	Sets errno. Uses static internal_state variable.
wcsrtombs	Sets errno.
wcstoimax	Sets errno via calls to other functions.
wcstombs	N.A.; skeleton function
wcstoumax	Sets errno via calls to other functions.
wctob	-
wctomb	N.A.; skeleton function
wctrans	-

Function	Not reentrant because
wctype	-
write	Calls _write

Table: C library reentrancy

Several functions in the C library are not reentrant due to the following reasons:

- The `iob[]` structure is static. This influences all I/O functions.

- The `ungetc[]` array is static. This array holds the characters (one for each stream) when `ungetc()` is called.

- The variable `errno` is globally defined. Numerous functions read or modify `errno`

- `_doprint` and `_doscan` use static variables for e.g. character counting in strings.

- Some string functions use locally defined (static) buffers. This is prescribed by ANSI.

- `malloc` uses a static heap space.

The following description discusses these items in more detail. The numbers at the beginning of each paragraph relate to the number references in the table above.

(1) iob structures

The I/O part of the C library is not reentrant by design. This is mainly caused by the static declaration of the `iob[]` array. The functions which use elements of this array access these elements via pointers (`FILE *`).

Building a multi-process system that is created in one link-run is hard to do. The C language scoping rules for external variables make it difficult to create a private copy of the `iob[]` array. Currently, the `iob[]` array has external scope. Thus it is visible in every module involved in one link phase. If these modules comprise several tasks (processes) in a system each of which should have its private copy of `iob[]`, it is apparent that the `iob[]` declaration should be changed. This requires adaptation of the library to the multi-tasking environment. The library modules must use a process identification as an index for determining which `iob[]` array to use. Thus the library is suitable for interfacing to that kernel only.

Another approach for the `iob[]` declaration problem is to declare the array static in one of the modules which create a task. Thus there can be more than one `iob[]` array is the system without having conflicts at link time. This brings several restrictions: Only the module that holds the declaration of the static `iob[]` can use the standard file handles `stdin`, `stdout` and `stderr` (which are the first three entries in `iob[]`). Thus all I/O for these three file handles should be located in one module.

(2) errno declaration

Several functions in the C library set the global variable `errno`. After completion of the function the user program may consult this variable to see if some error occurred. Since most of the functions that set `errno` already have a return type (this is the reason for using `errno`) it is not possible to check successful completion via the return type.

The library routines can set `errno` to the values defined in `errno.h`. See the file `errno.h` for more information.

`errno` can be set to ERR_FORMAT by the print and scan functions in the C library if you specify illegal format strings.

`errno` will never be set to ERR_NOLONG or ERR_NOPOINT since the C library supports long and pointer conversion routines for input and output.

`errno` can be set to ERANGE by the following functions: `exp()`, `strtol()`, `strtoul()` and `tan()`. These functions may produce results that are out of the valid range for the return type. If so, the result of the function will be the largest representable value for that type and `errno` is set to ERANGE.

`errno` is set to EDOM by the following functions: `acos()`, `asin()`, `log()`, `pow()` and `sqrt()`. If the arguments for these functions are out of their valid range (e.g. `sqrt( -1 )`), `errno` is set to EDOM.

`errno` can be set to ERR_POS by the file positioning functions `ftell()`, `fsetpos()` and `fgetpos()`.

(3) ungetc

Currently the ungetc buffer is static. For each file entry in the `iob[]` structure array, there is one character available in the buffer to unget a character.

(4) local buffers

`tmpnam()` creates a temporary filename and returns a pointer to a local static buffer. This is according to the ANSI definition. Changing this function such that it creates the name in a user specified buffer requires another calling interface. Thus the function would be no longer portable.

`strtok()` scans through a string and remembers that the string and the position in the string for subsequent calls. This function is not reentrant by design. Making it reentrant requires support of a kernel to store the information on a per process basis.

`rand()` generates a sequence of random numbers. The function uses the value returned by a previous call to generate the next value in the sequence. This function can be made reentrant by specifying the previous random value as one of the arguments. However, then it is no longer a standard function.

(5) malloc

Malloc uses a heap space which is assigned at locate time. Thus this implementation is not reentrant. Making a reentrant malloc requires some sort of system call to obtain free memory space on a per process basis. This is not easy to solve within the current context of the library. This requires adaptation to a kernel.

This paragraph on reentrancy applies to multi-process environments only. If reentrancy is required for calling library functions from an exception handler, another approach is required. For such a situation it is of no use to allocate e.g. multiple `iob[]` structures. In such a situation several pieces of code in the library have to be declared 'atomic': this means that interrupts have to be disabled while executing an atomic piece of code.

Chapter 13. List File Formats

This chapter describes the format of the assembler list file and the linker map file.

13.1. Assembler List File Format

The assembler list file is an additional output file of the assembler that contains information about the generated code. For details on how to generate a list file, see Section 6.4, *Generating a List File*.

The list file consists of a page header and a source listing.

Page header

The page header is repeated on every page:

```
TASKING SmartCode vx.yrz - ARC assembler  Build yymmddqq
Title                                                     Page 1

ADDR CODE       CYCLES  LINE SOURCE LINE
```

The first line contains version information. The second line can contain a title which you can specify with the assembler control $TITLE and always contains a page number. The third line is empty and the fourth line contains the headings of the columns for the source listing.

With the assembler controls $LIST ON/OFF, $PAGE, and with the assembler option **--list-format** you can format the list file.

Source listing

The following is a sample part of a listing. An explanation of the different columns follows below.

```
ADDR CODE       CYCLES  LINE SOURCE LINE
                         1          ; Module start
                         .
                         .
0000                     7          .section     .text
                         8          .global main
                         9 ; Function main
0000                    10 main:    .type    func
0000 FC1CC8B7 1     1   11          push     %blink
0004 0A20800F 1     2   12          mov      %r0,.1.str
     rrrrrrrr
000C rr0rrrrr 1     3   13          bl       printf
0010 4A200000 1     4   14          mov      %r0,0
0014 04141F34 1     5   15          pop      %blink
0018 2020C007 1     6   16          j        [%blink]
                        17          ; End of function
                        18          .endsec ; End of section
```

```
                            .
                            .
0000                   44 buf:    .ds  4
   |   RESERVED
0003
```

ADDR	This column contains the memory address. The address is a hexadecimal number that represents the offset from the beginning of a relocatable section or the absolute address for an absolute section. The address only appears on lines that generate object code.
CODE	This is the object code generated by the assembler for this source line, displayed in hexadecimal format. The displayed code need not be the same as the generated code that is entered in the object module. The code can also be relocatable code. In this case the letter '**r**' is printed for the relocatable code part in the listing. For lines that allocate space, the code field contains the text "RESERVED". For lines that initialize a buffer, the code field lists one value followed by the word "REPEATS".
CYCLES	The first number in this column is the number of instruction cycles needed to execute the instruction(s) as generated in the CODE field. The second number is the accumulated cycle count of this section.
LINE	This column contains the line number. This is a decimal number indicating each input line, starting from 1 and incrementing with each source line.
SOURCE LINE	This column contains the source text. This is a copy of the source line from the assembly source file.

For the `.SET` and `.EQU` directives the `ADDR` and `CODE` columns do not apply. The symbol value is listed instead.

13.2. Linker Map File Format

The linker map file is an additional output file of the linker that shows how the linker has mapped the sections and symbols from the various object files (`.o`) to output sections. Locate information is not present, because that is not available for an ARC project. External symbols are listed per space with their absolute address, both sorted on symbol and sorted on address. For details on how to generate a map file, see Section 7.10, *Generating a Map File*.

With the linker option **--map-file-format** you can specify which parts of the map file you want to see. To specify the same for the global map file, use linker option **--global-map-file-format**. Both options have the same defaults and accept the same arguments.

In Eclipse the linker map file (*project*.`mapxml`) is generated in the output directory of the build configuration, usually `Debug` or `Release`. You can open the map file by double-clicking on the file name.

Each page displays a part of the map file. You can use the drop-down list or the Outline view to navigate through the different tables and you can use the following buttons.

Icon	Action	Description
⇦	Back	Goes back one page in the history list.
⇨	Forward	Goes forward one page in the history list.
	Next Table	Shows the next table from the drop-down list.
	Previous Table	Shows the previous table from the drop-down list.

When you right-click in the view, a popup menu appears (for example, to reset the layout of a table). The meaning of the different parts is:

Tool and Invocation

This part of the map file contains information about the linker, its version header information, binary location and which options are used to call it.

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. This part is not available when you use MIL linking (control program option **--mil-link**).

Link Result

This part of the map file shows per object file how the link phase has mapped the sections from the various object files (`.o`) to output sections.

[in] 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. This part is not available when you use MIL linking (control program option **--mil-link**).

Call Graph

This part of the map file contains a schematic overview that shows how (library) functions call each other. To obtain call graph information, the assembly file must contain .CALLS directives.

The following example is a part of a call graph in the textual version of the map file (.map):

```
_START [0,104]
|
+-- main [4,104]
|   |
|   +-- printf [20,100]
|       |
|       +-- _doprint [16,80]
|           |
|           +-- _io_putc [4,64]
|               |
|               +-- fputc [4,60]
|                   |
|                   +-- _flsbuf [0,56]
|                       |
|                       +-- _dofls [20,56]
|                           |
|                           +-- _flsbuf.c:.cocofun_1 [0,0]
|                           |
|                           +-- _fflush *
|                           |
|                           +-- _host_write *
|
+-- exit [8,68]
```

- A * after a function name indicates that the call tree starting with this function is shown separately, with a * in front of the function name.

- A * in front of a function name indicates that the function is not considered a "root" in the call graph since it is called by one or more other functions.

- An additional R (not shown in this example) indicates this function is part of a recursive call chain. If both a leaf and the root of a tree are marked this way, all nodes in between are in a recursive chain.

- An '__INDIRECT__' entry (not shown in this example) indicates an indirect function call. It is not an actual function. Each function listed as a caller of the __INDIRECT__ placeholder symbol places a call through a function pointer. Each function listed as a callee of the __INDIRECT__ placeholder symbol has its address taken (and used).

- [ ] after a function contains information about the stack usage. The first field is the amount of stack used by the function and the second field is the amount of stack used by the function including its callees.

In the graphical version of the map file, you can expand or collapse a single node. Use the ⊞ / ⊟ buttons to expand/collapse all nodes in the call graph. Hover the mouse over a function (root, callee or node) to see information about the stack usage.

Icon	Meaning	Description
⚓	Root	This function is the top of the call graph. If there are interrupt handlers, there can be several roots.
🐾	Callee	This function is referenced by several No leaf functions. Right-click on the function and select **Expand all References** to see all functions that reference this function. Select **Back to Caller** to return to the calling function.
●	Node	A normal node (function) in the call graph.
➡	Caller	This function calls a function which is listed separately in the call graph. Right-click on the function and select **Go to Callee** to see the callee. Hover the mouse over the function to see a popup with all callees.

Overlay

This part of the map file shows how the stack is organized. This part also shows the locate overlay information if you used overlay groups in the linker script file.

Processor and Memory

This part of the map file shows the processor and memory information of the linker script file.

By default this part is not shown in the map file. You have to turn this part on manually with linker option **--map-file-format=+lsl** (processor and memory info). You can print this information to a separate file with linker option **--lsl-dump**.

You can expand or collapse a part of the information.

Locate Rules

This part of the map file shows the rules the linker uses to locate sections.

Address space   The names of the address spaces as defined in the linker script file (`*.lsl`). The names are constructed of the `derivative` name followed by a colon ':', the `core` name, another colon ':' and the `space` name.

Type	The rule type:

ordered/contiguous/clustered/unrestricted

Specifies how sections are grouped. By default, a group is 'unrestricted' which means that the linker has total freedom to place the sections of the group in the address space.

absolute

The section must be located at the address shown in the Properties column.

ranged

The section must be located anywhere in the address ranges shown in the Properties column; end addresses are not included in the range.

page

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 page

Both the ranged and the paged restriction apply. In the Properties column the range restriction is listed first, followed by the paged restriction between parenthesis.

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.
Prio	The locate priority of the rule. A higher priority value gives a rule precedence over a rule with a lower priority, but only if the two rules have the same type and the same properties. The relative order of rules of different types or different properties is not affected by this priority value. You can set the priority with the priority group attribute in LSL
Sections	The sections to which the rule applies;

restrictions between sections are shown in this column:

```
<    ordered
|    contiguous
+    clustered
```

For contiguous sections, the linker uses the section order as shown here. Clustered sections can be located in any relative order.

Removed Sections

This part of the map file shows the sections which are removed from the output file as a result of the optimization option to delete unreferenced sections and or duplicate code or constant data (linker option **--optimize=cxy**).

Section	The name of the section which has been removed.
File	The name of the input object file where the section is removed from.
Library	The name of the library where the object file is part of.
Symbol	The symbols that were present in the section.
Reason	The reason why the section has been removed. This can be because the section is unreferenced or duplicated.

Chapter 14. Object File Formats

The linker can generate machine code in several output formats: ELF/DWARF, Intel Hex, Motorola S-records and C array. The C array format is a special one, where the generated machine code is in the form of C code. The following sections describe each format.

14.1. ELF/DWARF Object Format

The TASKING toolset for ARC/PPU by default produces objects in the ELF/DWARF 3 format.

For a complete description of the ELF format, please refer to the *Tool Interface Standard (TIS)*.

For a complete description of the DWARF format, please refer to the *DWARF Debugging Information Format Version 3*. See http://dwarfstd.org/

14.2. Intel Hex Record Format

Intel Hex records describe the hexadecimal object file format for 8-bit, 16-bit and 32-bit microprocessors. The hexadecimal object file is an ASCII representation of an absolute binary object file. There are six different types of records:

- Data Record (8-, 16, or 32-bit formats)

- End of File Record (8-, 16, or 32-bit formats)

- Extended Segment Address Record (16, or 32-bit formats)

- Start Segment Address Record (16, or 32-bit formats)

- Extended Linear Address Record (32-bit format only)

- Start Linear Address Record (32-bit format only)

To generate an Intel Hex output file:

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

   The Properties dialog appears.

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

   In the right pane the Settings appear.

3. On the Tool Settings tab, select **Linker » Output Format**.

4. Enable the option **Generate Intel Hex format file**.

5. (Optional) Specify the **Size of addresses (in bytes) for Intel Hex records**.

6. (Optional) Enable or disable the option **Emit start address record**.

By default the linker generates records in the 32-bit format (4-byte addresses).

General Record Format

In the output file, the record format is:

:	*length*	*offset*	*type*	*content*	*checksum*

where:

:	is the record header.
length	is the record length which specifies the number of bytes of the *content* field. This value occupies one byte (two hexadecimal digits). The linker outputs records of 255 bytes (32 hexadecimal digits) or less; that is, *length* is never greater than 0xFF.
offset	is the starting load offset specifying an absolute address in memory where the data is to be located when loaded by a tool. This field is two bytes long. This field is only used for Data Records. In other records this field is coded as four ASCII zero characters ('0000').
type	is the record type. This value occupies one byte (two hexadecimal digits). The record types are:

Byte Type	Record Type
00	Data
01	End of file
02	Extended segment address (not used)
03	Start segment address (not used)
04	Extended linear address (32-bit)
05	Start linear address (32-bit)

content	is the information contained in the record. This depends on the record type.
checksum	is the record checksum. The linker computes the checksum by first adding the binary representation of the previous bytes (from length to content). The linker then computes the result of sum modulo 256 and subtracts the remainder from 256 (two's complement). Therefore, the sum of all bytes following the header is zero.

Extended Linear Address Record

The Extended Linear Address Record specifies the two most significant bytes (bits 16-31) of the absolute address of the first data byte in a subsequent Data Record:

:	02	0000	04	*upper_address*	*checksum*

The 32-bit absolute address of a byte in a Data Record is calculated as:

```
( address + offset + index ) modulo 4G
```

where:

address	is the base address, where the two most significant bytes are the upper_address and the two least significant bytes are zero.
offset	is the 16-bit offset from the Data Record.
index	is the index of the data byte within the Data Record (0 for the first byte).

Example:

```
:0200000400FFFB
 | |   | |   |_ checksum
 | |   | |_ upper_address
 | |   |_ type
 | |_ offset
 |_ length
```

Data Record

The Data Record specifies the actual program code and data.

:	*length*	*offset*	00	*data*	*checksum*

The *length* byte specifies the number of *data* bytes. The linker has an option (**--hex-record-size**) that controls the length of the output buffer for generating Data records. The default buffer length is 32 bytes.

The *offset* is the 16-bit starting load offset. Together with the address specified in the Extended Address Record it specifies an absolute address in memory where the data is to be located when loaded by a tool.

Example:

```
:0F0020000023222754E00754F04AF4FAE4E22C3
 | |   | |                          |_ checksum
 | |   | |_ data
 | |   |_ type
 | |_ offset
 |_ length
```

Start Linear Address Record

The Start Linear Address Record contains the 32-bit program execution start address.

:	04	0000	05	*address*	*checksum*

With linker option **--hex-format=S** you can prevent the linker from emitting this record.

Example:

```
:0400000500FF0003F5
 | |   | |          |_ checksum
 | |   | |_ address
 | |   |_ type
 | |_ offset
 |_ length
```

End of File Record

The hexadecimal file always ends with the following end-of-file record:

```
:00000001FF
 | |   | |_ checksum
 | |   |_ type
 | |_ offset
 |_ length
```

14.3. Motorola S-Record Format

To generate a Motorola S-record output file:

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

   The Properties dialog appears.

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

   In the right pane the Settings appear.

3. On the Tool Settings tab, select **Linker » Output Format**.

4. Enable the option **Generate S-records file**.

5. (Optional) Specify the **Size of addresses (in bytes) for Motorola S records**.

By default, the linker produces output in Motorola S-record format with three types of S-records (4-byte addresses): S0, S3 and S7. Depending on the size of addresses you can force other types of S-records. They have the following layout:

S0 - record

S0	length	0000	comment	checksum

A linker generated S-record file starts with an S0 record with the following contents:

```
        l a r c
S00700006C61726356
```

The S0 record is a comment record and does not contain relevant information for program execution.

where:

S0	is a comment record and does not contain relevant information for program execution.
length	represents the number of bytes in the record, not including the record type and length byte. This value occupies one byte (two hexadecimal digits).
comment	contains the name of the linker.
checksum	is the record checksum. The linker computes the checksum by first adding the binary representation of the bytes following the record type (starting with the *length* byte) to just before the checksum. Then the one's complement is calculated of this sum. The least significant byte of the result is the checksum. The sum of all bytes following the record type is 0xFF.

S1 / S2 / S3 - record

This record is the program code and data record for 2-byte, 3-byte or 4-byte addresses respectively.

S1	length	address	code bytes	checksum

S2	length	address	code bytes	checksum

S3	length	address	code bytes	checksum

where:

S1	is the program code and data record for 2-byte addresses.
S2	is the program code and data record for 3-byte addresses.
S3	is the program code and data record for 4-byte addresses (this is the default).
length	represents the number of bytes in the record, not including the record type and length byte. This value occupies one byte (two hexadecimal digits).
address	contains the code or data address.
code bytes	contains the actual program code and data.
checksum	is the record checksum. The checksum calculation is identical to S0.

Example:

```
S3070000FFFE6E6825
   | |          |   |_ checksum
   | |          |_ code
   | |_ address
   |_ length
```

S7 / S8 / S9 - record

This record is the termination record for 4-byte, 3-byte or 2-byte addresses respectively.

S7	length	address	checksum

S8	length	address	checksum

S9	length	address	checksum

where:

S7	is the termination record for 4-byte addresses (this is the default). S7 is the corresponding termination record for S3 records.
S8	is the termination record for 3-byte addresses. S8 is the corresponding termination record for S2 records.
S9	is the termination record for 2-byte addresses. S9 is the corresponding termination record for S1 records.
length	represents the number of bytes in the record, not including the record type and length byte. This value occupies one byte (two hexadecimal digits).
address	contains the program start address.
checksum	is the record checksum. The checksum calculation is identical to S0.

Example:

```
S70500000000FA
   | |          |_checksum
   | |_ address
   |_ length
```

14.4. C Array Format

The linker can emit the generated machine code in the form of C code. This is useful for the integration of ARC/PPU code in applications where the CPU is programmed with a non-TASKING compiler. Note that no symbolic debugging is possible unless you use the debug information from the ELF file.

The C array output format consists of a C source file with data encoded in a C array initializer combined with a C header file that contains necessary declarations and optionally provides access to the exported symbols of the program. You can use the C array format only for chip output files where, as with normal

hex files, each memory gets its own output file that contains data for that memory only. The intended target for output files in C array format is a programmable peripheral where an application compiled for the peripheral is imported into a host application as C code. This "host application" must initialize the programmable peripheral's memory using the data in the C array(s).

Generate the C array output format

To generate a C array output file on the command line use linker option **--chip-output=**_basename_**:CARR:32**. For example with the following call to the control program:

```
ccarc -t -Wl--chip-output=myproject:CARR:32 myproject.c
```

This results in the files `myproject_xrom.c` and `myproject_xrom.h`, where the basename is appended with an underscore and the full name of the memory represented in the file.

To generate a C array output file in the Eclipse IDE:

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

   The Properties dialog appears.

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

   In the right pane the Settings appear.

3. On the Tool Settings tab, select **Linker » Output Format**.

4. Enable the option **Generate C array file**.

5. (Optional) Enable the option **Emit list of exported symbols**.

C array source file

The C array source file contains an initialized array with the data of the memory. Each non-zero element of the C array corresponds to an assembled instruction or an initialized memory mapped variable. The array starts with the first initialized Minimal Addressable Unit (MAU) in the memory and ends with the last initialized MAU in the same memory. Any non-initialized MAUs in between are assigned the value 0. Sections that are marked as clear are emitted as zero values. Sections that are marked as scratch will not appear in the C array output unless they occupy space between initialized/cleared sections.

Example (part of) output of a C array source file:

```
#include "myproject_xrom.h"

unsigned long myproject_xrom[] = {
 0x00000988, /* 0 */
 0x00100018, /* 1 */
 0x00100018, /* 2 */
 0x00000000, /* 3 */
 0x00000050, /* 4 */
 0x00000009, /* 5 */
 0x00100068, /* 6 */
```

```
 0x00100068, /* 7 */
 0x00000000, /* 8 */
 0x00000050, /* 9 */
 0x00000012, /* 10 */
    ...
 0x20676E69, /* 853 */
 0x6D617865, /* 854 */
 0x2E656C70, /* 855 */
};
```

By default the C array data elements have data type `unsigned long`. You can overrule this default with linker option **--c-array-element-type**.

C array header file

The C array header file contains a define for the offset from the start of the memory where the data array starts (in MAUs). Furthermore it contains a define for the size of the data array in MAUs of the memory and an `extern` declaration of the data array that is defined in the C source file.

Example output of a C array header file:

```
#ifndef MYPROJECT_XROM_H
#define MYPROJECT_XROM_H

extern unsigned long myproject_xrom[];

/* Locations of symbols as index in the associated C array */

/* Offset to the start of the C array within the memory in bytes */
#define OFFSET_MYPROJECT_XROM 0

/* Size of the C array in bytes */
#define SIZE_MYPROJECT_XROM 3426

#endif /* MYPROJECT_XROM_H */
```

The C array header file can also contain a list of preprocessor variable definitions for each label mentioned in the ARC/PPU application. With the linker option **--hex-format=+c-array-symbols** the linker emits a list of exported symbols in the header file for each "C array" output C source file. An exported symbol is represented by a preprocessor definition where the name is the full basename of the file translated to upper case and with colon(s) replaced by underscore(s), followed by an underscore and the name of the exported variable, while the value is the index of the variable's location in the data array. Symbols with a name that is not guaranteed to be a valid C identifier are emitted inside a comment block.

Example:

```
/* Locations of symbols as index in the associated C array */
#define MYPROJECT_XROM_main 754
#define MYPROJECT_XROM__START 610
/* #define MYPROJECT_XROM_.vector.0 610 */
#define MYPROJECT_XROM__doprint 56
```

740

```
#define MYPROJECT_XROM___printf_int2 56
#define MYPROJECT_XROM_printf 774
#define MYPROJECT_XROM__doclose 597
#define MYPROJECT_XROM_fputc 728
#define MYPROJECT_XROM_myvar 10
...
#define MYPROJECT_XROM__doexit 802
#define MYPROJECT_XROM__do_quick_exit 802
#define MYPROJECT_XROM__dofree 802
#define MYPROJECT_XROM__Exit 54
#define MYPROJECT_XROM___init 738
#define MYPROJECT_XROM__lc_ub_table 825
```

If the C array is directly mapped to the ARC/PPU memory, the software of the host CPU can use these preprocessor variables for direct access of ARC/PPU program variables. For example, the host CPU could overwrite the value 0x12 of variable myvar with 0x14 using the following C code statement:

```
myproject_XROM[MYPROJECT_XROM_myvar] = 0x14;
```

14.5. Binary Object Format

With linker option **--chip-output=:BIN** you tell the linker to produce a binary output file for each memory chip.

The data of a binary output file represents the first MAU (minimal addressable unit) in the memory (at offset zero) up to the last data MAU of the application in the memory. Any memory location included in the file that is not occupied by application data is set to zero.

Chapter 15. Linker Script Language (LSL)

To make full use of the linker, you can write a script with information about the architecture of the target processor and locating information. The language for the script is called the *Linker Script Language (LSL)*. This chapter first describes the structure of an LSL file. The next section contains a summary of the LSL syntax. In the remaining sections, the semantics of the Linker Script Language is explained.

The TASKING linker is a target independent linker/locator that can simultaneously link and locate all programs for all cores available on a target board. The target board may be of arbitrary complexity. A simple target board may contain one standard processor with some external memory that executes one task. A complex target board may contain multiple standard processors and DSPs combined with configurable IP-cores loaded in an FPGA. Each core may execute a different program, and external memory may be shared by multiple cores.

LSL serves two purposes. First it enables you to specify the characteristics (that are of interest to the linker) of your specific target board and of the cores installed on the board. Second it enables you to specify how sections should be located in memory.

15.1. Structure of a Linker Script File

A script file consists of several definitions. The definitions can appear in any order.

The architecture definition (required)

In essence an *architecture definition* describes how the linker should convert logical addresses into physical addresses for a given type of core. If the core supports multiple address spaces, then for each space the linker must know how to perform this conversion. In this context a physical address is an offset on a given internal or external bus. Additionally the architecture definition contains information about items such as the (hardware) stack and the vector table.

This specification is normally written by TASKING. TASKING supplies LSL files in the `include.lsl` directory. The architecture definition of the LSL file should not be changed by you unless you also modify the core's hardware architecture. If the LSL file describes a multi-core system an architecture definition must be available for each different type of core.

See Section 15.4, *Semantics of the Architecture Definition* for detailed descriptions of LSL in the architecture definition.

The derivative definition

The *derivative definition* describes the configuration of the internal (on-chip) bus and memory system. Basically it tells the linker how to convert offsets on the buses specified in the architecture definition into offsets in internal memory. Microcontrollers and DSPs often have internal memory and I/O sub-systems apart from one or more cores. The design of such a chip is called a *derivative*.

When you build an ASIC or use a derivative that is not (yet) supported by the TASKING tools, you may have to write a derivative definition.

When you want to use multiple cores of the same type, you must instantiate the cores in a derivative definition, since the linker automatically instantiates only a single core for an unused architecture.

See Section 15.5, *Semantics of the Derivative Definition* for a detailed description of LSL in the derivative definition.

The processor definition

The *processor definition* describes an instance of a derivative. Typically the processor definition instantiates one derivative only (single-core processor). A processor that contains multiple cores having the same (homogeneous) or different (heterogeneous) architecture can also be described by instantiating multiple derivatives of the same or different types in separate processor definitions.

See Section 15.6, *Semantics of the Board Specification* for a detailed description of LSL in the processor definition.

The memory and bus definitions (optional)

Memory and bus definitions are used within the context of a derivative definition to specify internal memory and on-chip buses. In the context of a board specification the memory and bus definitions are used to define external (off-chip) memory and buses. Given the above definitions the linker can convert a logical address into an offset into an on-chip or off-chip memory device.

See Section 15.6.3, *Defining External Memory and Buses*, for more information on how to specify the external physical memory layout. *Internal* memory for a processor should be defined in the derivative definition for that processor.

The board specification

The processor definition and memory and bus definitions together form a *board specification*. LSL provides language constructs to easily describe single-core and heterogeneous or homogeneous multi-core systems. The board specification describes all characteristics of your target board's system buses, memory devices, I/O sub-systems, and cores that are of interest to the linker. Based on the information provided in the board specification the linker can for each core:

- convert a logical address to an offset within a memory device

- locate sections in physical memory

- maintain an overall view of the used and free physical memory within the whole system while locating

The section layout definition (optional)

The optional section layout definition enables you to exactly control where input sections are located. Features are provided such as: the ability to place sections at a given load-address or run-time address, to place sections in a given order, and to overlay code and/or data sections.

Which object files (sections) constitute the task that will run on a given core is specified on the command line when you invoke the linker. The linker will link and locate all sections of all tasks simultaneously. From the section layout definition the linker can deduce where a given section may be located in memory,

form the board specification the linker can deduce which physical memory is (still) available while locating the section.

See Section 15.8, *Semantics of the Section Layout Definition*, for more information on how to locate a section at a specific place in memory.

Skeleton of a Linker Script File

```
architecture architecture_name
{
    // Specification core architecture
}

derivative derivative_name
{
    // Derivative definition
}

processor processor_name
{
    // Processor definition
}

memory and/or bus definitions

section_layout space_name
{
    // section placement statements
}
```

15.2. Syntax of the Linker Script Language

This section describes what the LSL language looks like. An LSL document is stored as a file coded in UTF-8 with extension `.lsl`. Before processing an LSL file, the linker preprocesses it using a standard C preprocessor. Following this, the linker interprets the LSL file using a scanner and parser. Finally, the linker uses the information found in the LSL file to guide the locating process.

15.2.1. Preprocessing

When the linker loads an LSL file, the linker first processes it with a C-style prepocessor. As such, it strips C and C++ comments. Lines starting with the `#` character are taken as commands for the preprocessor. You can use the standard ISO C99 preprocessor directives, including:

```
#include "file"
#include <file>
```

Preprocess and include file *file* at this point in the LSL file.

For example:

```
#include "arch.lsl"
```

Preprocess and include the file `arch.lsl` at this point in the LSL file.

```
#if condition
#else
#endif
```

If the *condition* evaluates to a non-zero value, copy the following lines, up to an `#else` or `#endif` command, skip lines between `#else` and `#endif`, if present. If the condition evaluates to zero, skip the lines up to the `#else` command, or `#endif` if no `#else` is present, and copy the lines between the `#else` and `#endif` commands.

```
#ifdef identifier
#else
#endif
```

Same as `#if`, but with `defined(identifier)` as condition.

```
#error text
```

Causes a fatal error the given message (optional).

15.2.2. Lexical Syntax

The following lexicon is used to describe the syntax of the Linker Script Language:

`A ::= B`	=	*A* is defined as *B*
`A ::= B C`	=	*A* is defined as *B* and *C*; *B* is followed by *C*
`A ::= B │ C`	=	*A* is defined as *B* or *C*
`<B>`$^{0\|1}$	=	zero or one occurrence of *B*
`<B>`$^{>=0}$	=	zero of more occurrences of *B*
`<B>`$^{>=1}$	=	one of more occurrences of *B*
`IDENTIFIER`	=	a character sequence starting with 'a'-'z', 'A'-'Z' or '_'. Following characters may also be digits and dots '.'
`STRING`	=	sequence of characters not starting with \n, \r or \t
`DQSTRING`	=	" STRING " (double quoted string)
`OCT_NUM`	=	octal number, starting with a zero (`06`, `045`)
`DEC_NUM`	=	decimal number, not starting with a zero (`14`, `1024`)

```
HEX_NUM                =    hexadecimal number, starting with '0x' (0x0023, 0xFF00)
```

OCT_NUM, *DEC_NUM* and *HEX_NUM* can be followed by a **k** (kilo), **M** (mega), or **G** (giga).

Characters in **bold** are characters that occur literally. Words in *italics* are higher order terms that are defined in the same or in one of the other sections.

To write comments in LSL file, you can use the C style '/* */' or C++ style '//'.

15.2.3. Identifiers and Tags

```
arch_name          ::= IDENTIFIER
bus_name           ::= IDENTIFIER
core_name          ::= IDENTIFIER
derivative_name    ::= IDENTIFIER
file_name          ::= DQSTRING
group_name         ::= IDENTIFIER
heap_name          ::= section_name
map_name           ::= IDENTIFIER
mem_name           ::= IDENTIFIER
proc_name          ::= IDENTIFIER
section_name       ::= DQSTRING
space_name         ::= IDENTIFIER
stack_name         ::= section_name
symbol_name        ::= DQSTRING
tag_attr           ::= (tag<,tag>>=0)
tag                ::= tag = DQSTRING
```

A tag is an arbitrary text that can be added to a statement.

15.2.4. Expressions

The expressions and operators in this section work the same as in ISO C.

```
number             ::= OCT_NUM
                     | DEC_NUM
                     | HEX_NUM

expr               ::= number
                     | symbol_name
                     | unary_op expr
                     | expr binary_op expr
                     | expr ? expr : expr
                     | ( expr )
                     | function_call

unary_op           ::= !    // logical NOT
                     | ~    // bitwise complement
                     | -    // negative value
```

747

```
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
```

15.2.5. Built-in Functions

```
function_call      ::= absolute ( expr )
                    |  addressof ( addr_id )
                    |  checksum ( checksum_algo , expr , expr )
                    |  exists ( section_name )
                    |  max ( expr , expr )
                    |  min ( expr , expr )
                    |  sizeof ( size_id )

addr_id            ::= sect : section_name
                    |  group : group_name
                    |  mem : mem_name

checksum_algo      ::= crc32w

size_id            ::= sect : section_name
                    |  group : group_name
                    |  mem : mem_name
```

- Every space, bus, memory, section or group you refer to, must be defined in the LSL file.

- The addressof() and sizeof() functions with the **group** or **sect** argument can only be used in the right hand side of an assignment. The sizeof() function with the **mem** argument can be used anywhere in section layouts.

- The checksum() function can only be used in a **struct** statement.

You can use the following built-in functions in expressions. All functions return a numerical value. This value is a 64-bit signed integer.

absolute()

```
int absolute( expr )
```

Converts the value of *expr* to a positive integer.

```
absolute( "labelA"-"labelB" )
```

addressof()

```
int addressof( addr_id )
```

Returns the offset of *addr_id*, which is a named section, group, or memory in the address space of the section layout. If the referenced object is a group or memory, it must be defined in the LSL file. To get the offset of the section with the name `asect`:

```
addressof( sect: "asect")
```

This function only works in assignments and **struct** statements.

checksum()

```
int checksum( checksum_algo, expr, expr )
```

Returns the computed checksum over a contiguous address range. The first argument specifies how the checksum must be computed (see below), the second argument is an expression that represents the start address of the range, while the third argument represents the end address (exclusive). The value of the end address expression must be strictly larger than the value of the start address (i.e. the size of the checksum address range must be at least one MAU). Each address in the range must point to a valid memory location. Memory locations in the address range that are not occupied by a section are filled with zeros.

The only checksum algorithm (*checksum_algo*) currently supported is **crc32w**. This algorithm computes the checksum using a Cyclic Redundancy Check with the "CRC-32" polynomial 0xEDB88320. The input range is processed per 4-byte word. Those 4 bytes are passed to the checksum algorithm in reverse order if the target architecture is little-endian. For big-endian targets, this checksum algorithm is equal to a regular byte-wise CRC-32 implementation. Both the start address and end address values must be aligned on 4 MAUs. The behavior of this checksum algorithm is undefined when used in an address space that has a MAU size not equal to 8.

```
checksum( crc32w,
    addressof( mem:foo ),
    addressof( mem:foo ) + sizeof( mem:foo ) )
```

This function only works in **struct** statements.

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 hat 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 works in assignments and **struct** statements. The **mem** argument can be used anywhere in section layouts. If the referenced object is a group or memory, it must be defined in the LSL file.

15.2.6. 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.

15.2.7. Memory and Bus Definitions

mem_def   ::= **memory** *mem_name* <*tag_attr*>$^{0|1}$ **{** <*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*
        | **priority =** *number*
        | **exec_priority =** *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 **priority** (or **speed**) statement (if absent, the default value is 1).

- A *mem_def* contains zero or one **exec_priority** statement.

- A *mem_def* contains zero or one **fill** statement.

- A *mem_def* contains at least one *mapping*

mem_type   ::= **rom**   // attrs = rx
        | **ram**   // attrs = rw
        | **nvram** // attrs = rwx
        | **blockram**

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_name*>$^{0|1}$ **(** *map_descr* <**,** *map_descr*>$^{>=0}$ **)**

map_descr   ::= **dest =** *destination*
                      | **dest_dbits =** *range*
                      | **dest_offset =** *expr*
                      | **size =** *expr*
                      | **src_dbits =** *range*
                      | **src_offset =** *expr*
                      | **reserved**
                      | **priority =** *number*
                      | **exec_priority =** *number*
                      | *tag*

- A *map_descr* requires at least the **size** and **dest** statements.

- A *map_descr* contains zero or one **priority** statement (if absent, the default value is 0).

- A *map_descr* contains zero or one **exec_priority** statement.

- 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.

- The **reserved** statement is allowed only in mappings defined for a memory.

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.

15.2.8. 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* <*tag_attr*>$^{0|1}$ **{** <*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_descr
                             <, stack_descr ><sup>>=0</sup> )
```

- A *stack_def* defines a stack with the *stack_name* as a unique name.

```
stack_descr          ::= min_size = expr
                     | grows = direction
                     | align = expr
                     | fixed
                     | entry_points = entry_point_list
                     | attributes
                     | tag
```

```
entry_point_list  ::= symbol_name
                     | [ symbol_name <, symbol_name ><sup>>=0</sup> ]
```

- The **min_size** statement must be present.

- The **min_size** value must be 1 or greater.

- You can specify at most one **align** statement and one **grows** statement.

- Each stack definition can have one or more **entry_points** statements for stack estimation. The *symbol_name* corresponds to the caller name in the .CALLS directive as generated by the compiler.

```
heap_def             ::= heap heap_name ( heap_descr
                             <, heap_descr ><sup>>=0</sup> )
```

- A *heap_def* defines a heap with the *heap_name* as a unique name.

```
heap_descr           ::= min_size = expr
                     | grows = direction
                     | align = expr
                     | fixed
                     | attributes
                     | tag
```

- The **min_size** statement must be present.

- The **min_size** value must be 1 or greater.

754

- You can specify at most one **align** statement and one **grows** statement.

```
direction          ::= low_to_high
                     | high_to_low
```

- If you do not specify the **grows** statement, the stack and heap grow **low-to-high**.

```
copy_table_def     ::= copytable <( copy_table_descr
                            <, copy_table_descr >>=0 )>0|1
```

- A *space_def* contains at most one **copytable** statement.

- Exactly one copy table must be defined in one of the spaces.

```
copy_table_descr   ::= align = expr
                     | copy_unit = expr
                     | dest <space_name>0|1 = space_name
                     | page
                     | tag
```

- The **copy_unit** is defined by the size in MAUs in which the startup code moves data.

- The **dest** statement is only required when the startup code initializes memory used by another processor that has no access to ROM.

- A *space_name* refers to a defined address space.

```
start_addr         ::= 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
                     | tag
```

```
vector_def         ::= vector ( vector_spec <, vector_spec>>=0 );
```

755

```
vector_spec        ::= id = vector_id_spec
                     | fill = vector_value
                     | optional
                     | tag

vector_id_spec     ::= number
                     | [ range ] <, [ range ]>>=0

vector_value       ::= symbol_name
                     | [ number <, number>>=0 ]
                     | loop <[ expr ]>0|1

reserved_range     ::= reserved <tag_attr>0|1 expr .. expr ;
```

• The end address is not part of the range.

```
endianness_def     ::= endianness { <endianness_type;>>=1 }

endianness_type    ::= big
                     | little
```

15.2.9. Derivative Definition

```
derivative_definition
                   ::= derivative derivative_name
                       <( parameter_list )>0|1
                       <extends derivative_name <( argument_list )>0|1
                             <, derivative_name <( argument_list )>0|1>>=0 >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*.

15.2.10. 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.

15.2.11. Section Setup

```
section_setup       ::= section_setup space_ref <tag_attr>^{0|1}
                        { <section_setup_item>^{>=0} }

section_setup_item
                    ::= vector_table_statement
                      | reserved_range
                      | stack_def ;
                      | heap_def ;
                      | copy_table_def ;
                      | start_address ;
                      | modify linktime_modification

linktime_modification
                    ::= input ( input_modifier <,input_modifier>^{>=0} )
                        { <select_section_statement ; >^{>=0} }

input_modifier      ::= space = space_ref
                      | attributes = < <+|-> attribute>^{>0}
                      | copy
                      | nocopy
```

• An *input_modifier* contains at most one **space** statement.

• An *input_modifier* contains at most one **attributes** statement.

15.2.12. Section Layout Definition

section_definition ::= **section_layout** <*space_ref*>$^{0|1}$
                        <**(** *space_layout_properties* **)**>$^{0|1}$
                        { <*section_statement*>$^{>=0}$ **}**

- A section definition inside a space definition does not have a *space_ref*.

- All global section definitions have a *space_ref*.

space_ref   ::= <*proc_name*>$^{0|1}$ **:** <*core_name*>$^{0|1}$
                           : *space_name* <**|** *space_name*>$^{>=0}$

- If more than one processor is present, the *proc_name* must be given for a global section layout.

- If the section layout refers to a processor that has more than one core, the *core_name* must be given in the *space_ref*.

- A *proc_name* refers to a defined processor.

- A *core_name* refers to a defined core.

- A *space_name* refers to a defined address space.

space_layout_properties
                  ::= *space_layout_property* <**,** *space_layout_property* >$^{>=0}$

space_layout_property
                  ::= *locate_direction*
                   | *tag*

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}
                       | tag
```

- **+**_attribute_ means: select all sections that have this attribute.

- **-**_attribute_ means: select all sections that do not have this attribute.

```
special_section_statement
                     ::= heap heap_name <stack_heap_mods>^{0|1}
                       | stack stack_name <stack_heap_mods>^{0|1}
                       | copytable
                       | reserved section_name <reserved_specs>^{0|1}
```

- Special sections cannot be selected in load-time groups.

```
stack_heap_mods      ::= ( stack_heap_mod <, stack_heap_mod>^{>=0} )
```

```
stack_heap_mod       ::= size = expr
                       | tag
```

```
reserved_specs       ::= ( reserved_spec <, reserved_spec>^{>=0} )
```

```
reserved_spec        ::= attributes
                       | fill_spec
                       | size = expr
                       | alloc_allowed = absolute | ranged
```

- If a **reserved** section has attributes **r**, **rw**, **x**, **rx** or **rwx**, and no fill pattern is defined, the section is filled with zeros. If no attributes are set, the section is created as a scratch section (attributes **ws**, no image).

```
fill_spec            ::= fill = fill_values
```

```
fill_values          ::= expr
                       | [ expr <, expr>^{>=0} ]
```

```
aggregate_section_statement
                     ::= { <section_statement>^{>=0} }
                       | group_descr
```

759

```
                       | if_statement
                       | section_creation_statement
                       | struct_statement

group_descr        ::= group <group_name>⁰|¹ <( group_specs )>⁰|¹
                           section_statement
```

- For every group with a name, the linker defines a label.

- No two groups for address spaces of a core can have the same $group\_name$.

```
group_specs        ::= group_spec <, group_spec >>=⁰

group_spec         ::= group_alignment
                       | attributes
                       | copy
                       | nocopy
                       | group_load_address
                       | fill <= fill_values>⁰|¹
                       | group_page
                       | group_run_address
                       | group_type
                       | allow_cross_references
                       | priority = number
                       | tag
```

- The **allow-cross-references** property is only allowed for *overlay* groups.

- The **copy** and **nocopy** properties cannot be applied both to the same group.

- Sub groups inherit all properties from a parent group.

```
group_alignment    ::= align = expr

attributes         ::= attributes = <attribute>>=¹

attribute          ::= r    // readable sections
                       | w    // writable sections
                       | x    // executable code sections
                       | i    // initialized sections
                       | s    // scratch sections
                       | b    // blanked (cleared) sections
                       | p    // protected sections

group_load_address
                   ::= load_addr <= load_or_run_addr>⁰|¹

group_page         ::= page <= expr>⁰|¹
                       | page_size = expr <[ range ] <| [ range ]>>=⁰>⁰|¹

group_run_address  ::= run_addr <= load_or_run_addr>⁰|¹
```

```
group_type          ::= clustered
                      |  contiguous
                      |  ordered
                      |  overlay
```

- For *non-contiguous* groups, you can only specify `group_alignment` and `attributes`.

- The **overlay** keyword also sets the **contiguous** property.

- The **clustered** property cannot be set together with **contiguous** or **ordered** on a single group.

```
load_or_run_addr   ::= addr_absolute
                     | addr_range <| addr_range>^{>=0}
```

```
addr_absolute      ::= expr
                     | memory_reference [ expr ]
```

- An absolute address can only be set on *ordered* groups.

```
addr_range         ::= [ expr .. expr ]
                     | memory_reference
                     | memory_reference [ expr .. expr ]
```

- The parent of a group with an `addr_range` or **page** restriction cannot be **ordered**, **contiguous** or **clustered**.

- The end address is not part of the range.

```
memory_reference   ::= mem : <proc_name :>^{0|1} mem_name </ map_name>^{0|1}
```

- A `proc_name` refers to a defined processor.

- A `mem_name` refers to a defined memory.

- A `map_name` refers to a defined memory mapping.

```
if_statement       ::= if ( expr ) section_statement
                       <else section_statement>^{0|1}
```

```
section_creation_statement
                   ::= section section_name ( section_specs )
                       { <section_statement2>^{>=0} }
```

```
section_specs      ::= section_spec <, section_spec >^{>=0}
```

```
section_spec       ::= attributes
                     | fill_spec
                     | size = expr
                     | blocksize = expr
                     | overflow = section_name
                     | tag
```

```
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
                   | nocopy
                   | tag

struct_statement
                 ::= struct { <struct_item>>=0 }

struct_item       ::= expr : number ;
```

15.3. Expression Evaluation

Only *constant* expressions are allowed, including sizes, but not addresses, of sections in object files.

All expressions are evaluated with 64-bit precision integer arithmetic. The result of an expression can be absolute or relocatable. A symbol you assign is created as an absolute symbol. Symbol references are only allowed in symbol assignments and **struct** statements.

15.4. Semantics of the Architecture Definition

Keywords in the architecture definition

```
architecture
   extends
endianness         big  little
bus
   mau
   width
   map
space
   id
   mau
   align
   page_size
   page
   direction       low_to_high  high_to_low
   stack
      min_size
      grows         low_to_high  high_to_low
      align
      fixed
      entry_points
      attributes   b
   heap
      min_size
      grows         low_to_high  high_to_low
      align
      fixed
      attributes   b
   copytable
      align
      copy_unit
      dest
      page
   vector_table
      vector_size
      size
      id_symbol_prefix
      run_addr
      template
      template_symbol
      vector_prefix
      fill
      no_inline
      copy
      vector
         id
```

763

```
        fill        loop
        optional
reserved
start_address
    run_addr
    symbol
map

map
    dest            bus  space
    dest_dbits
    dest_offset
    size
    src_dbits
    src_offset
    priority
    exec_priority
```

15.4.1. Defining an Architecture

With the keyword **architecture** you define an architecture and assign a unique name to it. The name is used to refer to it at other places in the LSL file:

architecture *name*
{
    definitions
}

If you are defining multiple core architectures that show great resemblance, you can define the common features in a parent core architecture and extend this with a child core architecture that contains specific features. The child inherits all features of the parent. With the keyword extends you create a child core architecture:

architecture *name_child_arch* **extends** *name_parent_arch*
{
    definitions
}

A core architecture can have any number of parameters. These are identifiers which get values assigned on instantiation or extension of the architecture. You can use them in any expression within the core architecture. Parameters can have default values, which are used when the core architecture is instantiated with less arguments than there are parameters defined for it. When you extend a core architecture you can pass arguments to the parent architecture. Arguments are expressions that set the value of the parameters of the sub-architecture.

architecture *name_child_arch* (**parm1,parm2=1**)
        extends *name_parent_arch* (*arguments*)
{
    definitions
}

15.4.2. Defining Internal Buses

With the **bus** keyword you define a bus (the combination of data and corresponding address bus). The bus name is used to identify a bus and does not conflict with other identifiers. Bus descriptions in an architecture definition or derivative definition define *internal* buses. Some internal buses are used to communicate with the components outside the core or processor. Such buses on a processor have physical pins reserved for the number of bits specified with the **width** statements.

- The **mau** field specifies the MAU size (Minimum Addressable Unit) of the data bus. This field is required and must be non-zero.

- The **width** field specifies the width (number of address lines) of the data bus. The default value is the MAU size.

- The **map** keyword specifies how this bus maps onto another bus (if so). Mappings are described in Section 15.4.4, *Mappings*.

```
bus bus_name
{
     mau = 8;
     width = 8;
     map ( map_description );
}
```

15.4.3. Defining Address Spaces

With the **space** keyword you define a logical address space. The space name is used to identify the address space and does not conflict with other identifiers.

- The **id** field defines how the addressing space is identified in object files. In general, each address space has a unique ID. The linker locates sections with a certain ID in the address space with the same ID. This field is required.

- The **mau** field specifies the MAU size (Minimum Addressable Unit) of the space. This field is required and must be non-zero.

- The **align** value must be a power of two. The linker uses this value to compute the start addresses when sections are concatenated. An align value of *n* means that objects in the address space have to be aligned on *n* MAUs.

- The **page_size** field sets the page alignment and page size in MAUs for the address space. It must be a power of 2. The default value is 1. If one or more page ranges are supplied the supplied value only sets the page alignment. The ranges specify the available space in each page, as offsets to the page start, which is aligned at the page alignment.

  See also the **page** keyword in subsection Locating a group in Section 15.8.2, *Creating and Locating Groups of Sections*.

- With the optional **direction** field you can specify how all sections in this space should be located. This can be either from **low_to_high** addresses (this is the default) or from **high_to_low** addresses.

- The **map** keyword specifies how this address space maps onto an internal bus or onto another address space. Mappings are described in Section 15.4.4, *Mappings*.

Stacks and heaps

- The **stack** keyword defines a stack in the address space and assigns a name to it. The architecture definition must contain at least one stack definition. Each stack of a core architecture must have a unique name. See also the **stack** keyword in Section 15.8.3, *Creating or Modifying Special Sections*.

  The stack is described in terms of a minimum size (**min_size**) and the direction in which the stack grows (**grows**). This can be either from **low_to_high** addresses (stack grows upwards, this is the default) or from **high_to_low** addresses (stack grows downwards). The **min_size** is required.

  By default, the linker tries to maximize the size of the stacks and heaps. After locating all sections, the largest remaining gap in the space is used completely for the stacks and heaps. If you specify the keyword **fixed**, you can disable this so-called 'balloon behavior'. The size is also fixed if you used a stack or heap in the software layout definition in a restricted way. For example when you override a stack with another size or select a stack in an ordered group with other sections.

  Please note that the stack and heap in the ARC/PPU LSL files include the **fixed** keyword by default.

  A stack may have an **attributes** property with value **b**. Such a stack must be cleared at program startup. No other attributes are allowed.

  Optionally you can specify an alignment for the stack with the argument **align**. This alignment must be equal or larger than the alignment that you specify for the address space itself.

  For each stack, a stack size estimation may be computed (and listed in a map file) from a call graph. Each root node of the call graph is treated as a separate thread that can run independently from the other threads. Root nodes can be specified using the **entry_points** keyword. The estimated stack usage for a root node is the highest sum of stack usage values along a path to a leaf node. The total estimated stack usage of a link task is the sum of the calculated stack usage of such independent call graphs.

  A stack definition may have one or more **entry_points** statements that specify the code that uses that stack - all functions that are reachable (by calls) from the entry points are considered to be using the stack at run-time. Each symbol name specified as an entry point must match a node in the call graph, which must not have more than one caller. As a result of the entry point specification, the specific call (edge) is removed from the call graph (so the symbol becomes a root). A symbol may be declared as entry point for multiple stacks.

- 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 15.8.3, *Creating or Modifying Special Sections*.

Stacks and heaps are only generated by the linker if the corresponding linker labels are referenced in the object files.

See Section 15.8, *Semantics of the Section Layout Definition*, for information on creating and placing stack sections.

Copy tables

- The **copytable** keyword defines a copy table in the address space. The content of the copy table is created by the linker and contains the start address and size of all sections that should be initialized by the startup code. You must define exactly one copy table in one of the address spaces (for a core).

  Optionally you can specify an alignment for the copy table with the argument **align**. This alignment must be equal or larger than the alignment that you specify for the address space itself. If smaller, the alignment for the address space is used.

  The **copy_unit** argument specifies the size in MAUs of information chunks that are copied. If you do not specify the copy unit, the MAU size of the address space itself is used.

  The **dest** argument specifies the destination address space that the code uses for the copy table. The linker uses this information to generate the correct addresses in the copy table. The memory into where the sections must be copied at run-time, must be accessible from this destination space.

  Sections generated for the copy table may get a page restriction with the address space's page size, by adding the **page** argument.

Vector table

- The **vector_table** keyword defines a vector table with *n* vectors of size *m* (This is an internal LSL object similar to an LSL group.) The **run_addr** argument specifies the location of the first vector (id=0). This can be a simple address or an offset in memory (see the description of the run-time address in subsection Locating a group in Section 15.8.2, *Creating and Locating Groups of Sections*). A vector table defines symbols _lc_ub_foo and _lc_ue_foo pointing to start and end of the table.

```
vector_table "vector_table" (vector_size=m, size=n, run_addr=x, ...)
```

  See the following example of a vector table definition:

```
vector_table "vector_table" (vector_size = 4, size = 16, run_addr=0,
                    template=".text.handler.address",
                    template_symbol="_lc_vector_handler",
                    vector_prefix="_vector_",
                    id_symbol_prefix="foo",
                    no_inline,
                    /* default: empty, or */
                    fill="foo", /* or */
                    fill=[1,2,3,4], /* or */
                    fill=loop)
{
    vector (id=23, fill="main", optional);
    vector (id=12, fill=[0xab, 0x21, 0x32, 0x43]);
    vector (id=[1..11], fill=[0]);
    vector (id=[18..23], fill=loop);
}
```

The **template** argument defines the name of the section that holds the code to jump to a handler function from the vector table. This template section does not get located and is removed when the locate phase is completed. This argument is required.

The **template_symbol** argument is the symbol reference in the template section that must be replaced by the address of the handler function. This symbol name should start with the linker prefix for the symbol to be ignored in the link phase. This argument is required.

The **vector_prefix** argument defines the names of vector sections: the section for a vector with id *vector_id* is $(*vector_prefix*)$(*vector_id*). Vectors defined in C or assembly source files that should be included in the vector table must have the correct symbol name. The compiler uses the prefix that is defined in the default LSL file(s); if this attribute is changed, the vectors declared in C source files are not included in the vector table. When a vector supplied in an object file has exactly one relocation, the linker will assume it is a branch to a handler function, and can be removed when the handler is inlined in the vector table. Otherwise, no inlining is done. This argument is required.

With the optional **no_inline** argument the vectors handlers are not inlined in the vector table.

With the optional **copy** argument a ROM copy of the vector table is made and the vector table is copied to RAM at startup.

With the optional **id_symbol_prefix** argument you can set an internal string representing a symbol name prefix that may be found on symbols in vector handler code. When the linker detects such a symbol in a handler, the symbol is assigned the vector number. If the symbol was already assigned a vector number, a warning is issued.

The **fill** argument sets the default contents of vectors. If nothing is specified for a vector, this setting is used. See below. When no default is provided, empty vectors may be used to locate large vector handlers and other sections. Only one **fill** argument is allowed.

The **vector** field defines the content of vector with the number specified by **id**. If a range is specified for **id** ([p..q,s..t]) all vectors in the ranges (inclusive) are defined the same way.

With **fill=**_symbol_name_, the vector must jump to this symbol. If the section in which the symbol is defined fits in the vector table (size may be >*m*), locate the section at the location of the vector. Otherwise, insert code to jump to the symbol's value. A template interrupt handler section name **+** symbol name for the target code must be supplied in the LSL file.

fill=[_value(s)_**]**, fills the vector with the specified MAU values.

With **fill=loop** the vector jumps to itself. With the optional **[**_offset_**]** you can specify an offset from the vector table entry.

When the keyword **optional** is set on a vector specification with a symbol value and the symbol is not found, no error is reported. A default fill value is used if the symbol was not found. With other values the attribute has no effect.

Reserved address ranges

- The **reserved** keyword specifies to reserve a part of an address space even if not all of the range is covered by memory. See also the **reserved** keyword in Section 15.8.3, *Creating or Modifying Special Sections*.

Start address

* The `start_address` keyword specifies the start address for the position where the C startup code is located. When a processor is reset, it initializes its program counter to a certain start address, sometimes called the reset vector. In the architecture definition, you must specify this start address in the correct address space in combination with the name of the label in the application code which must be located here.

  The `run_addr` argument specifies the start address (reset vector). If the core starts executing using an entry from a vector table, and directly jumps to the start label, you should omit this argument.

  The `symbol` argument specifies the name of the label in the application code that should be located at the specified start address. The `symbol` argument is required. The linker will resolve the start symbol and use its value after locating for the start address field in IEEE-695 files and Intel Hex files. If you also specified the `run_addr` argument, the start symbol (label) must point to a section. The linker locates this section such that the start symbol ends up on the start address.

```
space space_name
{
    id = 1;
    mau = 8;
    align = 8;
    page_size = 1;
    stack name (min_size = 1k, grows = low_to_high);
    reserved start_address .. end_address;
    start_address ( run_addr = 0x0000,
                    symbol = "start_label" )
    map ( map_description );
}
```

15.4.4. Mappings

You can use a mapping when you define a space, bus or memory. With the `map` field you specify how addresses from the source (space, bus or memory) are translated to addresses of a destination (space, bus). The following mappings are possible:

* space => space

* space => bus

* bus => bus

* memory => bus

With a mapping you specify a range of source addresses you want to map (specified by a source offset and a size), the destination to which you want to map them (a bus or another address space), and the offset address in the destination.

* The `dest` argument specifies the destination. This can be a `bus` or another address `space` (only for a space to space mapping). This argument is required.

- The `src_offset` argument specifies the offset of the source addresses. In combination with size, this specifies the range of address that are mapped. By default the source offset is 0x0000.

- The `size` argument specifies the number of addresses that are mapped. This argument is required.

- The `dest_offset` argument specifies the position in the destination to which the specified range of addresses is mapped. By default the destination offset is 0x0000.

If you are mapping a bus to another bus, the number of data lines of each bus may differ. In this case you have to specify a range of source data lines you want to map (`src_dbits` = *begin..end*) and the range of destination data lines you want to map them to (`dest_dbits` = *first..last*).

- The `src_dbits` argument specifies a range of data lines of the source bus. By default all data lines are mapped.

- The `dest_dbits` argument specifies a range of data lines of the destination bus. By default, all data lines from the source bus are mapped on the data lines of the destination bus (starting with line 0).

A mapping can optionally have a name which can be referenced in an address assignment.

If you define a memory and the memory mapping must not be used by default when locating sections in address spaces, you can specify the `reserved` argument. This marks all address space areas that the mapping points to as reserved. If a section has an absolute or address range restriction, the reservation is lifted and the section may be located at these locations. This feature is only useful when more than one mapping is available for a range of memory addresses, otherwise the `memory` keyword with the same name would be used.

For example:

```
memory xrom
{
    mau = 8;
    size = 1M;
    type = rom;
    map cached   (dest=bus:mycore:local_bus, dest_offset=0x80000000,
                  size=1M);
    map uncached (dest=bus:mycore:local_bus, dest_offset=0xa0000000,
                  size=1M, reserved);
}
```

Mapping priority

If you define a memory you can set a locate priority on a mapping with the keywords `priority` and `exec_priority`. The values of these priorities are relative which means they add to the priority of memories. Whereas a priority set on the memory applies to all address space areas reachable through any mapping of the memory, a priority set on a mapping only applies to address space areas reachable through the mapping. The memory mapping with the highest priority is considered first when locating. To set only a priority for non-executable (data) sections, add a `priority` keyword with the desired value and an `exec_priority` set to zero. To set only a priority for executable (code) sections, simply set an `exec_priority` keyword to the desired value.

The default for a mapping **priority** is zero, while the default for **exec_priority** is the same as the specified **priority**. If you specify a value for **priority** in LSL it must be greater than zero. A value for **exec_priority** must be greater or equal to zero.

For more information about priority values see the description of the memory **priority** keyword.

```
memory myram
{
    mau = 8;
    size = 112k;
    type = ram;
    map (dest=bus:mycore:local_bus, dest_offset=0xd0000000,
            size=112k, priority=8, exec_priority=0);
    map (dest=bus:mycore:local_bus, dest_offset=0x70000000,
            size=112k);
}
```

From space to space

If you map an address space to another address space (nesting), you can do this by mapping the subspace to the containing larger space. In this example a small space of 64 kB is mapped on a large space of 16 MB.

```
space small
{
    id = 2;
    mau = 4;
    map (src_offset = 0, dest_offset = 0,
        dest = space : large, size = 64k);
}
```

From space to bus

All spaces that are not mapped to another space must map to a bus in the architecture:

```
space large
{
    id = 1;
    mau = 4;
    map (src_offset = 0, dest_offset = 0,
        dest = bus:bus_name, size = 16M );
}
```

From bus to bus

The next example maps an external bus called e_bus to an internal bus called i_bus. This internal bus resides on a core called mycore. The source bus has 16 data lines whereas the destination bus has only 8 data lines. Therefore, the keywords **src_dbits** and **dest_dbits** specify which source data lines are mapped on which destination data lines.

```
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.

15.5. Semantics of the Derivative Definition

Keywords in the derivative definition

```
derivative
   extends
core
   architecture
bus
   mau
   width
   map
memory
   type           reserved rom  ram  nvram  blockram
   mau
   size
   speed
   priority
   exec_priority
   fill
   map
section_layout
section_setup

   map
      dest        bus  space
```

```
        dest_dbits
        dest_offset
        size
        src_dbits
        src_offset
        priority
        exec_priority
        reserved
```

15.5.1. Defining a Derivative

With the keyword **derivative** you define a derivative and assign a unique name to it. The name is used to refer to it at other places in the LSL file:

```
derivative name
{
        definitions
}
```

If you are defining multiple derivatives that show great resemblance, you can define the common features in one or more parent derivatives and extend this with a child derivative that contains specific features. The child inherits all features of the parent (cores and memories). With the keyword **extends** you create a child derivative:

```
derivative name_child_deriv extends name_parent_derivs
{
        definitions
}
```

As with a core architecture, a derivative can have any number of parameters. These are identifiers which get values assigned on instantiation or extension of the derivative. You can use them in any expression within the derivative definition.

```
derivative name_child_deriv (parm1,parm2=1)
            extends name_parent_deriv (arguments)
{
        definitions
}
```

15.5.2. Instantiating Core Architectures

With the keyword **core** you instantiate a core architecture in a derivative.

• With the keyword **architecture** you tell the linker that the given core has a certain architecture. The architecture name refers to an existing architecture definition in the same LSL file.

For example, if you have two cores (called mycore_1 and mycore_2) that have the same architecture (called mycorearch), you must instantiate both cores as follows:

```
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);
}
```

15.5.3. Defining Internal Memory and Buses

With the keyword **memory** you define physical memory that is present on the target board. The memory name is used to identify the memory and does not conflict with other identifiers. It is common to define internal memory (on-chip) in the derivative definition. External memory (off-chip memory) is usually defined in the board specification (See Section 15.6.3, *Defining External Memory and Buses*).

- The **type** field specifies a memory type:

  - **rom**: read-only memory - it can only be written at load-time

  - **ram**: random access volatile writable memory - writing at run-time is possible while writing at load-time has no use since the data is not retained after a power-down

  - **nvram**: non volatile ram - writing is possible both at load-time and run-time

  - **blockram**: writing is possible both at load-time and run-time. Changes are applied in RAM, so after a full device reset the data in a blockram reverts to the original state.

  The optional **reserved** qualifier before the memory type, tells the linker not to locate any section in the memory by default. You can locate sections in such memories using an absolute address or range restriction (see subsection Locating a group in Section 15.8.2, *Creating and Locating Groups of Sections*).

- The **mau** field specifies the MAU size (Minimum Addressable Unit) of the memory. This field is required and must be non-zero.

- The **size** field specifies the size in MAU of the memory. This field is required.

- The **priority** field specifies a locate priority for a memory. The **speed** field has the same meaning but is considered deprecated. By default, a memory has its priority set to 1. The memories with the highest priority are considered first when trying to locate a rule. Subsequently, the next highest priority memories are added if the rule was not located successfully, and so on until the lowest priority that is available is reached or the rule is located. The lowest priority value is zero. Sections with an **ordered**

and/or **contiguous** restriction are not affected by the locate priority. If such sections also have a **page** restriction, the locate priority is still used to select a page.

- If an **exec_priority** is specified for a memory, the regular priority (either specified or its default value) does not apply to locate rules with only executable sections. Instead, the supplied value applies for such rules. Additionally, the **exec_priority** value is used for any executable unrestricted sections, even if they appear in an unrestricted rule together with non-executable sections.

- The **map** field specifies how this memory maps onto an (internal) bus. The mapping can have a name. Mappings are described in Section 15.4.4, *Mappings*.

- The optional **fill** field contains a bit pattern that the linker writes to all memory addresses that remain unoccupied during the locate process. The result of the expression, or list of expressions, is used as values to write to memory, each in MAU.

```
memory mem_name
{
    type = rom;
    mau = 8;
    fill = 0xaa;
    size = 64k;
    priority = 2;
    map map_name ( map_description );
}
```

With the **bus** keyword you define a bus in a derivative definition. Buses are described in Section 15.4.2, *Defining Internal Buses*.

15.6. Semantics of the Board Specification

Keywords in the board specification

```
processor
   derivative
bus
   mau
   width
   map
memory
   type            reserved  rom  ram  nvram  blockram
   mau
   size
   speed
   priority
   exec_priority
   fill
   map

   map
      dest          bus  space
```

```
        dest_dbits
        dest_offset
        size
        src_dbits
        src_offset
        priority
        exec_priority
        reserved
```

15.6.1. Defining a Processor

If you have a target board with multiple processors that have the same derivative, you need to instantiate each individual processor in a processor definition. This information tells the linker which processor has which derivative and enables the linker to distinguish between the present processors.

If you use processors that all have a unique derivative, you may omit the processor definitions. In this case the linker assumes that for each derivative definition in the LSL file there is one processor. The linker uses the derivative name also for the processor.

With the keyword **processor** you define a processor. You can freely choose the processor name. The name is used to refer to it at other places in the LSL file:

```
processor proc_name
{
    processor definition
}
```

15.6.2. Instantiating Derivatives

With the keyword **derivative** you tell the linker that the given processor has a certain derivative. The derivative name refers to an existing derivative definition in the same LSL file.

For example, if you have two processors on your target board (called myproc_1 and myproc_2) that have the same derivative (called myderiv), you must instantiate both processors as follows:

```
processor myproc_1
{
    derivative = myderiv;
}

processor myproc_2
{
    derivative = myderiv;
}
```

If the derivative definition has parameters you must specify the arguments that correspond with the parameters. For example myderiv1 expects two parameters which are used in the derivative definition:

```
processor myproc
{
```

```
    derivative = myderiv1 (2,4);
}
```

15.6.3. Defining External Memory and Buses

It is common to define external memory (off-chip) and external buses at the global scope (outside any enclosing definition). Internal memory (on-chip memory) is usually defined in the scope of a derivative definition.

With the keyword **memory** you define physical memory that is present on the target board. The memory name is used to identify the memory and does not conflict with other identifiers. If you define memory parts in the LSL file, only the memory defined in these parts is used for placing sections.

If no external memory is defined in the LSL file and if the linker option to allocate memory on demand is set then the linker will assume that all virtual addresses are mapped on physical memory. You can override this behavior by specifying one or more memory definitions.

```
memory mem_name
{
    type = rom;
    mau = 8;
    fill = 0xaa;
    size = 64k;
    priority = 2;
    map map_name ( map_description );
}
```

For a description of the keywords, see Section 15.5.3, *Defining Internal Memory and Buses*.

With the keyword **bus** you define a bus (the combination of data and corresponding address bus). The bus name is used to identify a bus and does not conflict with other identifiers. Bus descriptions at the global scope (outside any definition) define external buses. These are buses that are present on the target board.

```
bus bus_name
{
    mau = 8;
    width = 8;
    map ( map_description );
}
```

For a description of the keywords, see Section 15.4.2, *Defining Internal Buses*.

You can connect off-chip memory to any derivative: you need to map the off-chip memory to a bus and map that bus on the internal bus of the derivative you want to connect it to.

15.7. 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
   copytable
      align
      copy_unit
      dest
      page
   vector_table
      vector_size
      size
      id_symbol_prefix
      run_addr
      template
      template_symbol
      vector_prefix
      fill
      no_inline
      copy
      vector
         id
         fill       loop
         optional
   reserved
   start_address
      run_addr
      symbol
 modify input
      space
      attributes
      copy
      nocopy
```

15.7.1. Setting up a Section

With the keyword **section_setup** you can define stacks, heaps, copy tables, vector tables, start address and/or reserved address ranges outside their address space definition.

```
section_setup ::my_space
{
    vector table statements
    reserved address range
    stack definition
    heap definition
    copy table definition
    start adress
    input section modifications
}
```

See the subsections Stacks and heaps, Copy tables, Start address, Vector table and Reserved address ranges in Section 15.4.3, *Defining Address Spaces* for details on the keywords **stack**, **heap**, **copytable**, **vector_table** and **reserved**.

Input section modifications

Before sections are located and before selections defined in section_layout are performed, you can still modify a few section properties. These are:

- change the address space of a section

- add (+w) or remove (-w) the writable attribute

- add (+p) or remove (-p) the protected attribute

- add (+u) or remove (-u) the uncached attribute

- change a noclear/scratch section into a clear/blank section (attributes=+b) and vice versa (attributes=-b)

- prevent an initialized writable section from being copied from ROM to RAM, without making it read-only, using the **nocopy** keyword.

You cannot set the protected attribute on linker created sections like reserved sections and output sections.

Sections are selected the same way as in groups in a section_layout. Instead of attributes=+w you can use the **copy** keyword.

```
section_setup ::A
{
    modify input (space=::B, attributes=+w)
    {
        select "mysection";
    }
}
```

Note that the new address space must be used to select a modified section in a `section_layout`. To locate the section `mysection` in the example somewhere, it must be selected in a `section_layout` for space `::B`. If the link result is output to a file, for example by only linking or incremental linking, the modified properties are exported. So, when the resulting file is used in another invocation of the linker, the section can appear in a different address space.

15.8. Semantics of the Section Layout Definition

Keywords in the section layout definition

```
section_layout
   direction      low_to_high  high_to_low
group
   align
   attributes     + -  r w x b i s p
   copy
   nocopy
   fill
   ordered
   contiguous
   clustered
   overlay
   allow_cross_references
   load_addr
      mem
   run_addr
      mem
   page
   page_size
   priority
select
stack
   size
heap
   size
reserved
   size
   attributes     r w x
   fill
   alloc_allowed absolute ranged
copytable
section
   size
   blocksize
   attributes     r w x
   fill
   overflow
struct
   checksum
```

```
if
else
```

15.8.1. Defining a Section Layout

With the keyword **section_layout** you define a section layout for exactly one address space. In the section layout you can specify how input sections are placed in the address space, relative to each other, and what the absolute run and load addresses of each section will be.

You can define one or more section definitions. Each section definition arranges the sections in one address space. You can precede the address space name with a processor name and/or core name, separated by colons. You can omit the processor name and/or the core name if only one processor is defined and/or only one core is present in the processor. A reference to a space in the only core of the only processor in the system would look like "::my_space". A reference to a space of the only core on a specific processor in the system could be "my_chip::my_space". The next example shows a section definition for sections in the my_space address space of the processor called my_chip:

```
section_layout my_chip::my_space ( locate_direction )
{
    section statements
}
```

Locate direction

With the optional keyword **direction** you specify whether the linker starts locating sections from **low_to_high** (default) or from **high_to_low**. In the second case the linker starts locating sections at the highest addresses in the address space but preserves the order of sections when necessary (one processor and core in this example).

```
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.

15.8.2. Creating and Locating Groups of Sections

Sections are located per group. A group can contain one or more (sets of) input sections as well as other groups. Per group you can assign a mutual order to the sets of sections and locate them into a specific memory part.

```
group ( group_specifications )
{
    section_statements
}
```

With the `section_statements` you generally select sets of sections to form the group. This is described in subsection Selecting sections for a group.

Instead of selecting sections, you can also modify special sections like stack and heap or create a reserved section. This is described in Section 15.8.3, *Creating or Modifying Special Sections*.

With the `group_specifications` you actually locate the sections in the group. This is described in subsection Locating a group.

Selecting sections for a group

With the keyword **select** you can select one or more sections for the group. You can select a section by name or by attributes. If you select a section by name, you can use a wildcard selection pattern:

*	matches any number of characters
?	matches with a single character in the section name
\	escape character, takes the next character literally, you can use it to escape the special characters *, ?, [,], (,), | so it is not treated as a wildcard pattern
[abc]	matches with a single 'a', 'b' or 'c' character
[a-z]	matches with any single character in the range 'a' to 'z'
(*pattern*)	matches the specified *pattern*
pattern1 | *pattern2*	matches either *pattern1* or *pattern2* (or both)

```
group ( ... )
{
    select "mysection";
    select "*";
}
```

The first **select** statement selects the section with the name "`mysection`". The second **select** statement selects all sections that were not selected yet.

A section is selected by the first select statement that matches, in the union of all section layouts for the address space. Global section layouts are processed in the order in which they appear in the LSL file. Internal core architecture section layouts always take precedence over global section layouts.

When you use wildcards, the linker skips sections with an absolute address from the selection process, for example, a start section already having an absolute start address.

Note that when you select sections with an exact name (no wildcard special characters), all sections with that name are automatically protected against unreferenced section removal. With a selection using wildcard special characters, matching sections are selected, but matching sections that are unreferenced may be removed. For example, when you escape a wildcard special character *, ?, [,], (,), \| with \ it is not considered as a wildcard.

- The **attributes** field selects all sections that carry (or do not carry) the given attribute. With **+***attribute* you select sections that have the specified attribute set. With **-***attribute* you select sections that do not have the specified attribute set. You can specify one or more of the following attributes:

- **r** readable sections

- **w** writable sections

- **x** executable sections

- **i** initialized sections

- **b** sections that should be cleared at program startup

- **s** scratch sections (not cleared and not initialized)

- **p** protected sections

To select all read-only sections:

```
group ( ... )
{
    select (attributes = +r-w);
}
```

Keep in mind that all section selections are restricted to the address space of the section layout in which this group definition occurs.

- With the `ref_tree` field you can select a group of related sections. The relation between sections is often expressed by means of references. By selecting just the 'root' of tree, the complete tree is selected. This is for example useful to locate a group of related sections in special memory (e.g. fast memory). The (referenced) sections must meet the following conditions in order to be selected:

  1. The sections are within the section layout's address space

  2. The sections match the specified attributes

  3. The sections have no absolute restriction (as is the case for all wildcard selections)

  For example, to select the code sections referenced from `foo1`:

```
group refgrp (contiguous, run_addr=mem:ext_c)
{
    select ref_tree "foo1" (attributes=+x);
}
```

  If section `foo1` references `foo2` and `foo2` references `foo3`, then all these sections are selected by the selection shown above.

Locating a group

```
group group_name ( group_specifications )
{
   section_statements
}
```

With the `group_specifications` you actually define how the linker must locate the group. You can roughly define three things: 1) assign properties to the sections in a group like alignment and read/write attributes, 2) define the mutual order in the address space for sections in the group and 3) restrict the possible addresses for the sections in a group.

The linker creates labels that allow you to refer to the begin and end address of a group from within the application software. Labels `_lc_gb_`*group_name* and `_lc_ge_`*group_name* mark the begin and end of the group respectively, where the begin is the lowest address used within this group and the end is the highest address used. Notice that a group not necessarily occupies all memory between begin and end address. The given label refers to where the section is located at run-time (versus load-time).

1. Assign properties to the sections in a group like alignment and read/write attributes.

   These properties are assigned to all sections in the group (and subgroups) and override the attributes of the input sections.

   - The **align** field tells the linker to align all sections in the group according to the align value. The alignment of a section is first determined by its own initial alignment and the defined alignment for the address space. Alignments are never decreased, if multiple alignments apply to a section, the largest one is used.

   - The **attributes** field tells the linker to assign one or more attributes to all sections in the group. This overrules the default attributes. By default the linker uses the attributes of the input sections. You can set the **r**, **w**, or **rw** attributes and you can switch between the **b** and **s** attributes.

   - The **copy** field tells the linker to locate a read-only section in RAM and generate a ROM copy and a copy action in the copy table. This property makes the sections in the group writable which causes the linker to generate ROM copies for the sections.

   - The effect of the **nocopy** field is the opposite of the **copy** field. It prevents the linker from generating ROM copies of the selected sections. You cannot apply both **copy** and **nocopy** to the same statement.

2. Define the mutual order of sections in an LSL group.

   By default, a group is *unrestricted* which means that the linker has total freedom to place the sections of the group in the address space.

   > Note that when you use the linker optimization option **--optimize=+copytable-compression**, unrestricted sections affected by the copy table are located as if they were in a clustered LSL group. This option is enabled by default.

   - The **ordered** keyword tells the linker to locate the sections in the same order in the address space as they appear in the group (but not necessarily adjacent).

     Suppose you have an ordered group that contains the sections 'A', 'B' and 'C'. By default the linker places the sections in the address space like 'A' - 'B' - 'C', where section 'A' gets the lowest possible address. With **direction=high_to_low** in the **section_layout** space properties, the linker places the sections in the address space like 'C' - 'B' - 'A', where section 'A' gets the highest possible address.

- The **contiguous** keyword tells the linker to locate the sections in the group in a single address range. Within a contiguous group the input sections are located in arbitrary order, however the group occupies one contiguous range of memory. Due to alignment of sections there can be 'alignment gaps' between the sections.

  When you define a group that is both **ordered** and **contiguous**, this is called a *sequential* group. In a sequential group the linker places sections in the same order in the address space as they appear in the group and it occupies a contiguous range of memory.

- The **clustered** keyword tells the linker to locate the sections in the group in a number of *contiguous* blocks. It tries to keep the number of these blocks to a minimum. If enough memory is available, the group will be located as if it was specified as **contiguous**. Otherwise, it gets split into two or more blocks.

  If a contiguous or clustered group contains *alignment gaps*, the linker can locate sections that are not part of the group in these gaps. To prevent this, you can use the **fill** keyword. When no fill pattern is specified, the gaps are treated as reserved (scratch) space. You can have the gaps filled with data by supplying an expression argument to the **fill** property. The result of the expression, or list of expressions, is used as values to write to memory, each in a 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 in a group can be restricted either to a single absolute address, or to a number of address ranges (not including the end address). With an expression you can specify that the group should be located at the absolute address specified by the expression:

```
group (ordered, run_addr = 0xa00f0000)
```

A group with an absolute address must be ordered, the first section in the group is located at the specified absolute address.

You can use the '[*offset*]' variant to locate the group at the given absolute offset in memory:

```
group (ordered, run_addr = mem:A[0x1000])
```

A group with an absolute address must be ordered, the first section in the group is located at the specified absolute offset in memory.

A range can be an absolute space address range, written as **[** *expr .. expr* **]**, a complete memory device, written as **mem:***mem_name*, or a memory address range, **mem:**mem_name**[***expr .. expr* **]**

```
group (run_addr = mem:my_dram)
```

You can use the '|' to specify an address range of more than one physical memory device:

```
group (run_addr = mem:A | mem:B)
```

When used in top-level section layouts, a memory name refers to a board-level memory. You can select on-chip memory with **mem:***proc_name***:***mem_name*. If the memory has multiple parallel mappings towards the current address space, you can select a specific named mapping in the memory by appending */map_name* to the memory specifier. The linker then maps memory offsets only through that mapping, so the address(es) where the sections in the group are located are determined by that memory mapping.

```
group (run_addr = mem:CPU1:A/cached)
```

- The **load_addr** keyword changes the meaning of the section selection in the group: the linker selects the load-time ROM copy of the named section(s) instead of the regular sections. Just like **run_addr** you can specify an absolute address or an address range.

```
group (contiguous, load_addr)
{
```

```
    select "mydata";  // select ROM copy of mydata:
                      // "[mydata]"
}
```

The load-time and run-time addresses of a group cannot be set at the same time. If the load-time property is set for a group, the group (only) restricts the positioning at load-time of the group's sections. It is not possible to set the address of a group that has a not-unrestricted parent group.

The properties of the load-time and run-time start address are:

- At run-time, before using an element in an overlay group, the application copies the sections from their load location to their run-time location, but only if these two addresses are different. For non-overlay sections this happens at program start-up.

- The start addresses cannot be set to absolute values for unrestricted groups.

- For non-overlay groups that do not have an overlay parent, the load-time start address equals the run-time start address.

- For any group, if the run-time start address is not set, the linker selects an appropriate address.

- If an ordered group or sequential group has an absolute address and contains sections that have separate page restrictions (not defined in LSL), all those sections are located in a single page. In other cases, for example when an unrestricted group has an address range assigned to it, the paged sections may be located in different pages.

For overlays, the linker reserves memory at the run-time start address as large as the largest element in the overlay group.

- The **page** keyword tells the linker to place the group in one page. Instead of specifying a run-time address, you can specify a page and optional a page number. Page numbers start from zero. If you omit the page number, the linker chooses a page.

  The **page** keyword refers to pages in the address space as defined in the architecture definition.

- With the **page_size** keyword you can override the page alignment and size set on the address space. When you set the page size to zero, the linker removes simple (auto generated) page restrictions from the selected sections. See also the **page_size** keyword in Section 15.4.3, *Defining Address Spaces*.

- With the **priority** keyword you can change the order in which sections are located. This is useful when some sections are considered important for good performance of the application and a small amount of fast memory is available. The value is a number for which the default is 1, so higher priorities start at 2. Sections with a higher priority are located before sections with a lower priority, unless their relative locate priority is already determined by other restrictions like **run_addr** and **page**.

```
group (priority=2)
{
  select "importantcode1";
  select "importantcode2";
}
```

15.8.3. Creating or Modifying Special Sections

Instead of selecting sections, you can also create a reserved section or an output section or modify special sections like a stack or a heap. Because you cannot define these sections in the input files, you must use the linker to create them.

Stack

- The keyword **stack** tells the linker to reserve memory for the stack. The name for the stack section refers to the stack as defined in the architecture definition. If no name was specified in the architecture definition, the default name is stack.

  With the keyword **size** you can specify the size for the stack. If the size is not specified, the linker uses the size given by the **min_size** argument as defined for the stack in the architecture definition. Normally the linker automatically tries to maximize the size, unless you specified the keyword **fixed**.

  ```
  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 15.4.3, *Defining Address Spaces*.

Heap

- The keyword **heap** tells the linker to reserve a dynamic memory range for the malloc() function. Each heap section has a name. With the keyword **size** you can change the size for the heap. If the **size** is not specified, the linker uses the size given by the **min_size** argument as defined for the heap in the architecture definition. Normally the linker automatically tries to maximize the size, unless you specified the keyword **fixed**.

  ```
  group ( ... )
  {
      heap "myheap" ( size = 2k );
  }
  ```

  The linker creates two labels to mark the begin and end of the heap, **_lc_ub_***heap_name* for the begin of the heap and **_lc_ue_***heap_name* for the end of the heap. The linker allocates space for the heap when a reference to either of the section labels exists in one of the input object files.

Reserved section

- The keyword **reserved** tells the linker to create an area or section of a given size. The linker will not locate any other sections in the memory occupied by a reserved section, with some exceptions. Each reserved section has a name. With the keyword **size** you can specify a size for a given reserved area or section. If the size is set to 0, then no other properties can be set for the reserved 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 begin of the section and **_lc_ue_**_name_ for the end of the reserved section.

Output sections

- The keyword **section** tells the linker to accumulate sections obtained from object files ("input sections") into an output section of a fixed size in the locate phase. You can select the input sections with **select** statements. You can use groups inside output sections, but you can only set the **align**, **attributes**, **nocopy** and **load_addr** properties and the **load_addr** property cannot have an address specified.

The **fill** field contains a bit pattern that the linker writes to all unused space in the output section. When all input sections have initialized code or data you must specify a fill pattern. If you do not specify a fill pattern, all input sections must be scratch sections (not cleared and not initialized), or BSS sections. The fill pattern is aligned at the start of the output section.

In the following example, the sections myinput1 and myinput2 are assumed to have initialized data, so the **fill** keyword is needed on the output section.

As with a reserved section you can use the **attributes** field to set the access type of the output section.

```
group ( ... )
{
    section "myoutput" ( size = 4k, attributes = r,
                          fill = 0xaa )
    {
        select "myinput1";
        select "myinput2";
    }
}
```

The available room for input sections is determined by the **size**, **blocksize** and **overflow** fields. With the keyword **size** you specify the fixed size of the output section. The value of the **size** property must be greater than zero. Input sections are placed from output section start towards higher addresses (offsets). When the end of the output section is reached and one or more input sections are not yet placed, an error is emitted. If however, the **overflow** field is set to another output section, remaining sections are located as if they were selected for the overflow output section.

In the following example, the sections .data.tsk1.* and .data.tsk2.* do not contain initialized data, so the **fill** keyword should not be used on the output section.

```
group ( ... )
{
  section "tsk1_data" (size=4k, attributes=rw,
                        overflow = "overflow_data")
  {
        select ".data.tsk1.*"
  }
  section "tsk2_data" (size=4k, attributes=rw,
                        overflow = "overflow_data")
  {
        select ".data.tsk2.*"
  }
  section "overflow_data" (size=4k, attributes=rw)
  {
  }
}
```

With the keyword **blocksize** , the size of the output section will adapt to the size of its content. For example:

```
group flash_area (run_addr = 0x10000)
{
    section "flash_code" (blocksize=4k, attributes=rx,
                          fill=0)
    {
      select "*.flash";
    }
}
```

If the content of the section is 1 mau, the size will be 4 KiB, if the content is 11 KiB, the section will be 12 KiB, etc. If you use **size** in combination with **blocksize**, the **size** value is used as default (minimal) size for this section. If it is omitted, the default size will be of **blocksize**. It is not allowed to omit both **size** and **blocksize** from the section definition. A value of zero is allowed for the **size** property if it sets a default size.

The linker creates two labels to mark the begin and end of the section, **_lc_ub_***name* for the begin of the section and **_lc_ue_***name* for the end of the output section.

When the **copy** property is set on an enclosing group, a ROM copy is created for the output section and the output section itself is made writable causing it to be located in RAM by default. For this to work, the output section and its input sections must be read-only and the output section must have a **fill** property.

A copy table can also be inserted into an output section, but only if two additional conditions are met:

• The copy table is the last section added to the output section.

• There must be sufficient room in the output section to accommodate the additional size of the copy table.

A copy table will likely increase in size after being added to the output section, so if you would add sections after the copy table selection, this would overwrite part of the copy table. The linker will emit an error message if either of the conditions is not met.

```
group ( ... )
{
    section "myoutput_tbl" ( size = 4k, attributes = r, fill = 0)
    {
      select "myinput";
      select "table"; // select the copy table
    }
}
```

Copy table

• The keyword **copytable** tells the linker to select a section that is used as *copy table*. The content of the copy table is created by the linker. It contains the start address and length of all sections that should be initialized by the startup code.

The linker creates two labels to mark the begin and end of the section, **_lc_ub_table** for the begin of the section and **_lc_ue_table** for the end of the copy table. The linker generates a copy table when a reference to either of the section labels exists in one of the input object files.

Structures

- A **struct** statement in a **section_layout** creates a section and fills it with numbers that each occupy one or more MAUs. The new section must be named by providing a double-quoted string after the **struct** keyword. Each element has the form *expr* : *number* ;, where the expression provides the value to insert in the section and the number determines the number of MAUs occupied by the expression value. Elements are placed in the section in the order in which they appear in the **struct** body without any gaps between them. Multi-MAU elements are split into MAUs according to the endianness of the target. A **struct** section is read-only and it cannot be copied to RAM at startup (using the **copy** group attribute). No default alignment is set.

  For example,

```
struct "mystruct"
{
  0x1234                                      : 2;
  addressof( mem:foo )                        : 4;
  addressof( mem:foo ) + sizeof( mem:foo )    : 4;
  checksum( crc32w,
    addressof( mem:foo ),
    addressof( mem:foo ) + sizeof( mem:foo ) ) : 4}
}
```

15.8.4. Creating Symbols

You can tell the linker to create symbols before locating by putting assignments in the section layout definition. Symbol names are represented by double-quoted strings. Any string is allowed, but object files may not support all characters for symbol names. You can use two different assignment operators. With the simple assignment operator '**=**', the symbol is created unconditionally. With the '**:=**' operator, the symbol is only created if it already exists as an undefined reference in an object file.

The expression that represents the value to assign to the symbol may contain references to other symbols. If such a referred symbol is a special section symbol, creation of the symbol in the left hand side of the assignment will cause creation of the special section.

```
section_layout
{
   "_lc_bs" := "_lc_ub_stack";
    // when the symbol _lc_bs occurs as an undefined reference
    // in an object file, the linker allocates space for the stack
}
```

15.8.5. Conditional Group Statements

Within a group, you can conditionally select sections or create special sections.

- With the `if` keyword you can specify a condition. The succeeding section statement is executed if the condition evaluates to TRUE (1).

- The optional `else` keyword is followed by a section statement which is executed in case the if-condition evaluates to FALSE (0).

```
group ( ... )
{
   if ( exists( "mysection" ) )
      select "mysection";
   else
      reserved "myreserved" ( size=2k );
}
```

Chapter 16. CERT C Secure Coding Standard

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

This chapter contains an overview of the CERT C Secure Coding Standard recommendations and rules that are supported by the TASKING toolset.

For details see the CERT C Secure Coding Standard web site. For general information about CERT secure coding, see www.cert.org/secure-coding.

Identifiers

Each rule and recommendation is given a unique identifier. These identifiers consist of three parts:

* a three-letter mnemonic representing the section of the standard

* a two-digit numeric value in the range of 00-99

* the letter "C" indicates that this is a C language guideline

The three-letter mnemonic is used to group similar coding practices and to indicate to which category a coding practice belongs.

The numeric value is used to give each coding practice a unique identifier. Numeric values in the range of 00-29 are reserved for recommendations, while values in the range of 30-99 are reserved for rules.

C compiler invocation

With the C compiler option **--cert** you can enable one or more checks for the CERT C Secure Coding Standard recommendations/rules. With **--diag=cert** you can see a list of the available checks, or you can use a three-letter mnemonic to list only the checks in a particular category. For example, **--diag=pre** lists all supported checks in the preprocessor category.

16.1. 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.

16.2. 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 TASKING C compiler assumes an implicit declaration and continues translation after issuing a warning message (W505 or W535).

DCL32-C Guarantee that mutually visible identifiers are unique

The compiler encountered two or more identifiers that are identical in the first 31 characters. The ISO C99 standard allows a compiler to ignore characters past the first 31 in an identifier. Two distinct identifiers that are identical in the first 31 characters may lead to problems when the code is ported to a different compiler.

DCL35-C Do not invoke a function using a type that does not match the function definition

This warning is generated when a function pointer is set to refer to a function of an incompatible type. Calling this function through the function pointer will result in undefined behavior. Example:

```
void my_function(int a);
int main(void)
{
  int (*new_function)(int a) = my_function;
  return (*new_function)(10); /* the behavior is undefined */
}
```

16.3. 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.

797

EXP34-C   Ensure a null pointer is not dereferenced

> Attempting to dereference a null pointer results in undefined behavior, typically abnormal program termination.

EXP37-C   Call functions with the arguments intended by the API

> When a function is properly declared with function prototype information, an incorrect call will be flagged by the compiler. When there is no prototype information available at the call, the compiler cannot check the number of arguments and the types of the arguments. This message is issued to warn about this situation.

EXP38-C   Do not call offsetof() on bit-field members or invalid types

> The behavior of the offsetof() macro is undefined when the member designator parameter designates a bit-field.

16.4. Integers (INT)

INT30-C   Ensure that unsigned integer operations do not wrap

> A constant with an unsigned integer type is truncated, resulting in a wrap-around.

INT34-C   Do not shift a negative number of bits or more bits than exist in the operand

> The shift count of the shift operation may be negative or greater than or equal to the size of the left operand. According to the C standard, the behavior of such a shift operation is undefined. Make sure the shift count is in range by adding appropriate range checks.

INT35-C   Evaluate integer expressions in a larger size before comparing or assigning to that size

> If an integer expression is compared to, or assigned to a larger integer size, that integer expression should be evaluated in that larger size by explicitly casting one of the operands.

16.5. Floating Point (FLP)

FLP30-C   Do not use floating point variables as loop counters

> To avoid problems with limited precision and rounding, floating point variables should not be used as loop counters.

FLP35-C   Take granularity into account when comparing floating point values

> Floating point arithmetic in C is inexact, so floating point values should not be tested for exact equality or inequality.

FLP36-C   Beware of precision loss when converting integral types to floating point

> Conversion from integral types to floating point types without sufficient precision can lead to loss of precision.

16.6. Arrays (ARR)

ARR01-C Do not apply the sizeof operator to a pointer when taking the size of an array

> A function parameter declared as an array, is converted to a pointer by the compiler. Therefore, the sizeof operator applied to this parameter yields the size of a pointer, and not the size of an array.

ARR34-C Ensure that array types in expressions are compatible

> Using two or more incompatible arrays in an expression results in undefined behavior.

ARR35-C Do not allow loops to iterate beyond the end of an array

> Reading or writing of data outside the bounds of an array may lead to incorrect program behavior or execution of arbitrary code.

16.7. Characters and Strings (STR)

STR30-C Do not attempt to modify string literals

> Writing to a string literal has undefined behavior, as identical strings may be shared and/or allocated in read-only memory.

STR33-C Size wide character strings correctly

> Wide character strings may be improperly sized when they are mistaken for narrow strings or for multi-byte character strings.

STR34-C Cast characters to unsigned types before converting to larger integer sizes

> A signed character is sign-extended to a larger signed integer value. Use an explicit cast, or cast the value to an unsigned type first, to avoid unexpected sign-extension.

STR36-C Do not specify the bound of a character array initialized with a string literal

> The compiler issues this warning when the character buffer initialized by a string literal does not provide enough room for the terminating null character.

16.8. Memory Management (MEM)

MEM00-C Allocate and free memory in the same module, at the same level of abstraction

> The compiler issues this warning when the result of the call to malloc(), calloc() or realloc() is discarded, and therefore not free()d, resulting in a memory leak.

MEM08-C Use realloc() only to resize dynamically allocated arrays

> Only use realloc() to resize an array. Do not use it to transform an object to an object of a different type.

MEM30-C   Do not access freed memory

When memory is freed, its contents may remain intact and accessible because it is at the memory manager's discretion when to reallocate or recycle the freed chunk. The data at the freed location may appear valid. However, this can change unexpectedly, leading to unintended program behavior. As a result, it is necessary to guarantee that memory is not written to or read from once it is freed.

MEM31-C   Free dynamically allocated memory exactly once

Freeing memory multiple times has similar consequences to accessing memory after it is freed. The underlying data structures that manage the heap can become corrupted. To eliminate double-free vulnerabilities, it is necessary to guarantee that dynamic memory is freed exactly once.

MEM32-C   Detect and handle memory allocation errors

The result of realloc() is assigned to the original pointer, without checking for failure. As a result, the original block of memory is lost when realloc() fails.

MEM33-C   Use the correct syntax for flexible array members

Use the ISO C99 syntax for flexible array members instead of an array member of size 1.

MEM34-C   Only free memory allocated dynamically

Freeing memory that is not allocated dynamically can lead to corruption of the heap data structures.

MEM35-C   Allocate sufficient memory for an object

The compiler issues this warning when the size of the object(s) allocated by malloc(), calloc() or realloc() is smaller than the size of an object pointed to by the result pointer. This may be caused by a sizeof expression with the wrong type or with a pointer type instead of the object type.

16.9. Environment (ENV)

ENV32-C   All atexit handlers must return normally

The compiler issues this warning when an atexit() handler is calling a function that does not return. No atexit() registered handler should terminate in any way other than by returning.

16.10. 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.

16.11. Miscellaneous (MSC)

MSC32-C Ensure your random number generator is properly seeded

Ensure that the random number generator is properly seeded by calling srand().

Chapter 17. MISRA C Rules

This chapter contains an overview of the supported and unsupported MISRA C rules.

17.1. MISRA C:1998

This section lists all supported and unsupported MISRA C:1998 rules.

See also Section 4.7.2, *C Code Checking: MISRA C*.

A number of MISRA C rules leave room for interpretation. Other rules can only be checked in a limited way. In such cases the implementation decisions and possible restrictions for these rules are listed.

x means that the rule is not supported by the TASKING C compiler. (R) is a required rule, (A) is an advisory rule.

	1.	(R)	The code shall conform to standard C, without language extensions.
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.
63.	(A)	A `switch` expression should not represent a Boolean case.
64.	(R)	Every `switch` shall have at least one `case`.
65.	(R)	Floating-point variables shall not be used as loop counters.
66.	(A)	A `for` should only contain expressions concerning loop control. A violation is reported when the loop initialization or loop update expression modifies an object that is not referenced in the loop test.
67.	(A)	Iterator variables should not be modified in a `for` loop.
68.	(R)	Functions shall always be declared at file scope.
69.	(R)	Functions with variable number of arguments shall not be used.
70.	(R)	Functions shall not call themselves, either directly or indirectly. A violation will be reported for direct or indirect recursive function calls in the source file being checked. Recursion via functions in other source files, or recursion via function pointers is not detected.
71.	(R)	Function prototypes shall be visible at the definition and call.
72.	(R)	The function prototype of the declaration shall match the definition.
73.	(R)	Identifiers shall be given for all prototype parameters or for none.
74.	(R)	Parameter identifiers shall be identical for declaration/definition.
75.	(R)	Every function shall have an explicit return type.

76. (R) Functions with no parameters shall have a `void` parameter list.

77. (R) An actual parameter type shall be compatible with the prototype.

78. (R) The number of actual parameters shall match the prototype.

79. (R) The values returned by `void` functions shall not be used.

80. (R) Void expressions shall not be passed as function parameters.

81. (A) `const` should be used for reference parameters not modified.

82. (A) A function should have a single point of exit.

83. (R) Every exit point shall have a `return` of the declared return type.

84. (R) For `void` functions, `return` shall not have an expression.

85. (A) Function calls with no parameters should have empty parentheses.

86. (A) If a function returns error information, it should be tested.
    A violation is reported when the return value of a function is ignored.

87. (R) `#include` shall only be preceded by other directives or comments.

88. (R) Non-standard characters shall not occur in `#include` directives.

89. (R) `#include` shall be followed by either `<filename>` or `"filename"`.

90. (R) Plain macros shall only be used for constants/qualifiers/specifiers.

91. (R) Macros shall not be `#define`'d and `#undef`'d within a block.

92. (A) `#undef` should not be used.

93. (A) A function should be used in preference to a function-like macro.

94. (R) A function-like macro shall not be used without all arguments.

95. (R) Macro arguments shall not contain pre-preprocessing directives.
    A violation is reported when the first token of an actual macro argument is '#'.

96. (R) Macro definitions/parameters should be enclosed in parentheses.

97. (A) Don't use undefined identifiers in pre-processing directives.

98. (R) A macro definition shall contain at most one # or ## operator.

99. (R) All uses of the `#pragma` directive shall be documented.
    This rule is really a documentation issue. The compiler will flag all `#pragma` directives as violations.

100. (R) `defined` shall only be used in one of the two standard forms.

101. (A) Pointer arithmetic should not be used.

102. (A) No more than 2 levels of pointer indirection should be used.
     A violation is reported when a pointer with three or more levels of indirection is declared.

103. (R) No relational operators between pointers to different objects.
     In general, checking whether two pointers point to the same object is impossible. The compiler will only report a violation for a relational operation with incompatible pointer types.

104. (R) Non-constant pointers to functions shall not be used.

105. (R) Functions assigned to the same pointer shall be of identical type.

	106.	(R)	Automatic address may not be assigned to a longer lived object.
	107.	(R)	The null pointer shall not be de-referenced. A violation is reported for every pointer dereference that is not guarded by a NULL pointer test.
	108.	(R)	All `struct`/`union` members shall be fully specified.
	109.	(R)	Overlapping variable storage shall not be used. A violation is reported for every `union` declaration.
	110.	(R)	Unions shall not be used to access the sub-parts of larger types. A violation is reported for a `union` containing a `struct` member.
	111.	(R)	Bit-fields shall have type `unsigned int` or `signed int`.
	112.	(R)	Bit-fields of type `signed int` shall be at least 2 bits long.
	113.	(R)	All `struct`/`union` members shall be named.
	114.	(R)	Reserved and standard library names shall not be redefined.
	115.	(R)	Standard library function names shall not be reused.
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`/`atoi`/`atol` shall not be used.
	126.	(R)	The functions `abort`/`exit`/`getenv`/`system` shall not be used.
	127.	(R)	The time handling functions of library `<time.h>` shall not be used.

17.2. MISRA C:2004

This section lists all supported and unsupported MISRA C:2004 rules.

See also Section 4.7.2, *C Code Checking: MISRA C*.

A number of MISRA C rules leave room for interpretation. Other rules can only be checked in a limited way. In such cases the implementation decisions and possible restrictions for these rules are listed.

x means that the rule is not supported by the TASKING C compiler. (R) is a required rule, (A) is an advisory rule.

Environment

    1.1   (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/compilers/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.

5.5   (A)   No object or function identifier with static storage duration should be reused.

5.6   (A)   No identifier in one name space should have the same spelling as an identifier in another name space, with the exception of structure and union member names.

5.7   (A)   No identifier name should be reused.

Types

6.1   (R)   The plain `char` type shall be used only for storage and use of character values.

6.2   (R)   `signed` and `unsigned char` type shall be used only for the storage and use of numeric values.

6.3   (A)   `typedef`s that indicate size and signedness should be used in place of the basic types.

6.4   (R)   Bit-fields shall only be defined to be of type `unsigned int` or `signed int`.

6.5   (R)   Bit-fields of type `signed int` shall be at least 2 bits long.

Constants

7.1   (R)   Octal constants (other than zero) and octal escape sequences shall not be used.

Declarations and definitions

8.1   (R)   Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.

8.2   (R)   Whenever an object or function is declared or defined, its type shall be explicitly stated.

8.3   (R)   For each function parameter the type given in the declaration and definition shall be identical, and the return types shall also be identical.

8.4   (R)   If objects or functions are declared more than once their types shall be compatible.

8.5   (R)   There shall be no definitions of objects or functions in a header file.

8.6   (R)   Functions shall be declared at file scope.

8.7   (R)   Objects shall be defined at block scope if they are only accessed from within a single function.

8.8   (R)   An external object or function shall be declared in one and only one file.

8.9   (R)   An identifier with external linkage shall have exactly one external definition.

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 of the same signedness that is no wider than the underlying type of the expression.

10.4   (R)   The value of a complex expression of floating type may only be cast to a type that is no wider than the underlying type of the expression.

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.

814

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.

17.3. MISRA C:2012

This section lists all supported and unsupported MISRA C:2012 rules.

See also Section 4.7.2, *C Code Checking: MISRA C*.

A number of MISRA C rules leave room for interpretation. Other rules can only be checked in a limited way. In such cases the implementation decisions and possible restrictions for these rules are listed.

x means that the rule is not supported by the TASKING C compiler. (M) is a mandatory rule, (R) is a required rule, (A) is an advisory rule.

A standard C environment

1.1   (R)   The program shall contain no violations of the standard C syntax and constraints, and shall not exceed the implementation's translation limits.

1.2   (A)   Language extensions should not be used.

1.3   (R)   There shall be no occurrence of undefined or critical unspecified behavior.

Unused code

2.1   (R)   A project shall not contain unreachable code.

2.2   (R)   There shall be no dead code.

2.3   (A)   A project should not contain unused type declarations.

2.4   (A)   A project should not contain unused tag declarations.

2.5   (A)   A project should not contain unused macro declarations.

2.6   (A)   A function should not contain unused label declarations.

2.7   (A)   There should be no unused parameters in functions.

Comments

3.1   (R)   The character sequences `/*` and `//` shall not be used within a comment.

3.2   (R)   Line-splicing shall not be used in `//` comments.

Character sets and lexical conventions

4.1   (R)   Octal and hexadecimal escape sequences shall be terminated.

4.2   (A)   Trigraphs should not be used.

Identifiers

5.1   (R)   External identifiers shall be distinct.

5.2   (R)   Identifiers declared in the same scope and name space shall be distinct.

5.3   (R)   An identifier declared in an inner scope shall not hide an identifier declared in an outer scope.

5.4   (R)   Macro identifiers shall be distinct.

5.5   (R)   Identifiers shall be distinct from macro names.

5.6   (R)   A `typedef` name shall be a unique identifier.

5.7   (R)   A tag name shall be a unique identifier.

5.8   (R)   Identifiers that define objects or functions with external linkage shall be unique.

5.9   (A)   Identifiers that define objects or functions with internal linkage should be unique.

Types

6.1   (R)   Bit-fields shall only be declared with an appropriate type.

6.2   (R)   Single-bit named bit-fields shall not be of a signed type.

Literals and constants

7.1   (R)   Octal constants shall not be used.

7.2   (R)   A "u" or "U" suffix shall be applied to all integer constants that are represented in an `unsigned` type.

7.3   (R)   The lowercase character "l" shall not be used in a literal suffix trivial.

7.4   (R)   A string literal shall not be assigned to an object unless the object's type is "pointer to `const`-qualified `char`".

Declarations and definitions

8.1	(R)	Types shall be explicitly specified.
8.2	(R)	Function types shall be in prototype form with named parameters.
8.3	(R)	All declarations of an object or function shall use the same names and type qualifiers.
8.4	(R)	A compatible declaration shall be visible when an object or function with external linkage is defined.
8.5	(R)	An external object or function shall be declared once in one and only one file.
8.6	(R)	An identifier with external linkage shall have exactly one external definition.
8.7	(A)	Functions and objects should not be defined with external linkage if they are referenced in only one translation unit.
8.8	(R)	The `static` storage class specifier shall be used in all declarations of objects and functions that have internal linkage.
8.9	(A)	An object should be defined at block scope if its identifier only appears in a single function.
8.10	(R)	An inline function shall be declared with the `static` storage class.
8.11	(A)	When an array with external linkage is declared, its size should be explicitly specified.
8.12	(R)	Within an enumerator list, the value of an implicitly-specified enumeration constant shall be unique.
8.13	(A)	A pointer should point to a `const`-qualified type whenever possible.
8.14	(R)	The `restrict` type qualifier shall not be used.

Initialization

9.1	(M)	The value of an object with automatic storage duration shall not be read before it has been set.
9.2	(R)	The initializer for an aggregate or union shall be enclosed in braces.
9.3	(R)	Arrays shall not be partially initialized.
9.4	(R)	An element of an object shall not be initialized more than once.
9.5	(R)	Where designated initializers are used to initialize an array object the size of the array shall be specified explicitly.

The essential type model

10.1	(R)	Operands shall not be of an inappropriate essential type.
10.2	(R)	Expressions of essentially character type shall not be used inappropriately in addition and subtraction operations.
10.3	(R)	The value of an expression shall not be assigned to an object with a narrower essential type or of a different essential type category.
10.4	(R)	Both operands of an operator in which the usual arithmetic conversions are performed shall have the same essential type category.

10.5   (A)   The value of an expression should not be cast to an inappropriate essential type.

10.6   (R)   The value of a composite expression shall not be assigned to an object with wider essential type.

10.7   (R)   If a composite expression is used as one operand of an operator in which the usual arithmetic conversions are performed then the other operand shall not have wider essential type.

10.8   (R)   The value of a composite expression shall not be cast to a different essential type category or a wider essential type.

Pointer type conversions

11.1   (R)   Conversions shall not be performed between a pointer to a function and any other type.

11.2   (R)   Conversions shall not be performed between a pointer to an incomplete type and any other type.

11.3   (R)   A cast shall not be performed between a pointer to object type and a pointer to a different object type.

11.4   (A)   A conversion should not be performed between a pointer to object and an integer type.

11.5   (A)   A conversion should not be performed from pointer to `void` into pointer to object.

11.6   (R)   A cast shall not be performed between pointer to `void` and an arithmetic type.

11.7   (R)   A cast shall not be performed between pointer to object and a non-integer arithmetic type.

11.8   (R)   A cast shall not remove any `const` or `volatile` qualification from the type pointed to by a pointer.

11.9   (R)   The macro `NULL` shall be the only permitted form of integer null pointer constant.

Expressions

12.1   (A)   The precedence of operators within expressions should be made explicit.

12.2   (R)   The right hand operand of a shift operator shall lie in the range zero to one less than the width in bits of the essential type of the left hand operand.

12.3   (A)   The comma operator should not be used.

12.4   (A)   Evaluation of constant expressions should not lead to unsigned integer wrap-around.

12.5   (M)   The `sizeof` operator shall not have an operand which is a function parameter declared as "array of type".

Side effects

13.1   (R)   Initializer lists shall not contain persistent side effects.

13.2   (R)   The value of an expression and its persistent side effects shall be the same under all permitted evaluation orders.

13.3 (A) A full expression containing an increment (`++`) or decrement (`--`) operator should have no other potential side effects other than that caused by the increment or decrement operator.

13.4 (A) The result of an assignment operator should not be used.

13.5 (R) The right hand operand of a logical `&&` or `||` operator shall not contain persistent side effects.

13.6 (M) The operand of the `sizeof` operator shall not contain any expression which has potential side effects.

Control statement expressions

14.1 (R) A loop counter shall not have essentially floating type.

14.2 (R) A `for` loop shall be well-formed.

14.3 (R) Controlling expressions shall not be invariant.

14.4 (R) The controlling expression of an `if` statement and the controlling expression of an iteration-statement shall have essentially Boolean type.

Control flow

15.1 (A) The `goto` statement should not be used.

15.2 (R) The `goto` statement shall jump to a label declared later in the same function.

15.3 (R) Any label referenced by a `goto` statement shall be declared in the same block, or in any block enclosing the `goto` statement.

15.4 (A) There should be no more than one `break` or `goto` statement used to terminate any iteration statement.

15.5 (A) A function should have a single point of exit at the end.

15.6 (R) The body of an iteration-statement or a selection-statement shall be a compound-statement.

15.7 (R) All `if ... else if` constructs shall be terminated with an `else` statement.

Switch statements

16.1 (R) All `switch` statements shall be well-formed.

16.2 (R) A switch label shall only be used when the most closely-enclosing compound statement is the body of a `switch` statement.

16.3 (R) An unconditional `break` statement shall terminate every switch-clause.

16.4 (R) Every `switch` statement shall have a `default` label.

16.5 (R) A `default` label shall appear as either the first or the last switch label of a `switch` statement.

16.6 (R) Every `switch` statement shall have at least two switch-clauses.

16.7 (R) A switch-expression shall not have essentially Boolean type.

Functions

17.1   (R)   The features of `<stdarg.h>` shall not be used.

17.2   (R)   Functions shall not call themselves, either directly or indirectly.

17.3   (M)   A function shall not be declared implicitly.

17.4   (M)   All exit paths from a function with non-`void` return type shall have an explicit `return` statement with an expression.

17.5   (A)   The function argument corresponding to a parameter declared to have an array type shall have an appropriate number of elements.

17.6   (M)   The declaration of an array parameter shall not contain the `static` keyword between the `[ ]`.

17.7   (R)   The value returned by a function having non-`void` return type shall be used.

17.8   (A)   A function parameter should not be modified.

Pointers and arrays

18.1   (R)   A pointer resulting from arithmetic on a pointer operand shall address an element of the same array as that pointer operand.

18.2   (R)   Subtraction between pointers shall only be applied to pointers that address elements of the same array.

18.3   (R)   The relational operators `>`, `>=`, `<` and `<=` shall not be applied to objects of pointer type except where they point into the same object.

18.4   (A)   The `+`, `-`, `+=` and `-=` operators should not be applied to an expression of pointer type.

18.5   (A)   Declarations should contain no more than two levels of pointer nesting.

18.6   (R)   The address of an object with automatic storage shall not be copied to another object that persists after the first object has ceased to exist.

18.7   (R)   Flexible array members shall not be declared.

18.8   (R)   Variable-length array types shall not be used.

Overlapping storage

19.1   (M)   An object shall not be assigned or copied to an overlapping object.

19.2   (A)   The `union` keyword should not be used.

Preprocessing directives

20.1   (A)   `#include` directives should only be preceded by preprocessor directives or comments.

20.2   (R)   The `'`, `"` or `\` characters and the `/*` or `//` character sequences shall not occur in a header file name.

20.3   (R)   The #include directive shall be followed by either a `<filename>` or `"filename"` sequence.

20.4	(R)	A macro shall not be defined with the same name as a keyword.
20.5	(A)	`#undef` should not be used.
20.6	(R)	Tokens that look like a preprocessing directive shall not occur within a macro argument
20.7	(R)	Expressions resulting from the expansion of macro parameters shall be enclosed in parentheses.
20.8	(R)	The controlling expression of a `#if` or `#elif` preprocessing directive shall evaluate to 0 or 1.
20.9	(R)	All identifiers used in the controlling expression of `#if` or `#elif` preprocessing directives shall be `#define`'d before evaluation.
20.10	(A)	The `#` and `##` preprocessor operators should not be used.
20.11	(R)	A macro parameter immediately following a `#` operator shall not immediately be followed by a `##` operator.
20.12	(R)	A macro parameter used as an operand to the `#` or `##` operators, which is itself subject to further macro replacement, shall only be used as an operand to these operators.
20.13	(R)	A line whose first token is `#` shall be a valid preprocessing directive.
20.14	(R)	All `#else`, `#elif` and `#endif` preprocessor directives shall reside in the same file as the `#if`, `#ifdef` or `#ifndef` directive to which they are related.

Standard libraries

21.1	(R)	`#define` and `#undef` shall not be used on a reserved identifier or reserved macro name.
21.2	(R)	A reserved identifier or macro name shall not be declared.
21.3	(R)	The memory allocation and deallocation functions of `<stdlib.h>` shall not be used.
21.4	(R)	The standard header file `<setjmp.h>` shall not be used.
21.5	(R)	The standard header file `<signal.h>` shall not be used.
21.6	(R)	The Standard Library input/output functions shall not be used.
21.7	(R)	The `atof`, `atoi`, `atol` and `atoll` functions of `<stdlib.h>` shall not be used.
21.8	(R)	The library functions `abort`, `exit` and `system` of `<stdlib.h>` shall not be used.
21.9	(R)	The library functions `bsearch` and `qsort` of `<stdlib.h>` shall not be used.
21.10	(R)	The Standard Library time and date functions shall not be used
21.11	(R)	The standard header file `<tgmath.h>` shall not be used.
21.12	(A)	The exception handling features of `<fenv.h>` should not be used.
21.13	(M)	Any value passed to a function in `<ctype.h>` shall be representable as an `unsigned char` or be the value `EOF`.
21.14	(R)	The Standard Library function `memcmp` shall not be used to compare null terminated strings.
21.15	(R)	The pointer arguments to the Standard Library functions `memcpy`, `memmove` and `memcmp` shall be pointers to qualified or unqualified versions of compatible types

21.16 (R) The pointer arguments to the Standard Library function `memcmp` shall point to either a pointer type, an essentially signed type, an essentially unsigned type, an essentially Boolean type or an essentially enum type

21.17 (M) Use of the string handling functions from `<string.h>` shall not result in accesses beyond the bounds of the objects referenced by their pointer parameters.

21.18 (M) The `size_t` argument passed to any function in `<string.h>` shall have an appropriate value.

21.19 (M) The pointers returned by the Standard Library functions `localeconv`, `getenv`, `setlocale` or, `strerror` shall only be used as if they have pointer to const-qualified type.

21.20 (M) The pointer returned by the Standard Library functions `asctime`, `ctime`, `gmtime`, `localtime`, `localeconv`, `getenv`, `setlocale` or `strerror` shall not be used following a subsequent call to the same function.

Resources

x 22.1 (R) All resources obtained dynamically by means of Standard Library functions shall be explicitly released.

x 22.2 (M) A block of memory shall only be freed if it was allocated by means of a Standard Library function.

x 22.3 (R) The same file shall not be open for read and write access at the same time on different streams.

x 22.4 (M) There shall be no attempt to write to a stream which has been opened as read-only.

x 22.5 (M) A pointer to a `FILE` object shall not be dereferenced.

x 22.6 (M) The value of a pointer to a `FILE` shall not be used after the associated stream has been closed.

x 22.7 (R) The macro `EOF` shall only be compared with the unmodified return value from any Standard Library function capable of returning `EOF`.

x 22.8 (R) The value of `errno` shall be set to zero prior to a call to an errno-setting-function.

x 22.9 (R) The value of `errno` shall be tested against zero after calling an errno-setting-function.

x 22.10 (R) The value of `errno` shall only be tested when the last function to be called was an errno-setting-function.

Chapter 18. C Implementation-defined Behavior

The TASKING ARC C compiler fully supports the ISO C standard, but some parts of the ISO C standard are implementation-defined. This chapter describes how the implementation-defined areas and the locale-specific areas of the C language are implemented in the TASKING C compiler for ISO C99, ISO C11 and ISO C17. Below are some remarks on the other behaviors as mentioned in the standard.

Unspecified behavior

Unspecified behavior is the use of an unspecified value, or other behavior where the ISO C standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance. Some of the unspecified behaviors are relevant to users of the TASKING ARC C compiler. Some unspecified behaviors are specified in the ABI. The silicon vendor is responsible for the ABI.

Undefined behavior

Undefined behavior is behavior, upon use of a non-portable or erroneous program construct or of erroneous data, for which the ISO C standard imposes no requirements.

Some undefined behaviors may trigger a compiler error or warning. TASKING does not provide any guarantees about whether or not the compiler issues an error or warning. It is important to know whether your software contains undefined behaviors since this will make the source non-portable between compiler vendors and between other processors.

The MISRA C and CERT coding guides do not refer to undefined behaviors explicitly.

18.1. C99 Implementation-defined Behavior

Implementation-defined behavior is unspecified behavior where each implementation documents how the choice is made.

The following sections describe the implementation-defined characteristics. The section numbers listed in parenthesis refer to the corresponding sections in the ISO C99 standard. The order in this chapter is the same as used in Annex J.3 of the ISO/IEC 9899:1999 (E) standard.

18.1.1. Translation

- How a diagnostic is identified (3.10, 5.1.1.3).

  The C compiler diagnostics are explained in Section 4.8, *C Compiler Error Messages*.

- Whether each nonempty sequence of white-space characters other than new-line is retained or replaced by one space character in translation phase 3 (5.1.1.2).

  White-space is retained.

18.1.2. Environment

- The mapping between physical source file multibyte characters and the source character set in translation phase 1 (5.1.1.2).

  Use of variable length encoded characters in the source file in comments and string literals is permitted. For source file characters 0D and 1A the mapping is one-to-zero, for other characters a one-on-one mapping is done without interpretation of multibyte characters.

- The name and type of the function called at program startup in a freestanding environment (5.1.2.1).

  The function called at program startup (in `cstart.c`) is called `main`. The prototype for `main` in `cstart.c` is:

  ```
  extern int main( int argc, char *argv[] );
  ```

- The effect of program termination in a freestanding environment (5.1.2.1).

  Execution is halted if the program is executed under control of a debugger, otherwise the program will loop forever in function `_Exit()`.

- An alternative manner in which the `main` function may be defined (5.1.2.2.1).

  You can change the definition of `main` by altering file `cstart.c`.

- The values given to the strings pointed to by the `argv` argument to `main` (5.1.2.2.1).

  The strings get their values from the arguments given in file `cstart.c`. The program arguments are treated case sensitive.

- What constitutes an interactive device (5.1.2.3).

  The streams `stdin`, `stdout` and `stderr` are treated as interactive devices. The debugger uses these streams with File System Simulation (FSS) windows to interact.

- The set of signals, their semantics, and their default handling (7.14).

  The signals are described in Section 12.1.17, *signal.h*.

- Signal values other than `SIGFPE`, `SIGILL`, and `SIGSEGV` that correspond to a computational exception (7.14.1.1).

  There are no other values that correspond to a computational exception. All signal values are described in Section 12.1.17, *signal.h*.

- Signals for which the equivalent of `signal(sig, SIG_IGN);` is executed at program startup (7.14.1.1).

  By default the implementation does not ignore any signals at program startup.

- The set of environment names and the method for altering the environment list used by the `getenv` function (7.20.4.5).

There are no implementation-defined environment names that are used by the `getenv` function. A skeleton is provided for the `getenv` function in the C library, because the embedded environment has no operating system. The `getenv` function calls the name as a `void` function.

- The manner of execution of the string by the `system` function (7.20.4.6).

A skeleton is provided for the `system()` function in the C library, because the embedded environment has no operating system. The `system()` function calls the string as a `void` function.

18.1.3. Identifiers

- Which additional multibyte characters may appear in identifiers and their correspondence to universal character names (6.4.2).

No additional multibyte characters are supported in an identifier.

- The number of significant initial characters in an identifier (5.2.4.1, 6.4.2).

All characters in an identifier are significant.

18.1.4. Characters

- The number of bits in a byte (3.6).

There are eight bits in a byte.

- The values of the members of the execution character set (5.2.1).

Only 8-bit characters are supported. The values of the execution character set are the same as that of the source character set. The same representation value is used for each member in the characters sets except for the escape sequences.

- The unique value of the member of the execution character set produced for each of the standard alphabetic escape sequences (5.2.2).

The following table contains an overview of the escape sequences and their byte value in the execution character set.

Escape sequence	Value
\a	7
\b	8
\f	12
\n	10
\r	13
\t	9
\v	11

- The value of a `char` object into which has been stored any character other than a member of the basic execution character set (6.2.5).

  Any 8-bit value can be stored in a `char` object.

- Which of `signed char` or `unsigned char` has the same range, representation, and behavior as "plain" `char` (6.2.5, 6.3.1.1).

  By default "plain" `char` is the same as specifying `unsigned char`. With C compiler option **--schar** you can change the default to `signed char`.

- The mapping of members of the source character set (in character constants and string literals) to members of the execution character set (6.4.4.4, 5.1.1.2).

  The mapping is one-to-one. The values of the execution character set are the same as that of the source character set. The same representation value is used for each member in the characters sets except for the escape sequences.

- The value of an integer character constant containing more than one character or containing a character or escape sequence that does not map to a single-byte execution character (6.4.4.4).

  For any character constant containing more than one character, a warning is issued and the value is truncated to type `signed char`.

- The value of a wide character constant containing more than one multibyte character, or containing a multibyte character or escape sequence not represented in the extended execution character set (6.4.4.4).

  A wide character constant can contain at most two multibyte characters. Its value is the concatenation of the multibyte characters represented in a `signed short int`.

- The current locale used to convert a wide character constant consisting of a single multibyte character that maps to a member of the extended execution character set into a corresponding wide character code (6.4.4.4).

  By default, the "C" locale is used.

- The current locale used to convert a wide string literal into corresponding wide character codes (6.4.5).

  By default, the "C" locale is used.

- The value of a string literal containing a multibyte character or escape sequence not represented in the execution character set (6.4.5).

  All source characters can be represented in the execution character set.

18.1.5. Integers

- Any extended integer types that exist in the implementation (6.2.5).

  All types are described in Section 1.1, *Data Types*.

- Whether signed integer types are represented using sign and magnitude, two's complement, or one's complement, and whether the extraordinary value is a trap representation or an ordinary value (6.2.6.2).

  Signed integer types are represented in two's complement. The most significant bit is the sign bit. 1 is negative, 0 is positive.

- The rank of any extended integer type relative to another extended integer type with the same precision (6.3.1.1).

  All types are described in Section 1.1, *Data Types*.

- The result of, or the signal raised by, converting an integer to a signed integer type when the value cannot be represented in an object of that type (6.3.1.3).

  At compile time, when converting integer types and a value does not fit in a type, the compiler issues a warning and the value is truncated. At run-time no warning or signal is given and the value is truncated.

- The results of some bitwise operations on signed integers (6.5).

  The result of E1>>E2 is E1 right shifted E2 bit positions. If E1 has a signed type and a negative value, the shift behavior is implemented as an arithmetic shift. The empty position in the most significant bit is filled with a copy of the original most significant bit.

18.1.6. Floating-Point

- The accuracy of the floating-point operations and of the library functions in `<math.h>` and `<complex.h>` that return floating-point results (5.2.4.2.2).

  A `float` has an exponent of 8 bits and a significand of 24 bits. A `double` or `long double` has an exponent of 11 bits and a significand of 53 bits. This is conform IEEE-754 for single precision and double precision floating-point. Internally the compiler uses a significand of 80 bits. The results of floating-point operations are rounded to the nearest IEEE-754 format.

  The accuracy of `sqrt` is defined unknown.

- The rounding behaviors characterized by non-standard values of `FLT_ROUNDS` (5.2.4.2.2).

  No non-standard values are used.

- The evaluation methods characterized by non-standard negative values of `FLT_EVAL_METHOD` (5.2.4.2.2).

  `FLT_EVAL_METHOD` is defined as 0. No non-standard values are used.

- The direction of rounding when an integer is converted to a floating-point number that cannot exactly represent the original value (6.3.1.4).

  For FPU instructions, the rounding mode is used. For software floating-point instructions, the round to nearest method is used. `FLT_ROUNDS` is ignored.

- The direction of rounding when a floating-point number is converted to a narrower floating-point number (6.3.1.5).

The round to nearest method is used. `FLT_ROUNDS` is ignored.

- How the nearest representable value or the larger or smaller representable value immediately adjacent to the nearest representable value is chosen for certain floating constants (6.4.4.2).

  The round to nearest method is used. `FLT_ROUNDS` is ignored.

- Whether and how floating expressions are contracted when not disallowed by the `FP_CONTRACT` pragma (6.5).

  Pragma `FP_CONTRACT` is equivalent to compiler option **--fp-model=+contract**. This has only effect for fused multiply-and-accumulate (FMA) operations. FMA operations are not supported by the IEEE 754-1985 standard. The result of FMA operations is only rounded once at the end of the FMA. You can disable FMAs with the compiler option **--fp-model=-contract**.

- The default state for the `FENV_ACCESS` pragma (7.6.1).

  The default state of pragma `FENV_ACCESS` is "off". This pragma is ignored.

- Additional floating-point exceptions, rounding modes, environments, and classifications, and their macro names (7.6, 7.12).

  No additional floating-point exceptions, rounding modes, environments, and classifications are defined.

- The default state for the `FP_CONTRACT` pragma (7.12.2).

  The default state of pragma `FP_CONTRACT` is set by the **contract** flag of compiler option **--fp-model**. The default state is "on".

- Whether the "inexact" floating-point exception can be raised when the rounded result actually does equal the mathematical result in an IEC 60559 conformant implementation (F.9).

  No "inexact" floating-point exceptions are raised.

- Whether the "underflow" (and "inexact") floating-point exception can be raised when a result is tiny but not inexact in an IEC 60559 conformant implementation (F.9).

  No floating-point exceptions are raised.

18.1.7. Arrays and Pointers

- The result of converting a pointer to an integer or vice versa (6.3.2.3).

  All non-pointer conversions to and from a 32-bit pointer are implemented as a conversion to or from a 32-bit integer type.

- The size of the result of subtracting two pointers to elements of the same array (6.5.6).

  The size of `ptrdiff_t` is 32 bits. The difference in address location is expressed in bytes.

18.1.8. Hints

- The extent to which suggestions made by using the `register` storage-class specifier are effective (6.7.1).

  The compiler does not make assumptions based on the `register` storage-class specifier. So, basically this keyword is ignored, except that you cannot take the address of a `register` variable. The compiler issues an error in that case.

- The extent to which suggestions made by using the `inline` function specifier are effective (6.7.4).

  With the `inline` keyword you force the compiler to inline the specified function, regardless of the optimization strategy of the compiler itself. For more information see Section 1.10.2, *Inlining Functions: inline*.

18.1.9. Structures, Unions, Enumerations, and Bit-fields

- Whether a "plain" `int` bit-field is treated as a `signed int` bit-field or as an `unsigned int` bit-field (6.7.2, 6.7.2.1).

  By default an `int` bit-field is treated as `signed int`. You can still add the keyword `unsigned` to treat a particular `int` bit-field as `unsigned`.

- Allowable bit-field types other than `_Bool`, `signed int`, and `unsigned int` (6.7.2.1).

  All integer types as specified in Section 1.1, *Data Types* are allowable bit-field types.

- Whether a bit-field can straddle a storage-unit boundary (6.7.2.1).

  A bit-field cannot straddle a storage-unit boundary. If insufficient space remains, the bit-field is put into the next unit.

- The order of allocation of bit-fields within a unit (6.7.2.1).

  Allocation starts at the least significant bit up to the most significant bit. If the following bit-field fits within the same unit, it is allocated starting at the next available bit.

- The alignment of non-bit-field members of structures (6.7.2.1). This should present no problem unless binary data written by one implementation is read by another.

  The alignment of non-bit-field members of structures is the same as the alignment for data types as specified in Section 1.1, *Data Types*.

- The integer type compatible with each enumerated type (6.7.2.2).

  The compiler chooses the smallest suitable integer type (`char`, `unsigned char`, `short`, `unsigned short` or `int`).

18.1.10. Qualifiers

- What constitutes an access to an object that has `volatile`-qualified type (6.7.3).

Any reference to an object with `volatile` type results in an access. The order in which `volatile` objects are accessed is defined by the order expressed in the source code. References to non-volatile objects are scheduled in arbitrary order, within the constraints given by dependencies.

If the compiler option **--language=+volatile** (**-Av**) is set, all references to non-volatile objects result in an access before the access to a `volatile` object that occurs subsequently in the source file takes place. The volatile access acts as a memory barrier.

18.1.11. Preprocessing Directives

- How sequences in both forms of header names are mapped to headers or external source file names (6.4.7).

  Sequences in header names are mapped to file names as is. The backslash "\" is not interpreted as an escape sequence. The backslash "\" (Windows) or forward slash "/" (Windows and UNIX) is interpreted as a standard directory separator.

- Whether the value of a character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set (6.10.1).

  A character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set.

- Whether the value of a single-character character constant in a constant expression that controls conditional inclusion may have a negative value (6.10.1).

  A single-character character constant in a constant expression that controls conditional inclusion may have a negative value.

- The places that are searched for an included `< >` delimited header, and how the places are specified or the header is identified (6.10.2).

  How the compiler searches for include files is explained in Section 4.4, *How the Compiler Searches Include Files*.

- How the named source file is searched for in an included `" "` delimited header (6.10.2).

  How the compiler searches for include files is explained in Section 4.4, *How the Compiler Searches Include Files*.

- The method by which preprocessing tokens (possibly resulting from macro expansion) in a `#include` directive are combined into a header name (6.10.2).

  Preprocessing tokens in a `#include` directive are combined the same way as outside a `#include` directive.

- The nesting limit for `#include` processing (6.10.2).

  There is no nesting limit for `#include` processing.

- Whether the `#` operator inserts a `\` character before the `\` character that begins a universal character name in a character constant or string literal (6.10.3.2).

The # operator inserts a \ character before every \ character in a character constant or string literal.

- The behavior on each recognized non-STDC #pragma directive (6.10.6).

  All non-STDC pragmas are described in Section 1.8, *Pragmas to Control the Compiler*.

- The definitions for __DATE__ and __TIME__ when respectively, the date and time of translation are not available (6.10.8).

  The date and time of translation are always available, macros __DATE__ and __TIME__ are always defined.

18.1.12. Library Functions

- Any library facilities available to a freestanding program, other than the minimal set required by clause 4 (5.1.2.1).

  All library functions are described in Chapter 12, *Libraries*. Some functions are not completely implemented because their implementation depends on the context where your application will run. These functions are for example all I/O related functions. Where possible, these functions are implemented using file system simulation (FSS). This system can be used by the debugger to simulate an I/O environment which enables you to debug your application. If the application runs under control of the debugger and FSS is used, then the low-level behavior is equal to the host system's file access behavior.

  In the implementation in the C library, the basic sequences of a multibyte character consist of single bytes (MB_LEN_MAX is set to 1). If you want full multibyte support, you need to change the C library. See the notes in the header files stdio.h and wchar.h for more information.

- The format of the diagnostic printed by the assert macro (7.2.1.1).

  The assert() function is implemented as a macro in assert.h. The output is:

  Assertion failed: (*expression*) file *filename*, line *linenumber*

  when the parameter evaluates to zero.

- The representation of the floating-point status flags stored by the fegetexceptflag function (7.6.2.2).

  Floating-point exceptions are not supported. Function fegetexceptflag does nothing.

- Whether the feraiseexcept function raises the "inexact" floating-point exception in addition to the "overflow" or "underflow" floating-point exception (7.6.2.3).

  Floating-point exceptions are not supported. Function feraiseexcept does nothing.

- Strings other than "C" and " " that may be passed as the second argument to the setlocale function (7.11.1.1).

  No other strings are predefined. A NULL pointer as the second argument returns the "C" locale. Any other string than "C" or " " can be passed as the second argument to the setlocale function and results in NULL.

- The types defined for `float_t` and `double_t` when the value of the `FLT_EVAL_METHOD` macro is less than 0 or greater than 2 (7.12).

  The `FLT_EVAL_METHOD` macro can only have the values 0, 1 or 2.

- Domain errors for the mathematical functions, other than those required by this International Standard (7.12.1).

  No other domain errors exist, other than those required by the standard.

- The values returned by the mathematical functions on domain errors (7.12.1).

  On domain errors (`errno` is set to `EDOM`), the mathematical functions return a value as specified in the following table.

Math function	Return value on EDOM
`acos(|x| > 1.0)`	0.0
`asin(|x| > 1.0)`	0.0
`log(x < 0.0)`	-HUGE_VAL
`pow(x <= 0.0)`	0.0
`sqrt(x < 0.0)`	-NaN

- The values returned by the mathematical functions on underflow range errors, whether `errno` is set to the value of the macro `ERANGE` when the integer expression `math_errhandling & MATH_ERRNO` is nonzero, and whether the "underflow" floating-point exception is raised when the integer expression `math_errhandling & MATH_ERREXCEPT` is nonzero. (7.12.1).

  On underflow range errors, the mathematical functions return 0.0. `math_errhandling` is set to `MATH_ERRNO`. Trapping and non-trapping versions of the library are available. With a non-trapping library `errno` is not set to `ERANGE` on underflow range errors, with a trapping library no underflow exception is raised.

- Whether a domain error occurs or zero is returned when an `fmod` function has a second argument of zero (7.12.10.1).

  Zero (`0.0`) is returned when an `fmod` function has a second argument of zero.

- The base-2 logarithm of the modulus used by the `remquo` functions in reducing the quotient (7.12.10.3).

  The `remquo` function calculates at least 8 bits of the quotient.

- Whether the equivalent of `signal(sig, SIG_DFL);` is executed prior to the call of a signal handler, and, if not, the blocking of signals that is performed (7.14.1.1).

  The equivalent of `signal(sig, SIG_DFL);` is executed prior to the call of a signal handler.

- The null pointer constant to which the macro `NULL` expands (7.17).

  Macro `NULL` is defined as `(void *) 0`.

832

- Whether the last line of a text stream requires a terminating new-line character (7.19.2).

  Both a new-line character (`\n`) and end-of-file (`EOF`) are recognized as the termination character of a line.

- Whether space characters that are written out to a text stream immediately before a new-line character appear when read in (7.19.2).

  Space characters written to a stream immediately before a new-line character are preserved.

- The number of null characters that may be appended to data written to a binary stream (7.19.2).

  I/O related functions are implemented using file system simulation (FSS). This system can be used by the debugger to simulate an I/O environment. If the application runs under control of the debugger and FSS is used, then the low-level behavior is equal to the host system's file access behavior. The library does not append any null characters. It depends on the `open()` function on the host environment what happens. You can write your own `_open()` function if necessary.

- Whether the file position indicator of an append-mode stream is initially positioned at the beginning or end of the file (7.19.3).

  I/O related functions are implemented using file system simulation (FSS). This system can be used by the debugger to simulate an I/O environment. If the application runs under control of the debugger and FSS is used, then the low-level behavior is equal to the host system's file access behavior. Where the file position indicator of an append-mode stream is initially positioned depends on the `open()` function on the host environment. You can write your own `_open()` function if necessary.

- Whether a write on a text stream causes the associated file to be truncated beyond that point (7.19.3).

  I/O related functions are implemented using file system simulation (FSS). This system can be used by the debugger to simulate an I/O environment. If the application runs under control of the debugger and FSS is used, then the low-level behavior is equal to the host system's file access behavior. Whether a write on a text stream causes the associated file to be truncated beyond that point depends on how the low-level file routines are implemented in your application.

- The characteristics of file buffering (7.19.3).

  Files can be unbuffered, fully buffered or line buffered. What actually happens depends on how the low-level file routines are implemented in your application.

- Whether a zero-length file actually exists (7.19.3).

  This depends on how the low-level file routines are implemented in your application.

- The rules for composing valid file names (7.19.3).

  This depends on how the low-level file routines are implemented in your application.

- Whether the same file can be simultaneously open multiple times (7.19.3).

  This depends on how the low-level file routines are implemented in your application.

- The nature and choice of encodings used for multibyte characters in files (7.19.3).

Use of variable length encoded characters in files in comments and string literals is permitted.

- The effect of the `remove` function on an open file (7.19.4.1).

  This depends on how the low-level file routines are implemented in your application.

- The effect if a file with the new name exists prior to a call to the `rename` function (7.19.4.2).

  This depends on how the low-level file routines are implemented in your application.

- Whether an open temporary file is removed upon abnormal program termination (7.19.4.3).

  This depends on how the low-level file routines are implemented in your application.

- Which changes of mode are permitted (if any), and under what circumstances (7.19.5.4).

  The `freopen()` function first calls `fclose()` and then calls `_fopen()` with the new mode.

- The style used to print an infinity or NaN, and the meaning of any n-char or n-wchar sequence printed for a NaN (7.19.6.1, 7.24.2.1).

  The style used to print an infinity or NaN is `inf` and `nan` respectively (`INF` or `NAN` for the `F` conversion specifier). n-char or n-wchar sequences are not used for `nan`.

- The output for `%p` conversion in the `fprintf` or `fwprintf` function (7.19.6.1, 7.24.2.1).

  The argument is treated as having type `void *`. The value will be printed as a hexadecimal value, similar to `%x`.

- The interpretation of a – character that is neither the first nor the last character, nor the second where a ^ character is the first, in the scanlist for `%[` conversion in the `fscanf` or `fwscanf` function (7.19.6.2, 7.24.2.1).

  A – character is treated as a normal character.

- The set of sequences matched by a `%p` conversion and the interpretation of the corresponding input item in the `fscanf` or `fwscanf` function (7.19.6.2, 7.24.2.2).

  The format of `%p` matches the format of `%x`. The input for `%p` is a hexadecimal value, which is converted to a value with type `void *`.

- The value to which the macro `errno` is set by the `fgetpos`, `fsetpos`, or `ftell` functions on failure (7.19.9.1, 7.19.9.3, 7.19.9.4).

  If `errno` is set to a value depends on how the low-level file routines are implemented in your application.

- The meaning of any n-char or n-wchar sequence in a string representing a NaN that is converted by the `strtod`, `strtof`, `strtold`, `wcstod`, `wcstof`, or `wcstold` function (7.20.1.3, 7.24.4.1.1).

  An n-char or n-wchar sequence in a string representing a NaN is ignored.

- Whether or not the `strtod`, `strtof`, `strtold`, `wcstod`, `wcstof`, or `wcstold` function sets `errno` to `ERANGE` when underflow occurs (7.20.1.3, 7.24.4.1.1).

834

`errno` is set to `ERANGE` when underflow occurs and the value returned is 0.0.

- Whether the `calloc`, `malloc`, and `realloc` functions return a null pointer or a pointer to an allocated object when the size requested is zero (7.20.3).

  `NULL` is returned when a size of zero is requested.

- Whether open streams with unwritten buffered data are flushed, open streams are closed, or temporary files are removed when the `abort` or `_Exit` function is called (7.20.4.1, 7.20.4.4).

  When the `abort()` or `_Exit()` function is called, open streams with unwritten buffered data are not flushed, open streams are not closed, and temporary files are not removed.

- The termination status returned to the host environment by the `abort`, `exit`, or `_Exit` function (7.20.4.1, 7.20.4.3, 7.20.4.4).

  `exit()` and `_Exit()` use the input value as termination status. `abort()` calls `_Exit()` with `EXIT_FAILURE`.

- The value returned by the `system` function when its argument is not a null pointer (7.20.4.6).

  A skeleton is provided for the `system()` function in the C library, because the embedded environment has no operating system. The `system()` function returns the value 0.

- The local time zone and Daylight Saving Time (7.23.1).

  The default time zone is UTC. Daylight Saving Time is not available (`tm_isdst=-1`).

- The range and precision of times representable in `clock_t` and `time_t` (7.23).

  `clock_t` is defined as `unsigned long long`, `time_t` is defined as `unsigned long`. The resolution of the clock is defined by `CLOCKS_PER_SEC`, which value is hard-coded to 500000000 (500MHz).

- The era for the `clock` function (7.23.2.1).

  The `clock` function returns the current processor time. It reads the 64-bit real-time counter (RTC).

- The replacement string for the `%Z` specifier to the `strftime`, and `wcsftime` functions in the `"C"` locale (7.23.3.5, 7.24.5.1).

  `%Z` is replaced by the time zone name, by default UTC.

- Whether or when the trigonometric, hyperbolic, base-e exponential, base-e logarithmic, error, and log gamma functions raise the "inexact" floating-point exception in an IEC 60559 conformant implementation (F.9).

  The "inexact" floating-point exception is not supported.

- Whether the functions in `<math.h>` honor the rounding direction mode in an IEC 60559 conformant implementation (F.9).

  The round to nearest method is used. `FLT_ROUNDS` is defined as 1.

18.1.13. Architecture

- The values or expressions assigned to the macros specified in the headers `<float.h>`, `<limits.h>`, and `<stdint.h>` (5.2.4.2, 7.18.2, 7.18.3).

Macros in `<float.h>`:

Macro <float.h>	Value
FLT_RADIX	2
FLT_EVAL_METHOD	0
FLT_ROUNDS	1 (round to nearest)
FLT_MANT_DIG	24
FLT_DIG	6
FLT_EPSILON	1.19209290E-07F
FLT_MIN	1.17549435E-38F
FLT_MIN_EXP	-125
FLT_MIN_10_EXP	-37
FLT_MAX	3.40282347E+38F
FLT_MAX_EXP	+128
FLT_MAX_10_EXP	+38
[L]DBL_MANT_DIG	53
[L]DBL_DIG	15
[L]DBL_EPSILON	2.2204460492503131E-16
[L]DBL_MIN	2.2250738585072014E-308
[L]DBL_MIN_EXP	-1021
[L]DBL_MIN_10_EXP	-307
[L]DBL_MAX	1.7976931348623157E+308
[L]DBL_MAX_EXP	+1024
[L]DBL_MAX_10_EXP	+308
DECIMAL_DIG	17 (for double FP), 9 (for single FP)
FLT16_MANT_DIG	11
FLT16_DIG	3
FLT16_EPSILON	9.765625E-4F
FLT16_MIN	6.103515625E-05F
FLT16_MIN_EXP	-13
FLT16_MIN_10_EXP	-4
FLT16_MAX	65504.0F
FLT16_MAX_EXP	+16

Macro <float.h>	Value
FLT16_MAX_10_EXP	+4
FLT16_HAS_SUBNORM	1
FLT16_TRUE_MIN	FLT16_MIN
FLT16_DECIMAL_DIG	5

Macros in `<limits.h>`:

Macro <limits.h>	Value
CHAR_BIT	8
SCHAR_MIN	-SCHAR_MAX-1
SCHAR_MAX	0x7f
UCHAR_MAX	0xffU
CHAR_MIN	__CHAR_MIN (min value of 'plain' char)
CHAR_MAX	__CHAR_MAX (max value of 'plain' char)
MB_LEN_MAX	1
SHRT_MIN	-SHRT_MAX-1
SHRT_MAX	0x7fff
USHRT_MAX	0xffffU
INT_MIN	-INT_MAX-1
INT_MAX	0x7fffffff
UINT_MAX	0xffffffffU
LONG_MIN	-LONG_MAX-1
LONG_MAX	0x7fffffffL
ULONG_MAX	0xffffffffUL
LLONG_MIN	-LLONG_MAX-1
LLONG_MAX	0x7fffffffffffffffLL
ULLONG_MAX	0xffffffffffffffffULL

The limit macros in `<stdint.h>` for exact-width, minimum-width and fastest-width integer types have the same ranges as `char`, `short`, `int`, `long` and `long long`. Furthermore the following macros are defined:

Macro <stdint.h>	Value
INTPTR_MIN	INT32_MIN
INTPTR_MAX	INT32_MAX
UINTPTR_MAX	UINT32_MAX
INTMAX_MIN	INT64_MIN

Macro <stdint.h>	Value
INTMAX_MAX	INT64_MAX
UINTMAX_MAX	UINT64_MAX
PTRDIFF_MIN	__PTRDIFF_MIN
PTRDIFF_MAX	__PTRDIFF_MAX
SIG_ATOMIC_MIN	INT32_MIN
SIG_ATOMIC_MAX	INT32_MAX
SIZE_MAX	__SIZE_MAX
WCHAR_MIN	__WCHAR_MIN
WCHAR_MAX	__WCHAR_MAX
WINT_MIN	0
WINT_MAX	UINT32_MAX

- The number, order, and encoding of bytes in any object (when not explicitly specified in this International Standard) (6.2.6.1).

  All types are described in Section 1.1, *Data Types*.

- The value of the result of the sizeof operator (6.5.3.4).

  The value of the size of the data types is described in Section 1.1, *Data Types*. Divide the size by 8 to get bytes, because the table lists the size of the data types in bits.

18.2. C99 Locale-specific Behavior

Locale-specific behavior is behavior that depends on local conventions of nationality, culture, and language that each implementation documents.

The following items describe the locale-specific characteristics, as indicated in Annex J.4 of the ISO/IEC 9899:1999 (E) standard.

- Additional members of the source and execution character sets beyond the basic character set (5.2.1).

  The compiler accepts all one-byte characters in the host's default character set. Use of variable length encoded characters in the source file in comments and string literals is permitted.

  In the implementation in the C library, the basic sequences of a multibyte character consist of single bytes (MB_LEN_MAX is set to 1). If you want full multibyte support, you need to change the C library. See the notes in the header files stdio.h and wchar.h for more information.

- The presence, meaning, and representation of additional multibyte characters in the execution character set beyond the basic character set (5.2.1.2).

  Use of variable length encoded characters in the source file in comments and string literals is permitted.

- The shift states used for the encoding of multibyte characters (5.2.1.2).

  A multibyte character must be a single byte when in the initial shift state.

- The direction of writing of successive printing characters (5.2.2).

  The direction of writing depends on the application and the display device.

- The decimal-point character (7.1.1).

  The default decimal-point character is a '.'.

- The set of printing characters (7.4, 7.25.2).

  The set of printing characters are the characters for which the `isprint()` function returns true. Printing characters are characters in the range 32 (space) to 126.

- The set of control characters (7.4, 7.25.2).

  The set of control characters are the characters for which the `iscntrl()` function returns true. Control characters are characters in the range 0 to 31 and 127.

- The sets of characters tested for by the `isalpha`, `isblank`, `islower`, `ispunct`, `isspace`, `isupper`, `iswalpha`, `iswblank`, `iswlower`, `iswpunct`, `iswspace`, or `iswupper` functions (7.4.1.2, 7.4.1.3, 7.4.1.7, 7.4.1.9, 7.4.1.10, 7.4.1.11, 7.25.2.1.2, 7.25.2.1.3, 7.25.2.1.7, 7.25.2.1.9, 7.25.2.1.10, 7.25.2.1.11).

  The characters tested for are specified in the following table.

Function	Characters tested
`isalpha`	a-z, A-Z
`isblank`	`' '` (space), `'\t'` (tab)
`islower`	a-z
`ispunct`	!, ", #, $, %, &, ', (,), *, +, ,, -, ., /, :, ;, <, =, >, ?, @, [, \,], ^, _, `, {, |, }, ~
`isspace`	`' '` (space), `'\t'`, `'\n'`, `'\v'`, `'\f'`, `'\r'`
`isupper`	A-Z

- The native environment (7.11.1.1).

  The native environment is the same as the "C" locale.

- Additional subject sequences accepted by the numeric conversion functions (7.20.1, 7.24.4.1).

  No additional subject sequences are accepted.

- The collation sequence of the execution character set (7.21.4.3, 7.24.4.4.2).

  Only the "C" locale is supported. The `strcoll()` function is the same as the `strcmp()` function. The `wcscoll()` function is the same as the `wcscmp()` function.

- The contents of the error message strings set up by the `strerror` function (7.21.6.2).

  The error message strings returned by `strerror()` depend on the argument. Typically, the values for the argument come from `errno.h`. For a list of messages see Section 12.1.5, *errno.h*).

- The formats for time and date (7.23.3.5, 7.24.5.1).

  English names for months and days are used.

  `%c` is replaced by the following date and time representation: `%a %b %e %H:%M:%S %Y`

  `%x` is replaced by the following date representation: `%m/%d/%y`

  `%X` is replaced by the following time representation: `%H:%M:%S`

- Character mappings that are supported by the `towctrans` function (7.25.1).

  The character mappings supported by the `towctrans()` function are defined in `wctype.h`:`_to_lower` and `_to_upper`.

- Character classifications that are supported by the `iswctype` function (7.25.1).

  The character classifications supported by the `iswctype()` function are defined in `wctype.h`:`_alnum`, `_alpha`, `_cntrl`, `_digit`, `_graph`, `_lower`, `_print`, `_punct`, `_space`, `_upper`, `_xdigit` and `_blank`.

18.3. C11/C17 Implementation-defined Behavior

Implementation-defined behavior is unspecified behavior where each implementation documents how the choice is made.

The following sections describe the implementation-defined characteristics. The section numbers listed in parenthesis refer to the corresponding sections in the ISO C11 and C17 standard. The order in this chapter is the same as used in Annex J.3 of the ISO/IEC 9899:2011 (E) and ISO/IEC 9899:2018 (E) standard.

18.3.1. Translation

- How a diagnostic is identified (3.10, 5.1.1.3).

  The C compiler diagnostics are explained in Section 4.8, *C Compiler Error Messages*.

- Whether each nonempty sequence of white-space characters other than new-line is retained or replaced by one space character in translation phase 3 (5.1.1.2).

  White-space is retained.

18.3.2. Environment

- The mapping between physical source file multibyte characters and the source character set in translation phase 1 (5.1.1.2).

Use of variable length encoded characters in the source file in comments and string literals is permitted. For source file characters 0D and 1A the mapping is one-to-zero, for other characters a one-on-one mapping is done without interpretation of multibyte characters.

- The name and type of the function called at program startup in a freestanding environment (5.1.2.1).

  The function called at program startup (in `cstart.c`) is called `main`. The prototype for `main` in `cstart.c` is:

  ```
  extern int main( int argc, char *argv[] );
  ```

- The effect of program termination in a freestanding environment (5.1.2.1).

  Execution is halted if the program is executed under control of a debugger, otherwise the program will loop forever in function `_Exit()`.

- An alternative manner in which the `main` function may be defined (5.1.2.2.1).

  You can change the definition of `main` by altering file `cstart.c`.

- The values given to the strings pointed to by the `argv` argument to `main` (5.1.2.2.1).

  The strings get their values from the arguments given in file `cstart.c`. The program arguments are treated case sensitive.

- What constitutes an interactive device (5.1.2.3).

  The streams `stdin`, `stdout` and `stderr` are treated as interactive devices. The debugger uses these streams with File System Simulation (FSS) windows to interact.

- Whether a program can have more than one thread of execution in a freestanding environment (5.1.2.4).

  There is only a single thread of execution.

- The set of signals, their semantics, and their default handling (7.14).

  The signals are described in Section 12.1.17, *signal.h*.

- Signal values other than `SIGFPE`, `SIGILL`, and `SIGSEGV` that correspond to a computational exception (7.14.1.1).

  There are no other values that correspond to a computational exception. All signal values are described in Section 12.1.17, *signal.h*.

- Signals for which the equivalent of `signal(sig, SIG_IGN);` is executed at program startup (7.14.1.1).

  By default the implementation does not ignore any signals at program startup.

- The set of environment names and the method for altering the environment list used by the `getenv` function (7.22.4.5).

There are no implementation-defined environment names that are used by the `getenv` function. A skeleton is provided for the `getenv` function in the C library, because the embedded environment has no operating system. The `getenv` function calls the name as a `void` function.

- The manner of execution of the string by the `system` function (7.22.4.6).

  A skeleton is provided for the `system()` function in the C library, because the embedded environment has no operating system. The `system()` function calls the string as a `void` function.

18.3.3. Identifiers

- Which additional multibyte characters may appear in identifiers and their correspondence to universal character names (6.4.2).

  No additional multibyte characters are supported in an identifier.

- The number of significant initial characters in an identifier (5.2.4.1, 6.4.2).

  All characters in an identifier are significant.

18.3.4. Characters

- The number of bits in a byte (3.6).

  There are eight bits in a byte.

- The values of the members of the execution character set (5.2.1).

  Only 8-bit characters are supported. The values of the execution character set are the same as that of the source character set. The same representation value is used for each member in the characters sets except for the escape sequences.

- The unique value of the member of the execution character set produced for each of the standard alphabetic escape sequences (5.2.2).

  The following table contains an overview of the escape sequences and their byte value in the execution character set.

Escape sequence	Value
\a	7
\b	8
\f	12
\n	10
\r	13
\t	9
\v	11

- The value of a `char` object into which has been stored any character other than a member of the basic execution character set (6.2.5).

  Any 8-bit value can be stored in a `char` object.

- Which of `signed char` or `unsigned char` has the same range, representation, and behavior as "plain" `char` (6.2.5, 6.3.1.1).

  By default "plain" `char` is the same as specifying `unsigned char`. With C compiler option **--schar** you can change the default to `signed char`.

- The mapping of members of the source character set (in character constants and string literals) to members of the execution character set (6.4.4.4, 5.1.1.2).

  The mapping is one-to-one. The values of the execution character set are the same as that of the source character set. The same representation value is used for each member in the characters sets except for the escape sequences.

- The value of an integer character constant containing more than one character or containing a character or escape sequence that does not map to a single-byte execution character (6.4.4.4).

  For any character constant containing more than one character, a warning is issued and the value is truncated to type `signed char`.

- The value of a wide character constant containing more than one multibyte character or a single multibyte character that maps to multiple members of the extended execution character set, or containing a multibyte character or escape sequence not represented in the extended execution character set (6.4.4.4).

  A wide character constant can contain at most two multibyte characters. Its value is the concatenation of the multibyte characters represented in a `signed short int`.

- The current locale used to convert a wide character constant consisting of a single multibyte character that maps to a member of the extended execution character set into a corresponding wide character code (6.4.4.4).

  By default, the "C" locale is used.

- Whether differently-prefixed wide string literal tokens can be concatenated and, if so, the treatment of the resulting multibyte character sequence (6.4.5).

  Differently-prefixed wide string literals can be concatenated. The encoding prefix of the first literal determines the treatment of all literals.

- The current locale used to convert a wide string literal into corresponding wide character codes (6.4.5).

  By default, the "C" locale is used.

- The value of a string literal containing a multibyte character or escape sequence not represented in the execution character set (6.4.5).

  All source characters can be represented in the execution character set.

- The encoding of any of `wchar_t`, `char16_t`, and `char32_t` where the corresponding standard encoding macro (`__STDC_ISO_10646__`, `__STDC_UTF_16__`, or `__STDC_UTF_32__`) is not defined (6.10.8.2).

  The C compiler implements these typedefs with the following types:

Typedef	Implementation
`wchar_t`	`signed short`
`char16_t`	`unsigned short`
`char32_t`	`unsigned int`

18.3.5. Integers

- Any extended integer types that exist in the implementation (6.2.5).

  All types are described in Section 1.1, *Data Types*.

- Whether signed integer types are represented using sign and magnitude, two's complement, or ones' complement, and whether the extraordinary value is a trap representation or an ordinary value (6.2.6.2).

  Signed integer types are represented in two's complement. The most significant bit is the sign bit. 1 is negative, 0 is positive.

- The rank of any extended integer type relative to another extended integer type with the same precision (6.3.1.1).

  All types are described in Section 1.1, *Data Types*.

- The result of, or the signal raised by, converting an integer to a signed integer type when the value cannot be represented in an object of that type (6.3.1.3).

  At compile time, when converting integer types and a value does not fit in a type, the compiler issues a warning and the value is truncated. At run-time no warning or signal is given and the value is truncated.

- The results of some bitwise operations on signed integers (6.5).

  The result of E1>>E2 is E1 right shifted E2 bit positions. If E1 has a signed type and a negative value, the shift behavior is implemented as an arithmetic shift. The empty position in the most significant bit is filled with a copy of the original most significant bit.

18.3.6. Floating-Point

- The accuracy of the floating-point operations and of the library functions in `<math.h>` and `<complex.h>` that return floating-point results (5.2.4.2.2).

  A `float` has an exponent of 8 bits and a significand of 24 bits. A `double` or `long double` has an exponent of 11 bits and a significand of 53 bits. This is conform IEEE-754 for single precision and

double precision floating-point. Internally the compiler uses a significand of 80 bits. The results of floating-point operations are rounded to the nearest IEEE-754 format.

The accuracy of `sqrt` is defined unknown.

- The accuracy of the conversions between floating-point internal representations and string representations performed by the library functions in `<stdio.h>`, `<stdlib.h>`, and `<wchar.h>` (5.2.4.2.2).

The accuracy of the conversions is unknown.

- The rounding behaviors characterized by non-standard values of `FLT_ROUNDS` (5.2.4.2.2).

No non-standard values are used.

- The evaluation methods characterized by non-standard negative values of `FLT_EVAL_METHOD` (5.2.4.2.2).

`FLT_EVAL_METHOD` is defined as 0. No non-standard values are used.

- The presence or absence of subnormal numbers (5.2.4.2.2)

Subnormals support is characterized as present: macros FLT_HAS_SUBNORM, FLT16_HAS_SUBNORM, DBL_HAS_SUBNORM, and LDBL_HAS_SUBNORM are defined to 1.

- The direction of rounding when an integer is converted to a floating-point number that cannot exactly represent the original value (6.3.1.4).

For FPU instructions, the rounding mode is used. For software floating-point instructions, the round to nearest method is used. `FLT_ROUNDS` is ignored.

- The direction of rounding when a floating-point number is converted to a narrower floating-point number (6.3.1.5).

The round to nearest method is used. `FLT_ROUNDS` is ignored.

- How the nearest representable value or the larger or smaller representable value immediately adjacent to the nearest representable value is chosen for certain floating constants (6.4.4.2).

The round to nearest method is used. `FLT_ROUNDS` is ignored.

- Whether and how floating expressions are contracted when not disallowed by the `FP_CONTRACT` pragma (6.5).

Pragma `FP_CONTRACT` is equivalent to compiler option **--fp-model=+contract**. This has only effect for fused multiply-and-accumulate (FMA) operations. FMA operations are not supported by the IEEE 754-1985 standard. The result of FMA operations is only rounded once at the end of the FMA. You can disable FMAs with the compiler option **--fp-model=-contract**.

- The default state for the `FENV_ACCESS` pragma (7.6.1).

The default state of pragma `FENV_ACCESS` is "off". This pragma is ignored.

- Additional floating-point exceptions, rounding modes, environments, and classifications, and their macro names (7.6, 7.12).

  No additional floating-point exceptions, rounding modes, environments, and classifications are defined.

- The default state for the `FP_CONTRACT` pragma (7.12.2).

  The default state of pragma `FP_CONTRACT` is set by the **contract** flag of compiler option **--fp-model**. The default state is "on".

18.3.7. Arrays and Pointers

- The result of converting a pointer to an integer or vice versa (6.3.2.3).

  All non-pointer conversions to and from a 32-bit pointer are implemented as a conversion to or from a 32-bit integer type.

- The size of the result of subtracting two pointers to elements of the same array (6.5.6).

  The size of `ptrdiff_t` is 32 bits. The difference in address location is expressed in bytes.

18.3.8. Hints

- The extent to which suggestions made by using the `register` storage-class specifier are effective (6.7.1).

  The compiler does not make assumptions based on the `register` storage-class specifier. So, basically this keyword is ignored, except that you cannot take the address of a `register` variable. The compiler issues an error in that case.

- The extent to which suggestions made by using the `inline` function specifier are effective (6.7.4).

  With the `inline` keyword you force the compiler to inline the specified function, regardless of the optimization strategy of the compiler itself. For more information see Section 1.10.2, *Inlining Functions: inline*.

18.3.9. Structures, Unions, Enumerations, and Bit-fields

- Whether a "plain" `int` bit-field is treated as a `signed int` bit-field or as an `unsigned int` bit-field (6.7.2, 6.7.2.1).

  By default an `int` bit-field is treated as `signed int`. You can still add the keyword `unsigned` to treat a particular `int` bit-field as `unsigned`.

- Allowable bit-field types other than `_Bool`, `signed int`, and `unsigned int` (6.7.2.1).

  All integer types as specified in Section 1.1, *Data Types* are allowable bit-field types.

- Whether atomic types are permitted for bit-fields (6.7.2.1).

  Atomic types are not permitted for bit-fields.

- Whether a bit-field can straddle a storage-unit boundary (6.7.2.1).

  A bit-field cannot straddle a storage-unit boundary. If insufficient space remains, the bit-field is put into the next unit.

- The order of allocation of bit-fields within a unit (6.7.2.1).

  Allocation starts at the least significant bit up to the most significant bit. If the following bit-field fits within the same unit, it is allocated starting at the next available bit.

- The alignment of non-bit-field members of structures (6.7.2.1). This should present no problem unless binary data written by one implementation is read by another.

  The alignment of non-bit-field members of structures is the same as the alignment for data types as specified in Section 1.1, *Data Types*.

- The integer type compatible with each enumerated type (6.7.2.2).

  The compiler chooses the smallest suitable integer type (char, unsigned char, short, unsigned short or int).

18.3.10. Qualifiers

- What constitutes an access to an object that has volatile-qualified type (6.7.3).

  Any reference to an object with volatile type results in an access. The order in which volatile objects are accessed is defined by the order expressed in the source code. References to non-volatile objects are scheduled in arbitrary order, within the constraints given by dependencies.

  If the compiler option **--language=+volatile** (**-Av**) is set, all references to non-volatile objects result in an access before the access to a volatile object that occurs subsequently in the source file takes place. The volatile access acts as a memory barrier.

18.3.11. Preprocessing Directives

- The locations within #pragma directives where header name preprocessing tokens are recognized (6.4, 6.4.7).

  Within a #pragma directive, header name preprocessing tokens are not recognized.

- How sequences in both forms of header names are mapped to headers or external source file names (6.4.7).

  Sequences in header names are mapped to file names as is. The backslash "\" is not interpreted as an escape sequence. The backslash "\" (Windows) or forward slash "/" (Windows and UNIX) is interpreted as a standard directory separator.

- Whether the value of a character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set (6.10.1).

  A character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set.

- Whether the value of a single-character character constant in a constant expression that controls conditional inclusion may have a negative value (6.10.1).

  A single-character character constant in a constant expression that controls conditional inclusion may have a negative value.

- The places that are searched for an included < > delimited header, and how the places are specified or the header is identified (6.10.2).

  How the compiler searches for include files is explained in Section 4.4, *How the Compiler Searches Include Files*.

- How the named source file is searched for in an included " " delimited header (6.10.2).

  How the compiler searches for include files is explained in Section 4.4, *How the Compiler Searches Include Files*.

- The method by which preprocessing tokens (possibly resulting from macro expansion) in a `#include` directive are combined into a header name (6.10.2).

  Preprocessing tokens in a `#include` directive are combined the same way as outside a `#include` directive.

- The nesting limit for `#include` processing (6.10.2).

  There is no nesting limit for `#include` processing.

- Whether the `#` operator inserts a \ character before the \ character that begins a universal character name in a character constant or string literal (6.10.3.2).

  The `#` operator inserts a \ character before every \ character in a character constant or string literal.

- The behavior on each recognized non-STDC `#pragma` directive (6.10.6).

  All non-STDC pragmas are described in Section 1.8, *Pragmas to Control the Compiler*.

- The definitions for `__DATE__` and `__TIME__` when respectively, the date and time of translation are not available (6.10.8.1).

  The date and time of translation are always available, macros `__DATE__` and `__TIME__` are always defined.

18.3.12. Library Functions

- Any library facilities available to a freestanding program, other than the minimal set required by Clause 4 (5.1.2.1).

  All library functions are described in Chapter 12, *Libraries*. Some functions are not completely implemented because their implementation depends on the context where your application will run. These functions are for example all I/O related functions. Where possible, these functions are implemented using file system simulation (FSS). This system can be used by the debugger to simulate an I/O environment which enables you to debug your application. If the application runs under control

of the debugger and FSS is used, then the low-level behavior is equal to the host system's file access behavior.

In the implementation in the C library, the basic sequences of a multibyte character consist of single bytes (`MB_LEN_MAX` is set to 1). If you want full multibyte support, you need to change the C library. See the notes in the header files `stdio.h` and `wchar.h` for more information.

- The format of the diagnostic printed by the `assert` macro (7.2.1.1).

    The `assert()` function is implemented as a macro in `assert.h`. The output is:

    ```
    Assertion failed: (expression) file filename, line linenumber
    ```

    when the parameter evaluates to zero.

- The representation of the floating-point status flags stored by the `fegetexceptflag` function (7.6.2.2).

    Floating-point exceptions are not supported. Function `fegetexceptflag` does nothing.

- Whether the `feraiseexcept` function raises the "inexact" floating-point exception in addition to the "overflow" or "underflow" floating-point exception (7.6.2.3).

    Floating-point exceptions are not supported. Function `feraiseexcept` does nothing.

- Strings other than `"C"` and `""` that may be passed as the second argument to the `setlocale` function (7.11.1.1).

    No other strings are predefined. A NULL pointer as the second argument returns the "C" locale. Any other string than `"C"` or `""` can be passed as the second argument to the `setlocale` function and results in NULL.

- The types defined for `float_t` and `double_t` when the value of the `FLT_EVAL_METHOD` macro is less than 0 (7.12).

    The `FLT_EVAL_METHOD` macro can only have the values 0, 1 or 2.

- Domain errors for the mathematical functions, other than those required by this International Standard (7.12.1).

    No other domain errors exist, other than those required by the standard.

- The values returned by the mathematical functions on domain errors or pole errors (7.12.1).

    On domain errors (`errno` is set to `EDOM`), the mathematical functions return a value as specified in the following table.

Math function	Return value on EDOM
`acos(|x| > 1.0)`	0.0
`asin(|x| > 1.0)`	0.0
`log(x < 0.0)`	`-HUGE_VAL`
`pow(x <= 0.0)`	0.0

Math function	Return value on EDOM
`sqrt(x < 0.0)`	-NaN

- The values returned by the mathematical functions on underflow range errors, whether `errno` is set to the value of the macro `ERANGE` when the integer expression `math_errhandling & MATH_ERRNO` is nonzero, and whether the "underflow" floating-point exception is raised when the integer expression `math_errhandling & MATH_ERREXCEPT` is nonzero. (7.12.1).

  On underflow range errors, the mathematical functions return 0.0. `math_errhandling` is set to `MATH_ERRNO`. Trapping and non-trapping versions of the library are available. With a non-trapping library `errno` is not set to `ERANGE` on underflow range errors, with a trapping library no underflow exception is raised.

- Whether a domain error occurs or zero is returned when an `fmod` function has a second argument of zero (7.12.10.1).

  Zero (`0.0`) is returned when an `fmod` function has a second argument of zero.

- Whether a domain error occurs or zero is returned when a remainder function has a second argument of zero (7.12.10.2).

  A domain error occurs.

- The base-2 logarithm of the modulus used by the `remquo` functions in reducing the quotient (7.12.10.3).

  The `remquo` function calculates at least 8 bits of the quotient.

- Whether a domain error occurs or zero is returned when a `remquo` function has a second argument of zero (7.12.10.3).

  A domain error occurs.

- Whether the equivalent of `signal(sig, SIG_DFL);` is executed prior to the call of a signal handler, and, if not, the blocking of signals that is performed (7.14.1.1).

  The equivalent of `signal(sig, SIG_DFL);` is executed prior to the call of a signal handler.

- The null pointer constant to which the macro `NULL` expands (7.19).

  Macro `NULL` is defined as `(void *) 0`.

- Whether the last line of a text stream requires a terminating new-line character (7.21.2).

  Both a new-line character (`\n`) and end-of-file (`EOF`) are recognized as the termination character of a line.

- Whether space characters that are written out to a text stream immediately before a new-line character appear when read in (7.21.2).

  Space characters written to a stream immediately before a new-line character are preserved.

- The number of null characters that may be appended to data written to a binary stream (7.21.2).

  I/O related functions are implemented using file system simulation (FSS). This system can be used by the debugger to simulate an I/O environment. If the application runs under control of the debugger and FSS is used, then the low-level behavior is equal to the host system's file access behavior. The library does not append any null characters. It depends on the `open()` function on the host environment what happens. You can write your own `_open()` function if necessary.

- Whether the file position indicator of an append-mode stream is initially positioned at the beginning or end of the file (7.21.3).

  I/O related functions are implemented using file system simulation (FSS). This system can be used by the debugger to simulate an I/O environment. If the application runs under control of the debugger and FSS is used, then the low-level behavior is equal to the host system's file access behavior. Where the file position indicator of an append-mode stream is initially positioned depends on the `open()` function on the host environment. You can write your own `_open()` function if necessary.

- Whether a write on a text stream causes the associated file to be truncated beyond that point (7.21.3).

  I/O related functions are implemented using file system simulation (FSS). This system can be used by the debugger to simulate an I/O environment. If the application runs under control of the debugger and FSS is used, then the low-level behavior is equal to the host system's file access behavior. Whether a write on a text stream causes the associated file to be truncated beyond that point depends on how the low-level file routines are implemented in your application.

- The characteristics of file buffering (7.21.3).

  Files can be unbuffered, fully buffered or line buffered. What actually happens depends on how the low-level file routines are implemented in your application.

- Whether a zero-length file actually exists (7.21.3).

  This depends on how the low-level file routines are implemented in your application.

- The rules for composing valid file names (7.21.3).

  This depends on how the low-level file routines are implemented in your application.

- Whether the same file can be simultaneously open multiple times (7.21.3).

  This depends on how the low-level file routines are implemented in your application.

- The nature and choice of encodings used for multibyte characters in files (7.21.3).

  Use of variable length encoded characters in files in comments and string literals is permitted.

- The effect of the `remove` function on an open file (7.21.4.1).

  This depends on how the low-level file routines are implemented in your application.

- The effect if a file with the new name exists prior to a call to the `rename` function (7.21.4.2).

  This depends on how the low-level file routines are implemented in your application.

- Whether an open temporary file is removed upon abnormal program termination (7.21.4.3).

  This depends on how the low-level file routines are implemented in your application.

- Which changes of mode are permitted (if any), and under what circumstances (7.21.5.4).

  The `freopen()` function first calls `fclose()` and then calls `_fopen()` with the new mode.

- The style used to print an infinity or NaN, and the meaning of any n-char or n-wchar sequence printed for a NaN (7.21.6.1, 7.29.2.1).

  The style used to print an infinity or NaN is `inf` and `nan` respectively (`INF` or `NAN` for the `F` conversion specifier). n-char or n-wchar sequences are not used for `nan`.

- The output for %p conversion in the `fprintf` or `fwprintf` function (7.21.6.1, 7.29.2.1).

  The argument is treated as having type `void *`. The value will be printed as a hexadecimal value, similar to `%x`.

- The interpretation of a – character that is neither the first nor the last character, nor the second where a ^ character is the first, in the scanlist for `%[` conversion in the `fscanf` or `fwscanf` function (7.21.6.2, 7.29.2.1).

  A – character is treated as a normal character.

- The set of sequences matched by a `%p` conversion and the interpretation of the corresponding input item in the `fscanf` or `fwscanf` function (7.21.6.2, 7.29.2.2).

  The format of `%p` matches the format of `%x`. The input for `%p` is a hexadecimal value, which is converted to a value with type `void *`.

- The value to which the macro `errno` is set by the `fgetpos`, `fsetpos`, or `ftell` functions on failure (7.21.9.1, 7.21.9.3, 7.21.9.4).

  If `errno` is set to a value depends on how the low-level file routines are implemented in your application.

- The meaning of any n-char or n-wchar sequence in a string representing a NaN that is converted by the `strtod`, `strtof`, `strtold`, `wcstod`, `wcstof`, or `wcstold` function (7.22.1.3, 7.29.4.1.1).

  An n-char or n-wchar sequence in a string representing a NaN is ignored.

- Whether or not the `strtod`, `strtof`, `strtold`, `wcstod`, `wcstof`, or `wcstold` function sets `errno` to `ERANGE` when underflow occurs (7.22.1.3, 7.29.4.1.1).

  `errno` is set to `ERANGE` when underflow occurs and the value returned is 0.0.

- Whether the `calloc`, `malloc`, `realloc`, and `aligned_alloc` functions return a null pointer or a pointer to an allocated object when the size requested is zero (7.22.3).

  `NULL` is returned when a size of zero is requested.

- Whether open streams with unwritten buffered data are flushed, open streams are closed, or temporary files are removed when the `abort` or `_Exit` function is called (7.22.4.1, 7.22.4.4).

When the `abort()` or `_Exit()` function is called, open streams with unwritten buffered data are not flushed, open streams are not closed, and temporary files are not removed.

- The termination status returned to the host environment by the `abort`, `exit`, `_Exit`, or `quick_exit` function (7.22.4.1, 7.22.4.4, 7.22.4.5, 7.22.4.7).

  `abort()` calls `_Exit()` with `EXIT_FAILURE`.

  `exit()` and `quick_exit` call `_Exit()` with their input value.

  `_Exit()` returns the input value to the host environment.

- The value returned by the `system` function when its argument is not a null pointer (7.22.4.6).

  A skeleton is provided for the `system()` function in the C library, because the embedded environment has no operating system. The `system()` function returns the value 0.

- The range and precision of times representable in `clock_t` and `time_t` (7.27).

  `clock_t` is defined as `unsigned long long`, `time_t` is defined as `unsigned long`. The resolution of the clock is defined by `CLOCKS_PER_SEC`, which value is hard-coded to 500000000 (500MHz).

- The local time zone and Daylight Saving Time (7.27.1).

  The default time zone is UTC. Daylight Saving Time is not available (`tm_isdst=-1`).

- The era for the `clock` function (7.27.2.1).

  The `clock` function returns the current processor time. It reads the 64-bit real-time counter (RTC).

- The `TIME_UTC` epoch (7.27.2.5).

  The `timespec_get()` function is based on the `clock()` function. Therefore, the epoch is the starting time of the `clock()` function.

- The replacement string for the `%Z` specifier to the `strftime`, and `wcsftime` functions in the `"C"` locale (7.27.3.5, 7.29.5.1).

  `%Z` is replaced by the time zone name, by default UTC.

- Whether the functions in `<math.h>` honor the rounding direction mode in an IEC 60559 conformant implementation, unless explicitly specified otherwise (F.10).

  The round to nearest method is used. `FLT_ROUNDS` is defined as 1.

18.3.13. Architecture

- The values or expressions assigned to the macros specified in the headers `<float.h>`, `<limits.h>`, and `<stdint.h>` (5.2.4.2, 7.20.2, 7.20.3).

  Macros in `<float.h>`:

Macro <float.h>	Value
`FLT_RADIX`	2
`FLT_EVAL_METHOD`	0
`FLT_ROUNDS`	1 (round to nearest)
`FLT_MANT_DIG`	24
`FLT_DIG`	6
`FLT_EPSILON`	1.19209290E-07F
`FLT_MIN`	1.17549435E-38F
`FLT_MIN_EXP`	-125
`FLT_MIN_10_EXP`	-37
`FLT_MAX`	3.40282347E+38F
`FLT_MAX_EXP`	+128
`FLT_MAX_10_EXP`	+38
`FLT_HAS_SUBNORM`	1
`FLT_TRUE_MIN`	`FLT_MIN`
`FLT_DECIMAL_DIG`	9
`[L]DBL_MANT_DIG`	53
`[L]DBL_DIG`	15
`[L]DBL_EPSILON`	2.2204460492503131E-16
`[L]DBL_MIN`	2.2250738585072014E-308
`[L]DBL_MIN_EXP`	-1021
`[L]DBL_MIN_10_EXP`	-307
`[L]DBL_MAX`	1.7976931348623157E+308
`[L]DBL_MAX_EXP`	+1024
`[L]DBL_MAX_10_EXP`	+308
`[L]DBL_HAS_SUBNORM`	1
`[L]DBL_TRUE_MIN`	`DBL_MIN`
`[L]DBL_DECIMAL_DIG`	17 (for double FP), 9 (for single FP)
`DECIMAL_DIG`	17 (for double FP), 9 (for single FP)
`FLT16_MANT_DIG`	11
`FLT16_DIG`	3
`FLT16_EPSILON`	9.765625E-4F
`FLT16_MIN`	6.103515625E-05F
`FLT16_MIN_EXP`	-13
`FLT16_MIN_10_EXP`	-4

Macro <float.h>	Value
FLT16_MAX	65504.0F
FLT16_MAX_EXP	+16
FLT16_MAX_10_EXP	+4
FLT16_HAS_SUBNORM	1
FLT16_TRUE_MIN	FLT16_MIN
FLT16_DECIMAL_DIG	5

Macros in `<limits.h>`:

Macro <limits.h>	Value
CHAR_BIT	8
SCHAR_MIN	-SCHAR_MAX-1
SCHAR_MAX	0x7f
UCHAR_MAX	0xffU
CHAR_MIN	__CHAR_MIN (min value of 'plain' char)
CHAR_MAX	__CHAR_MAX (max value of 'plain' char)
MB_LEN_MAX	1
SHRT_MIN	-SHRT_MAX-1
SHRT_MAX	0x7fff
USHRT_MAX	0xffffU
INT_MIN	-INT_MAX-1
INT_MAX	0x7fffffff
UINT_MAX	0xffffffffU
LONG_MIN	-LONG_MAX-1
LONG_MAX	0x7fffffffL
ULONG_MAX	0xffffffffUL
LLONG_MIN	-LLONG_MAX-1
LLONG_MAX	0x7fffffffffffffffLL
ULLONG_MAX	0xffffffffffffffffULL

The limit macros in `<stdint.h>` for exact-width, minimum-width and fastest-width integer types have the same ranges as `char`, `short`, `int`, `long` and `long long`. Furthermore the following macros are defined:

Macro <stdint.h>	Value
INTPTR_MIN	INT32_MIN
INTPTR_MAX	INT32_MAX

Macro <stdint.h>	Value
UINTPTR_MAX	UINT32_MAX
INTMAX_MIN	INT64_MIN
INTMAX_MAX	INT64_MAX
UINTMAX_MAX	UINT64_MAX
PTRDIFF_MIN	__PTRDIFF_MIN
PTRDIFF_MAX	__PTRDIFF_MAX
SIG_ATOMIC_MIN	INT32_MIN
SIG_ATOMIC_MAX	INT32_MAX
SIZE_MAX	__SIZE_MAX
WCHAR_MIN	__WCHAR_MIN
WCHAR_MAX	__WCHAR_MAX
WINT_MIN	0
WINT_MAX	UINT32_MAX

- The result of attempting to indirectly access an object with automatic or thread storage duration from a thread other than the one with which it is associated (6.2.4).

  Threads are not supported (__STDC_NO_THREADS__ = 1).

- The number, order, and encoding of bytes in any object (when not explicitly specified in this International Standard) (6.2.6.1).

  All types are described in Section 1.1, *Data Types*.

- Whether any extended alignments are supported and the contexts in which they are supported (6.2.8).

  For automatic objects, an extended alignment of 8 is supported. For statically allocated objects, extended alignments greater than or equal to 8 are supported.

- Valid alignment values other than those returned by an _Alignof expression for fundamental types, if any (6.2.8).

  Any nonnegative integral power of two can be used as additional alignment value.

- The value of the result of the sizeof and _Alignof operators (6.5.3.4).

  The values of the size and alignment of the data types are described in Section 1.1, *Data Types*. Divide the numbers by 8 to get bytes.

18.4. C11/C17 Locale-specific Behavior

Locale-specific behavior is behavior that depends on local conventions of nationality, culture, and language that each implementation documents.

The following items describe the locale-specific characteristics, as indicated in Annex J.4 of the ISO/IEC 9899:2011 (E) and ISO/IEC 9899:2018 (E) standard.

- Additional members of the source and execution character sets beyond the basic character set (5.2.1).

  The compiler accepts all one-byte characters in the host's default character set. Use of variable length encoded characters in the source file in comments and string literals is permitted.

  In the implementation in the C library, the basic sequences of a multibyte character consist of single bytes (`MB_LEN_MAX` is set to 1). If you want full multibyte support, you need to change the C library. See the notes in the header files `stdio.h` and `wchar.h` for more information.

- The presence, meaning, and representation of additional multibyte characters in the execution character set beyond the basic character set (5.2.1.2).

  Use of variable length encoded characters in the source file in comments and string literals is permitted.

- The shift states used for the encoding of multibyte characters (5.2.1.2).

  A multibyte character must be a single byte when in the initial shift state.

- The direction of writing of successive printing characters (5.2.2).

  The direction of writing depends on the application and the display device.

- The decimal-point character (7.1.1).

  The default decimal-point character is a `'.'`.

- The set of printing characters (7.4, 7.30.2).

  The set of printing characters are the characters for which the `isprint()` function returns true. Printing characters are characters in the range 32 (space) to 126.

- The set of control characters (7.4, 7.30.2).

  The set of control characters are the characters for which the `iscntrl()` function returns true. Control characters are characters in the range 0 to 31 and 127.

- The sets of characters tested for by the `isalpha`, `isblank`, `islower`, `ispunct`, `isspace`, `isupper`, `iswalpha`, `iswblank`, `iswlower`, `iswpunct`, `iswspace`, or `iswupper` functions (7.4.1.2, 7.4.1.3, 7.4.1.7, 7.4.1.9, 7.4.1.10, 7.4.1.11, 7.30.2.1.2, 7.30.2.1.3, 7.30.2.1.7, 7.30.2.1.9, 7.30.2.1.10, 7.30.2.1.11).

  The characters tested for are specified in the following table.

Function	Characters tested
`isalpha`	a-z, A-Z
`isblank`	`' '` (space), `'\t'` (tab)
`islower`	a-z
`ispunct`	!, ", #, $, %, &, ', (,), *, +, ,, -, ., /, :, ;, <, =, >, ?, @, [, \,], ^, _, `, {, |, }, ~
`isspace`	`' '` (space), `'\t'`, `'\n'`, `'\v'`, `'\f'`, `'\r'`

Function	Characters tested
`isupper`	A-Z

- The native environment (7.11.1.1).

  The native environment is the same as the "C" locale.

- Additional subject sequences accepted by the numeric conversion functions (7.22.1, 7.29.4.1).

  No additional subject sequences are accepted.

- The collation sequence of the execution character set (7.24.4.3, 7.29.4.4.2).

  Only the "C" locale is supported. The `strcoll()` function is the same as the `strcmp()` function. The `wcscoll()` function is the same as the `wcscmp()` function.

- The contents of the error message strings set up by the `strerror` function (7.24.6.2).

  The error message strings returned by `strerror()` depend on the argument. Typically, the values for the argument come from `errno.h`. For a list of messages see Section 12.1.5, *errno.h*).

- The formats for time and date (7.27.3.5, 7.29.5.1).

  English names for months and days are used.

  `%c` is replaced by the following date and time representation: `%a %b %e %H:%M:%S %Y`

  `%x` is replaced by the following date representation: `%m/%d/%y`

  `%X` is replaced by the following time representation: `%H:%M:%S`

- Character mappings that are supported by the `towctrans` function (7.30.1).

  The character mappings supported by the `towctrans()` function are defined in `wctype.h`:`_to_lower` and `_to_upper`.

- Character classifications that are supported by the `iswctype` function (7.30.1).

  The character classifications supported by the `iswctype()` function are defined in `wctype.h`:`_alnum`, `_alpha`, `_cntrl`, `_digit`, `_graph`, `_lower`, `_print`, `_punct`, `_space`, `_upper`, `_xdigit` and `_blank`.

Chapter 19. C++ Implementation-defined Behavior

The TASKING C++ compiler for ARC/PPU fully supports the ISO C++14 standard, but some parts of the ISO C++14 standard are implementation-defined. This chapter describes how the implementation-defined areas and the implementation quantities of the C++ language are implemented in the TASKING C++ compiler for ISO C++14. It also describes the deviations from the C++11/C++14 standard.

19.1. C++14 Implementation-defined Behavior

Implementation-defined behavior is unspecified behavior where each implementation documents how the choice is made.

The section numbers listed refer to the corresponding sections in the ISO/IEC 14882:2014 standard.

1.3.6 diagnostic message

The C++ compiler diagnostics are explained in Section 5.3, *C++ Compiler Error Messages*.

1.3.10 implementation-defined behavior

This chapter describes the C++ implementation-defined behavior.

1.3.11 implementation limits

See Section 19.2, *C++14 Implementation Quantities*.

1.4 Implementation compliance

Freestanding implementation, LLVM libc++ standard library, with deviations as mentioned in Section 19.3, *Deviations from C++11/C++14*.

1.7 The C++ memory model

There are 8 bits in a byte.

1.9 Program execution

The streams `stdin`, `stdout` and `stderr` are treated as interactive devices.

1.10 Multi-threaded executions and data races

There is only a single thread of execution.

2.2 Phases of translation

1. Use of variable length encoded characters in the source file in comments and string literals is permitted. A one-on-one mapping is done without interpretation of multibyte characters.

2. Sources of translation units containing definitions of templates that are being instantiated are required to be available.

2.3 Character sets

Only 8-bit characters are supported. The values of the execution character set are the same as that of the source character set. The same representation value is used for each member in the characters sets except for the escape sequences.

2.9 Header names

How the C++ compiler searches for include files is explained in Section 5.2, *How the C++ Compiler Searches Include Files*.

2.11 Identifiers

No multibyte character or any other non standard character may appear in identifiers.

2.14.3 Character literals

No multibyte character is allowed in character literals.

2.14.4 Floating literals

The round to nearest representable value is used if the scaled floating point value is not representable.

2.14.5 String literals

Duplicated string literals are merged.

3.6.1 Main function

The function called at program startup (in `cstart.c`) is called `main`. The prototype for `main` in `cstart.c` is:

```
extern int main( int argc, char *argv[] );
```

3.6.2 Initialization of non-local variables

Dynamic initialization of a non-local variable with static storage duration is done before the first statement of main.

3.7.4.2 Deallocation functions

Any use of an invalid pointer value has undefined behavior.

3.7.4.3 Safely-derived pointers

Relaxed pointer safety is used.

3.9 Types

See the data types in Section 1.1, *Data Types*.

3.9.1 Fundamental types

By default `char` is the same as specifying `unsigned char`. With C++ compiler option **--schar** (**-s**) a plain `char` is the same as `signed char`.

A `float` has an exponent of 8 bits and a significand of 24 bits. A `double` or `long double` has an exponent of 11 bits and a significand of 53 bits. This is conform IEEE 754 for single precision and double precision floating-point. Internally the compiler uses a significand of 80 bits. The results of floating-point operations are rounded to the nearest IEEE 754 format.

3.9.2 Compound types

Pointers are 4 bytes.

3.11 Alignment

See the data types in Section 1.1, *Data Types*.

4.1 Lvalue-to-rvalue conversion

Any use of an invalid pointer value has undefined behavior.

4.7 Integral conversions

At compile time, when converting integer types and a value does not fit in a type, the compiler issues a warning and the value is truncated. At run-time no warning or signal is given and the value is truncated.

4.8 Floating point conversions

The round to nearest method is used. FLT_ROUNDS is ignored.

4.9 Floating-integral conversions

For FPU instructions, the rounding mode is used. For software floating-point instructions, the round to nearest method is used. FLT_ROUNDS is ignored.

4.13 Integer conversion rank

See the data types in Section 1.1, *Data Types*.

5.2.2 Function call

Passing a potentially-evaluated argument of class type having a non-trivial copy constructor, a non-trivial move contructor, or a non-trivial destructor, with no corresponding parameter is supported. A warning (W1300) is issued in any of these cases. The argument is passed as if it was trivially copyable, i.e. no copy/move constructors and destructors are invoked to pass the argument.

5.2.8 Type identification

Typeid expression results are only of type `std::type_info`. No derived name classes are provided.

5.2.10 Reinterpret cast

Mapping remains the same. Mapping of pointers to addresses and vice versa uses `int` or larger type.

Converting a function pointer to an object pointer or vice versa is supported.

5.3.3 Sizeof

See the data types in Section 1.1, *Data Types*.

5.3.4 New

See the data types in Section 1.1, *Data Types*.

5.7 Additive operators

`ptrdiff_t` is defined as the smaller of `int` or `long` that can hold a pointer value.

5.8 Shift operators

The result of `E1>>E2` is `E1` right shifted `E2` bit positions. If `E1` has a signed type and a negative value, the shift behavior is implemented as an arithmetic shift. The empty position in the most significant bit is filled with a copy of the original most significant bit.

7 Declarations

See Section 1.7, *Attributes*.

7.1.6.1 The cv-qualifiers

Any reference to an object with volatile type results in an access. The order in which volatile objects are accessed is defined by the order expressed in the source code. References to non-volatile objects are scheduled in arbitrary order, within the constraints given by dependencies.

7.1.6.2 Simple type specifiers

By default `char` is the same as specifying `unsigned char`. With C++ compiler option **--schar** (**-s**) a plain `char` is the same as `signed char`.

7.2 Enumeration declarations

The C++ compiler treats enumerated types as the smallest data type possible (`char`, `short` or `int`). This reduces code size.

7.4 The asm declaration

See Section 1.5, *Using Assembly in the C Source: __asm()*. Both `asm()` and `__asm()` entities are supported.

7.5 Linkage specifications

Routines declared with `extern "C"` will not have name mangling performed on them. The default mode, `extern "C++"`, performs name mangling. No other language linkages are supported.

8.4 Function definitions

The reserved identifier `__func__` expands to the current function name (as well as macro `__FUNCTION__`). An error is issued if it is used outside of a function.

9.6 Bit-fields

The bit-field layout, bit-field alignment, and struct alignment for bit-fields are defined in *section 2.1.4 Bitfields* of the *DesignWare ARCv2 System V ABI Supplement* [Version 4092-007, Nov 01, 2019, Synopsys, Inc.].

14 Templates

Only C and C++ linkages are supported.

14.7.1 Implicit instantiation

Recursive instantiations are limited to 64 by default.

15.5.1 The terminate() function

The stack is unwound before `terminate()` is called.

15.5.2 The unexpected() function

When `unexpected()` exits by throwing an exception the exception must not violate the exception specification that caused `unexpected()` to be called in the first place. If it does violate that exception

specification, `std::bad_exception` is thrown provided it is permitted by the violated exception specification. If it is not permitted, `terminate()` is called.

16.1 Conditional inclusion

A character constant in a constant expression that controls conditional inclusion matches the value of the same character constant in the execution character set. A single-character character constant in a constant expression that controls conditional inclusion may have a negative value.

16.2 Source file inclusion

Sequences in header names are mapped to file names as is. The backslash "\" is not interpreted as an escape sequence. The backslash "`\`" (Windows) or forward slash "`/`" (Windows and UNIX) is interpreted as a standard directory separator. How the C++ compiler searches for include files is explained in Section 5.2, *How the C++ Compiler Searches Include Files*.

16.6 Pragma directive

The C++ compiler supports the pragmas as explained in Section 2.7, *Pragmas to Control the C++ Compiler*.

16.8 Predefined macro names

The C++ compiler supports the predefined macros as explained in Section 2.8, *Predefined Macros*.

17.6.5.8 Reentrancy

Any function with `__attribute__((const))` is reentrant.

17.6.5.12 Restrictions on exception handling

See the C++14 library documentation at releases.llvm.org.

17.6.5.14 Value of error codes

See the C++14 library documentation at releases.llvm.org.

18.2 Types

`NULL` has a value of 0.

`ptrdiff_t` is defined as the smaller of `int` or `long` that can hold a pointer value.

`size_t` is defined as `unsigned int`.

18.3.1 In general

See header files `<limits>`, `<climits>` and `<cfloat>`.

18.3 Start and termination

`exit()` returns the value of argument status; `EXIT_SUCCESS` has a value of 0; `EXIT_FAILURE` has a value of 1.

18.6.2.1 Class bad_alloc

`bad_alloc::what()` returns an empty string.

18.6.2.2 Class bad_array_length

`bad_array_length::what()` is not implemented.

18.6.2.3 Class bad_array_new_length

`bad_array_new_length::what()` is not implemented.

18.7.1 Class type_info

`typeid(T).name()` returns the mangled name of type `T`.

18.7.2 Class bad_cast

`bad_cast::what()` returns an empty string.

18.7.3 Class bad_typeid

`bad_typeid::what()` returns an empty string.

18.8.1 Class exception

`exception::what()` returns an empty string.

18.8.2 Class bad_exception

`bad_exception::what()` returns an empty string.

18.10 Other runtime support

Any non POF (plain old functions) signal handler yields undefined behavior.

20.8.4 Pointer safety

See the C++14 library documentation at releases.llvm.org.

20.8.9.1 allocator members

See the data types in Section 1.1, *Data Types*.

20.8.11 Temporary buffers

See the data types in Section 1.1, *Data Types*.

20.9.2.2.1 shared_ptr

Throws `bad_alloc`.

20.10.9.1.4 Placeholders

There are 10 placeholders. See header file `<functional>` for details.

20.11.7.6 Other transformations

See the data types in Section 1.1, *Data Types*.

20.13.7.1 Class system_clock

Values are truncated to the required precision.

21.2.2 traits typedefs

See the C++14 library documentation at releases.llvm.org.

21.2.3.2 struct char_traits<char16_t>

See the C++14 library documentation at releases.llvm.org.

21.2.3.3 struct char_traits<char32_t>

See the C++14 library documentation at releases.llvm.org.

21.2.3.4 struct char_traits<wchar_t>

See the C++14 library documentation at releases.llvm.org.

21.4 Class template basic_string

See the C++14 library documentation at releases.llvm.org.

22.3.1 Class locale

See the C++14 library documentation at releases.llvm.org.

22.3.1.1.1 Type locale::category

See the C++14 library documentation at releases.llvm.org.

22.3.1.2 locale constructors and destructor

See the C++14 library documentation at releases.llvm.org.

22.3.1.3 locale members

See the C++14 library documentation at releases.llvm.org.

22.3.1.5 locale static members

See the C++14 library documentation at releases.llvm.org.

22.4.1.3 ctype specializations

```
table_size = 256
```

22.4.1.4 Class template codecvt

See the C++14 library documentation at releases.llvm.org.

22.4.5.1.2 time_get virtual functions

See the C++14 library documentation at releases.llvm.org.

22.4.5.3.1 time_put members

See the C++14 library documentation at releases.llvm.org.

22.4.5.3.2 time_put virtual functions

See the C++14 library documentation at releases.llvm.org.

22.4.7.1.2 messages virtual functions

See the C++14 library documentation at releases.llvm.org.

23.3.2.1 Class template array overview

See the C++14 library documentation at releases.llvm.org.

23.3.3.1 Class template deque overview

See the C++14 library documentation at releases.llvm.org.

23.3.4.1 Class template dynarray overview

See the C++14 library documentation at releases.llvm.org.

23.3.5.1 Class template forward_list overview

See the C++14 library documentation at releases.llvm.org.

23.3.6.1 Class template list overview

See the C++14 library documentation at releases.llvm.org.

23.3.7.1 Class template vector overview

See the C++14 library documentation at releases.llvm.org.

23.3.8 Class vector<bool>

See the C++14 library documentation at releases.llvm.org.

23.4.4.1 Class template map overview

See the C++14 library documentation at releases.llvm.org.

23.4.5.1 Class template multimap overview

See the C++14 library documentation at releases.llvm.org.

23.4.6.1 Class template set overview

See the C++14 library documentation at releases.llvm.org.

23.4.7.1 Class template multiset overview

See the C++14 library documentation at releases.llvm.org.

23.5.4.1 Class template unordered_map overview

See the C++14 library documentation at releases.llvm.org.

23.5.4.2 unordered_map constructors

See the C++14 library documentation at releases.llvm.org.

23.5.5.1 Class template unordered_multimap overview

See the C++14 library documentation at releases.llvm.org.

23.5.5.2 unordered_multimap constructors

See the C++14 library documentation at releases.llvm.org.

23.5.6.1 Class template unordered_set overview

See the C++14 library documentation at releases.llvm.org.

23.5.6.2 unordered_set constructors

See the C++14 library documentation at releases.llvm.org.

23.5.7.1 Class template unordered_multiset overview

See the C++14 library documentation at releases.llvm.org.

23.5.7.2 unordered_multiset constructors

See the C++14 library documentation at releases.llvm.org.

25.3.12 Random shuffle

See the C++14 library documentation at releases.llvm.org.

26.4.8 complex transcendentals

```
pow(0,0) = (NaN,NaN)
```

26.5.5 Engines and engine adaptors with predefined parameters

See the C++14 library documentation at releases.llvm.org.

26.5.6 Class random_device

See the C++14 library documentation at releases.llvm.org.

26.5.8.1 In general

See the C++14 library documentation at releases.llvm.org.

26.8 C library

See the C++14 library documentation at releases.llvm.org.

27.2.2 Positioning type limitations

See the C++14 library documentation at releases.llvm.org.

27.5 Iostreams base classes

See the C++14 library documentation at releases.llvm.org.

27.5.3.4 ios_base static members

See the C++14 library documentation at releases.llvm.org.

27.5.5.4 basic_ios flags functions

See the C++14 library documentation at releases.llvm.org.

27.7.2.1.3 Class basic_istream::sentry

See the C++14 library documentation at releases.llvm.org.

27.8.2.1 basic_stringbuf constructors

See the C++14 library documentation at releases.llvm.org.

27.8.2.4 Overridden virtual functions

See the C++14 library documentation at releases.llvm.org.

27.9.1.2 basic_filebuf constructors

See the C++14 library documentation at releases.llvm.org.

27.9.1.5 Overridden virtual functions

See the C++14 library documentation at releases.llvm.org.

28.5.1 Bitmask type syntax_option_type

See the C++14 library documentation at releases.llvm.org.

28.5.3 Implementation-defined error_type

Values of enum `error_type` are present in header `<regex>`.

28.7 Class template regex_traits

See the C++14 library documentation at releases.llvm.org.

28.8 Class template basic_regex

See the C++14 library documentation at releases.llvm.org.

28.10 Class template match_results

See the C++14 library documentation at releases.llvm.org.

30.2.3 Native handles

`native_handle_type` is defined either as `pthread_t` or as `pthread_mutex_t`. Both `pthread_t` and `pthread_mutex_t` are defined as `int`.

30.3.1 Class thread

`native_handle_type` is defined as `pthread_t`, which is defined as `int`.

30.4.1.2.1 Class mutex

`native_handle_type` is not used in this class.

30.4.1.2.2 Class recursive_mutex

`native_handle_type` is defined as `pthread_mutex_t`, which is defined as `int`.

30.4.1.3.1 Class timed_mutex

`native_handle_type` is not used in this class.

30.4.1.3.2 Class recursive_timed_mutex

`native_handle_type` is not used in this class.

30.5.1 Class condition_variable

`native_handle_type` is not used in this class.

30.6.1 Overview

Values of enum class `future_errc`:

```
future_already_retrieved = 1
promise_already_satisfied = 2
no_state = 3
broken_promise = 4
```

Values of enum class `launch`:

```
async = 1
deferred = 2
any = async | deferred
```

A.2 Lexical conventions

No multibyte character or any other non standard character may appear in identifiers.

C.1.10 Clause 16: preprocessing directives

Macro `__STDC__` always expands to 0.

C.4.2.4 Macro NULL

Macro NULL has a value of 0.

D.6 Old iostreams members

See the C++14 library documentation at releases.llvm.org.

19.2. C++14 Implementation Quantities

The following items describe the actual limits of the TASKING C++ compiler as indicated in Annex B, Implementation quantities of the ISO/IEC 14882:2014 standard. In most cases the C++ compiler is limited only by available memory on the user's PC. This is marked with "(no limit)".

Item	Limit
Nesting levels of compound statements, iteration control structures, and selection control structures	(no limit)
Nesting levels of conditional inclusion	(no limit)
Pointer, array, and function declarators (in any combination) modifying a class, arithmetic, or incomplete type in a declaration	(no limit)
Nesting levels of parenthesized expressions within a full-expression	(no limit)
Number of characters in an internal identifier or macro name	9999999
Number of characters in an external identifier	9999999
External identifiers in one translation unit	(no limit)
Identifiers with block scope declared in one block	(no limit)
Macro identifiers simultaneously defined in one translation unit	(no limit)
Parameters in one function definition	(no limit)
Arguments in one function call	(no limit)
Parameters in one macro definition	(no limit)
Arguments in one macro invocation	(no limit)
Characters in one logical source line	(no limit)
Characters in a string literal (after concatenation)	4294967295
Size of an object	4294967295

Item	Limit
Nesting levels for `#include` files	(no limit, however recursive includes of the same file are limited to 10 levels)
Case labels for a `switch` statement (excluding those for any nested `switch` statements)	(no limit)
Data members in a single class	(no limit)
Enumeration constants in a single enumeration	(no limit)
Levels of nested class definitions in a single member-specification	(no limit)
Functions registered by `atexit()`	32
Functions registered by `at_quick_exit()`	32
Direct and indirect base classes	(no limit)
Direct base classes for a single class	(no limit)
Members declared in a single class	(no limit)
Final overriding virtual functions in a class, accessible or not	(no limit)
Direct and indirect virtual bases of a class	(no limit)
Static members of a class	(no limit)
Friend declarations in a class	(no limit)
Access control declarations in a class	(no limit)
Member initializers in a constructor definition	(no limit)
Scope qualifications of one identifier	(no limit)
Nested external specifications	(no limit)
Recursive `constexpr` function invocations	255
Full-expressions evaluated within a core constant expression	(no limit)
Template arguments in a template declaration	65535
Recursively nested template instantiations, including substitution during template argument deduction (14.8.2)	64 (this is the default, but you can change it with C++ compiler option **--pending-instantiations**)
Handlers per `try` block	(no limit)
Throw specifications on a single function declaration	(no limit)
Number of placeholders (20.10.9.1.4)	10

19.3. Deviations from C++11/C++14

While the TASKING C++ compiler is designed to comply to the C++11 and C++14 standards, some language features contain restrictions that have been removed in later C++ standards. The TASKING

C++ compiler adheres to the newer C++ standard in the cases listed in this section. Also some deviations apply to the LLVM libc++ standard library.

Trivially copyable volatile qualified types

In C++11 and C++14, `volatile` qualified types are not considered as trivially copyable. This restriction is no longer present in the C++17 standard. The TASKING C++ compiler adheres to the C++17 standard in this regard, allowing trivially copyable `volatile` qualified types.

Noexcept specifier in typedefs and aliases

In C++11 and C++14, the `noexcept` specifier cannot be applied to typedefs and aliases . This restriction is no longer present in the C++17 standard. The TASKING C++ compiler adheres to the C++17 standard in this regard.

Typeid in constexpr

In C++14, `typeid` is not allowed in `constexpr` contexts if it applies to a polymorphic class. However, in C++20 it is allowed as long as it does not throw an exception, which is guaranteed unless it dereferences a pointer to a polymorphic class type. The TASKING C++ compiler adheres to the C++20 standard, allowing `typeid` applied to a polymorphic class type without using a pointer dereference.

Allocation size

In C++11 and C++14, `std::allocator_traits<Alloc>::max_size()` is defined to return `numeric_limits<size_type>::max()` if the template argument does not have a well-formed `max_size()` function. However, the `max_size()` allocator function must return the largest number that can possibly be passed to `<allocator>::allocate()` for a successful allocation, while `allocate(n)` allocates `n * sizeof(<allocator>::value_type)` bytes.

This is an inconsistency in the C++14 standard, which was fixed in C++17 by defining `std::allocator_traits<Alloc>::max_size()` as `numeric_limits<size_type>::max()/sizeof(value_type)`. The TASKING C++ compiler adheres to the C++17 standard in this regard.

\<functional\> member types of the std::reference_wrapper class template

In \<functional\>, the member types `argument_type`, `first_argument_type`, and `second_argument_type` of the `std::reference_wrapper` class template are deprecated in C++17 and removed in C++20. The TASKING standard C++ library does not provide these type declarations in any standard mode.

\<memory\> garbage collection

In \<memory\>, garbage collection is not supported. Therefore, type `std::pointer_safety` and functions `std:: get_pointer_safety()`, `std::declare_reachable()`, `std::undeclare_reachable()`, `std:: declare_no_pointers()`, and `std::undeclare_no_pointers()` are not declared.

\<memory\> atomic operations for std::shared_ptr

In \<memory\>, atomic operations for `std::shared_ptr` are not available. These are deprecated in C++20. Therefore, functions `std::atomic_is_lock_free(std::shared_ptr)`, `std::atomic_load(std::shared_ptr)`, `std::atomic_load_explicit(std::shared_ptr)`, `std::atomic_store(std::shared_ptr)`, `std::atomic_store_explicit(std::shared_ptr)`, `std::atomic_exchange(std::shared_ptr)`, `std::atomic_exchange_explicit(std::shared_ptr)`, `std::atomic_compare_exchange_weak(std::shared_ptr)`, `std::atomic_compare_exchange_strong(std::shared_ptr)`, `std::atomic_compare_exchange_weak_explicit(std::shared_ptr)`, and `std::atomic_compare_exchange_strong_explicit(std::shared_ptr)` are not declared.